

# Eine funktionale, synchrone Programmiersprache für eingebettete Echtzeitsysteme

Zur Erlangung des akademischen Grades eines  
**Doktor-Ingenieurs**

von der Fakultät für

Elektrotechnik und Informationstechnik  
der Universität Fridericiana Karlsruhe  
genehmigte

**Dissertation**

von

**Dipl.-Inform. Gerd Frick**  
aus Sobernheim

Tag der mündlichen Prüfung: 13. Juli 2006

Hauptreferent: Prof. Dr.-Ing. Klaus D. Müller-Glaser

Korreferent: Prof. Dr. Dr. h. c. Manfred Broy



**FZI-Publikation 2/2006**

FZI Forschungszentrum Informatik  
an der Universität Karlsruhe

## Publikation 2/2006

FZI Forschungszentrum Informatik  
an der Universität Karlsruhe  
Haid-und-Neu-Str. 10-14 · 76131 Karlsruhe

Copyright © 2006 by FZI · >ISSN 0944-3037<

# Kurzfassung

Software eingebetteter Echtzeitsysteme ist durch zeitbezogenes Verhalten und das Anwendungsgebiet der Steuerungs- und Regelungstechnik hinreichend speziell, daß allgemeine Programmiersprachen keine optimale Unterstützung bieten. Andererseits darf auf die Anwendung allgemeiner Prinzipien des Software Engineering bei komplexen eingebetteten Echtzeitsystemen nicht verzichtet werden.

Die in dieser Arbeit entwickelte höheren Programmiersprache FSPL (Functional Synchronous Programming Language) setzt sich zum Ziel, das Anwendungsgebiet der Steuerungs- und Regelungstechnik hinreichend in problemnahen Beschreibungsmitteln abzubilden und zu gleicher Zeit wesentliche Anforderungen aus Software-Engineering-Prinzipien an eine Programmiersprache zu erfüllen. Konkret sollen die Programmierung von quasi-kontinuierlichen Regelungen, reaktiven und sequentiellen Steuerungen in durchgängiger Form ermöglicht und die methodische Anwendung von Abstraktion auf dynamisches Verhalten und Modularisierung nach dem Geheimnisprinzip („information hiding“) unterstützt werden.

Methodisch motiviert führt die Arbeit dazu einerseits bekannte Modelle dynamischen Verhaltens in neuer Weise zu einem durchgängigen Satz von Beschreibungsmitteln zusammen, und greift andererseits auf bewährte allgemeine Konzepte und Abstraktionsmechanismen höherer Programmiersprachen zurück. Grundlage der Verhaltensbeschreibung ist ein zeitdiskretes, synchrones Modell, auf dem funktionale Beschreibungsmittel für quasi-kontinuierliches, reaktives und sequentielles Verhalten aufgebaut werden. Mechanismen zur funktionalen Abstraktion, Datenabstraktion, Kapselung und generischen Abstraktion machen Verhaltensabstraktion in Verbindung mit Modularisierung anwendbar und schützen benutzerdefinierte Abstraktionen. Die Methodik und der Sprachentwurf werden durch ein ausgearbeitetes Programmbeispiel validiert.



# Danksagung

Die vorliegende Arbeit entstand zu wesentlichen Teilen während meiner Zeit als wissenschaftlicher Mitarbeiter am Forschungszentrum Informatik an der Universität Karlsruhe (FZI) im Forschungsbereich Elektronische Systeme und Mikrosysteme (ESM).

Mein besonderer Dank gilt Herrn Professor Dr.-Ing. Klaus D. Müller-Glaser, der die Erstellung der Arbeit bei ESM ermöglichte, für seine langjährige und gerade in kritischen Phasen in entscheidender Weise zielführende Betreuung und seine anhaltende, außerordentliche Unterstützung der Arbeit. Gleichweise danke ich Herrn Professor Dr. Dr. h.c. Manfred Broy für seine freundliche Bereitschaft, das Korreferat zu übernehmen, für das der Arbeit entgegengebrachte Interesse und für seine konstruktiven Ratschläge, die zu einer Schärfung des Profils der Arbeit beigetragen haben.

Danken möchte ich auch meinen ehemaligen Abteilungsleitern Dr.-Ing. Eric Sax und Markus Kühl für die Förderung und kollegiale Unterstützung der Arbeit und die wohlwollend gewährten Freiräume, auch für die bereitgestellte Infrastruktur als Gastwissenschaftler nach dem Ausscheiden aus dem FZI. Besonderer Dank gilt auch meinen ehemaligen Kollegen Barbara Scherrer und Andreas Mayer, mit denen ich nicht nur in den Projekten zusammengearbeitet habe, auf deren Erfahrungen diese Arbeit aufbaut, sondern die sich auch als kritische Leser, Diskussionspartner und Motivatoren in einer schwierigen Phase sehr für das Gelingen dieser Arbeit engagiert haben. Gleichweise danke ich auch Dr.-Ing. Rico Dreier, Stefan Mirevski, Dr.-Ing. Axel Schairer und Dr.-Ing. Manfred Schölzke, die Teile dieser Arbeit engagiert gelesen und wertvolle Hinweise und Anregungen gegeben haben.

Die Arbeit vorlegen zu können verdanke ich nicht zuletzt den anhaltenden Gebeten Vieler, nächst Gott selbst, der gerecht ist in allen Seinen Wegen und gütig in allen Seinen Taten (Psalm 145,17).



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Programmierung eingebetteter Echtzeitsysteme . . . . .	1
1.1.1	Eingebettete Echtzeitsysteme . . . . .	1
1.1.2	Anwendungen der Steuerungs- und Regelungs- technik . . . . .	3
1.1.3	Software Engineering . . . . .	5
1.2	Ziele und eigener Beitrag . . . . .	7
1.3	Gliederung der Arbeit . . . . .	9
<b>2</b>	<b>Grundlagen</b>	<b>11</b>
2.1	Übersicht . . . . .	11
2.2	Programmiersprachen . . . . .	11
2.2.1	Grundparadigmen der Programmierung . . . . .	11
2.2.1.1	Imperative Programmierung . . . . .	12
2.2.1.2	Funktionale Programmierung . . . . .	12
2.2.1.3	Datenflußprogrammierung . . . . .	13
2.2.2	Konzepte höherer Programmiersprachen . . . . .	13
2.2.2.1	Statische Typsysteme . . . . .	14
2.2.2.2	Funktionale/prozedurale Abstraktion . . . . .	14
2.2.2.3	Datenabstraktion und Kapselung . . . . .	15
2.2.2.4	Generische Abstraktion und Polymorphie . . . . .	16

---

2.3	Echtzeitprogrammierung . . . . .	17
2.3.1	Echtzeitprogramme . . . . .	17
2.3.1.1	Berechnungen und Zeitfortschritt . . . . .	17
2.3.1.2	Nebenläufigkeit . . . . .	19
2.3.2	Echtzeitbetriebssysteme . . . . .	20
2.3.3	Echtzeitprogrammiersprachen . . . . .	20
2.3.3.1	Synchrone Sprachen . . . . .	20
2.4	Verhaltensmodelle für Prozeßsteuerungen . . . . .	21
2.4.1	Quasi-kontinuierliches Verhalten . . . . .	22
2.4.2	Reaktives Verhalten . . . . .	22
2.4.3	Sequentielles Verhalten . . . . .	23
<b>3</b>	<b>Methodisches Konzept</b>	<b>25</b>
3.1	Modularisierung von Prozeßsteuerungen . . . . .	25
3.2	Steuerung mit konstruktiver Verhaltensabstraktion . . . . .	26
3.3	Abstrakte Maschinen . . . . .	28
3.4	Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme . . . . .	30
<b>4</b>	<b>Stand der Technik</b>	<b>33</b>
4.1	Übersicht und Vorbemerkungen . . . . .	33
4.2	Imperative Sprachen . . . . .	35
4.2.1	Sequentielle Sprachen . . . . .	35
4.2.2	Nebenläufige Sprachen . . . . .	36
4.2.3	Synchrone imperative Sprachen . . . . .	43
4.3	Funktionale Sprachen . . . . .	45
4.3.1	Synchrone Datenflußsprachen . . . . .	45
4.3.2	Andere synchrone funktionale Sprachen . . . . .	48



---

4.3.3	Andere funktionale Sprachen . . . . .	49
4.4	Sprachen mit imperativem und funktionalem Paradigma	50
4.4.1	Synchrone imperativ/funktionale Sprachen . . . .	50
4.4.2	Andere imperativ/funktionale Sprachen . . . . .	53
4.5	Fazit . . . . .	53
<b>5</b>	<b>Sprachkonzept</b>	<b>57</b>
5.1	Anforderungen . . . . .	57
5.2	Echtzeitfähigkeit . . . . .	58
5.3	Durchgängige Beschreibungsmittel für Prozeßsteuerungen	58
5.4	Infrastruktur einer höheren Programmiersprache . . . .	61
5.5	Umsetzung in der Arbeit . . . . .	62
<b>6</b>	<b>Funktionale, synchrone Modellierung von Prozessen</b>	<b>63</b>
6.1	Übersicht . . . . .	63
6.2	Formalisierungskonzept . . . . .	64
6.3	Zeit und Verhalten . . . . .	67
6.3.1	Zeit . . . . .	67
6.3.2	Prozeß . . . . .	71
6.3.3	Ereignis . . . . .	74
6.4	Signalflüsse . . . . .	87
6.4.1	Übertragungssysteme . . . . .	87
6.4.2	Primitive Signale . . . . .	90
6.4.3	Statische Systeme . . . . .	91
6.4.4	Dynamische Systeme . . . . .	93
6.5	Reaktive Prozesse . . . . .	96
6.5.1	Phasen und Transitionen . . . . .	96
6.5.2	Phasenübergangssysteme . . . . .	100

---

6.6	Sequentielle Prozesse . . . . .	115
6.6.1	Aktionen und Kontrollstrukturen . . . . .	115
6.6.2	Ausnahmebehandlung . . . . .	124
6.7	Zusammenfassung . . . . .	126
6.8	Literatur . . . . .	129
<b>7</b>	<b>Die Programmiersprache FSPL</b>	<b>131</b>
7.1	Übersicht . . . . .	131
7.2	Spracharchitektur . . . . .	132
7.3	Objektsprache . . . . .	134
7.3.1	Daten und Berechnungen . . . . .	135
7.3.1.1	Wahrheitswerte (Booleans) . . . . .	136
7.3.1.2	Zahlen . . . . .	137
7.3.1.3	Schriftzeichen . . . . .	140
7.3.2	Zeit und Verhalten . . . . .	140
7.3.2.1	Zeit . . . . .	141
7.3.2.2	Prozeß . . . . .	142
7.3.2.3	Ereignis . . . . .	142
7.3.3	Signalflüsse . . . . .	144
7.3.4	Reaktive Prozesse . . . . .	147
7.3.5	Sequentielle Prozesse . . . . .	150
7.4	Metasprache . . . . .	152
7.4.1	Ausdrücke, Bezeichner und Typen . . . . .	152
7.4.2	Kardinalzahlen . . . . .	154
7.4.3	Deklarationen und Definitionen . . . . .	155
7.4.4	Funktionen . . . . .	158
7.4.5	Typsystem . . . . .	162
7.4.5.1	Primitive Typen . . . . .	163

---

7.4.5.2	Strukturierte Typen . . . . .	164
7.4.5.3	Typdefinitionen und -funktionen . . . . .	170
7.4.6	Generische Programmierung . . . . .	172
7.4.7	Module . . . . .	174
7.5	Hinweise zur Implementierung . . . . .	177
7.5.1	Laufzeitsystem . . . . .	178
7.5.2	Bibliothekskonzept . . . . .	179
7.5.3	Standard-Bibliothek . . . . .	180
7.5.4	Entwicklungswerkzeuge . . . . .	182
7.6	Zusammenfassung . . . . .	184
7.7	Literatur . . . . .	185
<b>8</b>	<b>Das Entwurfsmuster „Abstrakte Maschine“</b>	<b>187</b>
8.1	Übersicht . . . . .	187
8.2	Möglichkeiten der Verhaltensabstraktion . . . . .	187
8.3	Modulschnittstellen für abstrakte Maschinen . . . . .	190
8.4	Diskussion . . . . .	193
<b>9</b>	<b>Anwendungsbeispiele</b>	<b>195</b>
9.1	Übersicht . . . . .	195
9.2	Quasi-kontinuierliches Verhalten und Signalflüsse . . . . .	196
9.3	Reaktives Verhalten und Phasenübergangssysteme . . . . .	199
9.4	Sequentielles Verhalten und Prozeduren . . . . .	204
9.5	Daten- und Verhaltensabstraktion . . . . .	207
9.6	Modularität und Variabilität . . . . .	211
9.7	Diskussion . . . . .	212
<b>10</b>	<b>Zusammenfassung</b>	<b>215</b>

<b>11 Ausblick</b>	<b>217</b>
<b>A Mathematische Grundlagen</b>	<b>219</b>
A.1 Mengen und Funktionen . . . . .	219
A.1.1 Mengen . . . . .	219
A.1.2 Geordnete Paare . . . . .	224
A.1.3 Relationen . . . . .	224
A.1.4 Funktionen . . . . .	225
A.1.5 Tupel und Folgen . . . . .	227
A.1.6 Mengenprodukt . . . . .	228
A.1.7 Mengensumme . . . . .	229
A.1.8 Mengenpotenz . . . . .	229
A.1.9 Mehrstellige Relationen und Funktionen . . . . .	230
A.1.10 Operationen auf Tupeln und Folgen . . . . .	232
A.2 Aussagen und Prädikate . . . . .	233
<b>B Definition von FSPL</b>	<b>237</b>
B.1 Formalisierungskonzept . . . . .	238
B.1.1 Ausdrücke . . . . .	238
B.1.2 Syntax . . . . .	238
B.1.3 Typisierung . . . . .	243
B.1.4 Denotationale Semantik . . . . .	249
B.2 Funktionale Basissprache . . . . .	253
B.2.1 Ausdrücke und Bezeichner . . . . .	253
B.2.2 Typsystem . . . . .	254
B.2.3 Kardinalzahlen . . . . .	255
B.2.4 Primitive Datentypen . . . . .	256
B.2.5 Deklarationen . . . . .	265

---

B.2.6	Funktionen . . . . .	266
B.2.7	Definitionen . . . . .	268
B.2.8	Strukturierte Typen . . . . .	271
B.2.9	Typdefinitionen und -funktionen . . . . .	278
B.2.10	Generische Programmierung . . . . .	281
B.2.11	Module . . . . .	283
B.3	Echtzeit-Beschreibungsmittel . . . . .	288
B.3.1	Zeit und Verhalten . . . . .	289
B.3.1.1	Zeit . . . . .	289
B.3.1.2	Prozeß . . . . .	291
B.3.1.3	Ereignis . . . . .	293
B.3.2	Signalflüsse . . . . .	296
B.3.3	Reaktive Prozesse . . . . .	305
B.3.4	Sequentielle Prozesse . . . . .	311
B.4	Konkrete Syntax . . . . .	315
B.4.1	Layout . . . . .	316
B.4.2	Bezeichner . . . . .	316
B.4.3	Zahlen . . . . .	317
B.4.4	Zeichenketten . . . . .	317
B.4.5	Typen . . . . .	318
B.4.6	Ausdrücke . . . . .	320
<b>C</b>	<b>Programmbeispiel Kaffeemaschinensteuerung</b>	<b>327</b>
C.1	Überblick über die Anwendung . . . . .	328
C.2	Software-Architektur . . . . .	331
C.3	Basisbibliothek . . . . .	334
C.3.1	Standarddatentypen . . . . .	334
C.3.2	Systemkonstanten . . . . .	335

C.4	Domänenbibliothek . . . . .	335
C.4.1	Überwachungsfunktionen . . . . .	335
C.5	Implementierung der Anwendung . . . . .	336
C.5.1	Wasserzufluß . . . . .	336
C.5.2	Presse . . . . .	338
C.5.3	Brühgruppe . . . . .	343
C.5.4	Kaffeeprodukte . . . . .	347
C.5.5	Kaffeemaschine . . . . .	348
C.5.6	Tastatur . . . . .	354
C.5.7	Display . . . . .	357
C.5.8	Benutzerinteraktion . . . . .	358
C.5.9	Kaffeemaschinenanwendung . . . . .	364
C.5.10	Hauptprogramm . . . . .	373
C.6	Testmodule . . . . .	376
<b>Literatur</b>		<b>379</b>
<b>Lebenslauf</b>		<b>397</b>

# 1. Einleitung

## 1.1 Programmierung eingebetteter Echtzeitsysteme

### 1.1.1 Eingebettete Echtzeitsysteme

*Eingebettete Systeme* (vgl. [FMG02, MGFSK04]) sind Computersysteme, die als Bestandteil technischer Systeme anderer Art informationsverarbeitende Aufgaben – wie die Überwachung und Steuerung technischer Prozesse – wahrnehmen. Eingebettete Systeme sind beispielsweise in Kraftfahrzeugen, Haushaltsgeräten, medizinischen Geräten oder industriellen Anlagen anzutreffen und werden manchmal auch als *Steuergeräte* bezeichnet. Die eingebettete Hard- und Software ist häufig für die grundsätzliche Realisierbarkeit oder innovative Zusatzfunktionen der Gesamtsysteme verantwortlich. Nicht selten nehmen eingebettete Systeme Aufgaben wahr, deren fehlerhafte Ausführung eine beträchtliche Gefahr für Leben und Gesundheit von Menschen oder den Erhalt von Gütern darstellen kann.

*eingebettete  
Systeme*

Ein *eingebettetes* System hat als solches Verbindungsglieder zu seiner physikalischen Umgebung, die entsprechend ihrer Wirkungsrichtung als *Sensoren* bzw. *Aktoren* bezeichnet werden. Das System tritt dadurch in eine Wechselwirkung mit physikalischen Prozessen in seiner Umgebung, die physikalischen Gesetzmäßigkeiten (z. B. Erhaltungsgesetzen) unterliegen. Die Dynamik dieser Prozesse beruht im wesentlichen auf Umwandlung von Energie (mechanisch, elektrisch, thermisch

*Signal-  
verarbeitung*

usw.). Das eingebettete System selbst hingegen ist aus physikalischer Sicht ein *elektronisches* System, in dem Energie in Form von elektrischen *Signalen* nur als Informationsträger maßgeblich ist. Seine Dynamik besteht in der Verarbeitung von Eingangssignalen zu Ausgangssignalen. Insgesamt ergibt sich ein sogenanntes *mechatronisches* System (vgl. [Bre03]). Sensorik und Aktorik vermitteln dabei zwischen Energie- und Informationsübertragung, indem nichtelektrische oder energiereiche elektrische Größen in energiearme elektronische Signale umgewandelt werden und umgekehrt (siehe Abbildung 1.1).

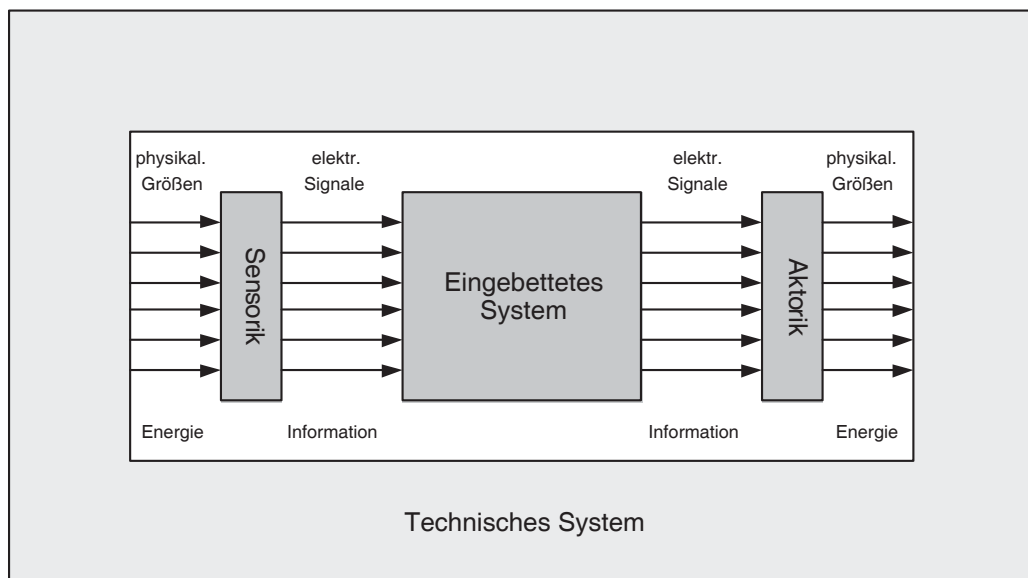


Abbildung 1.1: Eingebettetes System

### Computer

Signalverarbeitende Systeme stellen als informationsverarbeitende („rechnende“) Systeme einen Spezialfall für Computer-Systeme dar. Das gesteuerte technische System besitzt in seinem dynamischen Verhalten häufig *kontinuierliche* Eigenschaften (reellwertig auf einer reellen Zeitachse mit stetigem Verlauf), so daß kontinuierliche Signale zu verarbeiten sind. Computer-Hardware arbeitet jedoch im Regelfall wertdiskret (mit endlichen Zahlenrepräsentationen) und zeitdiskret (getaktet). Dennoch kann das Kontinuum der reellen Zahlen mit einer diskreten Menge von Gleit- oder Festkommazahlen für die meisten Praxisfälle hinreichend approximiert werden; in ähnlicher Weise kann durch ausreichend schnelle Abtastung ein quasi-zeitkontinuierliches Verhalten dargestellt werden.

### Software

Für einen weiten Anwendungsbereich stellt eine Kombination aus



standardisierter Hardware (Digitale Signalprozessoren, Mikroprozessoren, Mikrocontroller<sup>1</sup>) und individueller Software die zweckmäßigste Art der Realisierung des Kerns eines eingebetteten Computer-Systems dar und soll im weiteren als Regelfall angenommen werden. Sie steht im Gegensatz zu individuell entwickelter oder (re)konfigurierbarer Hardware.

Ein eingebettetes System läuft parallel zu seiner Umgebung und tauscht laufend über Signale Information mit ihr aus. Will das System mit seiner Umgebung schritthalten, muß die (diskrete) Signalverarbeitung *in Echtzeit* erfolgen, also synchron zu den entsprechenden Vorgängen in der Umgebung. Die technischen Prozesse bestimmen im wesentlichen die Geschwindigkeiten und dabei zulässige Toleranzen oder Reaktionszeiten. Ihre Einhaltung ist Bestandteil der korrekten Systemfunktion und kann genauso sicherheitsrelevant sein wie die logische und rechnerische Korrektheit der Systemausgaben in Abhängigkeit von den Eingaben. Software-Systeme, deren Verhalten derart an Zeitbedingungen geknüpft ist, werden als *Echtzeitsysteme* bezeichnet. Die zeitliche Dimension als wesentlicher Bestandteil des Verhaltens unterscheidet Echtzeitsysteme signifikant von anderen Anwendungen, für die Software entwickelt wird.

*Echtzeitsysteme*

Die Andersartigkeit der Software eingebetteter Echtzeitsysteme bei zugleich hohen Qualitätsanforderungen und ständig steigender Komplexität erfordert es, Besonderheiten des Anwendungsgebiets und die Methodik des Software Engineering gleichermaßen im Blick zu behalten. Gegenstand dieser Arbeit sind *Steuerungs- und Regelungssysteme* und ihre *Programmierung*. Weiterhin wird einschränkend angenommen, daß es sich um Programme für Mikrocontroller (oder Mikroprozessoren) handelt und daß sich eine Anwendung zusammenhängend durch ein Programm auf einem Rechner darstellen läßt (keine verteilten Systeme).

*Gegenstand der Arbeit*

### 1.1.2 Anwendungen der Steuerungs- und Regelungstechnik

Anwendungen der Steuerungs- und Regelungstechnik bilden einen

*Prozeßsteuerung*

---

<sup>1</sup>*Digitale Signalprozessoren* (DSP) sind dabei für Spezialfälle von Signalverarbeitung (z. B. Filter) optimiert, während *Mikroprozessoren* (µP) und *Mikrocontroller* (µC) über einen allgemeinen Befehlssatz verfügen. Mikrocontroller erweitern einen Mikroprozessor-Kern um typische weitere Rechnerkomponenten für eingebettete Systeme wie Echtzeituhren und Schnittstellen-Bausteine zu einem „System on Chip“ [Ber02].

wichtigen Spezialfall unter den eingebetteten Echtzeitsystemen. Ihre allgemeine Aufgabe ist es, technische Prozesse zu beobachten und ihnen ein bestimmtes Verhalten aufzuprägen (Steuerung) bzw. sie auf einem bestimmten Verhalten gegenüber Störeinflüssen zu stabilisieren (Regelung)<sup>2</sup>. Beide Arten von Anwendungen sollen unter dem Oberbegriff *Prozeßsteuerung* zusammengefaßt werden und treten häufig kombiniert auf. Sie stehen im Gegensatz zu Anwendungen z. B. der Kommunikationstechnik, bei der die Übertragung von Informationen im Mittelpunkt steht.

Für eine *Regelung* charakteristisch ist ein beständiger und quantitativer Vergleich des Ist-Verhaltens eines Prozesses mit seinem Soll-Verhalten, während bei *Steuerung* typischerweise<sup>3</sup> auf diskrete Ereignisse im Prozeß reagiert oder das Verhalten zeitlich vorgeplant wird. Regelung ist so ein quasi kontinuierlicher Vorgang, während bei Steuerung ein Prozeß in diskrete Phasen oder Aktionen zerfällt.

Drei Grundtypen von Aufgabenstellungen der Prozeßsteuerung sind erkennbar:

**Quasi-kontinuierliche Regelung** Eine – im Rahmen der notwendigen zeitlichen Diskretisierung – *kontinuierliche* Berechnung einer *Stellgröße* in Abhängigkeit von der Differenz zwischen einer *Ist-* und einer *Sollgröße* und ihrem zeitlichen Verlauf.

**Reaktive Steuerung** Das Nachführen eines – den Prozeß abbildenden – *Zustandes* und das Anpassen der – den Prozeß beeinflussenden – *Ausgabe* als Reaktion auf diskrete *Ereignisse* im Prozeß oder auf Zeitereignisse.

**Sequentielle Steuerung** Das Prägen eines Prozesses als ein *Ablauf* von *Aktionen*. Das Terminieren einer Aktion löst eine Folgeaktion aus.

---

<sup>2</sup>Steuerung kann dabei entweder wie im englischen Sprachraum als allgemeinerer Begriff (control) gelten, der auch den Fall der Regelung (closed loop control) beinhaltet, oder man verwendet beide Begriffe im Gegensatz zueinander, wobei dann mit Steuerung (open loop control) eine Nicht-Regelung gemeint ist. Steuerung und Regelung stehen also in einem ähnlichen Verhältnis zueinander wie *Relation* und *Funktion* in der Mathematik. Die Begriffe *offene* und *geschlossene Steuerung* sind zwar auch im deutschen Sprachraum anzutreffen [Gün97], aber eher unüblich. In dieser Arbeit wird der Begriff Steuerung sowohl in der allgemeinen wie in der speziellen Bedeutung verwendet, letzteres immer dann, wenn (wie im Rest des Abschnitts) unterschieden wird.

<sup>3</sup>Auch als *diskrete* Steuerung bezeichnet [WB05]. Es gibt auch kontinuierliche Steuerungen als Grenzfall von Regelungen (keine unbekannten Störeinflüsse).

Eine *sequentielle* Steuerung kann prinzipiell auf eine *reaktive* Steuerung zurückgeführt werden. Die Terminierung einer Aktion ist ein Ereignis, die Ausführung der Aktion spiegelt sich in einem Zustand und wird bestimmt durch die Ausgabe. Der Blickwinkel auf die Zusammenhänge ist bei der sequentiellen Steuerung aber ein anderer und bietet sich z. B. bei fertigungs- oder verfahrenstechnischen Abläufen als natürlicher an.

Die drei Grundtypen können nahezu beliebig kombiniert werden, um komplexere Prozeßsteuerungen zu bilden. So kann z. B. eine Steuerung reaktiv zwischen verschiedenen Betriebsmodi eines Systems umschalten, in denen unterschiedliche quasi-kontinuierliche Regelungen oder sequentielle Steuerungen (oder auch reaktive Steuerungen) zur Anwendung kommen. Oder ein überlagerter quasi-kontinuierlicher Regler gibt eine abstrakte Stellgröße vor, die auf eine Kombination von konkreteren Stellgrößen umgesetzt wird, die von unterlagerten Reglern eingestellt und reaktiv in Abhängigkeit von Schwellwerten (das Erreichen eines Schwellwerts wird zum Ereignis) koordiniert werden. Ein drittes Beispiel ist eine sequentielle Steuerung, in der eine Aktion einen dynamischen Verlauf einer Ausgabegröße beinhaltet, der quasi-kontinuierlich oder reaktiv (oder auch durch eine Sequenz von Unteraktionen) eingestellt wird.

Die Methoden der diskreten Steuerungstechnik sind informatiknah (z. B. Automatentheorie), während sich die kontinuierliche Regelungstechnik mathematischer Modelle (z. B. Differentialgleichungen) und Analysemethoden bedient, die sich von Kerngebieten der Software-Technik deutlich unterscheiden. Ein Standardwerk der Regelungstechnik ist [Föl92], stärker auf zeitdiskrete (quasi-kontinuierliche) Regelungen geht z. B. [Gün97] ein. Eine Einführung bietet auch [WB05], genauso in Prozeßsteuerungen allgemein.

### 1.1.3 Software Engineering

*Software Engineering* ist nach Parnas „the multi-person construction of multi-version programs“ [Par78]. Die Entwicklung und Weiterentwicklung großer und variantenreicher Systeme wirft einerseits ein Managementproblem auf (Anforderungsmanagement, Konfigurationsmanagement, Projektmanagement usw., siehe [Som01]), stellt andererseits aber auch hohe Anforderungen an den Entwurf. Die Software muß *änderbar*, *erweiterbar* und *reduzierbar* (vgl. [Par79]) sein. Mehrere *Varianten* eines Programms nebeneinander und Konzepte von Software-

*Produktlinien* [WL99] führen zu *Programmfamilien* [Par76] anstelle einzelner Programme. Ein hoher Grad an *Wiederverwendung* wird hierbei angestrebt. Nicht zuletzt müssen Arbeitspakete für einzelne Entwickler geschaffen werden, die weitgehend unabhängig voneinander bearbeitet werden können.

*Modularisierung*

Zentrales Konzept, um Systeme arbeitsteilig, für spätere Weiterentwicklung und mit hoher Wiederverwendung von Teilen zu entwickeln, ist das der *Modularisierung*. Das von Parnas eingeführte Verständnis von Modulen [Par72] ist unter dem Prinzip „Information Hiding“ bekannt. Es verbindet mit dem Organisationsprinzip der Arbeitsteilung die Entwurfsprinzipien *Dekomposition* und *Abstraktion*.

*Information Hiding*

Ein Modul ist einerseits ein Arbeitspaket, das von einem einzelnen Entwickler bearbeitet wird, und trägt andererseits als Komponente zu einem Gesamtsystem bei (Dekomposition). Zwischen Modulen gibt es Abhängigkeiten, wenn ein Modul Funktionalität eines anderen verwendet. Ziel von Modularisierung nach dem Information-Hiding-Prinzip (Abstraktion) ist es, Entwurfsentscheidungen so weit wie möglich voneinander zu entkopplern. Die gegenseitigen Annahmen, die Module übereinander machen können, sind zu minimieren (minimale Kopplung) und die von einer speziellen Entwurfsentscheidung direkt abhängigen Teile – und nur diese – an einer Stelle zusammenzufassen (maximale Kohäsion). Entwurfsentscheidungen gleichzustellen sind Anforderungen an das System, die mit hoher Wahrscheinlichkeit Änderungen unterliegen (vgl. [Mel02]).

*abstrakte Schnittstellen*

Die gegenseitigen Annahmen der Module werden als *Schnittstellen* explizit gemacht. Dem Information-Hiding-Prinzip (Abstraktionsprinzip) folgend, sind Schnittstellen möglichst abstrakt. Abstrakte Schnittstellen sind „stabil“, d. h. sie ändern sich mit weit geringerer Wahrscheinlichkeit als die „Geheimnisse“, die sie verbergen [Par78]. Solange sich die Schnittstellen nicht ändern, hat eine Änderung innerhalb eines Moduls keine Auswirkungen auf andere Module.

*Architektur*

Die Zerlegungsstruktur eines Systems in Module und ihre Kopplung heißen die *Architektur* des Systems (oder auch der System-Entwurf). Die in der Architektur enthaltenen *globalen* und die in den Modulen gekapselten *lokalen* Entwurfsentscheidungen für ein System sind komplementär. Eine gute Architektur zeichnet sich dadurch aus, daß sie bei Systemänderungen stabil bleibt, d. h. daß Systemänderungen in der Regel auf einzelne Module beschränkt bleiben. So ist für Tom DeMarco (nach [HR02]) Architektur „a framework for change“.

Um Information Hiding beim Architektur- und Modulentwurf praktizieren und eine stabile Architektur erzielen zu können, ist einerseits eine gute Durchdringung der Anwendungen und ihrer Anforderungen notwendig. Andererseits dürfen aber auch die zur Realisierung der Software zur Verfügung stehenden Mittel Abstraktion nicht im Wege stehen. Umgekehrt sollte auch das abgesicherte Einhalten von getroffenen Architekturentscheidungen (wie die ausschließliche Benutzung vorgesehener Schnittstellen ohne zusätzliche Annahmen) von den eingesetzten Werkzeugen möglichst gut unterstützt werden. *Ein*, wenn nicht *das* zentrale Werkzeug ist dabei die zur Programmerstellung verwendete Sprache.

## 1.2 Ziele und eigener Beitrag

Ziel der Arbeit ist ein Beitrag zur Programmierung von eingebetteten Echtzeitsystemen mit Mitteln, die den Belangen des Anwendungsgebiets und des Software Engineering gleichermaßen gerecht werden.

Ausgangspunkt der Arbeit ist ein – auf das Anwendungsgebiet zugeschnittenes – methodisches Konzept, das Abstraktion beim Entwurf in den Mittelpunkt stellt:

- Steuerung mit konstruktiver Verhaltensabstraktion und
- Modularität durch Abstrakte Maschinen.

Dabei wird der Begriff der konstruktiven Verhaltensabstraktion neu eingeführt. Aus Charakteristika des Anwendungsgebiets und dem methodischen Konzept werden Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme abgeleitet:

- Die Eignung für Echtzeitprogrammierung allgemein und speziell für Systeme mit harten Echtzeitbedingungen,
- adäquate Beschreibungsmittel für quasi-kontinuierliche Regelungen und reaktive und sequentielle Steuerungen in einer Durchgängigkeit, die konstruktive Verhaltensabstraktion ermöglicht, und
- Mechanismen zur funktionalen Abstraktion, zur Datenabstraktion und Kapselung sowie zur generischen Abstraktion in einer Allgemeinheit, die abstrakte Maschinen als Module ermöglicht.

Hauptgegenstand der Arbeit ist dann der Entwurf einer höheren, anwendungsnahe Programmiersprache, die die gestellten Anforderungen besser als bisher bekannte Sprachen erfüllt. Die Sprache mit dem Namen FSPL (Functional Synchronous Programming Language) ist hinsichtlich ihres Paradigmas der Programmierung eine rein funktionale Programmiersprache und hinsichtlich ihres Zeit- und Ausführungsmodells eine synchrone Sprache. Zur Validierung von Methodik und Sprachentwurf und zur Beleuchtung der Ergebnisse dient ein ausgearbeitetes Programmbeispiel, das von existierender Steuerungssoftware eines Kaffeeautomaten abgeleitet ist.

Alleinstellungsmerkmal und Beitrag dieser Arbeit ist eine Programmiersprache, die dedizierte Beschreibungsmittel für quasi-kontinuierliches, reaktives *und* sequentielles Steuerungsverhalten bereitstellt *und* Verhaltensabstraktion als Schnittkriterium für Module unterstützt. Damit wird gezeigt, daß ein Sprachentwurf möglich ist, der das Anwendungsgebiet der Steuerungs- und Regelungstechnik hinreichend vollständig und genau in problemnahen, konstruktiven Beschreibungsmitteln abbildet und zu gleicher Zeit wesentliche Anforderungen aus Software-Engineering-Prinzipien an eine Programmiersprache erfüllt und diese Prinzipien auf das Anwendungsgebiet anwendbar macht.

Die Wurzeln der hier vorgelegten Arbeit liegen in zwei Forschungsprojekten. Im COMTESSA-Projekt<sup>4</sup> (abgeschlossen 2002, Homepage: [COM02], Veröffentlichungen: [FMG01, FSMG01, FMG02, FMG03, MGFSK04]) wurde durch den Entwurf einer Zwischensprache eine semantische Integration dreier Modellierungssprachen aus dem Bereich eingebetteter Systeme<sup>5</sup> zum Modellaustausch durch Modellkonvertierung erzielt. Thema im IDESA-Projekt<sup>6</sup> (abgeschlossen 2004, Homepage: [IDE04], Veröffentlichungen: [FMG04, FSMG04]) war die modellbasierte Entwicklung eingebetteter Systeme auf Basis von UML 2.0 und Codegenerierung. Hierzu wurde eine Entwurfsmethodik und beispielhaft eine Software-Architektur entwickelt, deren Prinzipien zu dem in dieser Arbeit zugrundegelegten methodischen Konzept geführt haben.

---

<sup>4</sup>Gefördert vom Wirtschaftsministerium Baden-Württemberg im Rahmen des Verbundforschungsprojekts „Zukunftsoffensive Junge Generation“

<sup>5</sup>Untermengen der Sprachen der Werkzeuge Statemate von I-Logix, Simulink/Stateflow von The MathWorks und Rodon der Firma R.O.S.E. Informatik aus Heidenheim

<sup>6</sup>Gefördert von der Europäischen Kommission als Cooperative Research (CRAFT)-Forschungsprojekt innerhalb des 5. Rahmenprogramms (IST-2001-55024)

## 1.3 Gliederung der Arbeit

Kapitel 2 führt überblicksartig in Paradigmen und Konzepte von Programmiersprachen, Echtzeitprogrammierung und für das Anwendungsgebiet der Steuerungs- und Regelungstechnik wichtige Modelle dynamischen Verhaltens ein. Kapitel 3 stellt das methodische Konzept und die daraus abgeleiteten Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme vor. Kapitel 4 untersucht den Stand der Technik anhand dieser Kriterien und zeigt Verbesserungspotential auf.

Der Hauptteil der Arbeit beginnt mit dem Konzept der neu entwickelten Sprache (Kapitel 5) und zerfällt dann in zwei große Entwurfsschritte: In Kapitel 6 wird ein integriertes semantisches Modell entwickelt, das die durchgängige Modellierung quasi-kontinuierlichen, reaktiven und sequentiellen Verhaltens ermöglicht und Verhaltensabstraktion unterstützt. In Kapitel 7 wird daraus eine Programmiersprache entwickelt, die die Primitive zur Verhaltensmodellierung in Sprachkonstrukte umsetzt und die Anwendung des Abstraktionsprinzips durch allgemeine sprachliche Mittel zur Bildung und Absicherung von Abstraktionen unterstützt. Kapitel 7 selbst beschreibt den Entwurf der Sprache in seinen Grundzügen; eine vollständige Definition der Syntax und Semantik befindet sich in Anhang B. Einige mathematische Grundlagen für die Modellbildung und die Sprachdefinition sind in Anhang A dargestellt.

Ergänzt wird der Sprachentwurf in Kapitel 8 um ein Entwurfsmuster „Abstrakte Maschinen“, das das methodische Konzept in den Mechanismen der Sprache widerspiegelt. Kapitel 9 beleuchtet die Konzepte der entwickelten Sprache an kleinen Anwendungsbeispielen, die überwiegend dem ausgearbeiteten größeren Programmbeispiel entnommen sind, das vollständig in Anhang C angegeben wird und dem Referenzbeispiel aus dem im vorigen Abschnitt angeführten IDESA-Projekt entspricht. Kapitel 10 und 11 fassen die Ergebnisse und offene Fragestellungen zusammen.





## 2. Grundlagen

### 2.1 Übersicht

Abschnitt 2.2 gibt eine Klassifikation von Programmiersprachen nach ihrem grundlegenden Paradigma der Programmierung und beschreibt eine Auswahl von allgemeinen Konzepten und Mechanismen höherer Programmiersprachen, die für die Belange des Software Engineering als wichtig erachtet werden. Abschnitt 2.3 führt in die Problematik der Echtzeitprogrammierung ein. Abschnitt 2.4 gibt eine Übersicht zu grundlegenden Modellen dynamischen Verhaltens, die typische steuerungs- und regelungstechnische Problemstellungen repräsentieren. Verhalten dieser Art abzubilden wird im weiteren als Kriterium für die Nähe einer Sprache zum Anwendungsgebiet betrachtet.

### 2.2 Programmiersprachen

#### 2.2.1 Grundparadigmen der Programmierung

Eine Programmiersprache ist von einem zugrundeliegenden Berechnungsmodell geprägt, das die Denkweise bei der Programmierung (oder: das Programmierparadigma) bestimmt. Von den bekannten Paradigmen sind die *imperative*, die *funktionale* und die *Datenflußprogrammierung* für Echtzeitanwendungen eingebetteter Systeme relevant, wobei die hier betrachtete Variante der Datenflußprogrammierung sich als eingeschränkte Variante der funktionalen Programmierung auffassen läßt. Die *logische* oder *prädikative* Programmierung

(einzige Sprache mit Bekanntheitsgrad ist *Prolog*), die Programmierung als Beweisen in einem System von Tatsachen und Schlußfolgerungen auffaßt [E<sup>+</sup>88], ist für Echtzeitanwendungen der Steuerungs- und Regelungstechnik bislang ohne nennenswerte Bedeutung. Die *objektorientierte* Programmierung ist eine Unterart der imperativen Programmierung. Weitere Paradigmen werden in aktuellen Forschungsarbeiten untersucht (z. B. [WI02, SJG99, GJSB95, AMS05]); sie werden in dieser Arbeit nicht näher betrachtet.

### 2.2.1.1 Imperative Programmierung

Das weiterhin vorherrschende Paradigma in der Programmierung ist das *imperative*. *Imperative Programmiersprachen*, von Backus [Bac78] auch *Von-Neumann-Sprachen* genannt, sind im wesentlichen der Arbeitsweise eines Von-Neumann-Rechners mit Hauptspeicher und CPU nachempfunden:

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages... [Bac78, S. 615–616]

Elementarer Befehl und Charakteristikum für imperative Hochsprachen<sup>1</sup> ist also die Zuweisung (eines Wertes an eine Variable oder Speicherstelle). Ältere imperative Sprachen (z. B. PASCAL [Wir71] oder C [KR88]) zielen meist auf *prozedurale* Programmstrukturen nach den Prinzipien der strukturierten Programmierung nach Dijkstra [Dij72] ab; neuere imperative Sprachen (z. B. C++ [Str97] oder Java [AGH05]) sind in Erweiterung dessen meist *objektorientiert* (siehe Abschnitt 2.2.2.3).

### 2.2.1.2 Funktionale Programmierung

Charakteristikum für *funktionale* (oder *applikative*) Sprachen wie LISP [McC62, May88], ML [MTHM97] oder Haskell [Dav92, Has] ist das *Programmieren mit Funktionen*. Funktionen sind erstklassige Objekte und können somit selbst Argumente und Ergebnis der Anwendung von Funktionen sein (Funktionen höherer Ordnung). Durch sogenannte  $\lambda$ -Ausdrücke (siehe Anhang A.1.4 – nach dem auf Alonzo

<sup>1</sup>Auch Assembler-Sprachen können unmittelbar zu den imperativen Sprachen gezählt werden.

Church zurückgehenden  $\lambda$ -Kalkül [Chu41, Bar84]) können Funktionen anonym (ohne sie mit Namen zu definieren) in Form eines Ausdrucks angegeben werden. Anstelle einer Verknüpfung von Befehlen tritt zur Programmierung die Verknüpfung von Funktionen. *Schleifen* in prozeduralen Algorithmen entsprechen in der funktionalen Programmierung *rekursive* Funktionen.

Manche funktionale Programmiersprachen (wie z. B. LISP oder Standard ML) sind nicht *rein* funktional, sondern enthalten auch imperative Elemente (insbesondere die Zuweisung), deren Auswirkungen als „Seiteneffekte“ bei der Auswertung von Funktionen betrachtet werden.

Eine Einführung in die funktionale Programmierung bietet z. B. [Dav92].

### 2.2.1.3 Datenflußprogrammierung

*Programmieren mit Datenflüssen* ist eine spezielle, beschränkte Form der funktionalen Programmierung.<sup>2</sup> Datenflüsse sind endliche oder unendliche Sequenzen von Daten (auch *Ströme* genannt). In einer *Datenflußsprache* (wie z. B. Lucid [WA85]) ist ein Programm im wesentlichen die Repräsentation eines Datenflußgraphen, bei dem die gerichteten Kanten Datenflüsse und die Knoten funktionale Transformationen (Operationen) auf Datenflüssen darstellen, und der gewöhnlich Zyklen enthält. Zu den elementaren Operationen können neben solchen, die elementweise angewendet werden (z. B. arithmetisch-logische Operationen), beispielsweise auch Verschiebeoperationen gehören. Mit selbstdefinierten, datenflußwertigen Funktionen und lokalen, rekursiven Definitionen ist strukturierte Programmierung möglich.

In funktionalen Sprachen, die die unendliche Listen unterstützen (wie z. B. Haskell), sind die grundlegenden Mechanismen der Datenflußprogrammierung als Spezialfall enthalten. (Siehe [WA85, CP00].)

## 2.2.2 Konzepte höherer Programmiersprachen

Speziell unter Software-Engineering-Gesichtspunkten ist es wichtig, daß die Programmierung eingebetteter Echtzeitsysteme von einigen Konzepten höherer Programmiersprachen profitiert. Die im folgenden beschriebenen Konzepte enthalten Mechanismen, die die Kapselung

---

<sup>2</sup>Dabei wird hier nur eine spezielle Art der Datenflußprogrammierung betrachtet, die auch als *pipelined dataflow* bezeichnet wird [WA85]. Für eine allgemeinere Übersicht, siehe [JHM04].

von Entwurfsentscheidungen und eine systematische Wiederverwendung und Variantenbildung unterstützen (Abstraktions- und Variationsmechanismen).

Der Abschnitt über Typsysteme ist größtenteils ein Exzerpt aus [Pie02]. Die weiteren Abschnitte folgen in der Darstellung an einigen Stellen (teils unverändert) Watt [Wat04] oder Sebesta [Seb04], die das Thema umfassender abhandeln. Eine hervorragende theoretisch fundierte Darstellung von Programmiersprachenkonzepten bietet Reynolds [Rey98].

### 2.2.2.1 Statische Typsysteme

Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure software's correctness. [Lee00]

Typsysteme in Programmiersprachen sind eine leichtgewichtige formale Methode, um bestimmte Programmfehler auszuschließen. Programmiersprachliche Ausdrücke werden dabei klassifiziert nach der Art der Werte, die sie beschreiben. *Statische* Typsysteme sind so ausgelegt, daß alle Typprüfungen zur Kompilierzeit vorgenommen werden können.

Typen schützen die einer Programmiersprache eigenen und benutzerdefinierte Abstraktionen (z. B. abstrakte Datentypen). Ein Typsystem bildet so die Basis für die Ausgestaltung der weiteren Konzepte.

### 2.2.2.2 Funktionale/prozedurale Abstraktion

*Funktionen,  
Prozeduren*

Funktionen und Prozeduren in ihrer in Programmiersprachen bekanntesten Form sind Abstraktionen für *Berechnungen* (Unterprogramme). In imperativen Sprachen sind üblicherweise Prozeduren (Funktionsprozeduren und eigentliche Prozeduren<sup>3</sup>) anzutreffen, während funktionale Sprachen (einschließlich Datenflußsprachen) eine direkte Darstellung von Funktionen verwenden. Wenn man von dem Aufrufmechanismus absieht, können (eigentliche) Prozeduren mit Parametern als kommandowertige Funktionen betrachtet werden.

*Funktionen  
höherer  
Ordnung*

Eine Funktion in allgemeiner Form (siehe auch Anhang A.1.4 und

<sup>3</sup>Eine Funktionsprozedur wird als *Ausdruck* zur *Auswertung* aufgerufen, eine eigentliche Prozedur als *Kommando* zur *Ausführung*. Je nach Sprache treten für beides die Bezeichnungen *Funktion*, *Prozedur* und/oder *Methode* auf. Prozeduren („Funktionen“) in C und davon abgeleiteten Sprachen haben einen Hybridcharakter, da C Ausdrücke und Kommandos nicht strikt trennt.

A.1.9) kapselt einen Ausdruck beliebigen Typs mit freien Variablen beliebigen Typs, die als *Funktionsparameter* erscheinen. Bei jeder *Anwendung* der Funktion können die Parameter beliebig (im Rahmen der für sie spezifizierten Typen) belegt werden (*Argumente*). Treten Funktionen selbst als Argumente oder Ergebniswerte von Funktionen auf, spricht man von Funktionen *höherer Ordnung*.

Funktionen sind ein *Abstraktionsmechanismus* insofern, als sie die Details eines beliebig umfangreichen Ausdruck auf die Identität eines Funktionsobjekts und einen oder mehrere Parameter reduzieren<sup>4</sup>. Er enthält einen *Variationsmechanismus*, mit dem viele ähnliche Ausdrücke erzeugt werden können, in Form der Parameter.

### 2.2.2.3 Datenabstraktion und Kapselung

Among the last 45 years in programming methodologies and programming language design, data abstraction is one of the most profound. [Seb04]

*Abstrakte Datentypen* [Gut77] sind – informell ausgedrückt – Datentypen, deren Repräsentationen „privat“ sind, die aber mit „öffentlichen“ Operationen ausgestattet sind. Programmteile, die einen abstrakten Datentyp benutzen, können Objekte dieses Typs über die Operationen konstruieren und verwenden, haben jedoch keinen Zugriff auf die interne Repräsentation des Objekts.

*Abstrakte  
Datentypen*

Datenabstraktion baut auf funktionaler/prozeduraler Abstraktion auf, denn integraler und zentraler Bestandteil von jeder Datenabstraktion sind die Operationen, die (sofern es sich nicht um Konstanten handelt) Funktionen oder Prozeduren im Sinn von Abschnitt 2.2.2.2 sind.

Abstrakte Datentypen kapseln je einen Datentyp mit zugehörigen Operationen. Das Konzept kann verallgemeinert werden zu Kapselungen beliebiger Typen mit zugehörigen Operationen (*abstrakte Typen*; siehe Abschnitt 2.2.2.2 bezüglich der Allgemeinheit des Funktionskonzepts) und zu Kapselungen mehrerer zusammengehöriger Typen mit zugehörigen, auch typübergreifenden Operationen. Das verallgemeinerte Konzept wird von verschiedenen Autoren unterschiedlich als *Kapselungen*, *Pakete* oder *Module* bezeichnet. In dieser Arbeit wird, Reynolds [Rey98] folgend, der Begriff *Modul* verwendet.

*Module*

<sup>4</sup>Wird für die Funktion ein Name definiert, so repräsentiert die *Signatur* der Funktion, bestehend aus ihrem Namen und den Typen der Parameter und des Ergebnisses, syntaktisch die Abstraktion.

Die *Signatur* eines abstrakten Datentyps oder eines Moduls, bestehend aus dem oder den Typnamen und den Signaturen der Operationen, definiert syntaktisch die *Schnittstelle* des Moduls und kann als dessen Typ verstanden (und konzeptionell so im Typsystem verankert) werden. Offensichtlich kann es mehrere Module mit gleichem Typ (gleicher Schnittstelle) geben, die die abstrakten Typen unterschiedlich implementieren. Das Konzept der abstrakten Datentypen bzw. der Module mit abstrakten Typen enthält daher sowohl einen Abstraktionsmechanismus (die Implementierung der Typen und der Operationen wird unterdrückt) als auch einen Variationsmechanismus (alternative Modulimplementierungen zur gleichen Schnittstelle).

Objektorientierung in imperativen Sprachen ist eine weitere Variantenentwicklung des Konzepts der Datenabstraktion. Eine *Klasse* in Reinform definiert die Struktur einer Familie von Objekten, von denen jedes mit einer Identität ausgestattet ist und einen Satz privater, imperativer *Variablen* enthält, ausgestattet mit öffentlichen Operationen.<sup>5</sup> Mit Objektorientierung wird darüber hinaus noch das Konzept der *Vererbung* der Eigenschaften einer Klasse an *Unterklassen* (die zusätzliche Variablen und Operationen einführen) verbunden. Unterklassen und Vererbung realisieren eine Form von Polymorphie (siehe Abschnitt 2.2.2.4).

#### 2.2.2.4 Generische Abstraktion und Polymorphie

*Module mit Parametern*

Das Ergebnis von generischer Abstraktion sind *Module mit Parametern*. Solche Module (oder abstrakten Datentypen) werden erst instantiiert, bevor die darin enthaltenen Definitionen verwendbar sind. Klassische Beispiele sind abstrakte Datentypen für Stacks, Listen, Mengen usw., die unabhängig vom Datentyp der Elemente implementiert werden und für jeden beliebigen Datentyp instantiiert werden können.

*Typen als Parameter*

Sofern Module erstklassige Objekte sind, ist generische Abstraktion zum Teil bereits die Anwendung funktionaler Abstraktion im allgemeinen Sinn (siehe Abschnitt 2.2.2.2) auf Module. Damit können Konstanten, Funktionen oder andere Module als Parameter zu Modulen auftreten. Hinzukommen muß noch die Möglichkeit, *Typen als Parameter* zu verwenden.

Typen als Parameter erstklassiger Objekte, insbesondere von Funktionen, führen zu *parametrischer Polymorphie*. Polymorphie besagt, daß

<sup>5</sup>Die Typen der Variablen zusammengekommen können als Record-Typ interpretiert werden, der die Repräsentation eines abstrakten Datentyps darstellt.

eine Funktion oder Operation auf Objekte mehrerer Typen anwendbar ist. Man unterscheidet vier wesentliche Arten von Polymorphie [CW85]:

**Parametrische Polymorphie** Ein impliziter oder expliziter Typparameter der polymorphen Funktion bestimmt den Typ des Arguments für jede Anwendung der Funktion.

**Inklusionspolymorphie** Eine Operation ist auf Objekte eines Typs und auf Objekte von *Untertypen* anwendbar (die durch einen Untertyp repräsentierte Menge von Objekten ist eine Teilmenge der durch den Typ repräsentierten Menge). Diese Art von Polymorphie liegt bei Vererbung bzw. Unterklassen in objektorientierten Sprachen vor (vgl. Abschnitt 2.2.2.3).

**Überladung** Der selbe Bezeichner wird für verschiedene Funktionen verwendet. Der Kontext entscheidet bei der Anwendung, um welche Funktion es sich handelt. Diese Art von Polymorphie liegt in den meisten Programmiersprachen bei arithmetischen Operationen vor, wo das gleiche Symbol (z. B. +) für unterschiedliche arithmetische Datentypen verwendet wird.

**Automatische Typumwandlung** Das Argument der Funktion wird in den Typ des Parameters umgewandelt.

Im Rahmen des Typsystems bieten *Typfunktionen* (Typen als Parameter zu Typen) ferner die Möglichkeit, den Typ parametrisch polymorpher Objekte bezeichnen zu können.

Parametrische Polymorphie als Mittel für generische Abstraktion dehnt den Abstraktions- und Variationsmechanismus der funktionalen Abstraktion auf Typen als Parameter aus. Generische Abstraktion wird besonders als Mittel zur *Wiederverwendung* gesehen.

## 2.3 Echtzeitprogrammierung

### 2.3.1 Echtzeitprogramme

#### 2.3.1.1 Berechnungen und Zeitfortschritt

Die klassische Theorie der Informatik betrachtet ein Programm als Beschreibung einer *Berechnung*, die aus einer Eingabe eine Ausgabe erzeugt oder, mit anderen Worten, eine Funktion berechnet. Sie beschäftigt sich dann mit Fragestellungen wie Berechenbarkeit, Komplexität (Laufzeit und Speicherbedarf), Terminierung, Korrektheit usw. Die

Eingabe steht einmal am Anfang der Berechnung, die Ausgabe erfolgt am Ende der Berechnung, sofern die Berechnung terminiert.

Interaktive Systeme und speziell Echtzeitsysteme enthalten Programme, deren Ein- und Ausgaben sich über die Laufzeit verteilen und die im Normalfall nicht terminieren. Die Eingabe und die Ausgabe eines Echtzeitprogramms zerfallen in viele einzelne Ein- und Ausgaben zu diskreten Zeitpunkten, zwischen denen die Zeit fortschreitet. Das Programm steht dabei in Interaktion mit seiner Umgebung, bei eingebetteten Systemen also mit einem technischen Prozeß, aus dem sich die zeitlichen Bedingungen der Ein- und Ausgabe (*Echtzeitbedingungen*) ergeben. Bezogen auf die einzelnen Ausgaben oder internen Zustandsübergänge zerfällt das Programm in viele einzelne Berechnungen der klassischen Art.

*harte, feste und weiche Echtzeitbedingungen*

Da die einzelnen Berechnungen wie auch im klassischen Fall Zeit benötigen, muß die Ausführung der Berechnungen mit dem Zeitfortschritt bezüglich der Interaktion mit der Umgebung abgeglichen sein. Die Berechnungen müssen rechtzeitig terminieren, die Ein- und Ausgaben müssen „zeitlich korrekt“ [WB05] erfolgen. Man unterscheidet dabei „harte“, „feste“ und „weiche“ Echtzeitbedingungen. Ein Ergebniszeitpunkt für eine Berechnung ist *hart*, wenn ein Überschreiten katastrophale Auswirkungen haben kann, *fest*, wenn das Ergebnis dadurch nur wertlos wird und die Berechnung abgebrochen werden kann, und *weich*, wenn das Ergebnis auch nach Überschreiten der Zeit noch nützlich ist [Kop97]. Ein Echtzeitsystem heißt *hart*, wenn es mindestens eine harte Echtzeitbedingung einhalten muß, ansonsten *fest* bzw. *weich* [WB05, Kop97]. Die *worst-case execution time* (WCET) ist die maximale Dauer einer Berechnung unter allen anzunehmenden Bedingungen (vgl. [Kop97]).

*worst-case execution time*

Die zeitliche Abfolge der Berechnungen und die Art der Eingaben kann nach drei verschiedenen Mustern erfolgen:

1. Eine Berechnung wird *zyklisch so schnell wie möglich* ausgeführt. In jedem Durchlauf wird die Eingabe gelesen (Eingangssignale werden abgetastet). Aufgrund von i. a. eingabeabhängig schwankenden Berechnungszeiten erfolgt eine *nicht äquidistante* Abtastung, was zu schwer beherrschbaren Reglereigenschaften führen kann. Bei relativ zur Durchlaufzeit langsamen Umgebungsprozessen erfolgt die Abtastung häufiger als zur Regelung erforderlich; werden Ereignisse detektiert, erfolgt dies per *Busy Waiting*.



Da der Rechner beständig rechnet, ist der Stromverbrauch hoch. Aus Anwendungssicht ist ein solches Vorgehen nur in einfachen Fällen sinnvoll.

2. Eine Berechnung wird *periodisch in einem festen Zeitraster* ausgeführt (*Zeitsteuerung*). Bei jeder Ausführung werden Eingangssignale abgetastet (äquidistant). Aus Anwendungssicht ist die Zeit diskret, auch Ereignisse werden synchron zu diesem Zeitraster detektiert. Bei Unterabtastung können Ereignisse verloren gehen (das gleiche gilt auch bei Abtastung mit Maximalgeschwindigkeit). *Zeitsteuerung*
3. Eine Berechnung wird *ereignisgetrieben* angestoßen (*Ereignissteuerung*). Treten neue Ereignisse ein, bevor vorherige Berechnungen abgeschlossen sind, können sich Berechnungen verzögern (wenn und solange eine Ereignisspeicherung möglich ist) oder Ereignisse verloren gehen. *Ereignissteuerung*

Echtzeitbedingungen und Ressourcenanforderungen (z. B. Dimensionierung von Ereigniswarteschlangen) ergeben sich unter anderem aus Frequenzspektren von Signalen und sich daraus ableitenden Mindestabtastraten bzw. aus Frequenzbändern von Ereignissen und erforderlichen Reaktionszeiten.

Bei ereignisgesteuerten Systemen sind die zeitlichen Abläufe im System schwer planbar und manchmal auch schwer nachvollziehbar, während bei zeitgesteuerten Systemen das zeitliche Verhalten bereits vor der Laufzeit grundsätzlich planbar und auf die Einhaltung von Echtzeitbedingungen überprüfbar ist [Sch05]. Für Systeme mit harten oder festen Echtzeitbedingungen hat Zeitsteuerung darin einen entscheidenden Vorteil.

### 2.3.1.2 Nebenläufigkeit

Die Berechnungen eines größeren Echtzeitprogramms zerfallen wie klassische Berechnungen auch in zahlreiche Teilberechnungen, da typischerweise nicht alle Ausgangssignale eines eingebetteten Echtzeitsystems global von allen Eingangssignalen abhängen. Bezüglich ihrer zeitlichen Wiederholung und gespeicherter interner Zustände über die einzelnen Ausführungen hinweg sind viele der Berechnungen *nebenläufig*, etwa auch mit unterschiedlichen Abtastraten für verschiedene Signale oder angestoßen durch mehrere konkurrierende Ereignisse.

Die Struktur des Steuerungsprogramms kann eine Struktur des gesteuerten technischen Prozesses abbilden; Teilprozesse können dabei miteinander kommunizieren, d. h. entsprechende Berechnungen bedingen sich bezüglich ihrer Ausführung bzw. berechneter Zwischenergebnisse gegenseitig.

### 2.3.2 Echtzeitbetriebssysteme

Ein Echtzeitbetriebssystem bietet eine Ausführungsplattform für Echtzeitprogramme, die insbesondere Mechanismen für die zeit- oder ereignisgesteuerte, nebenläufige Ausführung von Berechnungen (dort gewöhnlich als *Tasks* oder *Prozesse* bezeichnet) bereitstellt. Darüber hinaus nimmt es weitere betriebssystemtypische Aufgaben wie Speicher- und Ein-/Ausgabeverwaltung wahr (siehe z. B. [WB05]).

Die durch ein Echtzeitbetriebssystem bereitgestellten Mechanismen können als Basis für die Anwendungsprogrammierung in einer Nicht-Echtzeitprogrammiersprache wie C dienen (siehe Abschnitt 4.2.1). Daneben kann ein Echtzeitbetriebssystem Teil der Ziellplattform für die Compilierung aus einer Echtzeitprogrammiersprache sein.

### 2.3.3 Echtzeitprogrammiersprachen

Echtzeitprogrammiersprachen enthalten wesentliche Aspekte des Zeitfortschritts und der Nebenläufigkeit (siehe Abschnitt 2.3.1) bereits in Ausdrucksmitteln der Sprache, so daß kein Rückgriff auf Betriebssystem- oder direkte Hardware-Mechanismen erforderlich ist (siehe die Sprachen in Abschnitt 4.2.2 und folgende). Die Echtzeiteigenschaften sind Teil der Programmsemantik.

Echtzeitprogrammiersprachen folgen im allgemeinen einem der in Abschnitt 2.2.1 beschriebenen Grundparadigmen und unter dem Echtzeitaspekt entweder dem Prinzip der Ereignis- oder der Zeitsteuerung (siehe Abschnitt 2.3.1.1). Unter den Sprachen mit Zeitsteuerung bilden die sogenannten *synchronen Sprachen* eine bekannte Untergruppe.

#### 2.3.3.1 Synchrone Sprachen

Die seit mehr als 20 Jahren entwickelten *synchronen* Sprachen (wie Esterel [BS91], Lustre [HCRP91] und Signal [GGBM91]) beruhen auf einem Systemmodell, das an getaktete Hardware-Schaltungen erinnert. Das Synchronitätsprinzip besagt dabei, daß Ausgaben von den Systemen *zeitgleich* mit den Eingaben (synchron) produziert werden,

wobei Ausgaben von Eingaben des gleichen Zeitpunkts abhängen können. Das System antwortet in Nullzeit, bezogen auf eine Folge von diskreten Zeitpunkten, die sich aus einem oder mehreren Takten ergibt. Systeme dieser Art lassen sich zu Systemen gleicher Art in einer Weise zusammensetzen, die *deterministische Nebenläufigkeit* erreicht. [BB91, BCE<sup>+</sup>03]

Die feste Taktung macht synchrone Programme auch zeitlich deterministisch und so für Systeme mit harten Echtzeitbedingungen besonders geeignet. Das vergleichsweise einfache mathematische Modell als Basis für die Semantik einer Echtzeitprogrammiersprache hat sich bei der erfolgreichen Entwicklung von Werkzeugen zur formalen Verifikation und dem Einsatz in sicherheitsrelevanten Anwendungen bewährt [BCE<sup>+</sup>03]. Synchrone Sprachen wurden sowohl als imperative als auch als Datenfluß- oder funktionale Sprachen entwickelt (siehe Kapitel 4, Abschnitte 4.2.3, 4.3.1, 4.3.2 und 4.4.1).

## 2.4 Verhaltensmodelle für Prozeßsteuerungen

Steuerungs- und regelungstechnische Anwendungen eingebetteter Echtzeitsysteme nehmen Aufgaben der *Prozeßsteuerung* wahr. Die allgemeine Aufgabe einer Prozeßsteuerung ist es, einen technischen Prozeß zu beobachten und ihm ein bestimmtes Verhalten aufzuprägen (Steuerung) bzw. ihn auf einem bestimmten Verhalten gegenüber Störeinflüssen zu stabilisieren (Regelung). Regelung ist ein quasi kontinuierlicher Vorgang mit einem beständigen und quantitativen Vergleich zwischen Ist- und Soll-Verhalten, während bei Steuerung ein Prozeß in diskrete Phasen zerfällt, die durch Ereignisse im Prozeß oder durch Zeitereignisse begrenzt werden. Die Abfolge der Phasen im Verhalten des Prozesses kann dabei zustandsorientiert (reaktive Steuerung) oder ablauforientiert (sequentielle Steuerung) ausgeprägt sein (vgl. Kapitel 1.1.2).

Prozeßverhalten und damit zusammenhängende Phänomene sind also Gegenstand der Betrachtung und der Programmierung bei den betrachteten Anwendungen. Um zu generierendes Verhalten (bzw. die Programmausgabe) in Abhängigkeit von beobachtetem Verhalten (bzw. der Programmeingabe) *problemnah* beschreiben zu können, benötigt man adäquate Modelle dynamischen Verhaltens. Die nachfolgend kurz beschriebenen, bewährten und bekannten Verhaltensmodelle (zu

den Formalismen siehe z. B. [WB05]) lassen sich den drei Typen von Prozeßsteuerungsaufgaben (quasi-kontinuierliche Regelung, reaktive Steuerung, sequentielle Steuerung) zuordnen.

### 2.4.1 Quasi-kontinuierliches Verhalten

*Differenzen-  
gleichungen*

Quasi-kontinuierliches Verhalten entsteht als Diskretisierung von kontinuierlichem (stetigem) Verhalten und ist das Standardmodell für (digitale) Regelungen. Das betrachtete Verhalten ist der zeitliche Verlauf physikalischer Größen (Variablen) in Beziehung zueinander. Die Stetigkeit bezieht sich auf diese funktionalen Beziehungen, die im kontinuierlichen Fall mathematisch mit Hilfe von Differentialgleichungen beschrieben werden, die sich in der diskretisierten (und daher für den Rechner zugänglichen Form) als *Differenzengleichungen* darstellen. Ein eindeutig aufgelöstes Gleichungssystem kann durch einen *Signalflußgraphen* dargestellt werden, der die funktionalen Abhängigkeiten zwischen den Variablen beschreibt. Signal und Variable sind dabei äquivalente Begriffe.

Die zeitliche Diskretisierung betrachtet Systeme als mit einem konstanten *Zeittakt* getaktet. Der aktuelle Zustand eines Signals ergibt sich im allgemeinen aus aktuellen Zuständen anderer Signale sowie aus Zuständen von Signalen zum vorhergehenden Zeitpunkt. Mit jedem Zeitschritt findet eine Neubestimmung der Zustände statt. Ein System ist dann eine Funktion, die Signale auf Signale abbildet. In der Theorie dieser sogenannten *zeitdiskreten Systeme* für reellwertige Signale werden Systeme auch mit Hilfe der *z-Transformation* und komplexwertiger Funktionen beschrieben (siehe z. B. [KJ02]).

Variablen bzw. Signale als Funktionen einer diskreten Zeit (modelliert durch  $\mathbb{N}$  oder  $\mathbb{Z}$ ), Zeitverschiebung um einen Zeitschritt sowie algebraische Beziehungen zwischen Variablen sind die für Differenzengleichungen wesentlichen Beschreibungselemente.

Das mathematische Modell trifft sich einerseits mit dem Synchronitätsprinzip der synchronen Sprachen (siehe Abschnitt 2.3.3.1) und andererseits auch mit dem Datenflußkonzept der Datenflußsprachen (mit Signalen als Datenflüssen und Signalflußgraphen als Datenflußgraphen, siehe Abschnitt 2.2.1.3).

### 2.4.2 Reaktives Verhalten

*endliche  
Automaten*

Reaktives Verhalten ist charakterisiert durch sprunghafte Änderun-

gen, ausgelöst durch Ereignisse. Man unterscheidet eine diskrete Menge von *Zuständen* (oder *Phasen*), in denen der Prozeß zwischen den *Ereignissen* verharret, und betrachtet Übergänge (*Transitionen*) zwischen den Zuständen (bzw. Phasen) als Reaktionen auf Ereignisse, zu denen auch Zeitereignisse gerechnet werden können.

Diese zustandsorientierte Form diskreten Verhaltens wird typischerweise formal mit Hilfe von (endlichen) *Automaten* (Zustands- bzw. Phasenübergangssystemen) beschrieben. Zustände (Phasen), Ereignisse und Transitionen sind die wesentlichen Beschreibungselemente. Ereignisse werden dabei auch als Eingaben betrachtet und oft ergänzt um Ausgaben, die Transitionen und/oder Zuständen zugeordnet werden.

### 2.4.3 Sequentielles Verhalten

Sequentielles Verhalten ist durch einen Ablauf charakterisiert, der sich als eine Sequenz von *Aktionen* auffassen läßt. Jede Aktion dauert eine gewisse Zeit, während der der Prozeß sich in einer bestimmten Weise verhält. Mit ihrer Terminierung setzt eine neue Aktion ein. Solche Abläufe werden typischerweise durch sequentielle Algorithmen (prozedural) beschrieben, die einfache Aktionssequenzen durch (bedingte und unbedingte) Sprünge oder mittels *Kontrollstrukturen* wie bedingte Ausführung, Verzweigung und Schleifen verknüpfen. Sequentielle Algorithmen können zu parallelen Algorithmen erweitert werden, um die Parallelausführung von Aktionen und nebenläufige Sequenzen zu beschreiben.

*sequentielle  
Algorithmen*

Eine Variante dieser ablauforientierten Form diskreten Verhaltens wird auch durch *Petrinetze* modelliert. Auch hier werden Aktionen (als *Transitionen* bezeichnet) betrachtet, die beliebig viel Zeit verbrauchen; allerdings steht die Nebenläufigkeit im Mittelpunkt. Eine Aktion wird nicht unmittelbar durch die Terminierung einer Vorgängeraktion angestoßen, sondern die Voraussetzung für die Ausführung einer Aktion ist das Vorhandensein einer Bedingung, die durch Terminierung ein oder mehrerer Aktionen hergestellt werden kann (Belegung von *Stellen* durch *Marken*). Aktionen starten (Transitionen *feuern*) selbsttätig (aber nicht notwendigerweise sofort), wenn die zugehörige Bedingung eintritt, und können sich auch gegenseitig ausschließen, weil das Feuern einer Transition die Markenbelegung (die Bedingungen) verändert.

*Petrinetze*



## 3. Methodisches Konzept

### 3.1 Modularisierung von Prozeßsteuerungen

Modularisierung ist ein zentrales Entwurfsprinzip im Hinblick darauf, große und/oder variantenreiche Systeme erstellbar und längerfristig beherrschbar zu machen (vgl. Kapitel 1.1.3). Leitgedanken der Modularisierung sind die Kapselung von zusammengehörigen Entwurfsentscheidungen und die Trennung von Belangen. Module müssen daher eine hohe (innere) *Kohäsion* und eine geringe (gegenseitige) *Kopplung* aufweisen. Sie werden über definierte *Schnittstellen* miteinander verbunden; interne Entwurfsentscheidungen sind Geheimnisse, die die Module voreinander verbergen (*Information Hiding*).

Module lassen sich getrennt entwickeln, ändern oder austauschen, solange keine Schnittstellen berührt werden. Module mit bewährter Funktionalität können wiederverwendet werden; je mehr ein Modul eingesetzt wird, desto gravierender sind schnittstellenrelevante Änderungen. Die Qualität einer Architektur mißt sich somit nicht zuletzt an der Stabilität der Schnittstellen. Schnittstellen abstrahieren von den Entwurfsentscheidungen der Module. Stabile Schnittstellen sind nur so detailliert wie nötig, so abstrakt wie möglich.

Bei einer Modularisierung besteht die Gefahr, sich auf *funktionale Dekomposition* zu beschränken. In der traditionellen Programmierung bedeutet das die Dekomposition der Algorithmen. Module sind dann Unterprogramme. Die gemeinsamen Datenstrukturen, auf denen

*funktionale Dekomposition vs. Modularisierung*

die Algorithmen arbeiten, werden dabei zum Bestandteil der Modulschnittstellen. Details der Datenrepräsentation werden so zu globalen, modulübergreifenden Entwurfsentscheidungen.

Die von Parnas [Par72] eingeführten Kriterien zur Modularisierung (mit denen auch der Begriff des „Information Hiding“ geprägt wurde) sehen dagegen vor, Datenstrukturen und zugehörige Operationen in eigenständigen Modulen zu kapseln. Entwurfsentscheidungen zu den Details der Datenstrukturen werden zu Geheimnissen von Modulen, deren Schnittstellen abstrakte Operationen auf den Daten für die Algorithmen bereitstellen. Der Abstraktionsgrad aller Schnittstellen steigt. Aus der Weiterführung der Idee der Datenkapselung entstand das verfeinerte Konzept der Abstrakten Datentypen, das eine der wesentlichen Grundlagen der Objektorientierung bildet (siehe Kapitel 2.2.2.3).

#### *Blöcke vs. Module*

Im Fall von Echtzeitverhalten, das durch Prozesse bzw. Signale und Ereignisse beschrieben wird, begegnet man bei der Frage der Modularisierung wiederum der funktionalen Dekomposition. Eine häufig gewählte Form dafür sind *Blockstrukturen* (hierarchische Signalflußgraphen). Dabei entspricht ein Block (eine signalverarbeitende Funktion) einem Algorithmus. Kommunikationspfade zwischen Blöcken (Signale oder auch Ereignisse) entsprechen Datenstrukturen. Die Schnittstellen zwischen den Blöcken bestehen folglich aus Signalen und Ereignissen. Wählt man Blöcke als Module, folgt die gleiche Tendenz wie bei Unterprogrammen in der traditionellen Programmierung: Geht die Ein-/Ausgabe eines Moduls über einzelne unstrukturierte Signale oder Ereignisse hinaus, gelangen zu viele Details in die Schnittstellen. Die genaue Zusammensetzung der Ein-/Ausgabesignale und -ereignisse gehört zu einer konkreten Realisierung einer eigentlich abstrakter zu betrachtenden Interaktion. Daraus kann gefolgert werden, daß Blockstrukturen zur Modularisierung nicht allgemein geeignet sind. Sie eignen sich allerdings sehr wohl zur Dekomposition von Prozessen oder signalwertigen Funktionen *innerhalb eines Moduls* (z. B. zur Beschreibung eines Regelungsalgorithmus) – genauso wie Unterprogramme zur Dekomposition von Algorithmen.

### **3.2 Steuerung mit konstruktiver Verhaltensabstraktion**

#### *Steuerungshierarchie*

In Kapitel 1.1.2 wurden für steuerungs- und regelungstechnische



Problemstellungen drei Grundtypen von Prozeßsteuerungen angegeben, nämlich quasi-kontinuierliche Regelung, reaktive und sequentielle Steuerung. Dabei wurde erwähnt, daß die drei Grundtypen nahezu beliebig kombiniert werden können, um komplexere Prozeßsteuerungen zu bilden. Treten regelungs- und steuerungstechnische Probleme unterschiedlicher Art in Kombination auf, dann ist im Normalfall eines dem anderen untergeordnet, so daß eine *Steuerungshierarchie* gebildet werden kann, bei der eine übergeordnete Prozeßsteuerung untergeordnete aktiviert oder Vorgaben macht, die von untergeordneten Prozeßsteuerungen umgesetzt werden.

Baut man eine Steuerungshierarchie über funktionale Dekomposition auf, werden die einzelnen Steuerungen zu Funktionsblöcken. Eine übergeordnete Steuerung produziert Signale als Ausgaben, die als Eingaben in untergeordnete Steuerungen eingehen. Die Signale kodieren eine Kommunikation zwischen zwei oder mehreren nebenläufigen Steuerungen, von denen eine implizit eine übergeordnete Funktion hat. Auf unterster Ebene kommunizieren die Steuerungen mit dem technischen Prozeß als einer weiteren, dazu nebenläufigen Komponente.

Eine andere Betrachtungsweise ist es, in der Steuerungshierarchie den gesteuerten Prozeß selbst auf unterschiedlichen Abstraktionsebenen zu sehen. Das Verhalten des Prozesses bzw. die Freiheitsgrade, wie der Prozeß gesteuert werden kann, werden schrittweise abstrakter betrachtet (*Verhaltensabstraktion*). Auf der untersten Ebene wird der Prozeß durch die Sensor-Signale und durch die Aktoren repräsentiert, die durch Signale angesteuert werden können. Die (zeitkontinuierlichen) Aktor-Signale sind sozusagen die applizierbaren Parameter des Prozesses. Darauf aufbauende Steuerungsfunktionen können eine abstraktere Sicht auf den Prozeß und seine Steuerbarkeit herstellen. So können z. B. aus den Sensor-Signalen Ereignisse extrahiert und bestimmte Aktor-Ansteuerungssignale (oder Abschnitte davon) vordefiniert werden, aus denen Verhalten diskret konstruiert werden kann (z. B. Phasen für ein Phasenübergangssystem oder Aktionen für einen sequentiellen Algorithmus). Der Prozeß ist sozusagen „programmierbar“ mit einer stets abstrakteren Programmierschnittstelle, die den Charakter von *Signalen* verlassen kann.

*Verhaltens-  
abstraktion*

Die Steuerungshierarchie nutzt von oben nach unten die verschiedenen Sichten *konstruktiv* für die Steuerung des Prozesses, indem die *atomaren* Verhaltensbestandteile einer Abstraktionsebene mit den

*konstruktive  
Verhaltens-  
abstraktion*

Mitteln (in Termini) der darunter liegenden Ebene realisiert werden (*konstruktive Verhaltensabstraktion*). Im Steuerungsprogramm wird der Prozeß auf mehreren Abstraktionsebenen beschrieben, ohne daß mehrere Sichten den gleichen Verhaltensumfang mit unterschiedlichem Detaillierungsgrad beschreiben. Die Sichten ergänzen sich redundanzfrei; Abstraktion wird mit Dekomposition verbunden. Das auf oberster Ebene beschriebene Verhalten wird schrittweise konkretisiert und dem Prozeß aufgeprägt, wobei alle Abstraktionen im Programm verbleiben.

So kann beispielsweise eine Prozedur (ein sequentieller Algorithmus) beschrieben werden, deren atomare Aktionen in der nächsttieferen Abstraktionsebene in reaktive Prozesse zerfallen, die als Phasenübergangssysteme (endliche Automaten) beschrieben werden. Die atomaren Phasen sind dann beispielsweise realisiert als Signale, zwischen denen umgeschaltet wird (die Signale müssen dabei für die Dauer einer Phase nicht konstant sein). Das phasenweise zusammengesetzte Signal dient dann etwa als Führungsgröße für einen quasi-kontinuierlichen Regler, dessen Stellgrößen an die Aktorik ausgegeben werden. Die Ereignisse der reaktiven Prozesse sind beispielsweise realisiert als Trigger auf Flanken boolescher Signale, die durch Verknüpfung (im Sinne eines Signalflußgraphen) aus elementaren Signalen hervorgehen, die von der Sensorik geliefert werden.

Steuerung und Beobachtung eines Prozesses stehen in engem Zusammenhang. Entlang einer Steuerungshierarchie zur Beeinflussung des Verhaltens eines Prozesses erfolgt daher im allgemeinen auch eine Beobachtung seines Ist-Verhaltens. So wie die Aktorik eine letzte Stufe der Konkretisierung in der Abstraktionshierarchie zur Prozeßsteuerung leistet, bildet die Sensorik auf der Seite der Prozeßbeobachtung die ersten Abstraktionen.

Man beachte, daß bei konstruktiver Verhaltensabstraktion direkt das *Verhalten eines Prozesses* stufenweise betrachtet und beschrieben wird. Im Unterschied dazu wird bei dem Blickwinkel der *Interaktionsverfeinerung* [Bro97, BS01] die *Steuerung als eine Funktion*, die abstrakte Eingabesignale auf abstrakte Ausgabesignale abbildet, schrittweise zu einer Funktion mit konkreten Signalen verfeinert.

### 3.3 Abstrakte Maschinen

Das Prinzip der konstruktiven Verhaltensabstraktion, Verhalten in abstrakten Termini zu beschreiben und diese atomaren Verhaltensbe-

standteile einer Abstraktionsebene mit den Mitteln (in Termini) der darunter liegenden Ebene zu realisieren, wird durch das Konzept der Abstrakten Maschinen mit dem Aspekt der Modularisierung verbunden.

Die Fortführung der Idee, aus den Freiheitsgraden, wie der Prozeß auf einer Abstraktionsebene gesteuert werden kann, Verhaltensprimitive zu bilden, aus denen abstrakteres Prozeßverhalten zusammengesetzt werden kann, besteht darin, diese als „Befehlssatz“ einer *abstrakten Maschine* anzusehen und als Schnittstelle eines Moduls zu exportieren. Das Modul ist nicht selbst eine Laufzeitkomponente, sondern stellt wie ein abstrakter Datentyp Operationen bereit, aus denen ein Programm konstruiert werden kann. Dazu können primitive Signale, aber auch Phasen, Ereignisse und Aktionen (im allgemeinen parametrierbar) gehören, aus denen Phasenübergangssysteme (Automaten) bzw. sequentielle Prozeduren konstruiert werden können.

Als *konkrete Maschine* wird dabei das gesteuerte physikalische System betrachtet, dem durch eine Steuerung (im Rahmen von bestimmten Toleranzen) ein bestimmtes Verhalten gegeben werden kann. Die Maschine kann dabei Fremdeinflüssen unterliegen, die die Steuerung durch Beobachtung des Ist-Verhaltens erkennen und berücksichtigen muß. Sensor-Signale liefern Informationen über das Ist-Verhalten der Maschine. Signale an die Aktoren beeinflussen das Verhalten der Maschine und können (im Rahmen der Spezifikationen der Aktoren) frei gewählt werden. Aufgabe der Steuerung ist es, aus dem gewünschten Soll-Verhalten (oder der Abweichung zwischen Soll- und Ist-Verhalten) und bekannten Eigenschaften der Maschine und der Wirkungsweise der Aktorik geeignete Aktor-Signale herzuleiten.

Das dazu auf unterschiedlichen Ebenen nötige Wissen wird durch den Entwurf einer Architektur aus *abstrakten Maschinen* so separiert und gekapselt, daß aus dem per Aktorik „programmierbaren“ Prozeß als konkreter Maschine stufenweise abstraktere Maschinen mit „höherwertigen“ Eigenschaften werden.

Vorbild sind die Abstrakten Maschinen nach Dijkstra [Dij68, Dij69], wo mit der konkreten Maschine der Rechner gemeint ist, auf dem die Software ausgeführt wird. Aufeinander aufbauende abstrakte Maschinen sind dort das logische Konzept für die Strukturierung des Betriebssystems. Im hier vorliegenden Fall der Prozeßsteuerung wird eine Maschine betrachtet, die selbst kein Rechner ist (der Rechner ist lediglich eingebettet und wirkt auf die Maschine ein). Aufeinander aufbauen-

de Teile der Steuerung bilden hier sozusagen ein immer spezielleres „Betriebssystem“ für den technischen Prozeß.

Im Gegensatz zu den Abstrakten Maschinen nach Dijkstra muß nicht an eine Schichtung von durchgängigen Abstraktionsebenen gedacht werden, die zu einer Sequenz  $M_0, \dots, M_n$  von Maschinen führt ( $M_0$  ist dabei die konkrete Maschine,  $M_1, \dots, M_n$  sind aufeinander aufbauende abstrakte Maschinen). Durch horizontale Dekomposition kann auch eine baumartige Struktur (nämlich eine Steuerungshierarchie, vgl. Abschnitt 3.2) etabliert werden. Die Maschine wird dabei in mehrere „Geräte“ zerlegt, die bis zu bestimmten Ebenen unabhängig voneinander betrachtet werden können und dann ggf. zu einem übergeordneten abstrakten Gerät aggregiert werden, um entsprechende physikalische Zusammenhänge im Prozeß abzubilden (diese Zusammenhänge werden zum Modulgeheimnis dieses abstrakten Geräts).

Durch eine Hierarchie abstrakter Geräte kann die gesamte Steuerungslogik schrittweise zu dem Verhalten der Maschine auf oberster Ebene verdichtet werden, das durch ein „Hauptprogram“ als abstrakte Gesamtsicht des Prozesses beschrieben wird. In welcher Reihenfolge dabei die einzelnen Abstraktionen entwickelt werden (top-down, bottom-up oder meet-in-the-middle [MG96]), ist durch die Methodik nicht festgelegt.

### **3.4 Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme**

Eine Sprache zur Programmierung eingebetteter Echtzeitsysteme muß sich entsprechend dem in Kapitel 1.2 gesteckten Ziel sowohl für das Anwendungsgebiet eignen als auch allgemeinen Belangen des Software Engineering gerecht werden. Die Problematik der Echtzeitprogrammierung wurde in Kapitel 2.3 erläutert. Die Nähe zu steuerungs- und regelungstechnischen Problemstellungen wurde durch die Verhaltensmodelle und Beschreibungsmittel in Kapitel 2.4 konkretisiert. Dem Stand der Technik bei Programmiersprachen entsprechende Konzepte und Mechanismen zur Unterstützung der Entwicklung großer und variantenreicher Software-Systeme wurden in Kapitel 2.2.2 beschrieben. Das in diesem Kapitel beschriebene methodische Konzept konkretisiert die Anwendung von Abstraktion und Modularisierung auf die Problemstellungen des Anwendungsgebiets.

Folgende Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme können festgehalten werden:

1. Die Eignung für Echtzeitprogrammierung (mit Nebenläufigkeit und Zeitfortschritt) allgemein und speziell für Systeme mit harten Echtzeitbedingungen (siehe Kapitel 2.3).
2. Adäquate Beschreibungsmittel für quasi-kontinuierliche Regelungen und reaktive und sequentielle Steuerungen (siehe Kapitel 1.1.2) nach den Verhaltensmodellen aus Kapitel 2.4 in einer Durchgängigkeit, die konstruktive Verhaltensabstraktion (siehe Abschnitt 3.2) ermöglicht.
3. Mechanismen zur funktionalen Abstraktion, zur Datenabstraktion und Kapselung sowie zur generischen Abstraktion (siehe Kapitel 2.2.2) in einer Allgemeinheit, die abstrakte Maschinen (siehe Abschnitt 3.3) als Module ermöglicht. Dabei darf ein Typsystem für statische Typprüfungen (siehe Kapitel 2.2.2.1) zur Absicherung der Abstraktionen vorausgesetzt werden.



## 4. Stand der Technik

### 4.1 Übersicht und Vorbemerkungen

Die vorliegende Arbeit ist motiviert daraus, daß heute verfügbare Sprachen den in Kapitel 3.4 gestellten Anforderungen nicht voll gerecht werden.

Die Software eingebetteter Echtzeitsysteme wurde lange Zeit überwiegend in Assembler-Sprachen geschrieben. Heute prägt die Sprache C den Stand der Praxis. Dennoch gibt es zahlreiche Hochsprachen, die für die Programmierung solcher Systeme eingesetzt werden oder dazu entwickelt wurden. Dieses Kriterium diene als Auswahlkriterium für die folgende Diskussion existierender Sprachen, aus der in Abschnitt 4.5 ein Fazit gezogen wird. Aufgrund der unübersichtlichen Vielzahl der entwickelten Programmiersprachen erhebt die Auswahl keinen Anspruch auf Vollständigkeit. Innerhalb verschiedener Klassen stehen einzelne Sprachen stellvertretend auch für ähnliche Sprachen. *Auswahl*

Sprachen außerhalb des imperativen und des funktionalen Paradigmas werden nicht betrachtet (z. B. Constraint-Programmierung). Nicht betrachtet werden auch Sprachen, die explizit für *verteilte* Systeme oder solche mit *weichen* Echtzeitbedingungen entwickelt wurden, wie z. B. Erlang [Arm97]. Weiterhin liegt der Fokus auf der Programmierung von Mikrocontrollern. So bleiben Sprachen unberücksichtigt, die auf spezielle Rechner-Architekturen abzielen, einschließlich der Programmiersprachen des IEC-Standards 61131-3 oder Grafcet [Dav95] für speicherprogrammierbare Steuerungen, sowie Hardwarebeschrei-

bungssprachen wie VHDL (IEEE-Standard 1164) oder Verilog (IEEE-Standard 1364-2001) und Sprachen für System-Level Design wie SystemC (IEEE-Standard 1666) oder SpecC [GZD<sup>+</sup>00].

#### *Modellierungssprachen als Programmiersprachen*

Bei den verbleibenden Sprachen muß der Begriff der Programmiersprache weit genug gefaßt werden. Neben herkömmlichen und so verstandenen Programmiersprachen kommen im Zuge von Methoden sogenannter „modellbasierter Entwicklung“ in jüngerer Zeit ausführbare Sprachen zum Einsatz, die als *Modellierungssprachen* verstanden werden. Ausführbare Modelle dienen ursprünglich zur Simulation oder als ausführbare Spezifikation; in Verbindung mit *Code-Generatoren* für Programmcode (meist in den Sprachen C oder C++), der dann automatisch weiterkompiliert wird, werden sie jedoch zum Software-Quellcode. Solche Verfahren werden mit Sprachen wie Simulink/Stateflow [The06d], ASCET [ETA06] oder UML [UML05a] produktiv für eingebettete Systeme eingesetzt. Modellierungssprachen mit verfügbarer Code-Generierung können daher im weiteren zu den Programmiersprachen gezählt werden. Graphische Syntaxelemente stehen dem nicht entgegen.

#### *Entwicklungsumgebungen*

Spezielle Programmiersprachen, insbesondere Modellierungssprachen mit Code-Generierung, sind heute oft integraler Bestandteil einer umfangreicheren Werkzeugumgebung. So bietet beispielsweise die Entwicklungsumgebung ASCET von ETAS [ETA06] mit ihrer eigenen Modellierungs- und Programmiersprache neben Simulation auf dem Entwicklungsrechner und C-Code-Generierung für verschiedene Zielplattformen (verschiedene Microcontroller, Echtzeitbetriebssysteme nach OSEK-Standard [OSE]) auch Funktionalität zur Konfiguration eines OSEK-Betriebssystems, zur Generierung von Meßtechnik-Beschreibungsdaten nach einem ASAM-Standard, eine Integration mit Spezial-Hardware für Rapid Prototyping, eine Integration mit Meß- und Kalibrierwerkzeugen usw. Ähnliche Entwicklungsumgebungen stehen auch mit anderen Sprachen zur Verfügung. Solche Programmiersprachen sind demnach nicht beliebig austauschbar und machen andererseits auch nur einen Teil des Leistungsumfangs der entsprechenden Werkzeuge aus. Die Bewertung nach den gestellten Anforderungen bezieht sich nur auf den Aspekt der Programmiersprache.

#### *Gliederung*

Die Gliederung der nachfolgenden Abschnitte folgt auf erster Ebene der Klassifikation nach Grundparadigmen der Programmierung gemäß Kapitel 2.2.1, wobei die Datenflußsprachen innerhalb der funktionalen Sprachen gesehen werden. Auf zweiter Ebene werden *synchrone*



Sprachen hervorgehoben, da sie auf Systeme mit harten Echtzeitbedingungen spezialisiert sind.

## 4.2 Imperative Sprachen

### 4.2.1 Sequentielle Sprachen

Rein sequentielle Sprachen wie **C** und **C++** unterstützen die in Abschnitt 2.3 beschriebenen Charakteristika von Echtzeitsystemen (Zeitfortschritt, Nebenläufigkeit) von Hause aus nicht. **C** ist dennoch die am weitesten verbreitete Sprache für eingebettete Software<sup>1</sup>. Die Nutzung sequentieller Sprachen setzt die direkte Interaktion mit Mechanismen eines darunterliegenden Betriebssystems voraus [BS05]. Alternativ kann auch in klassischer Art hardwarenaher Programmierung direkt – unter Verzicht auf ein Betriebssystem – mit Mechanismen der Hardware wie Timer-Bausteinen und Interrupt-Vektoren gearbeitet werden.

Die von Hardware und Betriebssystem bereitgestellten Mechanismen wie periodische Tasks/Prozesse und Interrupt-Service-Routinen führen klassische sequentielle Unterprogramme aus, die über vom Betriebssystem bereitgestellte Kommunikationsmechanismen oder schlicht über gemeinsame globale Variablen miteinander kommunizieren. Auf diese Weise werden Zeitfortschritt und Nebenläufigkeit realisiert. Um beim sogenannten Multitasking eine korrekte zeitliche Abfolge der Berechnungen und der Kommunikation und die Konsistenz gemeinsam genutzter Daten sicherzustellen, sind besondere Vorkehrungen (betriebssystemseitig und in der Anwendung) zu treffen.

Echtzeitprogrammierung dieser Art ist stark geprägt von den Ausführungsmechanismen, die über den Umfang der Programmiersprache selbst hinausgehen. Das kann z. B. beim Programmieren ein Denken in „Zeitscheiben“ zur Folge haben: Sequentieller Code wird so geschrieben, daß er periodisch ausgeführt wird; wesentliche Teile eines übergeordneten Kontrollflusses können sich dabei in Zustandsvariablen (in der Rolle von „Programmzählern“) verbergen. Und da eine Gesamtanwendung letztlich aus einer Menge mehr oder weniger lose gekoppelter Programme besteht, ist es grundsätzlich schwierig, die Anwendung zu verstehen, zu debuggen, zu warten oder auch bestimmte Echtzeiteigenschaften nachzuweisen [BB91].

<sup>1</sup>Im Jahr 2000 kam **C** laut empirischen Erhebungen (Lewis, 2001, zitiert in [Sch05]) auf einen Anteil von etwa 80%.

Sequentielle imperative Sprachen bieten keine besondere sprachliche Unterstützung für quasi-kontinuierliches oder reaktives Verhalten. Mit den beschriebenen Mechanismen zur Echtzeitprogrammierung können Signale und Zeittakt über imperative Variablen und Zeitscheiben (periodische Tasks/Prozesse) realisiert werden, wobei Zeitverschiebung gegenüber der Verwendung eines Wertes im gleichen Taktzyklus durch unterschiedliche Ausführungsreihenfolgen von Prozessen unterschieden werden kann. Automaten können durch Zustandsvariablen und Fallunterscheidungen implementiert werden. Auch (zeitverbrauchendes) sequentielles Verhalten kann nicht bei allen Betriebssystemen direkt mit den Kontrollstrukturen der sequentiellen Programmierung abgebildet werden, sondern muß gegebenenfalls, wie schon angedeutet, mit Zeitscheibenprogrammierung und Zustandsvariablen realisiert werden, um Wartezeiten zu überbrücken.

*Funktionen / Prozeduren erster Ordnung* werden von sequentiellen imperativen Sprachen allgemein unterstützt. C und C++ unterstützen auch *Funktionen als Argumente* (über *Funktionszeiger*). *Datenabstraktion / Kapselung* kann in C zwar simuliert werden, die Abstraktionen lassen sich jedoch nicht über das Typsystem absichern. In C++ steht dafür das *Klassenkonzept* zur Verfügung. *Generische Abstraktion* wird in C++ über *Templates* unterstützt<sup>2</sup>, die in Embedded C++ [Emb99], einer restriktiven Teilmenge von C++ für die Programmierung eingebetteter Systeme<sup>3</sup>, allerdings entfallen.

### 4.2.2 Nebenläufige Sprachen

Eine Reihe von Sprachen verbindet das imperative Paradigma mit Nebenläufigkeit als Ausdrucksmittel der Sprache selbst. In der Regel sind damit auch *zeitbezogene* Mechanismen verbunden, so daß Echtzeitprogrammierung insgesamt unterstützt wird, ohne daß ein direkter Zugriff auf Mechanismen eines Echtzeitbetriebssystems erforderlich ist. Die Echtzeitanwendung kann als ein zusammenhängendes Programm betrachtet werden.

Eine Gruppe innerhalb dieser Sprachen, die klassischen nebenläufigen Programmiersprachen, mit **Ada 95** [Ada95] und **Real-Time**

<sup>2</sup>Mit Vererbung, Überladung und automatischer Typumwandlung finden sich in C++ auch die drei anderen Arten von Polymorphie (vgl. Abschnitt 2.2.2.4).

<sup>3</sup>Embedded C++ entfernt aus C++ z. B. multiple Vererbung, Ausnahmebehandlung, Run-Time Type Identification (RTTI) und andere Fähigkeiten von C++, die als schwer erlernbar, zeitlich schwer vorhersagbar oder ressourcenbelastend angesehen wurden. Entsprechend wurden auch die Standardbibliotheken von C++ gekürzt.

**Java** [BBD<sup>+</sup>00], einer Erweiterung von Java, als aktuellen Vertretern, kennen ansonsten die Kontrollstrukturen der sequentiellen Sprachen. Mit Mechanismen zur Synchronisation und zur relativen oder absoluten Verzögerung kann das Warten auf ein Ereignis bzw. das Vergehen von Zeit in diesen Sprachen innerhalb eines sequentiellen Kontrollflusses realisiert werden. Damit ist reaktives und (zeitverbrauchendes) sequentielles Verhalten darstellbar. Sequentielle Algorithmen können direkt formuliert werden, Automaten müssen mit Kontrollstrukturen und ggf. Zustandsvariablen nachgebildet werden. Quasi-kontinuierliches Verhalten bleibt ununterstützt und muß ähnlich wie bei rein sequentiellen Sprachen in imperative Algorithmen übersetzt werden.

In bezug auf harte Echtzeitanforderungen nachteilig ist, daß diese Sprachen im wesentlichen *asynchron* und *nichtdeterministisch* sind: Durch Kommunikation ist zwar die Synchronisation zweier Prozesse möglich; wann aber die Kommunikation absolut stattfindet und in welcher Reihenfolge mehrere zeitgleich mögliche Kommunikationen stattfinden, ist offen und unvorhersagbar [BB91]. Deterministisches (Kommunikations-)Verhalten abzusichern erfordert im allgemeinen zusätzliche Maßnahmen (wie auch beim Programmieren mit Echtzeitbetriebssystemen) wie etwa feste Zeitraster und Einflußnahme auf das Scheduling mittels Prozeßprioritäten etc. Auf Basis aller verfügbaren Mechanismen empfiehlt z. B. [BS05, S. 339] Ada 95 durchaus für harte Echtzeitsysteme. Das Ravenscar-Profil für sicherheitsrelevante Echtzeitsysteme [Bur99] mit einer gezielten Auswahl von Möglichkeiten gibt dazu besondere Unterstützung.

Unter Software-Engineering-Gesichtspunkten stellen sich sowohl Ada als auch Java als sehr leistungsfähig dar. Schon Ada 83 [GH83], der erste Ada-Standard, ist speziell auch unter diesem Aspekt entwickelt worden und war eine der ersten Sprachen, die abstrakte Datentypen unterstützten [Seb04]. *Funktionen und Prozeduren erster Ordnung* gehören zu den Grundlagen, *anonymen Klassen* erlauben in Java die Bildung von *Abstraktionen höherer Ordnung*. *Module* heißen in Ada *Pakete*, in Java wird das Konzept durch *Klassen* realisiert. *Generische Abstraktion* gibt es in Ada in Form von *generischen Paketen*, und in Form von *generischen Klassen* auch in den neuesten Versionen von Java (siehe z. B. [Seb04]).

**Giotto** [HHK03] ist eine Sprache zur *zeitgesteuerten* Programmierung eingebetteter Systeme mit harten Echtzeitanforderungen. Ein Giotto-

Programm beschreibt die zeitgesteuerte Ausführung und Kommunikation nebenläufiger Berechnungen (Tasks). Die Tasks sind sequentielle Programme in einer Wirtssprache (z. B. C) und selbst nicht unterbrechbar. Sie werden periodisch ausgeführt. Kommunikation zwischen Tasks (sowie zwischen Tasks und externen Ein-/Ausgängen) findet über Ein-/Ausgangsvariablen (Ports) zu jeder Task statt. Die Kommunikation erfolgt zwischen Task-Ausführungen und deterministisch. Die Tasks sind nicht grundsätzlich alle gleichzeitig aktiv (d. h. sie werden periodisch aufgerufen), sondern Giotto-Programme zerfallen in verschiedene Modi. In jedem Modus ist eine Teilmenge der Tasks aktiv (mit modusspezifischer Periode) und spezifisch untereinander verbunden (durch Belegung der Eingangsvariablen). Es finden periodische, bedingte Moduswechsel (Transitionen) statt.

In einem Giotto-Programm gibt es keine Hierarchie (weder von kommunizierenden Tasks, noch im Zustandsautomaten) und keine Modularisierung. Abstraktionsmechanismen wie in Abschnitt 2.2.2 beschrieben, stehen nicht zur Verfügung (mit Ausnahme einer Typisierung der Ports). Giotto erscheint als eine Sprache für eine abstrakte Echtzeitbetriebssystemkonfiguration (implementierbar auch durch ein Mehrrechnersystem)<sup>4</sup> für den in Abschnitt 4.2.1 beschriebenen Stil der Programmierung, die mit strikter Zeitsteuerung arbeitet und deterministisches Zeit- und Kommunikationsverhalten erreicht. Quasi-kontinuierliches, reaktives und sequentielles Verhalten wird durch Zeitscheibenprogrammierung (sequentiell imperativ) wie in Abschnitt 4.2.1 abgebildet.

### Statecharts

Das Modellierungswerkzeug **Statemate** von I-Logix [ILo, HP99, HN96, ILo01] kann zusammen mit dem Codegenerator *MicroC* (früher: *Rhapsody in MicroC*) zur Programmierung eingebetteter Systeme eingesetzt werden. Im Mittelpunkt der Sprache von Statemate stehen Harel-*Statecharts* [Har87] zur Beschreibung reaktiven Verhaltens. Statecharts erweitern das Konzept der endlichen Automaten um Hierarchie, Nebenläufigkeit und (imperative) Aktionen. In einem hierarchischen Automaten kann ein Zustand in Unterzustände zerfallen und dabei selbst durch einen Automaten beschrieben werden. Bei Nebenläufigkeit beschreiben mehrere Teilautomaten ein Verhalten in zueinander orthogonalen Zustandsräumen; zwischen den nebenläufigen Automaten kann es Kommunikation über Ereignisse geben. Nebenläufigkeit kann auf jeder Hierarchieebene eines hierarchischen Automaten

<sup>4</sup>von den Autoren als „a software architecture of the implementation which specifies its functionality and timing“ bezeichnet

auftreten; die Kommunikation erfolgt ebenenübergreifend über den gesamten Automaten hinweg (event broadcasting). Ereignisse lösen Transitionen (Zustandsübergänge) und/oder *Aktionen* aus.

Aktionen sind Zuweisungen an (Daten-)Variablen, die Generierung von Ereignissen, das Starten oder Stoppen von *Aktivitäten* und Aufrufe von (sequentiellen) Subroutinen sowie zusammengesetzte Aktionen. Als Zusammensetzungen gibt es bedingte/alternative Aktionen, parallele Aktionen, wobei die parallelisierten Aktionen entweder einzeln aufgezählt oder iterativ beschrieben werden (z. B. für Array-Operationen).

Die Statecharts sind eingebettet in eine Hierarchie von Aktivitäten entsprechend einer *funktionalen Dekomposition* des Verhaltens eines Systems (beschrieben durch sogenannte Activity Charts). Ausgehend von einer Gesamtaktivität für das System wird bei jeder Zerlegung eine der Teilaktivitäten als „Kontroll-Aktivität“ ausgezeichnet; sie startet und stoppt ihre benachbarten Aktivitäten. Kontroll-Aktivitäten werden beschrieben durch Statecharts. Sonstige atomare Aktivitäten enthalten nur sogenannte „statische Reaktionen“ (imperative Aktionen, ausgelöst durch Ereignisse). Aktivitäten kommunizieren untereinander über Ereignisse und Variablen, die in atomaren Aktivitäten mit Statecharts bzw. statischen Reaktionen verarbeitet werden. Neben parallelen Aktionen und nebenläufigen Teilautomaten innerhalb eines Statechart bilden die Aktivitäten eine dritte Ebene der Nebenläufigkeit.

Das nebenläufige Verhalten ist deterministisch mit einer *Schrittsemantik*. Das Verhalten des gesamten Systems zerfällt in eine Folge von Ausführungsschritten, bei der alle Aktivitäten im Gleichtakt voranschreiten. In jedem Schritt werden systemweit alle anstehenden Aktionen parallel ausgeführt. Die Auswirkungen einer Aktion – speziell neu zugewiesene Werte von Variablen und ausgelöste Ereignisse – sind erst im nächsten Schritt sichtbar. Ereignisse sind für genau einen Schritt gültig. Statemate als Simulationswerkzeug kennt dabei zwei verschiedene Zeitmodelle. Im *synchronen* Zeitmodell (zeitgesteuert) entspricht jeder Ausführungsschritt einem Zeitschritt.<sup>5</sup> Im *asynchronen* Zeitmodell (ereignisgesteuert) werden ohne Zeitfortschritt so viele Schritte ausgeführt, bis intern kein Ereignis mehr vorliegt; dann wartet das System auf das nächste Ereignis von außen (sogenannter

---

<sup>5</sup>Es liegt jedoch keine Synchronität im Sinn des Synchronitätsprinzips der synchronen Sprachen vor, da Ausgaben mit einem Schritt Verzögerung auf die Eingaben folgen.

„Superschritt“).<sup>6</sup> Mit Timeout-Ereignissen (für Transitionen) und sogenannten „geplanten Aktionen“ (scheduled actions) kann zeitbezogenes Verhalten beschrieben werden.

Abweichend davon verwendet (Rhapsody in) MicroC auf oberster Ebene das Ausführungsmodell von OSEK [OSE] mit priorisierten asynchronen Tasks und Interrupt-Service-Routinen unter der Kontrolle des Echtzeitbetriebssystems. Die Schrittsemantik gilt dabei nur innerhalb einer Task oder Interrupt-Service-Routine und entsprechend dem asynchronen Zeitmodell des Simulators [ILo01]. Die Programmierung von Nebenläufigkeit erfolgt hier also zweistufig mit Konzepten von OSEK und von Statemate.

Statemate ist für die Beschreibung reaktiven Verhaltens konzipiert und unterstützt Automaten als Beschreibungsmittel direkt in seiner Sprache. Quasi-kontinuierliches und sequentielles Verhalten muß reaktiv simuliert werden. Ein quasi-kontinuierliches Subsystem wird dabei in einer einzigen Aktion gerechnet, die durch ein periodisch generiertes Ereignis angestoßen wird.<sup>7</sup> Sequentielle Algorithmen können durch Statecharts, strukturiert im Stil von Flußdiagrammen, nachgebildet werden (normale Subroutinen werden in Nullzeit in einem Schritt ausgeführt); bedingte Transitionen ermöglichen Verzweigungen und Schleifen.

Funktionale/prozedurale Abstraktion erster Ordnung ist innerhalb der Aktionssprache möglich (Subroutinen) und, indem sowohl Statecharts als auch Activity Charts als *generisch* definiert und mehrfach und mit Parametern instantiiert werden können. Mögliche Parameter generischer Charts sind Konstanten, Variablen zum Lesen oder Schreiben, Ereignisse zum Empfangen oder Senden und (bei Statecharts) Aktivitäten zum Beobachten, Starten und Stoppen. Statemate unterstützt benutzerdefinierte Datentypen (Summen und Produkte vordefinierter elementarer Datentypen), jedoch ohne Kapselungsmechanismen zur Datenabstraktion. Generische Abstraktion wird ebenfalls nicht unterstützt.

---

<sup>6</sup>Eine derartige Aufteilung eines Zeitschritts in Mikroschritte findet sich auch in Hardwarebeschreibungssprachen wie in VHDL und Verilog (vgl. [BCE<sup>+</sup>03]).

<sup>7</sup>Eine weniger effiziente Möglichkeit – ausschließlich zur Simulation – wäre, algebraisch-logische Verknüpfungen mit sogenannten „kombinatorischen Zuweisungen“ (einem Sprachelement von Statemate) auszudrücken, zeitverschiebende Elemente durch Aktionen zu rechnen, die durch periodische Ereignisse getriggert werden, und einen Zeitschritt grundsätzlich als „Superschritt“ zu rechnen.

Die konstruktive Teilmenge von **UML 2.0** [UML05a], die als Programmiersprache eingesetzt werden kann, ist eine objektorientierte Sprache mit Ähnlichkeit zu C++ und Java. Als zusätzliche strukturelle Konzepte finden sich „Assoziationen“ zwischen Klassen (als Konzept zurückgehend auf das Entity Relationship-Modell [Che76]), die zu Verbindungen zwischen Objekten instantiiert werden, und ein erweitertes Schnittstellen-Konzept für Klassen mit „Ports“ (neu in der Version 2.0) und Verbindungen zwischen Objekten von Port zu Port. Um ausführbare Modelle zu erhalten, wird UML herkömmlicherweise mit einer imperativen Wirtssprache (z. B. C++) verbunden, in der z. B. Operationen (Funktionen/Prozeduren, Methoden) ausformuliert werden. In der Version 2.0 besitzt UML jedoch auch eine vordefinierte abstrakte Syntax und Semantik für eine imperative „Aktionssprache“. Zur konstruktiven Verhaltensbeschreibung kennt UML darüber hinaus Zustandsdiagramme ähnlich den Statecharts (mit Varianten der Darstellung in Richtung SDL [SDL00]), jedoch ohne die deterministische Semantik der Nebenläufigkeit von Statemate (keine parallelen Aktionen). Nebenläufiges Verhalten ist insgesamt asynchron, vergleichbar mit Nebenläufigkeit in Ada oder Real-Time Java. Weiterhin gibt es in UML 2.0 eine „Aktivitätssprache“ mit Kontroll- und Datenflußaspekten und einer Petri-Netz-artigen Semantik (in UML 2.0 neu konzipiert). Sie enthält auch Kontrollstrukturen zur sequentiellen Programmierung und subsumiert die Aktionssprache als Spezialfall. Mit Aktionen zum Empfangen und Senden von Ereignissen bzw. Signalen kann reaktives zu sequentiellem Verhalten abstrahiert werden. Umkehrt kann einem Automaten-Zustand eine Aktivität, also sequentielles Verhalten zugeordnet werden.

Mit den Mitteln von UML 2.0<sup>8</sup> kann reaktives und sequentielles Verhalten im Sinn von Abschnitt 2.4 beschrieben werden. Quasi-kontinuierliches Verhalten wird nicht unterstützt; es muß wie bei den anderen bisher beschriebenen Sprachen in sequentielle Algorithmen übersetzt werden. UML unterstützt funktionale/prozedurale Abstraktion erster Ordnung für sequentielle Unterprogramme, Zustandsautomaten und Aktivitäten. Abstraktionen höherer Ordnung sind teilweise durch das von C++ übernommene Template-Konzept möglich. Daten-

---

<sup>8</sup>Zusätzlich wurde von der Object Management Group ein *UML Profile for Schedulability, Performance, and Time Specification* [UML05b] verabschiedet. Mit den UML 2.0 enthaltenen konstruktiven Elementen sind Zeit und Ereignisse als Verhaltens-elemente bereits abgedeckt. Das Profil bietet zusätzliche Beschreibungsmöglichkeiten für Analysemodelle.

abstraktion und Kapselung werden über das Klassenkonzept ermöglicht. Generische Abstraktion und Polymorphie gibt es ähnlich wie bei C++; zusätzlich kennt UML auch Templates für Pakete (Sammlungen von Definitionen).

Aus Sicht der wesentlichen Konzepte und der Werkzeugunterstützung ersetzt UML 2.x auch **ROOM** [SGW94], das ursprünglich in dem Werkzeug *ObjecTime* implementiert und anschließend als UML-Dialekt *UML-RT* [SR98] in dem Werkzeug *Rational Rose RealTime* angeboten wurde, das jetzt als Bestandteil von *Rational Rose Technical Developer* in Richtung UML 2.0 positioniert wird [IBM]. Auch von wesentlichen Beschreibungsmitteln von **SDL-2000** [SDL00] findet man Entsprechungen in UML 2.0, weshalb SDL hier nicht weiter betrachtet wird.

Die Sprache des Werkzeugs **ASCET** [ETA04] von ETAS ist objektorientiert imperativ, allerdings ohne das bei objektorientierten Sprachen übliche Konzept der Vererbung. ASCET besitzt eine C/Java-ähnliche sequentielle Teilsprache und unterstützt Nebenläufigkeit mit Konzepten und der Semantik des Echtzeitbetriebssystemstandards OSEK [OSE], so daß man ASCET als *OSEK-Programmiersprache* bezeichnen könnte. Zur Beschreibung reaktiven Verhaltens stehen hierarchische Automaten (ohne innere Nebenläufigkeit) mit einer Statechart-ähnlichen Notation zur Verfügung. Mit Blockdiagrammen (Signalflußgraphen) können regelungstechnische Algorithmen beschrieben werden, allerdings nicht funktional, sondern imperativ (Berechnungsreihenfolgen müssen explizit angegeben werden). ASCET unterstützt zur Simulation der Regelstrecke (d. h. außerhalb der eigentlichen Programmiersprache) auch die Modellierung zeitkontinuierlicher Systeme mit Differentialgleichungen und Blockdiagrammen.

Mit den sprachlichen Mitteln von ASCET wird die Beschreibung quasi-kontinuierlichen und reaktiven Verhaltens direkt unterstützt, jedoch ohne bei quasi-kontinuierlichem Verhalten von der imperativ-sequentiellen Realisierung zu abstrahieren. Zeitverbrauchendes sequentielles Verhalten muß wie bei sequentieller Programmierung mit Echtzeitbetriebssystem (siehe Abschnitt 4.2.1) nachgebildet werden. Funktionale Abstraktion erster Ordnung wird für sequentielle Programme über Methoden, für regelungstechnische Blöcke über Klassen unterstützt. Datenabstraktion kann über das Klassenkonzept erreicht werden. Funktionale Abstraktion höherer Ordnung und generische Abstraktion werden von ASCET nicht unterstützt.



**AutoFOCUS** [HSS96, HS01] ist ein prototypisches CASE-Werkzeug zur formalen Entwicklung (verteilter) eingebetteter Systeme, das Aspekte der FOCUS-Methode [BS01] implementiert. Systeme werden in AutoFOCUS als Hierarchie kommunizierender Automaten beschrieben, die über Datenflüsse (als Kanäle oder Ströme bezeichnet) miteinander verbunden sind. Die Automaten selbst sind hierarchische Zustandsübergangssysteme ohne interne Nebenläufigkeit, die auf Eingaben mit Ausgaben antworten und Zuweisungen an lokale Variablen vornehmen. Datentypen können in einer Haskell-ähnlichen funktionalen Sprache definiert werden. Verhalten ist in AutoFOCUS nicht konstruktiv zeitbehaftet. Allerdings können mit den in [HPS02] beschriebenen Echtzeiterweiterungen zeitliche Anforderungen (in annotierten Sequenzdiagrammen) formuliert werden, die bei der Ausführung eingehalten werden müssen. Die Beschreibungsmittel eignen sich damit für reaktives Verhalten, aber nur bedingt für Echtzeitanwendungen. Unterstützung für funktionale Abstraktion, Datenabstraktion und generische Abstraktion ist nicht verfügbar. Die jüngste Weiterentwicklung AutoFocus2 realisiert eine synchrone Datenflußsprache (siehe unter Abschnitt 4.3.1).

Die Liste der nebenläufigen imperativen Sprachen kann noch weit verlängert werden. Einige Sprachen lehnen sich an die Mechanismen von Hoare's *Communicating Sequential Processes* (CSP) [Hoa85] an (wie z. B. occam [Hoa88] oder TimedPL [FMO94]), greifen Ideen von Esterel, der bekanntesten synchronen imperativen Sprache (siehe Abschnitt 4.2.3), auf (so z. B. *Reactive C* [Bou91]) oder variieren Konzepte von Statecharts in Verbindung mit objektorientierter Programmierung (so z. B. ESP\* [SM05]). Eine gemeinsame Eigenschaft dieser Sprachen ist, daß sie Nebenläufigkeit und häufig Reaktivität sprachlich berücksichtigen, aber grundsätzlich keine Unterstützung für quasi-kontinuierliches Verhalten bieten.

### 4.2.3 Synchrone imperative Sprachen

**Esterel** [Ber00, BS91, BCE<sup>+</sup>03] gehört zusammen mit Lustre und Signal (siehe Abschnitt 4.3.1) zu den ältesten und prominentesten Vertretern der Familie der synchronen Sprachen (siehe Kapitel 2.3.3.1). Esterel-Programme sind blockstrukturiert, bestehend aus sequentiellen Anweisungen und Kontrollstrukturen im Stil der strukturierten Programmierung; Nebenläufigkeit wird durch den Parallisierungsoperator (`| |`) ausgedrückt. Aussprungpunkte (traps) und Präemption bei Eintreffen von *Signalen* (aborts) erlauben das Verlassen ganzer

Code-Blöcke. *Broadcasting* von Signalen sind der primäre Kommunikationsmechanismus zwischen nebenläufigen Programmteilen und mit der Umgebung. Signale kommunizieren Ereignisse, können dabei aber auch einen Wert tragen. Sie können gesendet, erwartet oder auf ihr Vorhandensein zum gegenwärtigen Zeitpunkt überprüft und ihr Wert bestimmt werden (ein Signal ist zu genau den Zeitpunkten vorhanden, zu denen es gesendet wird). Signale können nach dem Synchronitätsprinzip als Reaktion auf Signale zum gleichen Zeitpunkt gesendet werden; es gilt dabei eine Fixpunktsemantik, Kausalitätsverletzungen werden vom Compiler erkannt.

Die Beschreibungsmittel von Esterel sind auf reaktives und sequentielles Verhalten zugeschnitten. Die Nachbildung von quasi-kontinuierlichem Verhalten wird durch Signale mit Wert, den `pre`-Operator zum Zugriff auf den Wert eines Signals zum vorhergehenden Zeitpunkt und durch das Synchronitätsprinzip, das den Zugriff auf berechnete Signalwerte zum gleichen Zeitpunkt erlaubt, unterstützt. Signalflußgraphen können damit relativ einfach in Esterel-Programme übersetzt werden. Verhaltensabstraktion läßt sich durchgängig darstellen. Esterel unterstützt prozedurale Abstraktion erster Ordnung, wobei Unterprogramme hier *Module* heißen. Esterel besitzt keine eigenen Mechanismen zur Datenabstraktion<sup>9</sup> und Kapselung, kennt dementsprechend auch keine generische Abstraktion.

Der auf Esterel basierende Ansatz **ViPER** von Gunzert [GR99, Gun02] enthält ein Komponentenmodell, das gegenüber Esterel weitergehende Möglichkeiten der funktionalen Abstraktion bietet. ViPER bietet aber keine zusätzlichen Kapselungsmechanismen; Komponenten haben hier wie Esterel-Module stets Signalschnittstellen.

In der Literatur sind verschiedene Varianten von Esterel beschrieben. **SL** [BS96] erlaubt die direkte Einbettung von C-Code und schränkt die Kommunikationsmöglichkeiten gegenüber Esterel derart ein, daß Kausalitätsprobleme grundsätzlich unterdrückt werden und eine einfache Implementierung in Reactive C möglich ist [Ber00].

**PURR** [KRSW98] ist eine auf Esterel basierende Systembeschreibungssprache zur Spezifikation und Verifikation. Auch wenn es sich nicht um eine Programmiersprache handelt, bietet PURR aus Programmiersprachensicht zusätzlich erweiterte Möglichkeiten der funktionalen Abstraktion (generische Module) und abstrakte Datentypen,

<sup>9</sup>Die Implementierung von Datentypen, Funktionen und Prozeduren muß in einer Wirtssprache, z. B. C, erfolgen.

aber weiterhin keinen allgemeinen Kapselungsmechanismus, der auch dynamisches Verhalten mit einbezieht.

**SyncCharts** [And96] ist eine mit Esterel kompatible Statechart-ähnliche graphische Sprache, die den Nebenläufigkeits- und Kommunikationsaspekt im Stil von Statecharts anstelle der entsprechenden Kontrollstrukturen von Esterel handhabt. Aus Sicht der SyncCharts primitives Verhalten wird in Esterel beschrieben. Aus Sicht dieser Arbeit bringen SyncCharts gegenüber Esterel keine neue Mächtigkeit. Verhaltensabstraktion von in Esterel beschriebenem Verhalten zu reaktivem Verhalten in Form von SyncCharts ist möglich; umgekehrt können Ein-/Ausgangssignale von SyncCharts in Esterel-Modulen verwendet werden.

Ziel der Sprache **synERJY** [BPS04, BPS05] (Nachfolger von **sE** (synchronousEifel) [BP01] und **LEA** [HP98]) ist die Kombination des synchronen Ausführungsmodells mit Objektorientierung und die Integration verschiedener Beschreibungsstile. synERJY basiert auf einer Untermenge von Java und bringt Erweiterungen für synchrones Verhalten (sogenannte *reaktive Klassen*). Sequentielles und reaktives Verhalten kann im Stil von Esterel und in Automatenform beschrieben werden, quasi-kontinuierliches Verhalten im Stil von Lustre, wobei die Syntax von Esterel und Lustre weitgehend erhalten bleibt. Alle drei Stile können feingranular gemischt werden.

Funktionale Abstraktion erster Ordnung für synchrones Verhalten ermöglichen sogenannte reaktive Methoden (für sequentielle und reaktive Prozesse) und Knoten (für Datenflüsse); für Berechnungen sind es gewöhnliche Methoden. Die Modularisierung erfolgt in synERJY über das Klassenkonzept, das auch Datenabstraktion bereitstellt. Wie schon bei ViPER haben die sogenannten reaktiven Objekte jedoch ausschließlich Signalschnittstellen. Eine weitergehende Kapselung für dynamisches Verhalten (Abstrakte Maschinen) wird nicht unterstützt. Generische Abstraktion ist über Klassen mit Typparametern und funktionale Abstraktion höherer Ordnung ansatzweise über anonyme Klassen möglich.

## 4.3 Funktionale Sprachen

### 4.3.1 Synchrone Datenflußsprachen

Synchrone Datenflußsprachen arbeiten mit getakteten Datenflüssen, die bei jedem Zeitschritt einen Wert liefern. Getaktete Datenflüsse

entsprechen abgetasteten (zeitdiskreten) Signalen. Nebenläufigkeit ist implizit im Berechnungsmodell enthalten; alle Datenflüsse fließen parallel und synchron. Als Variante davon kann es in einem System bzw. Programm mehrere unterschiedliche Zeittakte geben. Übergänge zwischen verschiedenen Zeittakten können durch Bindeglieder (Operatoren) zur Abtastung bzw. Interpolation hergestellt werden.

In der synchronen Datenflußsprache **Lustre** [HCRP91] hat jeder Fluß einen Typ für seine Werte und einen Takt. Es gibt einen Basis-Takt, von dem langsamere Takte abgeleitet werden können. Dies erfolgt mittels boolescher Flüsse: Der abgeleitete Takt besteht dann aus der Folge von Zeitpunkten, zu denen der Fluß wahr liefert (nicht notwendigerweise äquidistant). Konstanten sind Flüsse mit konstantem Wert und Basis-Takt. Es gibt nur primitive Datentypen und Tupel; komplexere Datentypen können als abstrakte Datentypen von einer Wirtssprache importiert werden (ähnlich wie in Esterel). Eingebaute arithmetische Operatoren und von der Wirtssprache importierte Funktionen arbeiten punktweise auf Flüssen. Weiter gibt es zeitbezogene Operatoren für Zeitverschiebung (Verzögerung) um einen Schritt, Abtastung und Interpolation.

Die Beschreibungsmittel sind unmittelbar geeignet für quasi-kontinuierliches Verhalten. Um reaktives oder sequentielles Verhalten darstellen zu können, ist eine Codierung als Steuerwerk (wie in Hardware) nötig. Ereignisse können dabei als boolesche Datenflüsse mit Basis-Takt (wahr bei Ereignis) codiert werden. Lustre kennt funktionale Abstraktion erster Ordnung; Datenflußwertige Funktionen heißen hier *Knoten* (gedacht ist an Datenflußgraphen). Die Sprache hat keine eigenen Mechanismen zur Datenabstraktion, Kapselung und generischen Abstraktion.

**Mode-Automata** [MR03] als Erweiterung von Lustre erlauben die Beschreibung modalen Verhaltens mittels hierarchischer und nebenläufiger Automaten, die zwischen unterschiedlichen Definitionen der gleichen Datenflüsse umschalten. Transitionen sind dabei über Bedingungen definiert. Mit der Codierung von Ereignissen in booleschen Datenflüssen kann damit reaktives Verhalten in geeigneter Weise beschrieben werden und abstrahiert i. a. von quasi-kontinuierlichem Verhalten; sequentielles Verhalten kann durch Automaten einfacher nachgebildet werden. Eine ähnliche Erweiterung von Datenflußsprachen um Zustandsautomaten wird von [CPP05] vorgeschlagen.

Eine interessante Beobachtung bei der praktischen Anwendung von Mode-Automaten berichten die Autoren [MR03]: Es kommt vor, daß mehrere Automaten in unterschiedlichen Kontexten zwar unterschiedlichen Datenflußgleichungen, aber die gleiche Struktur besitzen. Sie schlagen für zukünftige Arbeiten eine Spracherweiterung vor, die diesbezügliche Wiederverwendung erlaubt. Als Kommentar dazu wäre funktionale Abstraktion höherer Ordnung als allgemein anwendbarer Mechanismus vorzuschlagen.

Die Sprache **Lucid Synchron** [CP96, CP00] ist eine funktionale Erweiterung von Lustre, die Funktionen höherer Ordnung und rekursive Funktionen für die Verarbeitung von Datenflüssen erlaubt. Mit dynamischen Netzwerken durch rekursive Funktionen löst sich Lucid Synchron aber explizit von der Anwendungsdomäne eingebetteter Echtzeitsysteme, da Rechenzeiten und Speicherbedarf im allgemeinen nicht mehr statisch beschränkt sind. [CGHP04] führt diese Arbeit weiter und erlaubt datenflußwertige Funktionen als erstklassige Werte, die in Datenflüssen übertragen werden können. Dabei wird auf dynamische rekonfigurierbare und aktualisierbare Systeme als Anwendung abgezielt.

In der mit Lustre vergleichbaren Sprache **Signal** [GGBM91] heißen Datenflüsse *Signale* und haben einen festen Basis-Takt. Wie bei Esterel kann ein Signal zu jedem Zeitpunkt entweder vorhanden oder nicht vorhanden sein. Takte können als Äquivalenzklassen von Signalen, die stets gleichzeitig präsent sind, aufgefaßt werden. Zur Signalverarbeitung stehen punktweise angewandte arithmetisch-logische Operatoren oder externe (d. h. in einer Wirtssprache definierte) Funktionen, Zeitverschiebung um beliebig viele Zeitschritte, Abtastung (gesteuert durch ein boolesches Signal) sowie das deterministische Mischen zweier Signale (bei gleichzeitiger Präsenz hat eines der Signale den Vorrang) zur Verfügung, ergänzt um diversen syntaktischen Zucker und erstklassige Vektor-Operationen. Ein System von Definitionsgleichungen für Signale wird als *Prozeß* verstanden. Die funktionale Abstraktion erster Ordnung davon heißt *Modell* (äquivalent zu einer signalverarbeitenden Funktion oder einem Block in Blockdiagrammnotation). Es können auch Arrays von Prozessen instantiiert werden.

Bezüglich der Verhaltensmodelle aus Abschnitt 2.4 und der Programmiersprachenkonzepte aus Abschnitt 2.2.2 ist Signal in den Beschreibungsmöglichkeiten äquivalent zu Lustre.

Das Berechnungsmodell von **AutoFocus2** [BRS05] arbeitet mit synchronen Datenflüssen und Takten im Stil von Lustre. AutoFocus2 ersetzt die Verhaltensmodellierung von AutoFOCUS in Form von imperativen hierarchischen Automaten (siehe unter Abschnitt 4.2.2) durch Datenflußgraphen und Modi im Stil von Mode-Automata. Systeme werden hierarchisch in Komponenten zerlegt, zwischen denen die Kommunikation um einen Zeitschritt versetzt erfolgt. Innerhalb einer Komponente kommunizieren die Funktionsblöcke eines Datenflußgraphen nach dem synchronen Modell. Bezüglich der Abstraktionsmechanismen ergibt sich keine Änderungen gegenüber AutoFOCUS.

### 4.3.2 Andere synchrone funktionale Sprachen

Ausgehend von der Anwendung der rein funktionalen Programmiersprache Haskell auf reaktive Anwendungen ohne Echtzeitbedingungen und mit einem zeitkontinuierlichen Verhaltensmodell [EH97, Hud00], wurde **FRP** (Functional Reactive Programming) [WH00] als in Haskell eingebettete domänenspezifische Sprache entwickelt. FRP kennt zeitkontinuierliche Signale (als Behaviours bezeichnet) und wertbehaftete diskrete Ereignisse und erlaubt die Beschreibung kontinuierlichen Verhaltens mit Diskontinuitäten, ausgelöst durch Ereignisse. Mit Real-Time FRP und Event-Driven FRP wurden die Grundkonzepte von FRP zur Verhaltensbeschreibung in Richtung einer Anwendung auf Echtzeit- und eingebettete Systeme weiterentwickelt.

**Real-Time FRP** (RT-FRP) [WTH01] arbeitet mit Signalen auf einer diskreten Zeitbasis und erlaubt die Beschreibung quasi-kontinuierlichen und reaktiven Verhaltens. Quasi-kontinuierliches Verhalten wird über einen Zeitverschiebungsoperator für Signale ermöglicht. Bei reaktivem Verhalten schalten Ereignisse zwischen Signalen um. Sequentielles Verhalten muß über reaktives Verhalten nachgebildet werden. **Event-Driven FRP** (E-FRP) [WTH02] ist eine Weiterentwicklung von RT-FRP, die von einem festen Zeittakt zur Steuerung durch externe Ereignisse übergeht und den Zeittakt nur noch als spezielles Ereignis beibehält. Signalwerte können sich nur durch Ereignisse ändern. Es wird angenommen, daß nie zwei Ereignisse gleichzeitig auftreten. Eine Nachbildung quasi-kontinuierlichen Verhaltens erscheint möglich, eine Möglichkeit zur Abstraktion von quasi-kontinuierlichem zu reaktivem Verhalten (ähnlich Mode-Automata) ist nicht ohne weiteres erkennbar. Beide Sprachen sind synchron nach dem Prinzip der synchronen Sprachen (siehe Abschnitt 2.3.3.1).

RT-FRP und E-FRP sind von Haskell gelöste, eigenständige funktionalen Sprachen für Verhalten, die auf einer beliebigen Basis-Sprache für Werteberechnungen aufsetzen. Die Forschungsarbeiten konzentrieren sich auf die syntaktischen und semantischen Aspekte der Verhaltensbeschreibung; Software-Engineering-Aspekte und Konzepte wie Datenabstraktion, Kapselung und generische Abstraktion werden in den Veröffentlichungen [WTH01, WTH02] nicht betrachtet. E-FRP unterstützt nur funktionale Programmierung erster Ordnung.

### 4.3.3 Andere funktionale Sprachen

Die als domänenspezifische Sprache für eingebettete Echtzeitsysteme entworfene funktionale Sprache **Hume** [HM03] verbindet kommunizierende endliche Automaten mit Petri-Netz-artiger Nebenläufigkeit. Ein Automat besitzt in Hume jedoch keinen Zustand (!). Untereinander vernetzte *Boxen* als Abstraktionen von endlichen Automaten bilden zyklisch ein Tupel von Eingaben auf ein Tupel von Ausgaben ab, beschrieben durch eine Funktion mit Pattern Matching. Ausgänge werden mit Eingängen der gleichen oder anderer Boxen verbunden. Eingänge puffern Werte mit Puffer-Größe 1. Zustandsspeicherung wird dadurch realisiert, daß ein Ausgang auf einen Eingang zurückgeführt wird, der ihn bis zur nächsten Ausführung der Box puffert. Eine Box ist erst ausführbar, wenn alle Eingänge belegt sind. Umgekehrt wird sie in ihrer Ausführung so lange blockiert, wie sie über einen Ausgang auf einen belegten Eingang schreiben würde. Alle Boxen arbeiten nebenläufig. Effektiv ist Hume damit eine Datenflußsprache. Das Programm terminiert, wenn keine Box mehr ausführbar ist.

Für die ein- und ausgegebenen Werte können algebraische Datentypen, ähnlich wie in Haskell oder ML, definiert werden. Hume unterscheidet neben einer *Ausdruckssprache* (für die Funktionen der Boxen) und einer *Koordinationssprache* (Boxen und ihre Verdrahtung) noch eine *Metasprache* zur Abstraktion und Applikation. Für Ausdrücke können Funktionen (auch höherer Ordnung) definiert werden. Aus Templates für Boxen und Makros für Verdrahtungen und struktureller Iteration können komplexe Netzwerke instantiiert werden, Boxen sind jedoch nicht hierarchisch. Templates sind der Veröffentlichung zufolge (ohne nähere Erklärung) polymorph.

Boxen unterstützen ein Konzept der Ausnahmebehandlung, wobei Ausnahmen abgefangen und von einer separaten Funktion behandelt werden, die in der Ausführung an die Stelle der normalen Funktion

tritt. Als spezielle Anwendung davon können *Timeouts* auf Berechnungen von Ausdrücken und sowie auf Eingänge und Ausgänge vorgegeben werden, die als Ausnahmen eintreffen. Timeouts auf Berechnungen können die Einhaltung von Deadlines überwachen, Timeouts auf Kommunikationen können für zeitperiodische Berechnungen benutzt werden. Für die Systemgrenze gibt es verschiedene Arten von Geräten (einschließlich Ports und Interrupts), die analog zu Ein- und Ausgängen von Boxen Quelle oder Ziel von Verdrahtungen sein können.

Die Beschreibungsmittel von Hume bieten weder für quasi-kontinuierliches noch für reaktives Verhalten eine direkte Unterstützung; diese ist nur für sequentielles Verhalten nach Art von Petri-Netzen erkennbar. Die Mechanismen zur funktionalen Abstraktion sind unklar ausgeprägt; Datenabstraktion, Kapselung und generische Abstraktion werden nicht unterstützt.

**Embedded Gofer** [WR95b, WR95a] ist eine Variante von Haskell mit Erweiterungen für Prozesse, die über Nachrichten kommunizieren, für Zugriff auf Ein-/Ausgabe-Register und für Interrupts (als spezielle Nachrichten behandelt). Die Nachrichtenverarbeitung erfolgt mit Fortsetzungsfunktionen (Continuations), mit denen relativ einfach reaktive und sequentielle Kontrollflüsse dargestellt werden können. Hierarchische Zustandsautomaten sind mangels Präemptionsmöglichkeiten jedoch nicht ohne weiteres darstellbar, was auch Verhaltensabstraktion verhindert. Die allgemeinen Abstraktionsmechanismen werden von Haskell geerbt (Funktionen höherer Ordnung und ein polymorphes Typsystem mit (nach heutigem Stand von Haskell) Typ-Klassen als Kapselungsmechanismus).

## 4.4 Sprachen mit imperativem und funktionalem Paradigma

### 4.4.1 Synchrone imperativ/funktionale Sprachen

Für eine Untermenge der Sprache(n) des Modellierungs- und Simulationswerkzeugs **Simulink** [The06d] von The MathWorks stehen Coder-Generatoren von den Firmen The MathWorks (Real-Time Workshop Embedded Coder) [The06b] und dSPACE (TargetLink) [dSP06] zur Verfügung, die die Programmierung eingebetteter Echtzeitsysteme in Simulink erlauben. Simulink basiert auf MATLAB [The06a],



einer imperativen Programmiersprache zur Array- und Matrixverarbeitung und setzt eine graphische Sprache (Blockdiagramme) zur Beschreibung zeitkontinuierlicher oder zeitdiskreter dynamische Systeme (auch gemischt und mit mehreren Takten) auf. Mit dem Teilwerkzeug **Stateflow** [The06c] kommt eine zweite graphische, Statechart-ähnliche Sprache zur Beschreibung ereignisdiskreter Systeme hinzu, die als atomare Subsysteme in Simulink eingebettet werden.

Die Sprache der Blockdiagramme (hierarchische Signalflußgraphen) wird ergänzt um eine umfangreiche und erweiterbare Blockbibliothek (angefangen bei arithmetisch-logischen Operationen). Sie enthält unter anderem generische Blöcke (polymorphe signalwertige Funktionen) für Integration (zeitkontinuierlich) und Zeitverschiebung (zeitdiskret) und verwandte Operationen, die eine Repräsentation von Differenzial- und Differenzengleichungen erlauben. Zeitkontinuierliche und zeitdiskrete Signale unterschiedlicher Taktraten können mit Abtast- und Haltegliedern aufeinander angepaßt werden. Für den zeitdiskreten Anteil der Sprache ist die Generierung von Echtzeit-Code möglich. Dieser Teil der Sprache ist eine synchrone Datenflußsprache, vergleichbar mit Lustre. Eine für regelungstechnische Anwendungen interessante Spezialität sind die Funktionsblöcke höherer Ordnung, die komplexwertige Übertragungsfunktionen (Laplace- bzw. Z-transformiert) als Parameter entgegennehmen können.

Das Übertragungsverhalten atomarer Blöcke kann imperativ in MATLAB oder der Wirtssprache (C) programmiert oder in Stateflow beschrieben werden. Diese Prozeduren werden innerhalb eines Zeitschritts ausgeführt. Das ansonsten funktionale Paradigma der Datenflußsprache wird aufgeweicht durch imperative Konstrukte, die Einfluß auf die zyklische Berechnung der Blöcke nehmen. So gibt es neben sogenannten *enabled subsystems*, die nur bei einem positivem Steuersignal an einem speziellen Eingang ausgeführt werden (ansonsten werden in dem Berechnungszyklus die Ausgangswerte beibehalten oder auf Initialwerte zurückgesetzt; ähnlich zu entscheiden ist bei Wieder Ausführung) auch Subsysteme, die nur bei Trigger-Ereignissen (definierten Flanken auf einem Steuersignal) oder durch einen Funktionsaufruf ausgeführt werden. Funktionsaufrufe können aus imperativem Code in atomaren Blöcken, von Stateflow-Automaten und von speziellen Funktionsaufruf-Generatoren oder Kontrollflußblöcken kommen. Es gibt dabei auch Kontrollflußblöcke für Alternative, Fallunterscheidung und Schleifen (äquivalent zu entsprechenden Konstrukten in

C), durch die imperativ-algorithmisch eine Berechnung pro Zeitschritt spezifiziert werden kann.

Mit Stateflow können hierarchische, nebenläufige Automaten mit einer speziellen, vermutlich stark von der C-Code-Generierung geprägten Semantik beschrieben werden. Ein Automat wird als Block in Simulink eingebunden und kann dabei Daten- und Ereignis-/Funktionsaufrufchnittstellen zum Kontext besitzen (beides in beide Richtungen). Die zyklische Berechnung des Automaten wird entweder durch ein oder mehrere Ereignisse oder über genau eine Funktionsaufrufchnittstelle getriggert. Innerhalb einer solchen Ausführung kann es Automaten-intern eine Ereigniskaskade geben. Stateflow erlaubt auch die graphische Definition von Funktionen (Prozeduren), die innerhalb von Stateflow-Blöcken verwendet werden können.

In Simulink/Stateflow ist die Beschreibung von quasi-kontinuierlichem und reaktivem Verhalten möglich. Sequentielles Verhalten muß reaktiv nachgebildet werden. Die Integration der Beschreibungsmittel erscheint wenig systematisch. In Simulink können Subsysteme parametrierbar als wiederverwendbare Blöcke definiert und in benutzerdefinierten Blockbibliotheken abgelegt werden. Dadurch wird funktionale Abstraktion erster Ordnung unterstützt; selbstdefinierte Funktionen höherer Ordnung sind nicht möglich. Stateflow bietet keine funktionale Abstraktion für Unterautomaten; es können nur ganze Automaten als Simulink-Blöcke mehrfach instantiiert werden. In Form der graphischen Funktionen bietet Stateflow aber prozedurale Abstraktion für Aktionen in Automaten. In MATLAB gibt es als primären Abstraktionsmechanismus Funktionen, die auch selbst Parameter und anonym sein können, und zur Datenabstraktion (ausgehend von verschiedenen skalaren Datentypen, Vektoren und Matrizen) Klassen. Funktionen können überladen werden, zwischen Klassen kann es Vererbung geben. Simulink-Blöcke können in einer eingeschränkten Form polymorph sein (signal propagation). In allgemeiner Form werden aber Kapselung und generische Abstraktion von MATLAB, Simulink und Stateflow nicht unterstützt.

Die objektorientierte synchrone Sprache **synERJY** (siehe Abschnitt 4.2.3) kann wegen ihres Datenflußanteils mit Knoten als Abstraktionskonzept auch zu den Sprachen mit imperativem und funktionalem Paradigma gezählt werden. Das imperative Paradigma prägt jedoch diese Sprache.

### 4.4.2 Andere imperativ/funktionale Sprachen

**Timber** [BCJ<sup>+</sup>02] ist eine objektorientiert imperative Sprache, basierend auf einem rein funktionalen Kern (einer Variante von Haskell). Über ein Nachrichtenkonzept, mit dem Objekte untereinander kommunizieren und das auch externe Ereignisse als Nachrichten behandelt, kann reaktives Verhalten dargestellt werden. Nachrichten sind Methoden-Ausführungen (Aktionen), denen Zeitbedingungen (Zeitintervall von frühestem Start bis zu spätester Terminierung) auferlegt werden können. Die Zeitbedingungen bestimmen ein dynamisches Scheduling für das Verteilen von Nachrichten, das das Ausführungsmodell von Timber bestimmt. Sequentielles und quasi-kontinuierliches Verhalten muß mit den gleichen Mitteln nachgebildet werden. Von dem funktionalen Kern erbt Timber funktionale Abstraktion höherer Ordnung, algebraische Datentypen, Typklassen und parametrische Polymorphie und unterstützt damit auch Datenabstraktion und generische Abstraktion.

## 4.5 Fazit

Die Evaluierung der in den vorangegangenen Abschnitten beschriebenen Sprachen gegenüber den in Kapitel 3.4 gestellten Anforderungen ist in den Tabellen 4.1 und 4.2 zusammengefaßt.

Hinsichtlich Vollständigkeit und Durchgängigkeit der Beschreibungsmittel erfüllen nur *synERJY* und – mit Abstrichen bei quasi-kontinuierlichem Verhalten – *Esterel* oder die Kombination aus *SynchCharts* und *Esterel* die Anforderungen.

Die Abstraktionsmechanismen von *Esterel* reichen allerdings nicht über funktionale Abstraktion hinaus und erfüllen die methodischen Anforderungen nicht. Die objektorientierte Sprache *synERJY* bietet zwar zusätzliche Mechanismen, stellt sie jedoch nicht so allgemein zur Verfügung, daß Abstrakte Maschinen damit realisiert werden könnten. Module sind bei beiden Sprachen auf Signalschnittstellen beschränkt und unterstützen damit nur Blockstrukturen (funktionale Dekomposition). Eine methodische Ausrichtung auf Abstraktion als Mittel zur Komplexitätsbeherrschung beim Entwurf von Prozeßsteuerungen wird von keiner der Sprachen in ausreichendem Maß unterstützt.

	harte Echtzeit	quasi- kont.	reakt.	sequ.	Verh.- abstr.
<i>Sequentielle Sprachen</i>					
C	-	-	-	(-)	-
C++	-	-	-	(-)	-
<i>Nebenläufige Sprachen</i>					
Ada95	(-)	-	(-)	+	-
Real-Time Java	(-)	-	(-)	+	-
Giotto	+	-	-	-	-
Statemate	(+)	-	+	-	-
UML 2.0	(-)	-	+	+	(+)
ASCET	+	(+)	+	-	-
AutoFocus	-	-	+	-	-
<i>Synchrone imperative Sprachen</i>					
Esterel, ViPER, SL, PURR	+	(-)	+	+	+
SynchCharts	+	(-)	+	(-)	+
SynERJY	+	+	+	+	+
<i>Synchrone Datenflußsprachen</i>					
Lustre	+	+	-	-	-
Mode-Automata	+	+	+	-	(+)
Lucid Synchrone	+	+	-	-	-
Signal	+	+	-	-	-
AutoFocus2	+	+	+	-	(+)
<i>Andere synchrone funktionale Sprachen</i>					
RT-FRP	+	+	+	-	(+)
E-FRP	(-)	-	+	-	-
<i>Andere funktionale Sprachen</i>					
Hume	+	-	-	(+)	-
Embedded Gofer	(-)	-	(+)	+	-
<i>Synchrone imperativ / funktionale Sprachen</i>					
Simulink/Stateflow	+	+	+	-	-
<i>Andere imperativ / funktionale Sprachen</i>					
Timber	(-)	-	(-)	-	-

Tabelle 4.1: Sprachenvergleich: Echtzeitfähigkeit und Beschreibungsmittel

	fkt. Abstr.	höherer Ordnung	Kaps.	gener. Abstr.	Abstr. Masch.
<i>Sequentielle Sprachen</i>					
C	+	(-)	-	-	-
C++	+	(-)	+	+	-
<i>Nebenläufige Sprachen</i>					
Ada95	+	-	+	+	-
Real-Time Java	+	(+)	+	+	-
Giotto	-	-	-	-	-
Statemate	+	-	-	-	-
UML 2.0	+	-	+	+	-
ASCET	+	-	+	-	-
AutoFocus	-	-	-	-	-
<i>Synchrone imperative Sprachen</i>					
Esterel, ViPER, SL, PURR	+	-	-	-	-
SynchCharts	+	-	-	-	-
SynERJY	+	(+)	(+)	+	-
<i>Synchrone Datenflußsprachen</i>					
Lustre, Mode-Automata	+	-	-	-	-
Lucid Synchrone	+	+	-	-	-
Signal	+	-	-	-	-
AutoFocus2	-	-	-	-	-
<i>Andere synchrone funktionale Sprachen</i>					
RT-FRP	+	+	-?	-?	-?
E-FRP	+	-	-?	-?	-
<i>Andere funktionale Sprachen</i>					
Hume	(+)	-	-	-	-
Embedded Gofer	+	+	+	+	-
<i>Synchrone imperativ / funktionale Sprachen</i>					
Simulink/Stateflow	+	(-)	(-)	(-)	-
<i>Andere imperativ / funktionale Sprachen</i>					
Timber	+	+	+	+	-

Tabelle 4.2: Sprachenvergleich: Abstraktionsmechanismen



# 5. Sprachkonzept

## 5.1 Anforderungen

Die in Kapitel 3.4 formulierten Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme können als eine Minimalforderung aus methodischer Sicht angesehen werden. Diese Anforderungen erstmals vollständig zu erfüllen und den methodischen Ansatz zu belegen, ist das Ziel des Neuentwurfs einer Programmiersprache FSPL (Functional Synchronous Programming Language), plakativ dargestellt in den Tabellen 5.1 und 5.2.

Dabei wird nicht der Anspruch erhoben, alle relevanten Anforderungen an eine Programmiersprache für eingebettete Echtzeitsysteme formuliert zu haben und zu erfüllen. So erfolgt etwa keine explizite Betrachtung von Code-Effizienz bei der Compilierung auf Zielplattformen (auch wenn die Sprache implizit dafür entworfen wurde) oder von definierten Schnittstellen zu anderen Sprachen (wie etwa C; siehe dazu die Hinweise in Kapitel 7.5.1). Die nur subjektiv zu bewertende Lesbarkeit der Syntax wurde genauso als weitere Anforderung verstanden wie eine formale, mathematische Semantik, um einen Ansatzpunkt für Konzepte zu formaler Verifikation zu bieten.

	harte Echtzeit	quasi- kont.	reakt.	sequ.	Verh.- abstr.
FSPL	+	+	+	+	+

Tabelle 5.1: FSPL: Echtzeitfähigkeit und Beschreibungsmittel

	fkt. Abstr.	höherer Ordnung	Kaps.	gener. Abstr.	Abstr. Masch.
FSPL	+	+	+	+	+

Tabelle 5.2: FSPL: Abstraktionsmechanismen

## 5.2 Echtzeitfähigkeit

*synchrones  
Systemmodell*

Im Hinblick auf die Eignung für Echtzeitprogrammierung (mit Nebenläufigkeit und Zeitfortschritt) allgemein und speziell für Systeme mit harten Echtzeitbedingungen (siehe Kapitel 2.3) macht sich FSPL das synchrone Systemmodell der synchronen Sprachen (siehe Kapitel 2.3.3.1) zunutze (daher FSPL). Auf Basis einer diskreten Zeit werden Systeme als getaktet betrachtet. Die Programmausführung erfolgt zeitgesteuert, Ereignisse werden auf den Basistakt synchronisiert.

## 5.3 Durchgängige Beschreibungsmittel für Prozeßsteuerungen

*integriertes Ver-  
haltensmodell*

Um adäquate Beschreibungsmittel für quasi-kontinuierliche Regelungen und reaktive und sequentielle Steuerungen (siehe Kapitel 1.1.2) nach den Verhaltensmodellen aus Kapitel 2.4 bereitzustellen, wird – wie nachfolgend skizziert – ein vertikal integriertes semantisches Modell für quasi-kontinuierliches, reaktives und sequentielles Verhalten entwickelt (die skizzierten Zusammenhänge sind in Kapitel 6 ausführlich beschrieben und formal modelliert). Die Art der Integration stellt dabei die Durchgängigkeit her, mit der konstruktive Verhaltensabstraktion ermöglicht wird. Bei der Wahl der Beschreibungsmittel für sequentielles Verhalten wird eine prozedurale Beschreibungsform gegenüber Petrinetzen als Verhaltensmodell bevorzugt.

*Prozesse =  
Signale =  
Variablen*

Auf Basis der diskreten Zeit werden *Prozesse*, *Signale* und *Variablen* als ein identisches Konzept betrachtet, nämlich als Funktionen der Zeit (von 0 bis  $\infty$ ) mit beliebigem Wertebereich. Zustände eines Prozesses sind gleichbedeutend mit Werten eines Signals bzw. einer Variablen. Die Begriffe Prozeß, Signal und Variable werden synonym verwendet. Mathematische Operationen (einschließlich einer Operation zur Zeitverschiebung um einen Takt) bilden Signale aus Signalen. Ein signalverarbeitendes System ist repräsentiert durch eine signalwertige Funktion; neben Signalen können auch *Ereignisse* als Eingabe auftreten, nicht jedoch als Ausgabe.



*Ereignisse* werden von Prozessen (Signalen, Variablen) unterschieden. Ereignisse finden zu diskreten Zeitpunkten synchron zum Zeittakt statt und können sich wiederholen - im Grenzfall zu jedem Zeitpunkt. Ereignisse sind jedoch keine booleschen Signale; sie können nicht zu Nicht-Ereignissen invertiert werden. Die Zeitpunkte, zu denen ein Ereignis positiv stattfindet, bieten eine vergrößerte Sicht auf die Zeit. Ereignisse können durch Signal- oder Zeit-Trigger spezifiziert und auf verschiedene Weise untereinander, mit Signal- oder mit Zeitbedingungen verknüpft werden.

*Ereignisse*  $\neq$   
*Signale*

Prozesse können abschnittsweise zusammengesetzt werden. Die Abschnitte werden durch Ereignisse begrenzt und durch *Phasen* definiert. Eine Phase ist eine prozeßwertige Funktion der Zeit; instantiiert man eine Phase zu einem bestimmten Zeitpunkt, beschreibt sie ein Prozeßverhalten ab diesem Zeitpunkt (bis  $\infty$ ). Atomare Phasen sind durch genau einen Prozeß definiert, den sie bei der Instantiierung wiedergeben. *Transitionen* sind Verknüpfungen zwischen zwei oder mehr Phasen und entsprechend vielen Ereignissen derart, daß bei Stattfinden eines der Ereignisse die Ausgangsphase verlassen und eine entsprechende Zielphase zu diesem Zeitpunkt instantiiert (gestartet) wird; das Ergebnis ist wiederum eine (zusammengesetzte) Phase. Durch explizite Instantiierung einer Phase zum Nullzeitpunkt ergibt sich ein Prozeß, der dann im Fall einer zusammengesetzten Phase abschnittsweise definiert ist. Rekurrente Transitionen (und somit das mehrfache Starten einer Phase innerhalb eines Prozesses) sind möglich. Ein System aus durch Transitionen miteinander verknüpften Phasen ist ein *Phasenübergangssystem*.

*Phasen*

Eine *Aktion* ist ein Paar aus einer Phase und einem Ereignis. Das Starten der Aktion startet die Phase, das Eintreten des Ereignisses beendet die Aktion. Eine Aktion entspricht einer Transition für zwei Phasen, angewendet auf die Ausgangsphase bei offen gelassener Zielphase. *Atomare* Aktionen sind durch genau eine Phase und ein Ereignis definiert. Aktionen können untereinander und mit Signalbedingungen zu Aktionen verknüpft werden. Eine Endlosschleife über einer Aktion ergibt eine Phase (Selbsttransition auf die Phase beim terminierenden Ereignis); die Fortsetzung einer Aktion durch eine Phase ergibt eine Phase (Transition zu einer Zielphase beim terminierenden Ereignis). Eine zusammengesetzte Aktion ist eine *Prozedur*.

*Aktionen*

Phasen und Ereignisse gehören konzeptionell zusammen, so daß drei Ebenen der Verhaltensbeschreibung entstehen, die miteinander ver-

*Verschränkung  
der Konzepte*

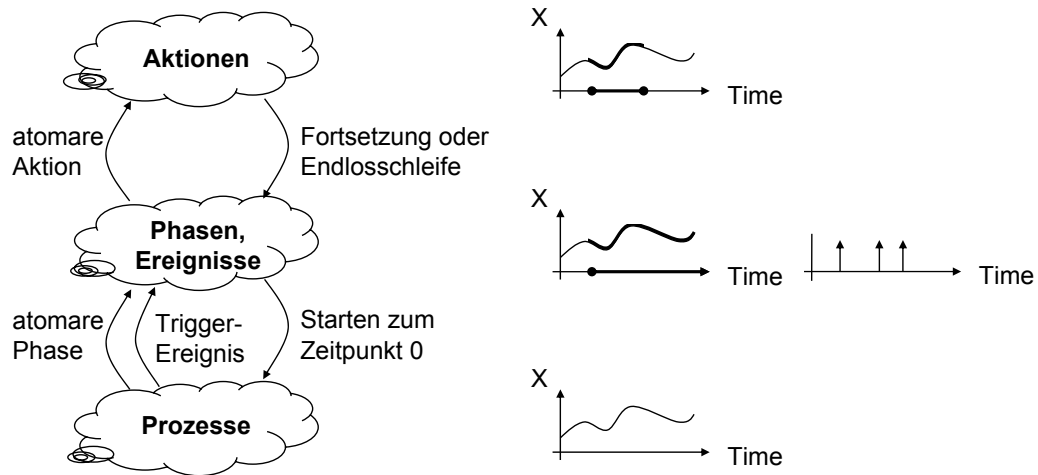


Abbildung 5.1: Verschränkung der Konzepte

schränkt sind (siehe Abbildung 5.1). Atomare Phasen entstehen aus Prozessen, Prozesse lösen Ereignisse aus, und das Starten einer Phase zum Zeitpunkt 0 erzeugt einen Prozeß aus einer Phase; eine atomare Aktion wird aus einer Phase und einem Ereignis gebildet, eine Aktion wird in eine Phase überführt durch Fortsetzung mit einer Phase oder durch eine Endlosschleife.

#### *Darstellung von Verhaltens- abstraktion*

Das in Kapitel 3.2 beschriebene Szenario konstruktiv eingesetzter Abstraktionsebenen kann mit den Konzepten des integrierten Verhaltensmodells aufgebaut werden. Der Übergang zwischen Abstraktionsebenen (die Realisierung atomarer Verhaltensbestandteile) wird genau durch die Verschränkung der Konzepte geleistet. Die durchgängig definierte Semantik der Beschreibungsmittel stellt sicher, daß Verknüpfungen von Elementen auf einer Abstraktionsebene eine (nicht explizit beschriebene, aber durch die Semantik gegebene) Entsprechung auf darunter liegenden Abstraktionsebenen haben. Eine Gesamtbeschreibung kann sich so redundanzfrei über mehreren Abstraktionsebenen erstrecken und zu einer konkreten Steuerung kompiliert werden.

Im skizzierten Beispiel aus Kapitel 3.2 kann es oberhalb der Prozedur wieder ein Phasenübergangssystem (z. B. für die Betriebsmodi einer Maschine) geben, das innerhalb einer Phase (eines Betriebsmodus) die Prozedur ausführt (neben anderen). Somit können die Beschreibungsebenen aus Abbildung 5.1 mehrfach in den Abstraktionsebenen einer Prozeßsteuerung auftreten.

Man beachte noch, daß sowohl Reaktionen auf Ereignisse als auch die Aktionen in sequentiellen Abläufen ohne ein Konzept wie die Zuweisung von Werten an Variablen auskommen. Die Modellierung von Prozessen erfolgt rein funktional (daher FSPL).

## 5.4 Infrastruktur einer höheren Programmiersprache

Um Mechanismen zur funktionalen Abstraktion, zur Datenabstraktion und Kapselung sowie zur generischen Abstraktion (siehe Kapitel 2.2.2) bereitzustellen, wird das im vorigen Abschnitt beschriebene Metamodell zur funktionalen, synchronen Verhaltensmodellierung zu einer höheren Programmiersprache (FSPL) umgesetzt und ausgebaut. Die verhaltensbezogenen Beschreibungsmittel, die die Sprache auf steuerungs- und regelungstechnische Anwendungen ausrichten, werden um allgemeine programmiertechnische Mechanismen ergänzt und syntaktisch repräsentiert.

Der Sprachentwurf bedient sich in weiten Teilen bei vorhandenen Konzepten höherer Programmiersprachen, vor allem aus dem Bereich der funktionalen Programmierung, die in neu zusammengestellter Form in Anwendung gebracht werden:

*Konstanten, Prozesse (Signale, Variablen), Ereignisse, Phasen, und Aktionen* werden als erstklassige Objekte behandelt und durch Literale und Konstruktoren beschrieben. *Bezeichner, Definitionen* und *Gültigkeitsbereiche* (einschließlich rekursiver Definitionen als generischem Mechanismus) stellen eine Basis-Infrastruktur zur Manipulation solcher Objekte bereit. Ein *statisches Typsystem* ermöglicht Prüfungen zur Compile-Zeit. *Funktionale Abstraktion* ( $\lambda$ -Abstraktion) erlaubt die Parametrierung von Objekten in allgemeiner Form; Funktionen sind erstklassige Objekte, können also als Parameter zu Funktionen höherer Ordnung auftreten. *Module* kapseln Typen und erstklassige Objekten zu Einheiten mit abstrakten Schnittstellen; auch Module sind erstklassige Objekte mit Modulschnittstellen als Typen. *Parametrische Polymorphie* unterstützt als ergänzender Mechanismus zur generischen Abstraktion die generische Programmierung. Die Sprache ist als funktionale Programmiersprache seiteneffektfrei.

So wird die sprachliche Implementierung der echtzeittypischen Beschreibungsmittel auf eine Art erreicht, die Abstraktion, Trennung von Belangen und generische Lösungen durchgängig ermöglicht und durch

ein Typsystem mit statischen Prüfungen absichert. Insbesondere wird der Entwurf von abstrakten Maschinen (siehe Abschnitt 3.3) als Module von der Programmiersprache unterstützt.

Der Entwurf der Programmiersprache FSPL (Functional Synchronous Programming Language) ist in Kapitel 7 und Anhang B beschrieben.

## 5.5 Umsetzung in der Arbeit

Die in den Abschnitten 5.2 und 5.3 vorgestellten Konzepte zur funktionalen, synchronen Modellierung von Prozessen werden im ersten Hauptteil der Arbeit (Kapitel 6) ausgearbeitet.

Im zweiten Hauptteil der Arbeit (Kapitel 7) wird die Implementierung der Konzepte in Form einer anwendungsnahen Programmiersprache FSPL (Functional Synchronous Programming Language) entsprechend den in Abschnitt 5.4 beschriebenen Konzepten dargestellt, indem der Sprachentwurf informell beschrieben wird. Die detaillierte, formale Definition der Syntax und der Semantik der Sprache findet sich in Anhang B.

Für das in Kapitel 3 beschriebene methodische Konzept der Steuerung mit konstruktiver Verhaltensabstraktion und der Modularisierung durch Abstrakte Maschinen wird in Kapitel 8 ein Entwurfsmuster „Abstrakte Maschine“ zur Anwendung mit der Sprache FSPL beschrieben.

Begleitend zur Arbeit wurden eine prototypische Implementierung der Sprache in Form eines Interpreters durchgeführt und ein größeres, von einem realen eingebetteten System abgeleitetes Anwendungsbeispiel in FSPL durchentwickelt, um die semantischen Konzepte, den Sprachentwurf und die Methodik zu validieren. Das Beispiel findet sich vollständig in Anhang C und wird ausschnittsweise in Kapitel 9 diskutiert.

# **6. Funktionale, synchrone Modellierung von Prozessen**

## **6.1 Übersicht**

Der Entwurf einer höheren Programmiersprache für ein bestimmtes Anwendungsgebiet (im Gegensatz zu einer allgemeinen Programmiersprache) setzt eine inhaltliche Erschließung dieses Anwendungsgebiets voraus. Dieser Aufgabe ist das vorliegende Kapitel gewidmet, das die wesentlichen semantischen Konzepte zur Beschreibung eingebetteter Echtzeitsysteme, die der Programmiersprache zugrundegelegt werden, aus pragmatischen Erwägungen heraus entwickelt („semantics first“).

Die entwickelten semantischen Modelle dienen der Beschreibung von zeitbehaftetem Verhalten wie es für Prozeßsteuerungen relevant ist. Ihr Entwurf stützt sich in Grundzügen auf die bekannten Verhaltensmodelle und Beschreibungsmittel, die in Kapitel 2.4 dargestellt wurden, führt jedoch die ausgewählten Konzepte zur Beschreibung quasi-kontinuierlichen, reaktiven und sequentiellen Verhaltens in neu abgestimmter Weise zu einem durchgängigen Beschreibungsansatz zusammen (siehe Kapitel 5.3). Grundlage bildet das synchrone Systemmodell (siehe Kapitel 5.2). Kennzeichnend für den Beschreibungsansatz ist die bidirektional-zyklische Einbettung (Verschränkung) der drei Verhaltensmodelle ineinander, so daß Schichtungen von Abstraktionsebenen

in beschriebenem Verhalten sehr flexibel ausgebildet werden können (Verhaltensabstraktion, siehe Kapitel 3.2). Ein zweites Kennzeichen ist der durchgängig funktionsorientierte Ansatz, der auf technisch motivierte, imperative Elemente wie das „Speichern“ von Werten oder das „Versenden“ von „Nachrichten“ verzichtet, so daß Systemverhalten auch in den reaktiven und sequentiellen Anteilen rein funktional durch Definition und Anwendung von Funktionen beschrieben wird.

Das Kapitel gliedert sich in drei Hauptteile für quasi-kontinuierliches (Abschnitt 6.4), reaktives (Abschnitt 6.5) und sequentielles Verhalten (Abschnitt 6.6). Die gemeinsame Basis dafür liefert der Zeit- und der Prozeßbegriff sowie ein Ereigniskonzept, die zuvor in Abschnitt 6.3 beschrieben werden. In Abschnitt 6.2 wird zunächst das Formalisierungskonzept für die mathematische Modellbildung beschrieben.

## 6.2 Formalisierungskonzept

Die Formalisierung der Modellbildung macht Gebrauch von mathematischen Grundlagen und Notationen, die im Anhang A eingeführt und zum Verständnis vorausgesetzt werden. Aus Gründen der Präzision und der Konsistenz ist es dabei unvermeidlich, daß die verwendete Notation an manchen Stellen von in der Elektrotechnik üblichen Schreibweisen (etwa für Funktionen) abweicht.

*Mengen und Operationen*

*Werte*

Die Erarbeitung inhaltlicher Begriffe und Zusammenhänge läuft auf die Definition spezieller Mengen und Operationen (Konstanten und Funktionen) im mathematischen Sinn hinaus, die der Klassifikation und Konstruktion spezieller mathematischer Objekte (Werte) zur Modellierung von Systemen dienen. Das dazu in dieser Arbeit primär verwendete Strukturierungsmittel ist an die Konzepte des *Abstrakten Datentyps* (ADT, siehe [Gut77]), der algebraischen Spezifikation [Wir90] und der mathematischen Logik [GS93, Kre91] angelehnt: Mengen und zugehörige Operationen werden zunächst in abstrakter Form spezifiziert, indem Bezeichner (Variablen) für sie eingeführt und die von ihnen geforderten Eigenschaften mit Hilfe logischer Formeln (oft in Gleichungsform) beschrieben werden.

*Spezifikation und Modell*

Jede konkrete Belegung dieser Variablen mit aus bekannten Objekten konstruierten Mengen und Operationen, die einer solchen *Spezifikation* genügt, bildet ein *Modell*. Die Existenz eines Modells weist die Erfüllbarkeit einer Spezifikation nach und ist Voraussetzung für ihre Sinnhaftigkeit. Zu jeder Spezifikation wird mindestens ein Modell

angegeben, so daß die Existenz der Basis für die Semantik der Sprache gesichert ist. Die Trennung der durch die Spezifikation geforderten Eigenschaften von zusätzlichen, die sich durch die konkrete Wahl des Modells ergeben, dient weniger der Vermeidung von Überspezifikation als der Verdeutlichung der semantischen Konzepte.

Zur Demonstration der Darstellungsweise wird im folgenden ein abstrakter Datentyp Boolean modelliert:

**Spezifikation 6.0 (Booleans)** für eine Menge Boolean und Objekte true, false, not, and und or:

$$\begin{aligned}
 \text{true} &\in \text{Boolean} \\
 \text{false} &\in \text{Boolean} \\
 \text{not} &\in \text{Boolean} \rightarrow \text{Boolean} \\
 \text{and} &\in \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \\
 \text{or} &\in \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \\
 \text{not true} &= \text{false} \\
 \text{not false} &= \text{true} \\
 \text{and } (x, \text{false}) &= \text{false} \\
 \text{and } (x, \text{true}) &= x \\
 \text{or } (x, \text{false}) &= x \\
 \text{or } (x, \text{true}) &= \text{true}
 \end{aligned}$$

□

Die in den Spezifikationsgleichungen auftretenden freien Variablen (Unbekannte; in diesem Fall  $x$ ) sind allquantifiziert zu verstehen. Ferner geben die in der ersten Hälfte der Spezifikation deklarierten Funktionalitäten der Operationen die Wertebereiche der Variablen vor. So ist z. B. die dritte Gleichung in Spezifikation 6.0 vervollständigt als

$$\forall x \in \text{Boolean} . \text{and } (x, \text{false}) = \text{false}$$

zu lesen.

Mit  $\text{Boolean} = \mathbb{B}$  kann ein triviales Modell angegeben werden, das die Spezifikation 6.0 erfüllt:

**Modell 6.0.1 (Booleans als Wahrheitswerte)** Die Menge Boolean und die Objekte true, false, not, and und or seien wie folgt definiert:

Boolean	$\stackrel{\text{def}}{=}$	$\mathbb{B}$
true	$\stackrel{\text{def}}{=}$	true
false	$\stackrel{\text{def}}{=}$	false
not	$\stackrel{\text{def}}{=}$	$\lambda x \in \mathbb{B} . \neg x$
and	$\stackrel{\text{def}}{=}$	$\lambda [x \in \mathbb{B}, y \in \mathbb{B}] . x \wedge y$
or	$\stackrel{\text{def}}{=}$	$\lambda [x \in \mathbb{B}, y \in \mathbb{B}] . x \vee y$

□

Ein zweites Modell genügt ebenfalls der Spezifikation 6.0:

**Modell 6.0.2 (Booleans als Zahlenwerte)** Die Menge Boolean und die Objekte true, false, not, and und or seien wie folgt definiert:

Boolean	$\stackrel{\text{def}}{=}$	$\{0, 1\}$
true	$\stackrel{\text{def}}{=}$	1
false	$\stackrel{\text{def}}{=}$	0
not	$\stackrel{\text{def}}{=}$	$\lambda x \in \{0, 1\} . 1 - x$
and	$\stackrel{\text{def}}{=}$	$\lambda [x \in \{0, 1\}, y \in \{0, 1\}] . x \cdot y$
or	$\stackrel{\text{def}}{=}$	$\lambda [x \in \{0, 1\}, y \in \{0, 1\}] . \min\{1, x + y\}$

□

Die Zählung der Modelle bezieht sich auf die zugehörigen Spezifikationen. Die Modelle 6.0.1 und 6.0.2 implementieren also Spezifikation 6.0. Jede Angabe eines Modells zu einer Spezifikation (zugeordnet durch die Zählweise) impliziert im folgenden jeweils die Behauptung, daß durch die angegebenen Definitionen die zugehörige Spezifikation erfüllt wird. Auf formale Beweise kann in den meisten Fällen verzichtet werden, weil die spezifizierten Eigenschaften wie im obigen Beispiel leicht nachzuvollziehen sind.

Die Notation innerhalb der Modellbildung verwendet serifenlose Schrift für die neu eingeführten Bezeichner (wie Boolean oder not im Beispiel) im Gegensatz zur Serifenschrift bei vordefinierten Operatoren aus der Mathematik (wie min).



## 6.3 Zeit und Verhalten

### 6.3.1 Zeit

Um Beschreibungsmittel für zeitbezogenes Verhalten definieren zu können, ist zuerst der Zeitbegriff zu präzisieren. Da *Echtzeitsysteme* im Blickpunkt stehen, ist die Definition einer *logischen* Zeit als einer geordneten Menge von abstrakten Zeitpunkten [Lam78], zu denen etwa Zustandsänderungen eintreten, nicht ausreichend. Die *Zeitdauer* von Vorgängen ist ein wesentlicher funktionaler Bestandteil dieser Systeme und bedarf einer Quantifizierung, die im Sinn der „echten“, physikalischen Zeit interpretiert werden kann. Wenn sich *Echtzeitsysteme* im Einsatz befinden, heißen sie deshalb so, weil die durch sie realisierten Abbildungen von Vorgängen (aus ihrer Umgebung) genauso lange dauern wie die Vorgänge selbst (vgl. [FAZ04]). Das „genauso lange“ ist nicht zuletzt eine Frage des Ermessens, die unmittelbar mit der Genauigkeit der Zeitangabe und der Zeiterfassung zusammenhängt.

Die physikalische Zeit wird als ein Kontinuum verstanden, das durch die Menge  $\mathbb{R}$  der reellen Zahlen modelliert werden kann. Dabei sind Zeitpunkte  $t \in \mathbb{R}$  relativ zu einem Nullzeitpunkt  $0 \in \mathbb{R}$  zu definieren. Dieser ist geeignet zu wählen, etwa als den Zeitpunkt der Inbetriebnahme eines Systems. Es reicht daher im allgemeinen aus, das Zeitkontinuum in der Betrachtung auf  $\mathbb{R}_+$  zu beschränken. Zur Quantifizierung von Zeitintervallen kann auf die Zeiteinheit des internationalen metrischen Systems [Tay95] zurückgegriffen werden. Ein  $\Delta t = 1 \in \mathbb{R}$  entspricht dann dem Zeitintervall einer Sekunde.

*physikalische  
Zeit*

Der Aufwand, der z. B. für Atomuhren betrieben wird, macht deutlich, daß die Zeit – wie andere physikalische Phänomene auch – praktischen Beschränkungen der Meßbarkeit unterliegt und nur mit endlicher Genauigkeit aufgelöst werden kann. Es ist mithin das Wesen von Uhren, eine Diskretisierung der Zeit vorzunehmen. Die Grundlage dafür bildet ein als fest angenommener Takt, der abgezählt wird, seien es nun die Schwingungen eines Cäsiumatoms, eines Schwingquarzes oder einer Unruh.

*Diskretisierung*

Die benötigte Präzision der Zeitauflösung hängt vom Verwendungszweck ab, d. h. von den Geschwindigkeiten bzw. Frequenzen der zu messenden oder zu steuernden Vorgänge<sup>1</sup>. Umgekehrt schränkt die

<sup>1</sup>Das *Abtasttheorem* gibt dazu ein hinreichendes Maß für den Zeittakt an: Ein aus der Überlagerung von Sinusschwingungen unterschiedlicher Frequenzen zusammen-

verfügbare Zeitauflösung die Bandbreite des Frequenzspektrums der Vorgänge ein, die zuverlässig zeitgerecht verarbeitet werden können.

#### *Zeitbasis eingebetteter Systeme*

Eingebetteten Systemen liegt auf unterschiedliche Weise eine Zeitbasis zugrunde. Falls vorhanden besitzt eine *Echtzeituhr* eine bestimmte Zeitauflösung für ablesbare Zeitwerte. Bei einem üblichen Schwingquarz mit 32,768 kHz ist eine theoretische Zeitauflösung von ca. 30  $\mu$ s möglich. *Hardware-Timer* nehmen in der Regel den Prozessortakt als Zeitbasis, der beispielsweise bei 100 MHz eine minimale Zeitauflösung von 10 ns bietet. Betriebssysteme stellen häufig einen eigenen, vergrößerten Systemtakt bereit (sogenannte Betriebssystem-Uhr), der die Zeitbasis für *Software-Timer* bietet und typischerweise im Millisekundenbereich liegt [ETF03]. Schließlich gibt die *worst-case execution time* (WCET, siehe Kapitel 2.3.1.1) eine anwendungsspezifische Maximalfrequenz für Systemeingaben vor, sofern harte Echtzeitbedingungen vorliegen.

#### *Synchronität*

Insgesamt arbeiten eingebettete Echtzeitsysteme also stets mit Diskretisierungen der Zeit, also einem Zeitraster, auf das alle Zeitangaben zu „runden“ sind. Bei Echtzeitanforderungen kann dann z. B. exakte Zeiteinhaltung innerhalb der Genauigkeit dieses Rasters gefordert werden. Gerade die Betrachtung von Systemantwortzeiten bei der Ausführung von Echtzeitsoftware, legt weiterhin nahe, daß die Zeitauflösung für die Definition des Systemverhaltens grundsätzlich als eine anwendungsspezifische Entwurfsentscheidung betrachtet werden kann. Dies bietet umgekehrt den Vorteil, daß von Berechnungszeiten (also Systemantwortzeiten) abstrahiert werden kann, wenn diese prinzipiell innerhalb der Zeitungenauigkeit verschwinden. Ein System antwortet dann auf eine Eingabe innerhalb des gleichen Zeitintervalls, d. h. logisch zum gleichen Zeitpunkt wie die Eingabe; der Wert eines Ausgangssignals hängt potentiell vom Wert eines Eingangssignals *zum gleichen Zeitpunkt* ab (Synchronitätsprinzip, siehe Kapitel 2.3.3.1).

#### *Zeit als abstrakter Datentyp*

Mit  $\mathbb{R}_+$  als Modell der echten Zeit einerseits und dem Charakteristikum einer prinzipiell diskreten und entwurfsabhängig aufgelösten Zeitbasis für Echtzeitsysteme andererseits erweist es sich als zweckmäßig, die Zeit als eine abstrakte Zeitmenge Time mit zugehörigen Operationen zu modellieren:

---

gesetztes Signal kann bei einer Abtastung mit mindestens dem Doppelten der maximalen im Signal enthaltenen Frequenz aus den Abtastwerten vollständig rekonstruiert werden (siehe z. B. [Rup93, KJ02]).

**Spezifikation 6.1 (Zeit)** für eine Menge *Time* und Objekte *time*, *step*, *tick*, *toTime*:

$$\begin{aligned}
 \text{time} &\in \text{Time} \rightarrow \mathbb{R}_+ \\
 \text{step} &\in \mathbb{R}_+ \setminus \{0\} \\
 \text{tick} &\in \text{Time} \rightarrow \mathbb{N} \\
 \text{toTime} &\in \mathbb{N} \rightarrow \text{Time} \\
 \text{time } t &= \text{step} \cdot (\text{tick } t) \\
 \text{toTime } (\text{tick } t) &= t \\
 \text{tick } (\text{toTime } n) &= n
 \end{aligned}
 \quad \square$$

*time* bildet jeden Zeitpunkt der Zeitmenge *Time*, der *Systemzeit*, auf das durch die nichtnegativen reellen Zahlen modellierte Zeitkontinuum der *echten Zeit* ab. *step* bezeichnet die Schrittweite (Taktperiode), durch die die Zeitauflösung der diskreten Systemzeit bestimmt wird. *tick* liefert die Zeit gemessen in Zeitschritten. Der Operator *toTime* erlaubt umgekehrt die Angabe von Zeitwerten als Vielfache der Schrittweite.

Mit  $\text{Time} = \mathbb{N}$  und einer Konstante für die Taktperiode kann ein einfaches Modell angegeben werden, das die Spezifikation 6.1 erfüllt:

**Modell 6.1.1 (Diskrete Zeit mit Taktperiode  $\Delta t$ )** Sei  $\Delta t \in \mathbb{R}_+ \setminus \{0\}$  beliebig, aber fest. Die Menge *Time* und die Objekte *time*, *step*, *tick*, *toTime* seien wie folgt definiert:

$$\begin{aligned}
 \text{Time} &\stackrel{\text{def}}{=} \mathbb{N} \\
 \text{step} &\stackrel{\text{def}}{=} \Delta t \\
 \text{time} &\stackrel{\text{def}}{=} \lambda n \in \mathbb{N}. n \cdot \text{step} \\
 \text{tick} &\stackrel{\text{def}}{=} \text{id}_{\mathbb{N}} \\
 \text{toTime} &\stackrel{\text{def}}{=} \text{id}_{\mathbb{N}}
 \end{aligned}
 \quad \square$$

Wie in der Theorie zeitdiskreter Signale und Systeme üblich (siehe z. B. [KJ02]), dienen hier also die natürlichen Zahlen als Zeitmodell. Zeitdiskrete Prozesse sind dann Funktionen von  $\mathbb{N}$  (siehe Abschnitt 6.3.2), die nach Anhang A.1.5 auch als Folgen aufgefaßt werden können. Der

hier Time genannte abstrakte Datentyp, der dem Rest des Prozeßmodells zugrundeliegt, verbirgt jedoch bewußt solche Zusammenhänge. So wird es möglich, die zeitdiskrete *Abtastung* von Signalen mit möglicherweise unterschiedlichen Schrittweiten als eigenständiges Konzept der Signalverarbeitung von der diskreten Zeitbasis für Signale bzw. Prozesse zu unterscheiden (vgl. Abschnitt 6.4.4).

Der Erhöhung der Lesbarkeit für Konstanten aus Time dient folgende alternative Notation:

**Definition 6.1** Sei  $n \in \mathbb{N}$ .

$$n_{\text{Time}} \stackrel{\text{def}}{=} \text{toTime } n \quad \square$$

Die folgende Definition wird dort Verwendung finden, wo nichtleere Zeitintervalle betrachtet werden:

**Definition 6.2**

$$\text{Time}_+ \stackrel{\text{def}}{=} \text{Time} \setminus \{0_{\text{Time}}\} \quad \square$$

### Operatoren

Dem vereinfachten Rechnen mit Zeitwerten dienen folgende Operatoren:

**Definition 6.3 (Rechenoperationen für Zeitwerte)** Seien  $t, t' \in \text{Time}$  und  $n \in \mathbb{N}$ .

$$\begin{aligned} t +_{\text{Time}} t' &\stackrel{\text{def}}{=} \text{toTime } (\text{tick } t + \text{tick } t') \\ t -_{\text{Time}} t' &\stackrel{\text{def}}{=} \text{toTime } (\max\{0, \text{tick } t - \text{tick } t'\}) \\ n *_{\text{Time}} t &\stackrel{\text{def}}{=} \text{toTime } (n \cdot (\text{tick } t)) \\ t /_{\text{Time}} n &\stackrel{\text{def}}{=} \text{toTime } ((\text{tick } t) \text{ div } n) \end{aligned} \quad \square$$

In Anlehnung an mathematische Konventionen sollen  $*_{\text{Time}}$  und  $/_{\text{Time}}$  stärker assoziieren als  $+_{\text{Time}}$  und  $-_{\text{Time}}$  („Punkt vor Strich“). Weiterhin können noch Vergleichsoperatoren auf Time definiert werden:

**Definition 6.4 (Zeitvergleiche)** Seien  $t, t' \in \text{Time}$ .

$$t \underset{\text{Time}}{=} t' \stackrel{\text{def}}{=} \text{time } t = \text{time } t'$$

$$t \underset{\text{Time}}{\neq} t' \stackrel{\text{def}}{=} \text{time } t \neq \text{time } t'$$

$$t \underset{\text{Time}}{<} t' \stackrel{\text{def}}{=} \text{time } t < \text{time } t'$$

$$t \underset{\text{Time}}{\leq} t' \stackrel{\text{def}}{=} \text{time } t \leq \text{time } t'$$

$$t \underset{\text{Time}}{\geq} t' \stackrel{\text{def}}{=} \text{time } t \geq \text{time } t'$$

$$t \underset{\text{Time}}{>} t' \stackrel{\text{def}}{=} \text{time } t > \text{time } t'$$

□

In Anlehnung an übliche mathematische Notation soll bei Bedarf eine abgekürzte Schreibweise für Doppelvergleiche zulässig sein. So steht also etwa  $t_0 \underset{\text{Time}}{<} t \underset{\text{Time}}{<} t_1$  für  $(t_0 \underset{\text{Time}}{<} t) \wedge (t \underset{\text{Time}}{<} t_1)$ . Schließlich können noch

Minimum- und Maximumbildung übertragen werden:

**Definition 6.5** Sei  $M \subseteq \text{Time}$  mit  $M \neq \emptyset$ .

$$\min_{\text{Time}} M \stackrel{\text{def}}{=} \text{toTime } (\min\{\text{tick } t \mid t \in M\})$$

$$\max_{\text{Time}} M \stackrel{\text{def}}{=} \text{toTime } (\max\{\text{tick } t \mid t \in M\})$$

□

### 6.3.2 Prozeß

Ein Prozeß ist ein *Vorgang*, etwa das Verhalten eines Objekts oder der Verlauf einer Sache, allgemein die Veränderung eines Zustands im Verlauf der Zeit. Mathematisch gesehen ist ein Prozeß eine *Funktion der Zeit* und wird beschrieben durch die Zeitmenge *Time*, eine *Zustandsmenge*  $X$  und eine Funktion

$$x \in \text{Time} \rightarrow X, \tag{6.1}$$

wobei der Prozeß als nichtterminierend angenommen wird.  $X$  heißt auch *Zustandsraum*. Während die Zeit linear, eindimensional ist, kann ein Zustandsraum auch mehrdimensional oder von beliebig tiefer Struktur sein.

*Größe,  
Variable*

Ein wichtiges Beispiel für einen Prozeß ist der Verlauf einer einzelnen zeitveränderlichen physikalischen *Größe* (*Variablen*), etwa einer Temperatur oder einer elektrischen Spannung. Der Zustand des Prozesses ist der augenblickliche Wert der Variablen. Auch im allgemeinen Fall beliebig komplexer Zustandsräume ist es je nach Bedeutung entweder natürlicher, von einem *Prozeß* und von *Zuständen* oder aber von einer *Variablen* und von *Werten* zu sprechen. *Prozesse* und *Variablen* (oder *Größen*) sollen daher als synonym behandelt werden. Der Begriff des Prozesses („Vorgangs“) verbindet sich mit einer zeitbetonten, der der Variablen („Veränderlichen“) oder Größe mit einer zustands- oder wertbetonten Sicht auf einen zweidimensionalen Sachverhalt.

*Signal*

Der in der Systemtheorie (siehe z. B. [Unb97]) neben dem Begriff der Variablen häufig verwendete Begriff des *Signals* stellt ein weiteres Synonym dar, auch wenn bei der mathematischen Definition häufig die Einschränkung gemacht wird, daß es sich bei  $X$  um einen skalaren Wertebereich handelt. Wie schon in bezug auf elektronische Signale bemerkt, liegt eine semantische Auszeichnung in dem *vermittelnden* oder *repräsentativen* Charakter von Signalen als Informationsträger. Es wird daher bei der Betrachtung von *indirekten* Größen häufig angemessen sein, von Signalen zu sprechen. So gibt z. B. das Signal einer Lichtschranke indirekt Aufschluß über eine Bewegung.

*Prozeß als  
abstrakter  
Datentyp*

Das Prozeßmodell nach (6.1) besagt, daß ein Prozeß durch eine Funktion der Zeit beschrieben wird. Der Zustand eines Prozesses  $x$  zum Zeitpunkt  $t$  ist der Funktionswert  $x\ t$ . Diese wesentliche Eigenschaft wird zum Ausdruck gebracht, wenn man auch Prozesse als abstrakten Datentyp spezifiziert, den das Modell implementiert:

**Spezifikation 6.2 (Prozeß)** für eine Menge  $\text{Process}_X$  und eine Funktion  $\text{state}_X$  zu jeder Zustandsmenge  $X$ :

$$\text{state}_X \in \text{Process}_X \rightarrow \text{Time} \rightarrow X \quad \square$$

Ein Ausdruck

$$\text{state}_X\ x\ t$$

*Zustandsmenge*

steht dabei für den Zustand (in  $X$ ) des Prozesses  $x$  zum Zeitpunkt  $t$ . Unter einer *Zustandsmenge* soll eine beliebige Menge (Semantik) verstanden werden, die einen Zustandsraum modelliert (Pragmatik). Die Mengen  $\text{Process}_X$  heißen auch *Prozeßmengen*. Es folgt das Prozeßmodell gemäß (6.1):

*Prozeßmenge*

**Modell 6.2.1 (Prozeß als Funktion der Zeit)** Zu jeder Zustandsmenge  $X$  seien  $\text{Process}_X$  und  $\text{state}_X$  wie folgt definiert:

$$\begin{aligned}\text{Process}_X &\stackrel{\text{def}}{=} \text{Time} \rightarrow X \\ \text{state}_X &\stackrel{\text{def}}{=} \text{id}_{\text{Time} \rightarrow X}\end{aligned}\quad \square$$

Zwei Prozesse sollen dann als gleich betrachtet werden, wenn sie zu jedem Zeitpunkt den gleichen Zustand aufweisen. Neben Gleichheit ist auch abschnittsweise Gleichheit eine interessante Eigenschaft, wobei Abschnitte über Zeitpunkte definiert werden:

*Vergleichsoperatoren*

**Definition 6.6 (Vergleich von Prozessen)** Seien  $X$  eine Zustandsmenge und  $x, x' \in \text{Process}_X$ .

$$\begin{aligned}x &\stackrel{\text{def}}{=} x' \stackrel{\text{def}}{=} \forall t \in \text{Time}. \text{state}_X x t = \text{state}_X x' t \\ x &\neq x' \stackrel{\text{def}}{=} \neg(x \stackrel{\text{def}}{=} x') \\ x &\stackrel{t_0}{|} x' \stackrel{\text{def}}{=} \forall t \in \text{Time}. (t \stackrel{\text{def}}{=} t_0 \Rightarrow \text{state}_X x t = \text{state}_X x' t) \\ x &\stackrel{t_0}{=} x' \stackrel{\text{def}}{=} \forall t \in \text{Time}. (t < t_0 \Rightarrow \text{state}_X x t = \text{state}_X x' t) \\ x &\stackrel{t_0 t_1}{|} x' \stackrel{\text{def}}{=} \forall t \in \text{Time}. (t_0 \stackrel{\text{def}}{=} t < t_1 \Rightarrow \text{state}_X x t = \text{state}_X x' t)\end{aligned}\quad \square$$

Als *Konstruktoren* sollen Operatoren zur Konstruktion eines komplexeren Werts aus einfacheren bezeichnet werden. Im Gegensatz dazu entnehmen *Selektoren* einfachere Werte aus komplexeren. So ist z. B. in der Sprache der Mathematik der Tupel-Operator  $[]$  ein Konstruktor, während die Indizierung eines Tupels (d. i. die Anwendung des Tupels als Funktion auf einen Indexwert) eine Selektion darstellt.

*Konstruktoren und Selektoren*

Es mag auffallen, daß mit  $\text{state}_X$  lediglich ein Selektor definiert wurde, ein Konstruktor für Prozeßobjekte jedoch fehlt. Die konstruktive Beschreibung von Prozeßverhalten basiert auf der Verknüpfung von primitiven Prozessen mittels geeigneter Operatoren. Beides liefern die in den nachfolgenden Abschnitten eingeführten Beschreibungsmittel.

### 6.3.3 Ereignis

Während ein Prozeß einen Verlauf aufweist, der jedem Zeitpunkt einen Zustand zuordnet, sind Ereignisse dadurch charakterisiert, daß sie zu einzelnen, diskreten Zeitpunkten stattfinden. Ereignisse sind häufig relativ zu dem Zustandsverlauf eines Prozesses definiert, z. B. als das Eintreten einer Bedingung, das Erreichen eines Schwellwerts oder eine Wertänderung innerhalb eines diskreten Wertebereichs (Zustandsübergang). Wesentlich ist ihr *punktuel*er Charakter.

Begrifflich liegt eine gewisse Schwierigkeit vor, da man mit „Ereignis“ sowohl einen einzelnen Zeitpunkt auszeichnen kann als auch einen Umstand, der sich wiederholen kann. Das Ticken einer Uhr könnte im ersten Sinn als eine *Folge* von Ereignissen, im zweiten Sinn als *ein* periodisch wiederkehrendes Ereignis aufgefaßt werden<sup>2</sup>. Je nach Kontext wird der Begriff im folgenden in der einen oder der anderen Bedeutung verwendet. Die Modellbildung selbst folgt der zweiten Bedeutung.

#### *Ereignismodell*

In diesem Sinn ist ein Ereignis durch eine endliche oder unendliche Folge von Zeitpunkten charakterisiert: Zu diesen Zeitpunkten findet das Ereignis statt. Für die Folge ist zu fordern, daß sie streng monoton steigend ist, damit sie die Vorkommnisse zeitlich ordnet und ein Ereignis nicht mehrfach zum gleichen Zeitpunkt auftreten kann. Weiterhin soll ein Ereignis in jedem endlichen Zeitintervall nur endlich oft vorkommen (was sogenannte Zeno-Folgen<sup>3</sup> ausschließt). Infolge des diskreten Zeitmodells mit konstanten Zeitschritten (siehe Spezifikation 6.1) ist diese letzte Eigenschaft allerdings in der strengen Monotonie enthalten, die zugleich eine Maximalfrequenz für jedes Ereignis impliziert (maximal ein Stattfinden pro Zeitschritt). Ereignisse finden synchron zum Zeittakt statt.

#### *Ereignis als abstrakter Datentyp*

Ereignisse sind ein Hilfskonzept zur Beschreibung von Prozessen. Unter praktischen Gesichtspunkten besteht daher in der Detektion ihres Stattfindens das primäre Anliegen bei der Analyse von Ereignissen. Die Folge der Zeitpunkte ergibt sich daraus, ist aber für sich von geringem Interesse. Der nachfolgend angegebene ADT Event wird deshalb im wesentlichen durch eine Operation happened bestimmt, die das nächste

<sup>2</sup>Die Singularform des Beispiels („das Ticken“) weist dabei im Deutschen den zweiten Sinn auf, während man im Englischen dem ersten Sinn gefolgt wäre (Pluralform „ticks“).

<sup>3</sup>Folgen unendlich vieler (Zeit-)Intervalle mit endlicher Summe; nach dem Paradoxon des altgriechischen Philosophen Zeno (Zenon) von Elea (ca. 490 v. Chr. – 430 v. Chr.) über Achill und die Schildkröte [UBH04]



Auftreten des Ereignisses nach einem gegebenen Zeitpunkt beschreibt.  $\text{happened } e \ t_0$  ist ein Prädikat über der Zeit, das wahr wird (und bleibt), sobald das Ereignis  $e$  zum nächsten Mal nach dem Zeitpunkt  $t_0$  stattfindet:

**Spezifikation 6.3 (Ereignis)** für eine Menge *Event* und eine Funktion *happened*:

$$\text{happened} \in \text{Event} \rightarrow \text{Time} \rightarrow \text{Time} \rightarrow \mathbb{B}$$

$$t \underset{\text{Time}}{\leq} t_0 \Rightarrow \neg(\text{happened } e \ t_0 \ t)$$

$$(\text{happened } e \ t_0 \ t) \wedge (t \underset{\text{Time}}{<} t') \Rightarrow \text{happened } e \ t_0 \ t'$$

□

Die beiden Bedingungen in Spezifikation 6.3 machen  $\text{happened } e \ t_0$  zu einer monotonen Funktion (wenn man auf  $\mathbb{B}$  die Ordnung  $\text{false} < \text{true}$  definiert). Sie kann entweder konstant  $\text{false}$  liefern oder aber zu einem bestimmten Zeitpunkt auf  $\text{true}$  wechseln. Dieser Zeitpunkt wird, falls vorhanden, als nächstes Stattfinden des Ereignisses  $e$  nach dem Zeitpunkt  $t_0$  gedeutet.

Man beachte, daß durch  $\text{happened}$  kein Ereignis bestimmt werden kann, daß zum Nullzeitpunkt stattfindet. Anders ausgedrückt: Der Zeitpunkt 0 kann kein Ereignis tragen. Dies verträgt sich damit, daß ein Ereignis in seiner Bedeutung mit einer *Veränderung* oder dem *Eintreten* eines Zustands assoziiert wird. Der Startzeitpunkt 0 trägt genau den allerersten Zustand. „Veränderung“ oder „Eintreten“ setzt aber einen vorherigen Zustand voraus.

Da infolge des diskreten Zeitmodells ein Ereignis zu jedem Zeitpunkt außer dem Nullzeitpunkt stattfinden kann und im Extremfall zu all diesen Zeitpunkten stattfindet, mag es nahe liegen, ein Ereignis wie ein Spezialfall eines booleschen Prozesses zu modellieren:

**Modell 6.3.1 (Ereignis, absolut)**

$$\text{Event} \stackrel{\text{def}}{=} \text{Time} \rightarrow \mathbb{B}$$

$$\begin{aligned} \text{happened} \stackrel{\text{def}}{=} & \lambda e \in \text{Event} . \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} . \\ & \exists t' \in \text{Time} . (t_0 \underset{\text{Time}}{<} t' \underset{\text{Time}}{\leq} t) \wedge e \ t' \end{aligned}$$

□

Hierbei ist ein Ereignis als boolesche Funktion für einen Zeitpunkt, der nicht der Nullzeitpunkt ist, genau dann wahr, wenn das Ereignis zu diesem Zeitpunkt stattfindet. Es ist leicht nachzuweisen, daß Modell 6.3.1 die Bedingungen aus Spezifikation 6.3 erfüllt. Alternativ kann ein Ereignis als eine Funktion mit zwei Zeitparametern modelliert werden, die genau diese Bedingungen erfüllt:

**Modell 6.3.2 (Ereignis, relativ)**

$$\begin{aligned} \text{Event} &\stackrel{\text{def}}{=} \{e \in \text{Time} \rightarrow \text{Time} \rightarrow \mathbb{B} \mid \forall t_0, t, t' \in \text{Time}. \\ &\quad (t \underset{\text{Time}}{\leq} t_0 \Rightarrow \neg(e \ t_0 \ t)) \wedge \\ &\quad ((e \ t_0 \ t) \wedge (t \underset{\text{Time}}{<} t') \Rightarrow e \ t_0 \ t')\} \\ \text{happened} &\stackrel{\text{def}}{=} \text{id}_{\text{Event}} \end{aligned}$$

□

Die beiden Modelle sind nicht äquivalent. Wie im Folgenden noch festgestellt wird, gibt es Ereignisse, die sich mit Modell 6.3.2, nicht aber mit Modell 6.3.1 darstellen lassen. Dies ist insofern wichtig als in Spezifikation 6.3 wie schon in Spezifikation 6.2 lediglich ein Selektor, nicht aber ein Konstruktor spezifiziert wurde. Auch Ereignisse werden aus primitiven Ereignissen und Verknüpfungen konstruiert, die erst im folgenden noch definiert werden.

Der wesentliche Unterschied zwischen den Modellen besteht darin, daß beim ersten Modell ein Ereignis die Zeitpunkte seines Stattfindens absolut bestimmt, während sie beim zweiten Modell relativ zu dem Zeitpunkt  $t_0$  definierbar sind, der sich aus dem Kontext der Ereignisdetektion ergibt.

Vergleichsoperatoren

Auch für Ereignisse können Vergleichsoperatoren definiert werden. Zusätzlich zur Gleichheit läßt sich auch eine partielle Ordnungsrelation definieren:

**Definition 6.7 (Vergleich von Ereignissen)** Seien  $e, e' \in \text{Event}$ .

$$\begin{aligned} e &\underset{\text{Event}}{=} e' &\stackrel{\text{def}}{=} &\forall t_0, t \in \text{Time}. \text{happened } e \ t_0 \ t = \text{happened } e' \ t_0 \ t \\ e &\underset{\text{Event}}{\neq} e' &\stackrel{\text{def}}{=} &\neg(e \underset{\text{Event}}{=} e') \\ e &\underset{\text{Event}}{\leq} e' &\stackrel{\text{def}}{=} &\forall t_0, t \in \text{Time}. \text{happened } e' \ t_0 \ t \Rightarrow \text{happened } e \ t_0 \ t \\ e &\underset{\text{Event}}{\geq} e' &\stackrel{\text{def}}{=} &e' \underset{\text{Event}}{\leq} e \end{aligned}$$

□

$\leq_{\text{Event}}$  kann als „findet stets früher oder gleichzeitig statt“ gelesen werden. Es gilt:

$$e =_{\text{Event}} e' \Rightarrow e \leq_{\text{Event}} e' \quad (6.2)$$

$$e =_{\text{Event}} e' \Rightarrow e \geq_{\text{Event}} e'. \quad (6.3)$$

Zur Konstruktion von Ereignissen wird nun im folgenden ein erweiterbarer Satz von *Ereignisgeneratoren* eingeführt. Die Definition dieser Funktionen, die in den abstrakten Datentyp Event hinein abbilden, ist abhängig vom gewählten Modell für Event.

*Ereignis-  
generatoren*

Ein triviales Ereignis ist das Ereignis, das niemals stattfindet:

**Spezifikation 6.4 (Unmögliches Ereignis)** für ein Objekt never :

$$\text{never} \in \text{Event}$$

$$\text{happened never } t_0 \ t = \text{false} \quad \square$$

Spezifikation 6.4 kann als Erweiterung zu Spezifikation 6.3 aufgefaßt werden. Zur Implementierung können beide Ereignismodelle entsprechend erweitert werden. Zunächst die Erweiterung von Modell 6.3.1, anschließend die von Modell 6.3.2:

**Modell 6.4.1 (Unmögliches Ereignis zu Modell 6.3.1)**

$$\text{never} \stackrel{\text{def}}{=} \lambda t \in \text{Time} . \text{false} \quad \square$$

**Modell 6.4.2 (Unmögliches Ereignis zu Modell 6.3.2)**

$$\text{never} \stackrel{\text{def}}{=} \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} . \text{false} \quad \square$$

Als kritisch erweist sich ein Generator für Timeout-Ereignisse:

**Spezifikation 6.5 (Timeout)** für eine Funktion after:

$$\text{after} \in \text{Time}_+ \rightarrow \text{Event}$$

$$\text{happened (after } \Delta t) t_0 \ t = t \underset{\text{Time}}{\geq} (t_0 + \Delta t)$$

□

Timeout-Ereignisse sind relativ: Die Beantwortung der Frage, wann eine angegebene Zeitspanne verstrichen ist, hängt vom Zeitpunkt der Fragestellung ab.  $\text{after } \Delta t$  tritt genau um  $\Delta t$  nach dem Zeitpunkt ein, zu dem die Beobachtung des Ereignisses beginnt.

Das Verhalten von  $\text{after}$  ist nachweisbar unverträglich mit Modell 6.3.1 für Ereignisse. Denn gibt es eine Funktion  $e \in \text{Time} \rightarrow \mathbb{B}$ , so daß  $\text{happened } e \ t_0 \ t = t \underset{\text{Time}}{\geq} (t_0 + \Delta t)$  für alle  $t_0, t \in \text{Time}$  und ein beliebiges  $\Delta t \in \text{Time}_+$ , so gilt:

$$\text{happened } e \ 0_{\text{Time}} \ \Delta t = \text{true} \quad (6.4)$$

$$\text{happened } e \ 0_{\text{Time}} \ (\Delta t - 1) = \text{false} \quad (6.5)$$

$$\text{happened } e \ 1_{\text{Time}} \ \Delta t = \text{false}. \quad (6.6)$$

Daraus folgt nach Definition von  $\text{happened}$ :

$$\exists t' \in \text{Time}. (0_{\text{Time}} < \underset{\text{Time}}{t'} \leq \underset{\text{Time}}{\Delta t}) \wedge e \ t' \quad (6.7)$$

$$\forall t' \in \text{Time}. (0_{\text{Time}} < \underset{\text{Time}}{t'} \leq \underset{\text{Time}}{(\Delta t - 1)}) \Rightarrow \neg(e \ t') \quad (6.8)$$

$$\forall t' \in \text{Time}. (1_{\text{Time}} < \underset{\text{Time}}{t'} \leq \underset{\text{Time}}{\Delta t}) \Rightarrow \neg(e \ t'). \quad (6.9)$$

Aus (6.7) und (6.8) folgt  $e \ \Delta t$  im Widerspruch zu (6.9).

Modell 6.3.1 ist also für eine Erweiterung zur Implementierung von Spezifikation 6.5 nicht mächtig genug. Dagegen stellt sich Modell 6.3.2 als geeignet heraus:

### Modell 6.5.1 (Timeout)

$$\text{after} \stackrel{\text{def}}{=} \lambda \Delta t \in \text{Time}_+. \lambda t_0 \in \text{Time}. \lambda t \in \text{Time}. t \underset{\text{Time}}{\geq} (\underset{\text{Time}}{t_0} + \underset{\text{Time}}{\Delta t}) \quad \square$$

Bei Modell 6.3.2 für Event bleibend, werden nun noch weitere Ereignisgeneratoren definiert. Das bereits als Beispiel erwähnte Ticken einer Uhr kann als periodisches Ereignis modelliert werden. Uhren-Ereignisse (auch als *Takt*-Ereignisse zu bezeichnen) stellen eine dezidierte Vergrößerung der Zeit über die diskrete Zeitbasis hinaus dar und können zyklische Vorgänge wie z. B. periodische Messungen (d. h. die Abtastung eines Signals) oder die periodische Neuberechnung von Stellwerten steuern.

Ein Takt ist definiert durch eine Periodendauer  $\Delta t$  und eine Phasenverschiebung  $t_1$ :

**Spezifikation 6.6 (Takt)** für eine Funktion clock:

$$\text{clock} \in \text{Time} \rightarrow \text{Time}_+ \rightarrow \text{Event}$$

$$\text{happened}(\text{clock } t_1 \Delta t) t_0 t = \begin{cases} t \geq t_1 & \text{falls } t_1 > t_0 \\ \text{happened}(\text{clock } (t_1 + \Delta t) \Delta t) t_0 t & \text{sonst} \end{cases}$$

□

Mit  $t_1 = 0_{\text{Time}}$  findet das erste Ereignis bei  $t = \Delta t$  statt. Die Spezifikation kann unmittelbar für eine rekursive Definition übernommen werden, weil die Rekursion stets nach endlich vielen Schritten abbricht, da wegen  $\Delta t \in \text{Time}_+$  das wiederholte Aufaddieren von  $\Delta t$  über jedes  $t_0$  hinausführt:

**Modell 6.6.1 (Takt)**

$$\text{clock} \stackrel{\text{def}}{=} \lambda t_1 \in \text{Time}. \lambda \Delta t \in \text{Time}_+. \lambda t_0 \in \text{Time}. \lambda t \in \text{Time}. \begin{cases} t \geq t_1 & \text{falls } t_1 > t_0 \\ (\text{clock } (t_1 + \Delta t) \Delta t) t_0 t & \text{sonst} \end{cases}$$

□

Der nächste Ereignisgenerator detektiert das Eintreten einer Bedingung, worunter ein Zustandswechsel eines booleschen Prozesses von false nach true zu verstehen ist: eine sogenannte *steigende Flanke* eines booleschen Signals.

**Spezifikation 6.7 (Trigger)** für eine Funktion trigger:

$$\text{trigger} \in \text{Process}_{\mathbb{B}} \rightarrow \text{Event}$$

$$\text{happened}(\text{trigger } x) t_0 t = \begin{cases} \text{false} & \text{falls } t \leq t_0 \\ \text{true} & \text{sonst, falls} \\ \text{happened}(\text{trigger } x) t_0 (t - 1) & \\ (\text{state}_{\mathbb{B}} x t) \wedge \neg(\text{state}_{\mathbb{B}} x (t - 1)) & \text{sonst} \end{cases}$$

□

Auch hier kann aus der Spezifikation unmittelbar eine rekursive Definition abgeleitet werden. Die Rekursion terminiert, weil  $t$  mit jedem Schritt um  $1_{\text{Time}}$  näher an  $t_0$  heranrückt, solange  $t > t_0$ :

### Modell 6.7.1 (Trigger)

$$\text{trigger} \stackrel{\text{def}}{=} \lambda c \in \text{Process}_{\mathbb{B}} . \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} . \begin{cases} \text{false} & \text{falls } t \leq t_0 \\ \text{true} & \text{sonst, falls} \\ & (\text{trigger } c) t_0 (t - 1) \\ (\text{state}_{\mathbb{B}} c t) \wedge \neg(\text{state}_{\mathbb{B}} c (t - 1)) & \text{sonst} \end{cases}$$

□

### Verknüpfung von Ereignissen

*Selektives Warten* auf mehrere Ereignisse setzt die ODER-Verknüpfbarkeit von Ereignissen voraus. Ein ODER-Ereignis hat stattgefunden, sobald eines der Ereignisse stattgefunden hat. Analog hat ein UND-Ereignis stattgefunden, sobald alle Ereignisse mindestens einmal stattgefunden haben. UND-Ereignisse können zur *Synchronisation* verwendet werden.

**Spezifikation 6.8 (ODER-Ereignis, UND-Ereignis)** für Funktionen  $\bigvee_{\text{Event}}$  und  $\bigwedge_{\text{Event}}$ , für die Infix-Notation angenommen wird:

$$\begin{aligned} \bigvee_{\text{Event}}, \bigwedge_{\text{Event}} &\in \text{Event} \times \text{Event} \rightarrow \text{Event} \\ \text{happened } (e_0 \bigvee_{\text{Event}} e_1) t_0 t &= (\text{happened } e_0 t_0 t) \vee (\text{happened } e_1 t_0 t) \\ \text{happened } (e_0 \bigwedge_{\text{Event}} e_1) t_0 t &= (\text{happened } e_0 t_0 t) \wedge (\text{happened } e_1 t_0 t) \end{aligned}$$

□

Wie leicht nachzuweisen ist, gelten die Assoziativgesetze

$$(e_0 \bigvee_{\text{Event}} e_1) \bigvee_{\text{Event}} e_2 = e_0 \bigvee_{\text{Event}} (e_1 \bigvee_{\text{Event}} e_2) \quad (6.10)$$

$$(e_0 \bigwedge_{\text{Event}} e_1) \bigwedge_{\text{Event}} e_2 = e_0 \bigwedge_{\text{Event}} (e_1 \bigwedge_{\text{Event}} e_2) \quad (6.11)$$

und die Distributivgesetze

$$\underset{\text{Event}}{e_0} \wedge (\underset{\text{Event}}{e_1} \vee \underset{\text{Event}}{e_2}) = (\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1}) \vee (\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_2}) \quad (6.12)$$

$$(\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1}) \wedge \underset{\text{Event}}{e_2} = (\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_2}) \vee (\underset{\text{Event}}{e_1} \wedge \underset{\text{Event}}{e_2}) \quad (6.13)$$

$$\underset{\text{Event}}{e_0} \vee (\underset{\text{Event}}{e_1} \wedge \underset{\text{Event}}{e_2}) = (\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1}) \wedge (\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_2}) \quad (6.14)$$

$$(\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1}) \vee \underset{\text{Event}}{e_2} = (\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_2}) \wedge (\underset{\text{Event}}{e_1} \vee \underset{\text{Event}}{e_2}), \quad (6.15)$$

so daß ferner in der Schreibweise zur Eliminierung von Klammern Assoziativität angenommen werden kann, also

$$(\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1}) \vee \underset{\text{Event}}{e_2} = \underset{\text{Event}}{e_0} \vee (\underset{\text{Event}}{e_1} \vee \underset{\text{Event}}{e_2}) = \underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1} \vee \underset{\text{Event}}{e_2}$$

und

$$(\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1}) \wedge \underset{\text{Event}}{e_2} = \underset{\text{Event}}{e_0} \wedge (\underset{\text{Event}}{e_1} \wedge \underset{\text{Event}}{e_2}) = \underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1} \wedge \underset{\text{Event}}{e_2},$$

aber bei gemischtem Auftreten von  $\underset{\text{Event}}{\vee}$  und  $\underset{\text{Event}}{\wedge}$  stets Klammerung gefordert werden muß.

Ferner gilt:

$$\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1} \underset{\text{Event}}{\geq} \underset{\text{Event}}{e_0} \quad (6.16)$$

$$\underset{\text{Event}}{e_0} \wedge \underset{\text{Event}}{e_1} \underset{\text{Event}}{\geq} \underset{\text{Event}}{e_1} \quad (6.17)$$

$$\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1} \underset{\text{Event}}{\leq} \underset{\text{Event}}{e_0} \quad (6.18)$$

$$\underset{\text{Event}}{e_0} \vee \underset{\text{Event}}{e_1} \underset{\text{Event}}{\leq} \underset{\text{Event}}{e_1}. \quad (6.19)$$

Es folgt noch ein Modell zur Spezifikation 6.8:

#### Modell 6.8.1 (ODER-Ereignis, UND-Ereignis)

$$\underset{\text{Event}}{\vee} \stackrel{\text{def}}{=} \lambda [e_0 \in \text{Event}, e_1 \in \text{Event}]. \lambda t_0 \in \text{Time}. \lambda t \in \text{Time}. \\ (e_0 \ t_0 \ t) \vee (e_1 \ t_0 \ t)$$

$$\underset{\text{Event}}{\wedge} \stackrel{\text{def}}{=} \lambda [e_0 \in \text{Event}, e_1 \in \text{Event}]. \lambda t_0 \in \text{Time}. \lambda t \in \text{Time}. \\ (e_0 \ t_0 \ t) \wedge (e_1 \ t_0 \ t)$$

□

Die Assoziativität der binären ODER- und UND-Verknüpfungen rechtfertigt ihre Verallgemeinerung:

**Definition 6.8** Sei  $I = \{j_0, \dots, j_n\}$  eine endliche, nichtleere Menge (Indexmenge). Zu jedem  $i \in I$  sei ein  $e_i \in \text{Event}$ .

$$\begin{aligned} \bigvee_{\text{Event}} \emptyset &\stackrel{\text{def}}{=} \text{never} \\ \bigvee_{\text{Event}} (\lambda i \in I. e_i) &\stackrel{\text{def}}{=} e_{j_0} \bigvee_{\text{Event}} \dots \bigvee_{\text{Event}} e_{j_n} \\ \bigwedge_{\text{Event}} (\lambda i \in I. e_i) &\stackrel{\text{def}}{=} e_{j_0} \bigwedge_{\text{Event}} \dots \bigwedge_{\text{Event}} e_{j_n} \end{aligned}$$

□

Während die UND-Verknüpfung zweier Ereignisse das Eintreten der beiden Ereignisse in beliebiger Reihenfolge (unter Einschluß der Gleichzeitigkeit) beschreibt, werden bei der Ereignissequenz zwei Ereignisse nacheinander in bestimmter Reihenfolge erwartet:

**Spezifikation 6.9 (Ereignissequenz)** für eine Funktion  $+$ <sub>Event</sub>, für die Infix-Notation angenommen wird:

$$\begin{aligned} +_{\text{Event}} &\in \text{Event} \times \text{Event} \rightarrow \text{Event} \\ \text{happened } (e_0 +_{\text{Event}} e_1) t_0 t &= \begin{cases} \text{false} & \text{falls } \neg(\text{happened } e_0 t_0 t) \\ \text{happened } e_1 t_1 t & \text{sonst} \end{cases} \\ \text{wobei } t_1 &\stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } e_0 t_0 t\} \end{aligned}$$

□

### Modell 6.9.1

$$\begin{aligned} +_{\text{Event}} &\stackrel{\text{def}}{=} \lambda [e_0 \in \text{Event}, e_1 \in \text{Event}] . \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} . \\ &\quad \begin{cases} \text{false} & \text{falls } \neg(e_0 t_0 t) \\ e_1 t_1 t & \text{sonst} \end{cases} \\ \text{wobei } t_1 &\stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid e_0 t_0 t\} \end{aligned}$$

□



$e_0 +_{\text{Event}} e_1$  bezeichnet das Ereignis  $e_1$ , das auf das Ereignis  $e_0$  folgt bzw. die Sequenz der Ereignisse  $e_0$  und  $e_1$ . Es gilt:

$$e_0 +_{\text{Event}} e_1 \geq_{\text{Event}} e_0. \quad (6.20)$$

Als Spezialfall tritt die  $n$ -fache Wiederholung eines Ereignisses ( $n \geq 1$ ) auf:

**Definition 6.9 (Ereigniswiederholung)** Seien  $n \in \mathbb{N} \setminus \{0\}$  und  $e \in \text{Event}$ .

$$n *_{\text{Event}} e \stackrel{\text{def}}{=} \underbrace{e +_{\text{Event}} \dots +_{\text{Event}} e}_n$$

□

Aus (6.20) folgt per vollständiger Induktion (für  $n, m \in \mathbb{N} \setminus \{0\}$  und  $e \in \text{Event}$ ):

$$n \leq m \Rightarrow (n *_{\text{Event}} e) \leq_{\text{Event}} (m *_{\text{Event}} e). \quad (6.21)$$

Ein logisches Gegenstück zur Ereignissequenz bildet der Ereignisausschluß. Hierbei wird erwartet, daß ein Ereignis eintritt, bevor ein anderes Ereignis eintritt:

**Spezifikation 6.10 (Ereignisausschluß)** für eine Funktion  $-_{\text{Event}}$ , für die Infix-Notation angenommen wird:

$$\begin{aligned} -_{\text{Event}} &\in \text{Event} \times \text{Event} \rightarrow \text{Event} \\ \text{happened } (e_0 -_{\text{Event}} e_1) t_0 t &= (\text{happened } e_0 t_0 t) \wedge \neg(\text{happened } e_1 t_0 t) \end{aligned}$$

□

### Modell 6.10.1

$$\begin{aligned} -_{\text{Event}} &\stackrel{\text{def}}{=} \lambda [e_0 \in \text{Event}, e_1 \in \text{Event}]. \lambda t_0 \in \text{Time}. \lambda t \in \text{Time}. \\ &\quad (e_0 t_0 t) \wedge \neg(e_1 t_0 t) \end{aligned}$$

□

Es gilt:

$$e_0 \underset{\text{Event}}{-} e_1 \underset{\text{Event}}{\geq} e_0. \quad (6.22)$$

Mit  $\underset{\text{Event}}{-}$  und  $\text{after}$  kann eine Timeout-Variante konstruiert werden, die logisch als NICHT-Ereignis (ergänzend zum UND- und ODER-Ereignis) aufgefaßt werden kann:

$$(\text{after } \Delta t) \underset{\text{Event}}{-} e \quad (6.23)$$

bezeichnet das Ereignis, das eintritt, wenn das Ereignis  $e$  nicht binnen einer Frist von  $\Delta t$  eintritt.

*Bewachte* Ereignisse sind Ereignisse, deren Eintreten nur dann verwendet wird, wenn zum Zeitpunkt des Eintretens zugleich eine Bedingung wahr, d. h. der Wert eines booleschen Signals `true` ist.

**Spezifikation 6.11 (Bewachtes Ereignis)** für eine Funktion *guard*:

$$\text{guard} \in \text{Process}_{\mathbb{B}} \rightarrow \text{Event} \rightarrow \text{Event}$$

$$\text{happened} (\text{guard } c \ e) \ t_0 \ t = \begin{cases} \text{false} & \text{falls } \neg(\text{happened } e \ t_0 \ t) \\ \text{true} & \text{sonst, falls } \text{state}_{\mathbb{B}} \ c \ t_1 = \text{true} \\ \text{happened} (\text{guard } c \ e) \ t_1 \ t & \text{sonst} \end{cases}$$

$$\text{wobei } t_1 \stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } e \ t_0 \ t\}$$

□

Auch hier leitet sich mit rekursiver Definition unmittelbar ein Modell aus der Spezifikation ab. Die Rekursion terminiert, da  $t_0 < t_1 \leq t$   $\text{Time} \quad \text{Time}$

und deshalb  $t \underset{\text{Time}}{-} t_0$  gegen  $0_{\text{Time}}$  strebt (wird  $t_0 \underset{\text{Time}}{=} t$ , so ergibt sich  $\neg(\text{happened } e \ t_0 \ t)$  und damit `false`).

**Modell 6.11.1 (Bewachtes Ereignis)**

$$\text{guard} \stackrel{\text{def}}{=} \lambda c \in \text{Process}_{\mathbb{B}} . \lambda e \in \text{Event} . \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} .$$

$$\begin{cases} \text{false} & \text{falls } \neg(e \ t_0 \ t) \\ \text{true} & \text{sonst, falls } \text{state}_{\mathbb{B}} \ c \ t_1 = \text{true} \\ (\text{guard } c \ e) \ t_1 \ t & \text{sonst} \end{cases}$$

$$\text{wobei } t_1 \stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid e \ t_0 \ t\}$$

□

guard liefert sich ein interessantes Wechselspiel mit after: Ein bewachtes after-Ereignis tritt dann ein, wenn das erste Vielfache der Wartezeit verstrichen ist, zu dessen Ende die Bedingung wahr ist.

**Beispiel 6.1** Sei  $c \in \text{Process}_{\mathbb{B}}$  ein boolescher Prozeß, für den

$$\text{state}_{\mathbb{B}} c t = \begin{cases} \text{false} & \text{für } t < 5_{\text{Time}} \\ \text{true} & \text{sonst} \end{cases}$$

gilt, und sei ferner ein Ereignis  $e$  definiert durch

$$e \stackrel{\text{def}}{=} \text{guard } c \text{ (after } 3_{\text{Time}}).$$

Dann gilt:

$$\text{happened } e \text{ } 0_{\text{Time}} t = t \geq 6_{\text{Time}}. \quad \square$$

Allgemein gilt für  $e \in \text{Event}$  und  $c \in \text{Process}_{\mathbb{B}}$ :

$$\text{guard } c e \geq_{\text{Event}} e. \quad (6.24)$$

Im Gegensatz zum bewachten Ereignis entscheidet das *bedingte* Ereignis zwischen zwei Ereignissen nach dem Wert eines booleschen Signals zu Beginn der Beobachtung. Dies setzt, ähnlich wie beim Konstruktor after für Timeout-Ereignisse, ein relatives Ereignismodell wie das gewählte Modell 6.3.2 voraus:

**Spezifikation 6.12 (Bedingtes Ereignis)** für eine Funktion  $\text{ifElseEvent}$ :

$$\text{ifElseEvent} \in \text{Process}_{\mathbb{B}} \rightarrow \text{Event} \rightarrow \text{Event} \rightarrow \text{Event}$$

$$\begin{aligned} \text{happened } (\text{ifElseEvent } c e_0 e_1) t_0 = \\ \begin{cases} \text{happened } e_0 t_0 & \text{falls } \text{state}_{\mathbb{B}} c t_0 = \text{true} \\ \text{happened } e_1 t_0 & \text{sonst} \end{cases} \quad \square \end{aligned}$$

**Modell 6.12.1 (Bedingtes Ereignis)**

$$\text{ifElseEvent} \stackrel{\text{def}}{=} \lambda c \in \text{Process}_{\mathbb{B}} . \lambda e_0 \in \text{Event} . \lambda e_1 \in \text{Event} . \lambda t_0 \in \text{Time} .$$

$$\begin{cases} e_0 \ t_0 & \text{falls } \text{state}_{\mathbb{B}} \ c \ t_0 = \text{true} \\ e_1 \ t_0 & \text{sonst} \end{cases}$$

□

Abschließend soll noch ein weiterer Typ von Timeout-Ereignissen definiert werden, nämlich das Überwachen eines Ereignisses auf seine regelmäßige Wiederholung im Sinn einer „Watchdog“-Funktion. Das Timeout-Ereignis trifft dann ein, wenn ein gegebenes Ereignis nicht innerhalb einer definierten Frist seit seinem letzten Stattfinden bzw. seit Beginn der Beobachtung auftritt.

**Spezifikation 6.13 (Watchdog-Timeout)**

$$\text{watchdog} \in \text{Event} \rightarrow \text{Time}_+ \rightarrow \text{Event}$$

$$\text{happened} (\text{watchdog } e \ \Delta t) \ t_0 \ t = \begin{cases} \text{false} & \text{falls } t <_{\text{Time}} (t_0 +_{\text{Time}} \Delta t) \\ \text{true} & \text{sonst, falls} \\ & \neg(\text{happened } e \ t_0 \ (t_0 +_{\text{Time}} \Delta t)) \\ \text{happened } (\text{watchdog } e \ \Delta t) \ t_1 \ t & \text{sonst} \end{cases}$$

$$\text{wobei } t_1 \stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } e \ t_0 \ t\}$$

□

Auch hier wieder leitet sich mit rekursiver Definition unmittelbar ein Modell aus der Spezifikation ab. Die Rekursion terminiert aus den gleichen Gründen wie bei Modell 6.11.1.

**Modell 6.13.1 (Watchdog-Timeout)**

$$\text{watchdog} \stackrel{\text{def}}{=} \lambda e \in \text{Event} . \lambda \Delta t \in \text{Time}_+ . \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} .$$

$$\begin{cases} \text{false} & \text{falls } t <_{\text{Time}} (t_0 +_{\text{Time}} \Delta t) \\ \text{true} & \text{sonst, falls } \neg(e \ t_0 \ (t_0 +_{\text{Time}} \Delta t)) \\ (\text{watchdog } e \ \Delta t) \ t_1 \ t & \text{sonst} \end{cases}$$

$$\text{wobei } t_1 \stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid e \ t_0 \ t\}$$

□

## 6.4 Signalflüsse

### 6.4.1 Übertragungssysteme

Die Betrachtung von *Signalen* als Wertverläufe (im Gegensatz zu dem Synonym *Prozeß* als Zustandsentwicklung) führt zu einem Denkansatz des *Rechnens* mit Signalen, der sich für Anwendungen wie Regelungsalgorithmen eignet. Die Verarbeitung von Signalen – die Berechnung von Signalen aus Signalen – wird ausgedrückt in Form von funktionalen Abhängigkeiten, die auf einen Satz vordefinierter Operationen zurückgeführt werden. Zur Beschreibung quasi-kontinuierlichen Verhaltens kommen dazu z. B. Differenzengleichungen zum Tragen (vgl. Kapitel 2.4.1), die mit einer signalwertigen Operation „Zeitverschiebung“ dargestellt werden (siehe Abschnitt 6.4.4).

*Rechnen mit Signalen*

Die vordefinierten Operationen sind typischerweise signalwertige Konstanten oder Funktionen

$$f \in \text{Process}_{X_0} \rightarrow \cdots \rightarrow \text{Process}_{X_{n-1}} \rightarrow \text{Process}_{X_n}$$

oder auch

$$f \in (\text{Process}_{X_0} \times \cdots \times \text{Process}_{X_{n-1}}) \rightarrow \text{Process}_{X_n}$$

für gewisse Zustandsmengen  $X_0, \dots, X_n$  mit  $n \geq 0$ . Auf ihrer Basis können weitere, „höherwertige“ Funktionen definiert werden, die Signale auf Signale abbilden.

Durch eine (einstellige) signalwertige Funktion

*Übertragungssysteme*

$$f \in \text{Process}_X \rightarrow \text{Process}_Y$$

wird ein *Übertragungssystem* (siehe z. B. [Rup93]) repräsentiert, das ein *Eingangssignal* in ein *Ausgangssignal* „überträgt“. Ein Eingangssignal trägt zu jedem Zeitpunkt einen Eingangs- oder *Eingabewert*, ein Ausgangssignal einen Ausgangs- oder *Ausgabewert*. Die Eigenschaften dieses einfachen Falles lassen sich für mehrstellige signalwertige Funktionen (mehrere Ein- und/oder Ausgangssignale) verallgemeinern.

Die Berechnung signalwertiger Funktionen *in Echtzeit* sowie das in-

*Kausalität*

tuitive Verständnis von Prozeßsteuerungen legt eine praktische Beschränkung auf: Ein Ausgabewert kann nicht von einem *zukünftigen* Eingabewert abhängen. Systeme müssen also *kausal* sein, d. h. die funktionalen Abhängigkeiten dürfen einen Zustand nur auf zeitgleiche und vergangene Zustände (die in irgendeiner Form „gespeichert“ werden müssen) zurückführen, nicht aber auf zukünftige. Ein Übertragungssystem, beschrieben durch eine Funktion

$$f \in \text{Process}_X \rightarrow \text{Process}_Y,$$

heißt *kausal*, wenn für jedes  $t_0 \in \text{Time}$  aus

$$\forall t \underset{\text{Time}}{\leq} t_0 . \text{state}_X x t = \text{state}_X x' t$$

folgt:

$$\text{state}_Y (f x) t_0 = \text{state}_Y (f x') t_0,$$

d. h.  $\text{state}_Y (f x) t_0$  (die Ausgabe zum Zeitpunkt  $t_0$ ) hängt von keinem  $\text{state}_X x t$  (der Eingabe zum Zeitpunkt  $t$ ) für  $t > t_0$  ab. Bei der Definition der primitiven Operationen zur Verarbeitung von Signalen muß sichergestellt werden, daß signalwertige Funktionen per Konstruktion diese Eigenschaft besitzen.

*statische und  
dynamische  
Systeme*

Weiterhin heißt das System *statisch*, wenn für alle  $t, t' \in \text{Time}$  aus

$$\text{state}_X x t = \text{state}_X x t'$$

folgt:

$$\text{state}_Y (f x) t = \text{state}_Y (f x) t'.$$

*algebraische  
Funktionen*

Ansonsten heißt es *dynamisch*. Bei statischen Systemen hängen die Ausgaben also nur von den zeitgleichen Eingaben ab, während dynamische Systeme die „Historie“ der Eingaben und/oder den aktuellen Zeitpunkt berücksichtigen. Signalwertige Funktionen, deren Ausgaben nur von zeitgleichen Eingaben abhängen, heißen auch *algebraisch*.

Primitive Operationen zur Definition signalwertiger Konstanten (primitive Signale) und Funktionen für statische und dynamische Systeme werden in den nachfolgenden Abschnitten 6.4.2 bis 6.4.4 eingeführt.

*Signalflüsse*

Signalwertige Ausdrücke, also Ausdrücke, die aus primitiven Signalen, signalwertigen Variablen (Bezeichnern) und Anwendungen von signalwertigen Funktionen bestehen, werden in der Regelungstechnik oft in

graphischer Form notiert. Dazu kann ein Satz von (rekursiven) Definitionen für Signale – also ein Gleichungssystem – als ein gerichteter Hypergraph dargestellt werden, wobei jede Hyperkante maximal eine Quelle und beliebig viele Senken haben kann (*Signalflußgraph*). Freie (d. h. erst im Kontext gebundene) signalwertige Variablen sind Hyperkanten ohne Quelle; die definierten signalwertigen Variablen sind Hyperkanten mit Quelle (oder fallen mit existierenden Hyperkanten zusammen, wenn die Variable direkt als Wert einer anderen Variable definiert wird). Signalwertige Konstanten sind Hyperknoten mit Eingangsgrad 0 und Ausgangsgrad 1; signalwertige Funktionen sind Hyperknoten mit Eingangsgrad  $\geq 1$  und Ausgangsgrad  $\geq 1$ , wobei ein Eingangsgrad  $> 1$  wahlweise mit Tupeln von Signalen oder mit Currying (vgl. Anhang A.1.9) und ein Ausgangsgrad  $> 1$  mit Tupeln von Signalen korrespondieren kann (vgl. Abschnitt 6.4.3). Ein Pfad in einem Signalflußgraphen wird als *Signalfluß* bezeichnet.

**Beispiel 6.2** Folgender Satz von rekursiven Signaldefinitionen läßt sich wie in Abbildung 6.1 darstellen:

$$\begin{aligned} u &= f \ x \ v \\ v &= g \ u \\ y &= h \ u \end{aligned}$$

□

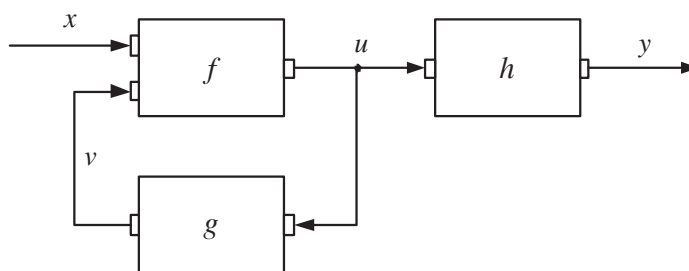


Abbildung 6.1: Signalflußgraph

Man beachte, daß bei der Beschreibung dynamischer Systeme durch rekursive Definition ein Signal von sich selbst abhängig gemacht werden kann (Rückkopplung). Der Signalflußgraph enthält dann Zyklen. Damit das System wohldefiniert ist, kann jedoch kein einzelner Signalwert rekursiv von sich selbst abhängen. Ist dies irrtümlicherweise der Fall, spricht man von einer *algebraischen Schleife*. Auf zyklischen Signalflüssen muß stets eine Operation liegen, die eine Zeitverzögerung herstellt, so daß ein Signalwert von einem *früheren* Wert des gleichen Signals abhängt.

*algebraische  
Schleifen*

### 6.4.2 Primitive Signale

Ein Signal liefert zu jedem Zeitpunkt einen Wert aus einem bestimmten Wertebereich. Zu den einfachsten Funktionen (hier nämlich der Zeit) gehören zweifelsohne die konstanten Funktionen und die Identitätsfunktion.

Ein *konstantes Signal* (eine *Konstante*) besitzt nur einen einzigen Wert, den es konstant beibehält. Er kann einer beliebigen Zustandsmenge entnommen sein. Die folgende polymorphe, d. h. für beliebige (Zustands-) Mengen  $X$  definierte Funktion macht einen beliebigen Wert aus  $X$  zu einer Konstanten:

**Spezifikation 6.14 (Konstanter Prozeß)** für eine Funktion  $\text{const}_X$  zu jeder Zustandsmenge  $X$ :

$$\begin{aligned}\text{const}_X &\in X \rightarrow \text{Process}_X \\ \text{state}_X (\text{const}_X k) t &= k\end{aligned}$$

□

Spezifikation 6.14 kann als Erweiterung von Spezifikation 6.2 aufgefaßt werden. Die entsprechende Erweiterung zu Modell 6.2.1 ergibt sich unmittelbar aus der Spezifikation:

#### Modell 6.14.1

$$\text{const}_X \stackrel{\text{def}}{=} \lambda k \in X. \lambda t \in \text{Time}. k$$

□

Prozeßkonstruktoren wie  $\text{const}_X$  werden nachfolgend stets, ohne dies jeweils zu erwähnen, als Ergänzung zu Spezifikation 6.2 und Modell 6.2.1 angegeben.

*Zeit  
als Signal*

Nachdem die Zeitmenge als Datentyp eingeführt wurde, kann auch die Zeit selbst als Signal aufgefaßt werden:

**Spezifikation 6.15 (Zeit als Signal)** für ein Objekt  $\text{timeProcess}$ :

$$\begin{aligned}\text{timeProcess} &\in \text{Process}_{\text{Time}} \\ \text{state}_{\text{Time}} \text{ timeProcess } t &= t\end{aligned}$$

□

Das Zeitsignal wird modelliert durch die Identitätsfunktion:



**Modell 6.15.1**

$$\text{timeProcess} \stackrel{\text{def}}{=} \text{id}_{\text{Time}} \quad \square$$

Mit den beiden Konstruktoren  $\text{const}_X$  und  $\text{timeProcess}$  können erste signalwertige Objekte dargestellt werden. Es wird sich herausstellen, daß der in Abschnitt 6.4.3 eingeführte Konstruktor  $\text{apply}_{X \rightarrow Y}$ , angewendet auf  $\text{timeProcess}$ , geeignet ist, jedes beliebige Signal zu konstruieren, sofern sein Verlauf als Funktion der Zeit explizit bekannt ist. Der Konstantenkonstruktor  $\text{const}_X$  wird dadurch redundant, entspricht aber unmittelbar dem wichtigsten Fall.

**6.4.3 Statische Systeme**

Das Wesen statischer Systeme besteht darin, Eingabewerte zeitgleich auf Ausgabewerte abzubilden, unabhängig vom Zeitpunkt und von Werten zu anderen Zeitpunkten. Ein statisches System mit der Systemfunktion

$$f \in \text{Process}_X \rightarrow \text{Process}_Y$$

ist also im wesentlichen durch eine Funktion

$$f' \in X \rightarrow Y$$

bestimmt, für die gilt:

$$\text{state}_Y (f \ x) \ t = f' (\text{state}_X \ x \ t)$$

(für alle  $x \in \text{Process}_X$  und alle  $t \in \text{Time}$ ).  $f$  wendet sozusagen  $f'$  „punktweise“ auf ein Eingangssignal  $x$  an. Man sagt auch, die Funktion  $f'$  wird auf die Signalebene „gehoben“ (Funktionslifting). Die Verallgemeinerung dieses Vorgangs liefert der Konstruktor  $\text{apply}$ : *Lifting*

**Spezifikation 6.16 (Funktionslifting)** für eine Funktion  $\text{apply}_{X \rightarrow Y}$  für jede Zustandsmenge  $X$  und jede Zustandsmenge  $Y$ :

$$\begin{aligned} \text{apply}_{X \rightarrow Y} &\in (X \rightarrow Y) \rightarrow \text{Process}_X \rightarrow \text{Process}_Y \\ \text{state}_Y (\text{apply}_{X \rightarrow Y} \ f \ x) \ t &\stackrel{\text{def}}{=} f (\text{state}_X \ x \ t) \end{aligned} \quad \square$$

**Modell 6.16.1**

$$\text{apply}_{X \rightarrow Y} \stackrel{\text{def}}{=} \lambda f \in X \rightarrow Y. \lambda x \in \text{Process}_X. \lambda t \in \text{Time}. f \ (x \ t) \quad \square$$

Für die oben angeführten Funktionen  $f$  und  $f'$  gilt damit:

$$f = \text{apply}_{X \rightarrow Y} f'.$$

(Wie leicht nachzuweisen ist, ist  $f$  als System kausal.)

So läßt sich beispielsweise ein Verstärker beschreiben, der sein Eingangssignal mit einem konstanten Faktor multipliziert:

$$\begin{aligned} \text{gain}_{\mathbb{R}} &\in \mathbb{R} \rightarrow \text{Process}_{\mathbb{R}} \rightarrow \text{Process}_{\mathbb{R}} \\ \text{gain}_{\mathbb{R}} &\stackrel{\text{def}}{=} \lambda k \in \mathbb{R}. \text{apply}_{\mathbb{R} \rightarrow \mathbb{R}} (\lambda x \in \mathbb{R}. k \cdot x). \end{aligned} \quad (6.25)$$

Für einen konkreten Faktor  $k$  ist dann  $\text{gain}_{\mathbb{R}} k$  eine signalwertige Funktion, die reellwertige Signale auf reellwertige Signale abbildet.

Eine Besonderheit für die Anwendung von  $\text{apply}$  bietet der elementare Prozeß  $\text{timeProcess}$  (siehe Spezifikation 6.15): Hat man eine Funktion  $s \in \text{Time} \rightarrow X$ , so wird durch

$$\text{apply}_{\text{Time} \rightarrow X} s \text{ timeProcess}$$

ein Signal erzeugt, so daß genau

$$\text{state}_X (\text{apply}_{\text{Time} \rightarrow X} s \text{ timeProcess}) = s.$$

Der Wertverlauf des Signals wird also der Funktion  $s$  entnommen. So entsteht z. B. ein Sinus-Signal durch:

$$\sin \in \text{Process}_{\mathbb{R}} \quad (6.26)$$

$$\sin \stackrel{\text{def}}{=} \text{apply}_{\text{Time} \rightarrow \mathbb{R}} (\sin \circ \text{time}) \text{ timeProcess}. \quad (6.27)$$

### Mehrstelligkeit

Funktionen werden in dieser Arbeit stets als einstellig betrachtet; *mehrstellige* Funktionen werden mit Hilfe von Tupeln oder über Curry-ing dargestellt (siehe Anhang A.1.9). Es reicht aus, das Anheben von Funktionen auf die Signalebene für einen der beiden Mechanismen zu unterstützen, denn auf jeder Ebene können die Funktionsvarianten einfach ineinander überführt werden. Tupel eignen sich dabei nicht nur für mehrstellige Eingaben, sondern auch für mehrstellige Ausgaben (Übertragungssysteme mit mehreren Eingängen und mehreren Ausgängen). An die Stelle von Tupeln können auch Records und Arrays (d. i. endliche Folgen über einer Menge, vgl. Kapitel 7.4.5.2) treten.

Um nun mehrstellige Funktionen der Wertebereichsebene auf mehrstellige Funktionen der Signalebene anzuheben, sind Transformationen zwischen Tupeln (Records, Arrays) von Signalen und Signalen von Tupeln (Records, Arrays) erforderlich. Polymorph für alle Produkte von Zustandsmengen leisten dies die folgenden Operationen:

*Produkt-  
transformation*

**Spezifikation 6.17 (Produkttransformation)** für Funktionen  $\text{zip}_{\prod(\lambda i \in I. X_i)}$  und  $\text{unzip}_{\prod(\lambda i \in I. X_i)}$  zu jeder Mengenabbildung  $\lambda i \in I. X_i$ , die jedem Index  $i$  aus einer Indexmenge  $I$  eine Zustandsmenge  $X_i$  zuordnet:

$$\begin{aligned} \text{zip}_{\prod(\lambda i \in I. X_i)} &\in \prod(\lambda i \in I. \text{Process}_{X_i}) \rightarrow \text{Process}_{\prod(\lambda i \in I. X_i)} \\ \text{unzip}_{\prod(\lambda i \in I. X_i)} &\in \text{Process}_{\prod(\lambda i \in I. X_i)} \rightarrow \prod(\lambda i \in I. \text{Process}_{X_i}) \\ (\text{state}_{\prod(\lambda i \in I. X_i)} (\text{zip}_{\prod(\lambda i \in I. X_i)} x) t) i &= \text{state}_{X_i} (x i) t \\ \text{state}_{X_i} ((\text{unzip}_{\prod(\lambda i \in I. X_i)} x) i) t &= (\text{state}_{\prod(\lambda i \in I. X_i)} x t) i \end{aligned}$$

□

**Modell 6.17.1** Für alle Mengenabbildungen  $\lambda i \in I. X_i$  seien die Funktionen  $\text{zip}_{\prod(\lambda i \in I. X_i)}$  und  $\text{unzip}_{\prod(\lambda i \in I. X_i)}$  wie folgt definiert:

$$\begin{aligned} \text{zip}_{\prod(\lambda i \in I. X_i)} &\stackrel{\text{def}}{=} \lambda x \in \prod(\lambda i \in I. \text{Process}_{X_i}). \lambda t \in \text{Time}. \\ &\quad \lambda i \in I. \text{state}_{X_i} (x i) t \\ \text{unzip}_{\prod(\lambda i \in I. X_i)} &\stackrel{\text{def}}{=} \lambda x \in \text{Process}_{\prod(\lambda i \in I. X_i)}. \lambda i \in I. \\ &\quad \lambda t \in \text{Time}. (\text{state}_{\prod(\lambda i \in I. X_i)} x t) i \end{aligned}$$

□

Auch diese Operationen sind kausalitätserhaltend. Mit ihnen lassen sich mehrstellige Operationen wie z. B. arithmetische Verknüpfungen realisieren:

$$\begin{aligned} \text{add}_{\mathbb{R}} &\in \text{Process}_{\mathbb{R}} \times \text{Process}_{\mathbb{R}} \rightarrow \text{Process}_{\mathbb{R}} \\ \text{add}_{\mathbb{R}} &\stackrel{\text{def}}{=} (\text{apply}_{\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}} (\lambda [x \in \mathbb{R}, y \in \mathbb{R}]. x + y)) \circ \text{zip}_{\mathbb{R} \times \mathbb{R}}. \end{aligned} \tag{6.28}$$

#### 6.4.4 Dynamische Systeme

Konstanten und statische Systeme (wie in den vorangehenden Abschnitten definiert) induzieren in einem System von Prozessen bzw.

Signalen selbst keine Zustandsänderungen. Zustandsänderungen sind allein auf Prozesse mit zeitveränderlichem Zustand (also nicht-konstante Signale) und auf dynamische Systeme zurückzuführen. Als bisher einziges nicht-konstantes Signal wurde das Zeitsignal eingeführt, von dem sich beliebige Signale ableiten lassen, deren Verlauf als Funktion der Zeit bekannt ist. Wenn sich so auch dynamisches Übertragungsverhalten konstruieren läßt, sind Signalflüsse mit diesen Möglichkeiten auf die Vorwärtsrichtung beschränkt. Rückgekoppelte Systeme wie etwa in Regelungen machen den Rückgriff auf „vergangene“ Signalwerte erforderlich, was als „Speichern“ eines Zustandes verstanden werden kann.

*Differenzengleichungen*

So genügt z. B. ein Integrator für reellwertige, zeitdiskrete Signale der Differenzengleichung

$$y_{k+1} = y_k + x_k \cdot \Delta t \quad (6.29)$$

mit der Anfangsbedingung

$$y_0 = 0, \quad (6.30)$$

wobei das Ausgangssignal  $y = [y_0, y_1, \dots]$  durch Integration aus dem Eingangssignal  $x = [x_0, x_1, \dots]$  entsteht;  $\Delta t$  ist dabei das Abtastintervall.

Gewöhnliche lineare Differenzengleichungen  $n$ -ter Ordnung haben allgemein die Form [WB05]:

$$y_{k+n} + a_1 \cdot y_{k+n-1} + \dots + a_n \cdot y_k = b_0 \cdot x_{k+m} + \dots + b_m \cdot x_k \quad (6.31)$$

mit der Kausalitätsbedingung  $m \leq n$ .

Jede solche Differenzengleichung  $n$ -ter Ordnung läßt sich in ein System von Differenzengleichungen erster Ordnung überführen. Es reicht also aus, das gewählte diskrete Zeitmodell zugrunde zu legen und die Zeitverschiebung von Signalen um einen Zeitschritt ( $\Delta t = \text{step}$ ) darzustellen. Nach der Verschiebung beträgt der Zeitwert des Signals genau den Wert des unverschobenen Signals zu dem nächstfrüheren Zeitpunkt; für den Zeitpunkt 0 muß ein Startwert für das verschobene Signal bereitgestellt werden:

**Spezifikation 6.18 (Zeitverschiebung)** für eine Funktion  $\text{delay}_X$  zu jeder Zustandsmenge  $X$ :

$$\text{delay}_X \in X \rightarrow \text{Process}_X \rightarrow \text{Process}_X$$

$$\text{state}_X (\text{delay}_X x_0 x) 0_{\text{Time}} = x_0$$

$$\text{state}_X (\text{delay}_X x_0 x) (t + 1)_{\text{Time}} = \text{state}_X x t$$

□

**Modell 6.18.1** Für jede Zustandsmenge  $X$  sei  $\text{delay}_X$  wie folgt definiert:

$$\text{delay}_X \stackrel{\text{def}}{=} \lambda x_0 \in X . \lambda x \in \text{Process}_X . \lambda t \in \text{Time} . \begin{cases} x_0 & \text{falls } t = 0_{\text{Time}} \\ \text{state}_X x (t - 1_{\text{Time}}) & \text{sonst} \end{cases}$$

□

Nur die Zeitverschiebung mit  $\text{delay}_X$  macht es möglich, daß Signale rekursiv definiert werden können.

**Beispiel 6.3** Der (zeitdiskrete) Integrator für reellwertige Signale aus (6.30) und (6.30) läßt sich mit  $\text{delay}_{\mathbb{R}}$  definieren als:

$$\begin{aligned} \text{integrate}_{\mathbb{R}} &\in \text{Process}_{\mathbb{R}} \rightarrow \text{Process}_{\mathbb{R}} \\ \text{integrate}_{\mathbb{R}} x &\stackrel{\text{def}}{=} y \\ \text{wobei } y &\stackrel{\text{def}}{=} \text{delay}_{\mathbb{R}} 0 (\text{add}_{\mathbb{R}} [y, \text{gain}_{\mathbb{R}} \text{ step } x]) . \end{aligned}$$

Zur Definition von  $y$  siehe auch Abbildung 6.2. Abbildung 6.3 zeigt das gleiche Beispiel in der Notation von MATLAB Simulink. □

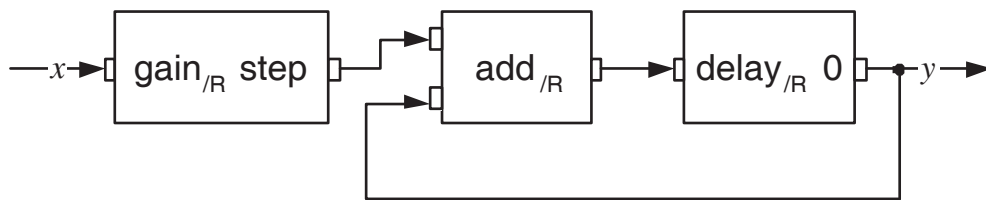


Abbildung 6.2: Signalflußgraph zu Beispiel 6.3

Die Schrittweite der Abtastung und Zeitverschiebung ist durch die bei der Sprachdefinition vorgenommene einheitliche Festlegung von  $\text{step}$  (auf 1 ms) konstant und sehr klein. In konkreten Anwendungen werden daher im allgemeinen Zeitverschiebungen nach dem gleichen Grundprinzip aber mit verbreiterten Abtastintervallen benötigt. Auch unterschiedliche Abtastraten innerhalb des gleichen Systems (sogenannte *multirate systems*) gehören zu den üblichen Entwurfsentscheidungen.

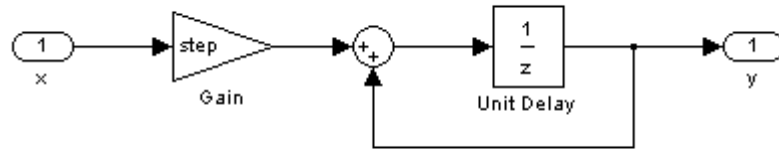


Abbildung 6.3: Beispiel 6.3 in Simulink

Die Beschreibungsmittel aus Abschnitt B.3.3 werden es ermöglichen, eine Verallgemeinerung von  $\text{delay}_X$  auf Abtastperioden  $> \text{step}$  und aperiodische Abtastung zu konstruieren. Die in Beispiel 6.7 beschriebene Funktion  $\text{sample}_X$  tastet ein Signal bei jedem Stattfinden eines beliebigen Ereignisses ab. Ein Ereignis kann periodisch sein (z. B. clock) oder aber beliebig aperiodisch stattfinden. Als Grenzfall (bei einem Ereignis, das zu jedem Zeitpunkt  $\neq 0_{\text{Time}}$  stattfindet, etwa bei clock  $0_{\text{Time}} \ 1_{\text{Time}}$ ) enthält  $\text{sample}_X$  das Verhalten von  $\text{delay}_X$ , mit dessen Hilfe es konstruiert ist.

## 6.5 Reaktive Prozesse

Ergänzend zu dem Begriff des Prozesses wurden in Abschnitt B.3.1.3 *Ereignisse* eingeführt. Als Hilfsmittel zur Beschreibung von Prozessen zeichnen sie besondere Zeitpunkte im Verhalten eines Prozesses aus und sind geeignet, dieses in Abschnitte (*Phasen*) zu unterteilen. Ereignisabhängige Übergänge (*Transitionen*) zwischen Phasen können als *Reaktionen* auf Ereignisse gedeutet werden.

### 6.5.1 Phasen und Transitionen

#### Phasen

Ein Prozeß definiert sich dann aus einer endlichen oder unendlichen Folge von Phasen, wobei eine Phase für ein nichtleeres Zeitintervall das Prozeßverhalten bestimmt. Bei endlichen Phasenfolgen ist das letzte Intervall rechtsseitig offen, erstreckt sich also bis ins Unendliche (siehe Abbildung 6.4). Eine Phase beginnt zum Nullzeitpunkt oder zum Zeitpunkt einer Transition (d. h. des Stattfindens eines Ereignisses); sie endet zum Zeitpunkt einer Transition oder nie.

Mit der Vorstellung eines in Phasen eingeteilten Prozesses verbindet sich, daß das Prozeßverhalten – wie bei einer abschnittsweise definierten Funktion – innerhalb einer Phase einheitlich auf bestimmte

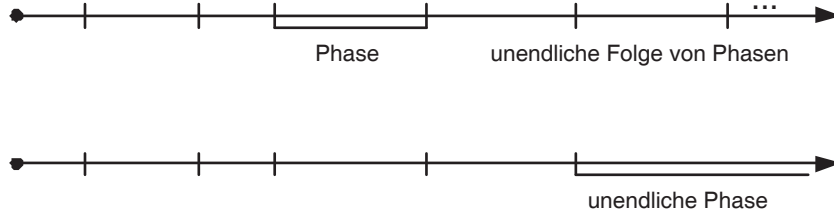


Abbildung 6.4: Phasen

Weise beschreibbar ist und daß die Transitionen dazwischen diskrete Wechsel darstellen. Eine Transition löst ein bestimmtes, einheitliches Verhalten durch ein neues Verhalten ab. Dies kann so präzisiert werden, daß für eine Phase ein *Abschnitt* eines bestimmten (den ganzen Zeitstrahl überdeckenden) *Prozesses* „eingebildet“ wird und sich der Gesamtprozeß als Verkettung solcher Abschnitte ergibt.

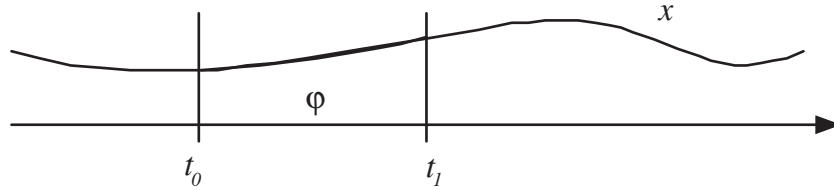


Abbildung 6.5: Phase

Eine Phase  $\varphi$  kann also in erster Näherung beschrieben werden durch einen Prozeß  $x \in \text{Process}_X$ , aus dem der Ausschnitt gebildet wird, einen Startzeitpunkt  $t_0$  und einen optionalen Endzeitpunkt  $t_1$  (siehe Abbildung 6.5). Sei  $x_\varphi$  der phasendefinierte Prozeß, der sich aus  $\varphi$  und anderen Phasen zusammensetzt. Dann soll gelten:

$$x \stackrel{t_0 \ t_1}{\mid=} x_\varphi.$$

$\text{Process}_X$

Da der Endzeitpunkt einer endlichen Phase stets zugleich der Startzeitpunkt einer neuen Phase ist und der Zeitpunkt durch das Stattfinden eines Ereignisses bestimmt wird, kann die Angabe eines Endzeitpunktes ersetzt werden durch die Angabe eines Ereignisses  $e$ , dessen nächstes Stattfinden nach dem Startzeitpunkt  $t_0$  den Endzeitpunkt  $t_1$  markiert, also

$$t_1 = \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } e \ t_0 \ t\}.$$

Falls das Ereignis nach  $t_0$  nicht mehr stattfindet (wie dies beim Ereignis *never* stets der Fall ist), ist die Phase unendlich.

Für das Verhalten innerhalb der Phase sind der Prozeß  $x$  und die Lage des Ausschnitts maßgeblich, die durch den Startzeitpunkt  $t_0$  bestimmt wird. Die Dauer der Phase wird durch das Ereignis  $e$  bestimmt. Die Wahl des Zeitpunktes  $t_0$  einer Phase fällt für die erste Phase eines phasendefinierten Prozesses zwangsläufig auf  $t_0 = 0_{\text{Time}}$ ; für alle nachfolgenden Phasen fällt  $t_0$  mit dem Ereigniszeitpunkt der sie einleitenden Transition, also  $t_1$  der vorangehenden Phase zusammen. Der frei verfügbare Parameter ist der Prozeß  $x$ .

Eine Festlegung von  $x$  unabhängig von  $t_0$  hat zur Folge, daß der Zustandsverlauf von  $x_\varphi$  innerhalb der Phase  $\varphi$  von beiden Parametern abhängt (so beginnt die Phase etwa mit dem Zustand  $\text{state}_X x t_0$ ). Um einen gleichbleibenden relativen Verlauf unabhängig von der absoluten Lage von  $t_0$  zu erzielen, müßte  $x$  ein jeweils um  $t_0$  verschobenes Signal  $x'$  sein (siehe Abbildung 6.6). Allgemeiner betrachtet wird es nötig sein,  $x$  in Abhängigkeit von  $t_0$  wählen zu können.

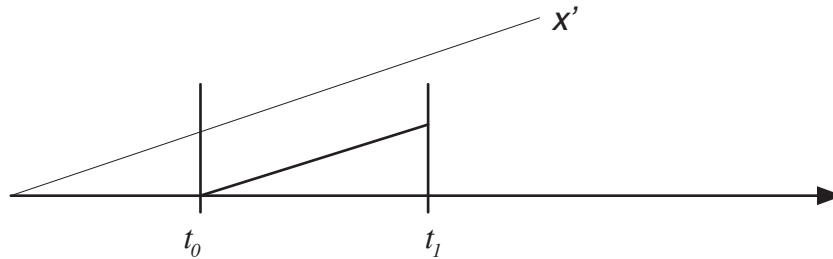


Abbildung 6.6: Phase: Verhalten abhängig vom Startzeitpunkt definiert

Ein zweites Kriterium zur Bestimmung von  $x$  mag der Systemzustand, speziell der Wert eines Signals  $x'$ , zum Zeitpunkt  $t_0$  sein. Im Beispiel aus Abbildung 6.7 hält die Phase den Wert des Signals  $x'$  zum Zeitpunkt  $t_0$  konstant. Wählt man als  $x'$  das Zeitsignal, so stellt sich eine Abhängigkeit vom Startzeitpunkt als Spezialfall der Abhängigkeit vom Systemzustand zum Startzeitpunkt dar, denn  $t_0 = \text{state}_{\text{Time}} \text{ timeProcess } t_0$ .

### Transitionen

Mit den aufgezeigten Mechanismen der Parametrierung können mit einer endlichen Anzahl von Phasenbeschreibungen beliebig viele und zudem beliebig viele verhaltensverschiedene Phasen instantiiert werden. Zu ihrer Verkettung zu einer Phasenfolge zur Konstruktion eines



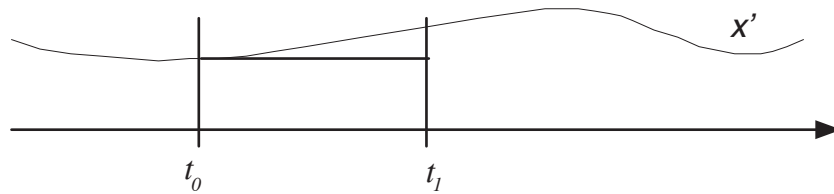


Abbildung 6.7: Phase: Verhalten abhängig vom Systemzustand zum Startzeitpunkt definiert

abschnittsweise definierten Prozesses müssen die *Transitionen* in endlicher Form beschrieben werden. Eine Transition ist der Wechsel von einer Phase zur nächsten, ausgelöst durch ein Ereignis. Fraglich ist nun, ob die Beschreibung des Ereignisses zur Beschreibung der Phase oder zur Beschreibung der Transition gehört. Im ersten Fall definiert eine Phase ihre Terminierung selbst und die Transition stellt lediglich die Sequenz dar; die Phase terminiert „aktiv“. Im zweiten Fall wird die Phase für eine potentiell unendliche Dauer gestartet und „passiv“ durch die Transition terminiert. Im ersten Fall kann eine Verzweigung in der Auswahl der Folgephase nur in Abhängigkeit vom Systemzustand definiert werden; im zweiten Fall kann eine Phase in Konkurrenz mehrerer Ereignisse terminiert werden, die zu unterschiedlichen Folgephasen führen. Im ersten Fall liegt eine *sequentielle*, im zweiten Fall eine *reaktive* Ablaufsteuerung vor.

Für die erste Art haben sich die Kontrollstrukturen der *strukturierten Programmierung* bewährt; in der zweiten Art lassen sich *endliche Automaten* erkennen. Beide Formen der Ablaufsteuerung haben ihre Berechtigung und Anwendung und werden im folgenden durch die entworfenen Beschreibungsmittel zur Verfügung gestellt. Sequentielle Abläufe werden in Abschnitt 6.6 behandelt, reaktive Steuerungen in Abschnitt 6.5.2.

Phasenweise definierte Prozesse der zweiten Art sollen im weiteren als *reaktiv* bezeichnet werden. Das Verhalten solcher Prozesse „reagiert“ mit Phasenwechseln auf Ereignisse. Die erste Art der phasenweisen Definition von Prozessen abstrahiert von der Ereignissteuerung; die Ereignisse verschwinden in atomaren *Aktionen*, die mit ereignisunabhängigen Kontrollstrukturen verkettet werden. Die Prozesse dieser Art heißen *sequentiell*.

*reaktive  
Prozesse*

### 6.5.2 Phasenübergangssysteme

Transitionssysteme nach Art endlicher Automaten sind beschreibbar als gerichtete Graphen mit Wurzel. Knoten repräsentieren Phasen, und Kanten sind durch Ereignisse beschriftet. Eine Phasenfolge entspricht dann einem von der Wurzel ausgehenden Pfad im Graphen, wobei die Kanten die Transitionen darstellen; Zyklen im Graphen ermöglichen unendliche Folgen von Phasen. Das Eintreffen der an den ausgehenden Kanten notierten Ereignisse bestimmt den Pfad. Da die Ereignisse mehrerer ausgehender Kanten eines Knotens gleichzeitig eintreffen können, ist zur Herstellung von Eindeutigkeit (deterministischem Verhalten) eine Priorisierung der Ereignisse bzw. eine Ordnung der ausgehenden Kanten erforderlich. Abbildung 6.8 zeigt ein Beispiel; der Wurzelknoten ist im Stil von UML markiert, die Priorisierung der Transitionen erfolge im Uhrzeigersinn ab der 12-Uhr-Position (z. B.  $e_1, e_2, e_0$  für  $\varphi_0$ ).

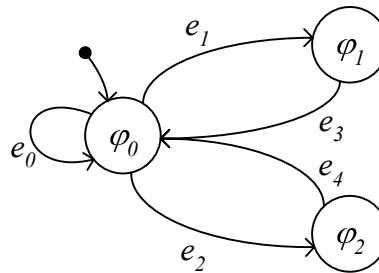


Abbildung 6.8: Transitionssystem

Ist – wie in FSPL durch die Metasprache gegeben – die Möglichkeit rekursiver Definition vorhanden, können gerichtete Graphen wie Bäume induktiv beschrieben werden. Ein Baum ist entweder ein Blattknoten oder ein innerer Knoten mit  $\geq 1$  Kanten zu Bäumen. Anders ausgedrückt ist ein Baum ein Knoten mit  $\geq 0$  Knoten zu Bäumen. So ist ein gerichteter Graph ein Knoten mit  $\geq 0$  ausgehenden Kanten zu gerichteten Graphen. Oder aber: Ein gerichteter Graph ist entweder ein Terminalknoten (ein Knoten ohne ausgehende Kanten) oder ein Knoten mit  $\geq 1$  Kanten zu gerichteten Graphen. Anders als bei Bäumen kann es gerichtete Graphen ohne Terminalknoten geben, und die Knoten können mehr als eine eingehende Kante aufweisen. Mit Rekursion läßt sich das Beispiel aus Abbildung 6.8 in Definitionen nach Abbildung 6.9 zerlegen.

Festzuhalten bleibt, daß Phasen in reaktiven Prozessen unabhängig von den sie terminierenden Ereignissen definiert werden sollen. Die

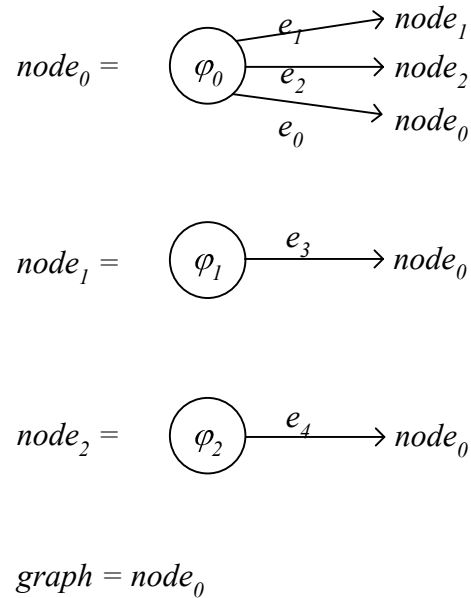


Abbildung 6.9: Rekursive Definition eines gerichteten Graphen

Verknüpfung von beiden bleibt dem Konzept selbstterminierender Aktionen in sequentiellen Prozessen vorbehalten. *Die Rolle eines Knotens in einem Phasenübergangssystem ist daher verschieden von der Beschreibung des Verhaltens der dem Knoten zugeordneten Phasen.* So kann eine einmal definierte Verhaltensbeschreibung in mehreren Knoten verwendet werden (siehe Abbildung 6.10), die sich auch in unterschiedlichen Graphen befinden können.

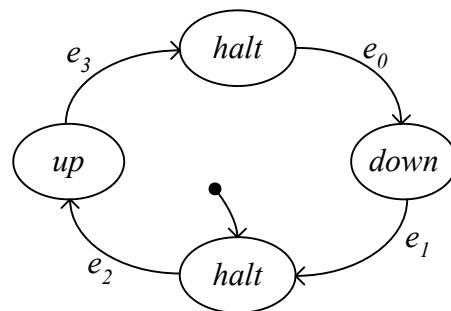


Abbildung 6.10: Wiederverwendung von Phasendefinitionen

Andererseits definiert das durch den gerichteten Graphen beschriebene Phasenübergangssystem ein unendliches Prozeßverhalten als Phasenfolge. Es kann wahlweise als Prozeß oder auch als (zusammengesetzte) Phase gedeutet werden, die zum Zeitpunkt 0 oder (innerhalb ei-

ner anderen Phasenfolge) zu einem späteren Zeitpunkt gestartet werden kann. Die zweite Deutung entspricht einer induktiven Definition, bei der der Graph von gleichem Typ ist wie ein einzelner Knoten. Sie hat zudem den Vorteil, daß Teilgraphen so ohne weiteres nicht nur horizontal, sondern auch vertikal zusammengesetzt werden können: Ein Graph (bzw. eine zusammengesetzte Phase) kann als Knoten eines übergeordneten Graphen (bzw. Phasenübergangssystems) auftreten (siehe Abbildung 6.11).

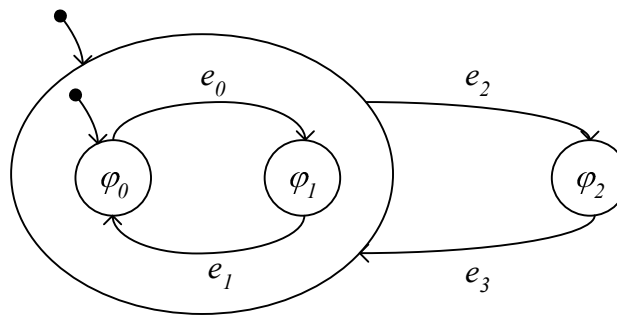


Abbildung 6.11: Hierarchisch zusammengesetzte Phasen

So ergibt sich, daß eine Phase entweder „atomar“ als generischer Ausschnitt aus einem Prozeß oder zusammengesetzt durch ein Phasenübergangssystem definiert werden kann, das Phasen mittels Transitionen verknüpft. Die Definition einer atomaren Phase ist insofern generisch, als der zugrundegelegte Prozeß allgemein vom Systemzustand zum Startzeitpunkt der Phase abhängt. Phasen sind stets als nichtterminierend definiert; Phasenwechsel, die innerhalb zusammengesetzter Phasen auftreten, brechen ein unendlich fortsetzbares Verhalten ab und setzen es mit einem zweiten, potentiell unendlichen Verhalten fort. Ergänzend muß ein Mechanismus existieren, eine Phase zum Zeitpunkt 0 als Prozeß zu starten.

Als Variante einer Transition kann auch der zu einem Ereigniszeitpunkt erreichte Zustand als Endzustand festgehalten werden. Die Phase setzt sich dabei mit einem konstanten Zustand fort (siehe Abbildung 6.12). Der Vorgang kann als eine Art von Terminierung aufgefaßt werden, die jedoch von Terminierung in einer sequentiellen Ablaufsteuerung zu unterscheiden ist.

*abstrakter  
Datentyp*

Als Zusammenfassung aus den Überlegungen zu Phasen, Transitionen und Phasenübergangssystemen kann nun ein abstrakter Datentyp angegeben werden, der Phasen als neue semantische Kategorie einführt:

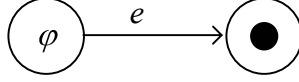


Abbildung 6.12: Endzustand

**Spezifikation 6.19 (Phase)** für Mengen  $\text{Phase}_X$ ,  $\text{Continuation}_X$  und  $\text{Transition}_X$  und Funktionen  $\text{phase}_X$ ,  $\text{valueInPhase}_{Y,X}$ ,  $\text{switch}_X$ ,  $\text{start}_X$ ,  $\text{goto}_X$ ,  $\text{wait}_X$  und  $\text{when}_X$  zu jeder Zustandsmenge  $X$  und (im Fall von  $\text{valueInPhase}_{Y,X}$ ) zu jeder Zustandsmenge  $Y$ :

$$\begin{aligned}
 \text{phase}_X &\in \text{Process}_X \rightarrow \text{Phase}_X \\
 \text{valueInPhase}_{Y,X} &\in \text{Process}_Y \rightarrow (Y \rightarrow \text{Phase}_X) \rightarrow \text{Phase}_X \\
 \text{switch}_X &\in \text{Phase}_X \rightarrow \text{Transition}_X^+ \rightarrow \text{Phase}_X \\
 \text{start}_X &\in \text{Phase}_X \rightarrow \text{Time} \rightarrow \text{Process}_X \\
 \text{goto}_X &\in \text{Phase}_X \rightarrow \text{Continuation}_X \\
 \text{wait}_X &\in \text{Continuation}_X \\
 \text{when}_X &\in \text{Event} \rightarrow \text{Continuation}_X \rightarrow \text{Transition}_X
 \end{aligned}$$

$$\begin{aligned}
 \text{start}_X (\text{phase}_X x) t_0 &\stackrel{t_0}{\underset{\text{Process}_X}{|}}= x \\
 \text{start}_X (\text{valueInPhase}_{Y,X} y \varphi_y) t_0 &\stackrel{t_0}{\underset{\text{Process}_X}{|}}= \text{start}_X (\varphi_y (\text{state}_Y y t_0)) t_0
 \end{aligned}$$

$$t \underset{\text{Time}}{\geq} t_0 \Rightarrow \text{state}_X (\text{start}_X (\text{switch}_X \varphi [\text{when}_X e_0 \psi_0, \dots, \text{when}_X e_n \psi_n]) t_0) t =
 \begin{cases} \text{state}_X (\text{start}_X \varphi t_0) t & \text{falls } \neg(\text{happened } (e_0 \underset{\text{Event}}{\vee} \dots \vee e_n) t_0 t) \\ \text{state}_X (\text{start}_X \varphi_k t_1) t & \text{sonst} \end{cases}$$

wobei

$$\begin{aligned}
 t_1 &\stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } (e_0 \underset{\text{Event}}{\vee} \dots \vee e_n) t_0 t\} \\
 x_1 &\stackrel{\text{def}}{=} \text{state}_X (\text{start}_X \varphi t_0) t_1 \\
 k &\stackrel{\text{def}}{=} \min\{i \in 0..n \mid \text{happened } e_i t_0 t_1\} \\
 \varphi_i &\stackrel{\text{def}}{=} \begin{cases} \phi_i & \text{falls } \psi_i = \text{goto}_X \phi_i \\ \text{phase}_X (\text{const}_X x_1) & \text{falls } \psi_i = \text{wait}_X \end{cases} \\
 &\text{für } i \in 0..n
 \end{aligned}$$

□

Fortsetzung

Der Konstruktor  $\text{phase}_X$  erzeugt eine atomare Phase aus einem unabhängig vom Startzeitpunkt  $t_0$  gegebenen Prozeß, während  $\text{valueInPhase}_{Y,X}$  den Wert eines Signals zum Startzeitpunkt einer Phase als konstanten Parameter übergibt. Als Spezialfall kann  $\text{valueInPhase}_{Y,X}$  mit  $\text{phase}_X$  kombiniert werden, um eine startzustandsabhängige atomare Phase zu konstruieren.  $\text{switch}_X$  erzeugt eine zusammengesetzte Phase aus einer Ausgangsphase und  $n$  konkurrierenden Transitionen. Jede Transition wird bestimmt durch ein Ereignis und eine *Fortsetzung*, wobei eine Fortsetzung entweder durch eine Zielphase oder als „Warten“ (Übergang in einen Endzustand) bestimmt ist. Welches Prozeßverhalten ab einem Startzeitpunkt die Phase definiert, wird durch  $\text{start}_X$  ermittelt. Speziell überführt  $\text{start}_X \varphi 0_{\text{Time}}$  eine Phase  $\varphi$  in einen (vollständig definierten) Prozeß und wird so zum Prozeßkonstruktor; für  $t_0 \neq 0_{\text{Time}}$  dient  $\text{start}_X$  lediglich der Spezifikation von  $\text{switch}_X$  und anderer Operatoren in noch folgenden Spezifikationen.

Bei der folgenden Angabe eines Modells wird Modell 6.2.1 für Prozesse zugrundegelegt:

**Modell 6.19.1** Für alle Zustandsmengen  $X$  und  $Y$  seien  $\text{Phase}_X$ ,  $\text{Continuation}_X$  und  $\text{Transition}_X$ ,  $\text{phase}_X$ ,  $\text{valueInPhase}_{Y,X}$ ,  $\text{switch}_X$ ,  $\text{start}_X$ ,  $\text{goto}_X$ ,  $\text{wait}_X$  und  $\text{when}_X$  wie folgt definiert:

$$\begin{aligned}
\text{Phase}_X &\stackrel{\text{def}}{=} \text{Time} \rightarrow \text{Process}_X \\
\text{Continuation}_X &\stackrel{\text{def}}{=} \text{Phase}_X + \mathbb{U} \\
\text{Transition}_X &\stackrel{\text{def}}{=} \text{Event} \times \text{Continuation}_X \\
\\ 
\text{start}_X &\stackrel{\text{def}}{=} \lambda \varphi \in \text{Phase}_X . \lambda t \in \text{Time} . \varphi t \\
\text{phase}_X &\stackrel{\text{def}}{=} \lambda x \in \text{Process}_X . \lambda t_0 \in \text{Time} . x \\
\text{valueInPhase}_{Y,X} &\stackrel{\text{def}}{=} \lambda y \in \text{Process}_Y . \lambda \varphi_y \in (Y \rightarrow \text{Phase}_X) . \lambda t_0 \in \text{Time} . \\
&\quad \varphi_y (\text{state}_Y y t_0) t_0 \\
\\ 
\text{switch}_X &\stackrel{\text{def}}{=} \lambda \varphi \in \text{Phase}_X . \lambda [[e_0, \psi_0], \dots, [e_n, \psi_n]] \in (\text{Event} \times (\text{Phase}_X + \mathbb{U}))^+ . \\
&\quad \lambda t_0 \in \text{Time} . \lambda t \in \text{Time} . \\
&\quad \begin{cases} \varphi t_0 t & \text{falls } \neg(\text{happened } (e_0 \vee \dots \vee e_n) t_0 t) \\ & \text{Event} \qquad \qquad \text{Event} \\ \varphi_k t_1 t & \text{sonst} \end{cases}
\end{aligned}$$

wobei

$$\begin{aligned}
t_1 &\stackrel{\text{def}}{=} \min_{\text{Time}} \{t \in \text{Time} \mid \text{happened } (e_0 \underset{\text{Event}}{\vee} \dots \underset{\text{Event}}{\vee} e_n) t_0 t\} \\
x_1 &\stackrel{\text{def}}{=} \text{state}_X (\text{start}_X \varphi t_0) t_1 \\
k &\stackrel{\text{def}}{=} \min \{i \in 0..n \mid \text{happened } e_i t_0 t_1\} \\
\varphi_i &\stackrel{\text{def}}{=} \begin{cases} \phi_i & \text{falls } \psi_i = [0, \phi_i] \\ \text{phase}_X (\text{const}_X x_1) & \text{sonst} \end{cases} \\
&\quad \text{für } i \in 0..n \\
\text{goto}_X &\stackrel{\text{def}}{=} \lambda \varphi \in \text{Phase}_X . [0, \varphi] \\
\text{wait}_X &\stackrel{\text{def}}{=} [1, \text{unit}] \\
\text{when}_X &\stackrel{\text{def}}{=} \lambda e \in \text{Event} . \lambda \psi \in \text{Continuation}_X . [e, \psi]
\end{aligned}$$

□

(Für die Definition von  $\text{switch}_X$  wurde eine in Anlehnung an (A.79) für Folgen erweiterte Lambda-Notation benutzt.)

Auch für Phasen sollen Vergleichsoperatoren definiert werden. Neben Gleichheit wird wie bei Prozessen abschnittsweise Gleichheit definiert, wobei hier Abschnitte über Ereignisse definiert werden: *Vergleichsoperatoren*

**Definition 6.10 (Vergleich von Phasen)** Seien  $X$  eine Zustandsmenge,  $\varphi, \varphi' \in \text{Phase}_X$  und  $e, e' \in \text{Event}$ .

$$\begin{aligned}
\varphi &=_{\text{Phase}_X} \varphi' \stackrel{\text{def}}{=} \forall t_0, t \in \text{Time} . \\
&\quad (t \underset{\text{Time}}{\geq} t_0 \Rightarrow \\
&\quad \text{state}_X (\text{start}_X \varphi t_0) t = \text{state}_X (\text{start}_X \varphi' t_0) t) \\
&= \forall t_0 \in \text{Time} . \\
&\quad \text{start}_X \varphi t_0 \underset{\text{Process}_X}{\Big|}^{t_0} \text{start}_X \varphi' t_0
\end{aligned}$$

$$\varphi \not\equiv_{\text{Phase}_X} \varphi' \stackrel{\text{def}}{=} \neg(\varphi \equiv_{\text{Phase}_X} \varphi')$$

$$\varphi \stackrel{e}{\models}_{\text{Phase}_X} \varphi' \stackrel{\text{def}}{=} \forall t_0, t \in \text{Time} .$$

$$((\text{happened } e \ t_0 \ t) \Rightarrow \text{state}_X (\text{start}_X \varphi \ t_0) \ t = \text{state}_X (\text{start}_X \varphi' \ t_0) \ t)$$

$$\varphi \stackrel{e}{\models}_{\text{Phase}_X} \varphi' \stackrel{\text{def}}{=} \forall t_0, t \in \text{Time} .$$

$$(t \geq_{\text{Time}} t_0 \wedge \neg(\text{happened } e \ t_0 \ t) \Rightarrow$$

$$\text{state}_X (\text{start}_X \varphi \ t_0) \ t = \text{state}_X (\text{start}_X \varphi' \ t_0) \ t)$$

$$\varphi \stackrel{e \ e'}{\models}_{\text{Phase}_X} \varphi' \stackrel{\text{def}}{=} \forall t_0, t \in \text{Time} .$$

$$((\text{happened } e \ t_0 \ t) \wedge \neg(\text{happened } (e +_{\text{Event}} e') \ t_0 \ t) \Rightarrow$$

$$\text{state}_X (\text{start}_X \varphi \ t_0) \ t = \text{state}_X (\text{start}_X \varphi' \ t_0) \ t) \quad \square$$

Zerfallen zwei Phasen in Abschnitte, auf denen sie jeweils gleich sind, sind sie insgesamt gleich. So gilt für  $\varphi, \varphi' \in \text{Phase}_X, e, e' \in \text{Event}$  mit  $e'' \stackrel{\text{def}}{=} e +_{\text{Event}} e'$ :

$$\varphi \stackrel{e}{\models}_{\text{Phase}_X} \varphi' \wedge \varphi \stackrel{e \ e'}{\models}_{\text{Phase}_X} \varphi' \wedge \varphi \stackrel{e''}{\models}_{\text{Phase}_X} \varphi' \quad (6.32)$$

$$\Rightarrow \varphi \equiv_{\text{Phase}_X} \varphi'$$

$$\varphi \stackrel{e}{\models}_{\text{Phase}_X} \varphi' \wedge \varphi \stackrel{e'}{\models}_{\text{Phase}_X} \varphi' \wedge e' \leq_{\text{Event}} e \quad (6.33)$$

$$\Rightarrow \varphi \equiv_{\text{Phase}_X} \varphi'.$$

Das folgende Beispiel zeigt die rekursive Definition von Phasen durch ein Gleichungssystem:



**Beispiel 6.4** *Eine einfache Ampelschaltung, bei der ein Knopfdruck die Grünphase beendet und dann zeitgesteuert ein Zyklus bis zurück zur Grünphase durchlaufen wird, sei wie folgt beschrieben:*

$$\begin{aligned}
buttonPressed &\in \text{Event} \\
\Delta t_{yellow} &\in \text{Time} \\
\Delta t_{red} &\in \text{Time} \\
\Delta t_{redyellow} &\in \text{Time} \\
L &= \mathbb{B}^3 \\
green &= \text{phase}_L (\text{const}_L [\text{false}, \text{false}, \text{true}]) \\
yellow &= \text{phase}_L (\text{const}_L [\text{false}, \text{true}, \text{false}]) \\
red &= \text{phase}_L (\text{const}_L [\text{true}, \text{false}, \text{false}]) \\
redyellow &= \text{phase}_L (\text{const}_L [\text{true}, \text{true}, \text{false}]) \\
greenPhase &= \text{switch}_L green \\
&\quad [\text{when}_L buttonPressed \quad (\text{goto}_L yellowPhase)] \\
yellowPhase &= \text{switch}_L yellow \\
&\quad [\text{when}_L (\text{after } \Delta t_{yellow}) \quad (\text{goto}_L redPhase)] \\
redPhase &= \text{switch}_L red \\
&\quad [\text{when}_L (\text{after } \Delta t_{red}) \quad (\text{goto}_L redyellowPhase)] \\
redyellowPhase &= \text{switch}_L redyellow \\
&\quad [\text{when}_L (\text{after } \Delta t_{redyellow}) \quad (\text{goto}_L greenPhase)] \\
lights &= \text{start}_L greenPhase \ 0_{\text{Time}}
\end{aligned}$$

Bei gegebenem Ereignis  $buttonPressed$  und Zeitkonstanten  $\Delta t_{yellow}$ ,  $\Delta t_{red}$  und  $\Delta t_{redyellow}$  und auf Basis der durch  $L$  gegebenen Codierung der Ampelzustände als Tripel von Wahrheitswerten (für das Leuchten der drei Ampellichter in der bekannten Reihenfolge) beschreibt  $lights$  den Verlauf der Ampelzustände als Prozeß.  $\square$

**Beispiel 6.5 (Beispielrechnung zu Beispiel 6.4)** *Gegeben seien*

$$\begin{aligned}
\text{happened } buttonPressed \ 0_{\text{Time}} \ t &= t \underset{\text{Time}}{\geq} 1000_{\text{Time}} \\
\Delta t_{yellow} &= 2000_{\text{Time}} \\
\Delta t_{red} &= 30000_{\text{Time}} \\
\Delta t_{redyellow} &= 1000_{\text{Time}}
\end{aligned}$$

Dann ist beispielsweise

```

stateL lights 5000Time =
stateL (startL greenPhase 0Time) 5000Time =
stateL (startL (switchL green [whenL buttonPressed (gotoL yellowPhase)]) 0Time) 5000Time =
stateL (startL yellowPhase 1000Time) 5000Time =
stateL (startL (switchL yellow [whenL (after 2000Time) (gotoL redPhase)]) 1000Time) 5000Time =
stateL (startL redPhase 3000Time) 5000Time =
stateL (startL (switchL red [whenL (after 3000Time) (gotoL redyellowPhase)]) 3000Time) 5000Time =
stateL (startL red 3000Time) 5000Time =
stateL (startL (phaseL (constL [true, false, false]))) 3000Time) 5000Time =
stateL (constL [true, false, false]) 5000Time =
[true, false, false].

```

□

Die rekursive Definition von *greenPhase*, *yellowPhase*, *redPhase* und *redyellowPhase* sowie die Konstruktion des Gesamtprozesses *lights* ist graphisch in Abbildung 6.13 dargestellt.

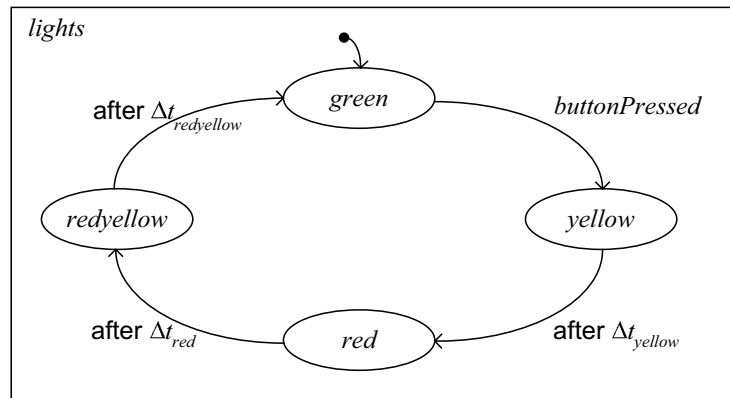


Abbildung 6.13: Beispiel 6.4

Ein weiteres Beispiel enthält Transitionen mit konkurrierenden Ereignissen (siehe auch Abbildung 6.14):

**Beispiel 6.6** Eine vertikale Linearbewegung kann durch die drei Ereignisse *up*, *down* und *stop* gesteuert werden. Im Haltezustand konkurrieren *up* und *down* miteinander.

$$\begin{aligned}
 up, down, stop &\in \text{Event} \\
 D &= \{-1, 0, 1\} \\
 moveUp &= \text{phase}_D (\text{const}_D 1) \\
 moveDown &= \text{phase}_D (\text{const}_D (-1)) \\
 halt &= \text{phase}_D (\text{const}_D 0) \\
 upwards &= \text{switch}_D moveUp \quad [\text{when}_D stop \quad (\text{goto}_D halting)] \\
 downwards &= \text{switch}_D moveDown \quad [\text{when}_D stop \quad (\text{goto}_D halting)] \\
 halting &= \text{switch}_D halt \quad [\text{when}_D up \quad (\text{goto}_D upwards), \\
 &\quad \text{when}_D down \quad (\text{goto}_D downwards)] \\
 motion &= \text{start}_D halting 0_{\text{Time}}
 \end{aligned}$$

□

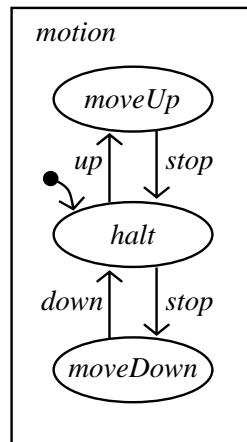


Abbildung 6.14: Beispiel 6.6

Als drittes Beispiel zeigt eine Funktion zur ereignisgetriebenen Abtastung, wie reaktives Verhalten zur Signalverarbeitung dienen kann. Die Funktion kann an Stelle von  $\text{delay}_X$  in Signalflüssen auftreten:

**Beispiel 6.7 (Ereignisgetriebene Signal-Abtastung)** Für jede Zustandsmenge  $X$  sei eine Funktion

$$\text{sample}_X \in \text{Event} \rightarrow X \rightarrow \text{Process}_X \rightarrow \text{Process}_X$$

wie folgt definiert:

$$\begin{aligned}
 \text{sample}_X e x_0 x &\stackrel{\text{def}}{=} \text{start}_X (\text{sampleCycle } x_0) 0_{\text{Time}} \\
 \text{wobei} \\
 \text{sampleCycle} &\in X \rightarrow \text{Phase}_X \\
 \text{sampleCycle } x_1 &\stackrel{\text{def}}{=} \text{switch}_X (\text{phase}_X (\text{const}_X x_1)) \\
 &\quad [\text{when}_X e (\text{goto}_X \text{takeValue})] \\
 \text{takeValue} &\in \text{Phase}_X \\
 \text{takeValue} &\stackrel{\text{def}}{=} \text{valueInPhase}_{X,X} (\text{delay}_X x_0 x) \text{sampleCycle}
 \end{aligned}$$

Damit kann eine Variante des Integrators aus Beispiel 6.3 mit frei wählbarem Abtastintervall gebildet werden (siehe auch Abbildung 6.15):

$$\begin{aligned}
 \text{sampleIntegrate}_{\mathbb{R}} &\in \text{Time}_+ \rightarrow \text{Process}_{\mathbb{R}} \rightarrow \text{Process}_{\mathbb{R}} \\
 \text{sampleIntegrate}_{\mathbb{R}} \Delta t x &\stackrel{\text{def}}{=} y \\
 \text{wobei } y &\stackrel{\text{def}}{=} \text{sample}_{\mathbb{R}} (\text{clock } 0_{\text{Time}} \Delta t) 0 (\text{add}_{\mathbb{R}} [\text{gain}_{\mathbb{R}} (\text{time } \Delta t) x, y])
 \end{aligned}$$

□

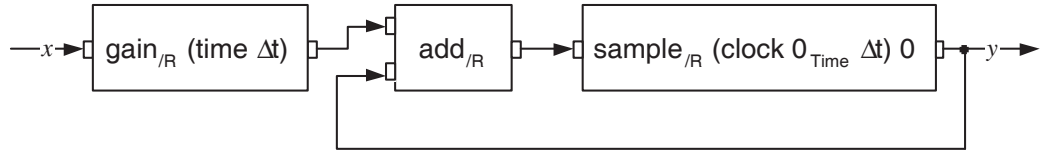


Abbildung 6.15: Signalflußgraph zu  $\text{sampleIntegrate}_{\mathbb{R}}$  aus Beispiel 6.7

*bedingte  
Phasenwechsel*

Als spezieller Anwendungsfall für  $\text{valueInPhase}$  stellt sich die Verzweigung einer Transition dar, bei der in Abhängigkeit vom Wert eines booleschen Signals zum Transitionszeitpunkt eine von zwei unterschiedlichen Folgephasen erreicht wird. Solche bedingten Phasenwechsel können zwar auch mittels bewachter Ereignisse (siehe Spezifikation 6.11) realisiert werden, wie das folgende Beispiel zeigt. Hier wird ein Ereignis  $e$  gegensätzlich durch  $c$  bzw. die Negation von  $c$  bewacht ( $c \in \text{Process}_{\mathbb{B}}$ ,  $\varphi, \varphi_0, \varphi_1 \in \text{Phase}_X$ ):

$$\begin{aligned}
 \text{switch}_X \varphi &[\text{when}_X (\text{guard } c e) (\text{goto}_X \varphi_0), \\
 &\quad \text{when}_X (\text{guard } (\text{apply}_{\mathbb{B} \rightarrow \mathbb{B}} \neg c) e) (\text{goto}_X \varphi_1)].
 \end{aligned} \tag{6.34}$$

Eine direktere Möglichkeit, den Sachverhalt auszudrücken, bietet aber ein Verzweigungskonstruktor für Phasen, der nachfolgend mit Hilfe von  $\text{valueInPhase}$  definiert wird:

**Definition 6.11** für eine Funktion  $\text{ifElsePhase}_X$  für jede Zustandsmenge  $X$ :

$$\text{ifElsePhase}_X \in \text{Process}_{\mathbb{B}} \rightarrow \text{Phase}_X \rightarrow \text{Phase}_X \rightarrow \text{Phase}_X$$

$$\text{ifElsePhase}_X c \varphi_0 \varphi_1 \stackrel{\text{def}}{=} \text{valueInPhase}_{\mathbb{B}, X} c \left( \lambda c_0 \in \mathbb{B}. \begin{cases} \varphi_0 & \text{falls } c_0 = \text{true} \\ \varphi_1 & \text{sonst} \end{cases} \right)$$

□

Mit  $\text{ifElsePhase}_X$  läßt sich (6.34) ersetzen durch

$$\text{switch}_X \varphi [\text{when}_X e (\text{goto}_X (\text{ifElsePhase}_X c \varphi_0 \varphi_1))]. \quad (6.35)$$

Abbildung 6.16 zeigt in Anlehnung an Statecharts eine graphische Notation für einen Ausdruck  $\text{ifElsePhase}_X c \varphi_0 \varphi_1$ .

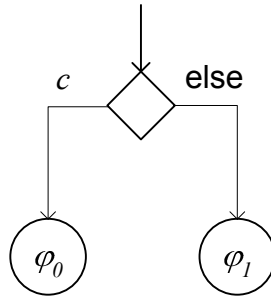


Abbildung 6.16: Bedingte Phase ( $\text{ifElsePhase}_X c \varphi_0 \varphi_1$ )

Bei Kenntnis von Statecharts [Har87] stellt sich die Frage, ob mit dem hier entwickelten Modell für Phasenübergangssysteme nicht nur *Hierarchie* (siehe Abbildung 6.11), sondern auch *Parallelität* dargestellt werden kann. Parallelität wird durch den folgenden Konstruktor *orthogonalize* eingeführt, der die Produkttransformation für Prozesse (siehe Spezifikation 6.17) auf Phasen überträgt. Die Zusammenfassung einer Struktur von Phasen zu einer Phase führt in der Anwendung dazu, daß alle Phasen gleichzeitig betreten werden. Eine Rücktransformation gibt es bei Phasen nicht. Spezifikation 6.19 und Modell 6.19.1 werden wie folgt erweitert:

**Spezifikation 6.20 (Phasenparallelisierung)** für Funktionen  $\text{orthogonalize}_{\prod(\lambda i \in I. X_i)}$  zu jeder Mengenabbildung  $\lambda i \in I. X_i$ , die jedem Index  $i$  aus einer Indexmenge  $I$  eine Zustandsmenge  $X_i$  zuordnet:

$$\text{orthogonalize}_{\prod(\lambda i \in I. X_i)} \in \prod(\lambda i \in I. \text{Phase}_{X_i}) \rightarrow \text{Phase}_{\prod(\lambda i \in I. X_i)}$$

$$\text{start}_X (\text{orthogonalize}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . \varphi_i)) t_0 \stackrel{t_0}{\models}_{\text{Process}_X} \text{zip}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . \text{start}_X \varphi_i t_0) \quad \square$$

**Modell 6.20.1** Für alle Mengenabbildungen  $\lambda i \in I . X_i$  seien die Funktionen  $\text{orthogonalize}_{\prod(\lambda i \in I . X_i)}$  wie folgt definiert:

$$\text{orthogonalize}_{\prod(\lambda i \in I . X_i)} \stackrel{\text{def}}{=} \lambda \varphi \in \prod(\lambda i \in I . \text{Phase}_{X_i}) . \lambda t_0 \in \text{Time} . \text{zip}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . \text{start}_{X_i} (\varphi i) t_0) \quad \square$$

Abbildung 6.17 zeigt eine graphische Notation in Anlehnung für Statecharts mit dem Ausdruck  $\text{orthogonalize}_{X_0 \times X_1} [\varphi_0, \varphi_1]$  für  $\varphi_0 \in \text{Phase}_{X_0}$  und  $\varphi_1 \in \text{Phase}_{X_1}$  als Beispiel.

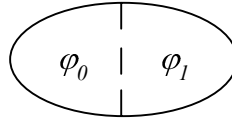


Abbildung 6.17: Parallelität von Phasen

*lokale  
Variablen*

Bei der Parallelisierung setzen sich die nebenläufigen Phasen zu einer Phase zusammen, die das Verhalten eines Prozesses mit multiplikativ zusammengesetztem Zustandsraum beschreibt. Im Gegensatz dazu bilden *lokale Variablen* (lokale Prozesse) eine Art unterordnender Nebenläufigkeit: Für die Verweildauer eines Prozesses in einer Phase wird temporär ein weiterer phasendefinierter Prozeß erzeugt, der als Hilfsvariable zur Definition der Phase des erstgenannten Prozesses dient. Spezifikation 6.19 und Modell 6.19.1 werden wie folgt erweitert:

**Spezifikation 6.21 (Lokale Variable)** für eine Funktion  $\text{variableInPhase}_{Y,X}$  zu je zwei Zustandsmengen  $Y$  und  $X$ :

$$\text{variableInPhase}_{Y,X} \in \text{Phase}_Y \rightarrow (\text{Process}_Y \rightarrow \text{Phase}_X) \rightarrow \text{Phase}_X$$

$$\text{start}_X (\text{variableInPhase}_{Y,X} \phi \varphi_\phi) t_0 \stackrel{t_0}{\models}_{\text{Process}_X} \text{start}_X (\varphi_\phi (\text{start}_Y \phi t_0)) t_0$$

□

**Modell 6.21.1** Für je zwei Zustandsmengen  $Y$  und  $X$  sei die Funktion  $\text{variableInPhase}_{Y,X}$  wie folgt definiert:

$$\begin{aligned} \text{variableInPhase}_{Y,X} &\stackrel{\text{def}}{=} \lambda \phi \in \text{Phase}_Y . \lambda \varphi_\phi \in (\text{Process}_Y \rightarrow \text{Phase}_X) . \\ &\lambda t_0 \in \text{Time} . \varphi_\phi (\phi t_0) t_0 \end{aligned}$$

□

**Beispiel 6.8** Mit einem lokalen Ereigniszähler läßt sich die Wirkung des Operators  $\underset{\text{Event}}{*}$  nachbilden. Sei dazu

$$\text{count} \in \mathbb{N} \rightarrow \text{Event} \rightarrow \text{Phase}_{\mathbb{N}}$$

rekursiv definiert durch

$$\text{count } n \ e \stackrel{\text{def}}{=} \text{switch}_{\mathbb{N}} (\text{phase}_{\mathbb{N}} (\text{const}_{\mathbb{N}} n)) [\text{when}_{\mathbb{N}} e (\text{goto}_{\mathbb{N}} (\text{count } (n+1) e))].$$

So ist etwa für  $n \in \mathbb{N} \setminus \{0\}$

$$\text{switch}_X \varphi [\text{when}_X (n \underset{\text{Event}}{*} e) \psi]$$

gleichbedeutend mit

$$\begin{aligned} &\text{variableInPhase}_{\mathbb{N},X} (\text{count } 0 \ e) \\ &(\lambda c \in \text{Process}_{\mathbb{N}} . \\ &\text{switch}_X \varphi [\text{when}_X (\text{guard } (\text{apply}_{\mathbb{N} \rightarrow \mathbb{B}} (\lambda c \in \mathbb{N} . c = n) c) e) \psi]). \quad \square \end{aligned}$$

Lokale Auswertung von Signalen im Sinn von  $\text{valueInPhase}_{Y,X}$  und lokale Variablen im Sinn von  $\text{variableInPhase}_{Y,X}$  lassen sich von Phasen auf Ereignisse übertragen. Die Auswertung von Ereignissen hängt in gleicher Weise wie die von Phasen von einem Startzeitpunkt  $t_0$  ab, zu dem Signale ausgewertet oder Phasen als lokale Prozesse (Variablen) gestartet werden können. Im Sinne einer erhöhten Orthogonalität der Beschreibungsmittel werden daher Spezifikation 6.3 und Modell 6.3.2 wie folgt ergänzt:

*lokale Auswertung und lokale Variablen für Ereignisse*

**Spezifikation 6.22 (Ereignislokale Auswertung und Variablen)** für Funktionen  $\text{valueInEvent}_X$  und  $\text{variableInEvent}_X$  zu jeder Zustandsmenge  $X$ :

$$\begin{aligned} \text{valueInEvent}_X &\in \text{Process}_X \rightarrow (X \rightarrow \text{Event}) \rightarrow \text{Event} \\ \text{variableInEvent}_X &\in \text{Phase}_X \rightarrow (\text{Process}_X \rightarrow \text{Event}) \rightarrow \text{Event} \\ \text{happened } (\text{valueInEvent}_X \ x \ e_x) \ t_0 \ t &= \text{happened } (e_x (\text{state}_X \ x \ t_0)) \ t_0 \ t \\ \text{happened } (\text{variableInEvent}_X \ \varphi \ e_\varphi) \ t_0 \ t &= \text{happened } (e_\varphi (\text{start}_X \ \varphi \ t_0)) \ t_0 \ t \end{aligned}$$

□

**Modell 6.22.1**

$$\begin{aligned}
\text{valueInEvent}_X &\stackrel{\text{def}}{=} \lambda x \in \text{Process}_X . \lambda e_x \in X \rightarrow \text{Event} . \lambda t_0 \in \text{Time} . \\
&\quad e_x (\text{state}_X \varphi t_0) t_0 \\
\text{variableInEvent}_X &\stackrel{\text{def}}{=} \lambda \varphi \in \text{Phase}_X . \lambda e_\varphi \in \text{Process}_X \rightarrow \text{Event} . \lambda t_0 \in \text{Time} . \\
&\quad e_\varphi (\text{start}_X \varphi t_0) t_0
\end{aligned}$$

□

Damit läßt sich Beispiel 6.8 dahingehend vereinfachen, daß der Operator  $\underset{\text{Event}}{*}$  direkt nachgebildet wird:

**Beispiel 6.9** *Sei*

$$\text{count} \in \mathbb{N} \rightarrow \text{Event} \rightarrow \text{Phase}_{\mathbb{N}}$$

wie in Beispiel 6.8 definiert. Dann ist für  $n \in \mathbb{N} \setminus \{0\}$

$$\begin{aligned}
n \underset{\text{Event}}{*} e &= \\
&\text{variableInEvent}_{\mathbb{N}} (\text{count } 0 \ e) \\
&\quad (\lambda c \in \text{Process}_{\mathbb{N}} . \text{guard} (\text{apply}_{\mathbb{N} \rightarrow \mathbb{B}} (\lambda c \in \mathbb{N} . c = n) \ c) \ e).
\end{aligned}$$

□

**Beispiel 6.10** Mit  $\text{valueInEvent}_{\text{Time}}$  und  $\text{timeProcess}$  läßt sich der Ereignisgenerator *after* nachbilden:

$$\begin{aligned}
\text{after} &= \lambda \Delta t \in \text{Time}_+ . \\
&\quad \text{valueInEvent}_{\text{Time}} \text{ timeProcess} (\lambda t_0 \in \text{Time} . \text{trigger } x)
\end{aligned}$$

wobei

$$x \stackrel{\text{def}}{=} \text{apply}_{\text{Time} \times \text{Time} \rightarrow \mathbb{B}} \left( \underset{\text{Time}}{=} \right) [\text{timeProcess}, \text{const}_{\text{Time}} (t_0 \underset{\text{Time}}{+} \Delta t)]$$

□

In ähnlicher Weise wie in Beispiel 6.10 können weitere vordefinierte Ereignisgeneratoren (wie z. B. *watchdog*) auf primitivere Ereignisse, Signale und Phasen zurückgeführt werden. Insbesondere wird aber auch die individuelle Konstruktion neuer Ereignisgeneratoren sehr flexibel möglich.



## 6.6 Sequentielle Prozesse

Programmierung wird im landläufigen Verständnis häufig gleichgesetzt mit der Erstellung *sequentieller* Programme, der Ausformulierung von Algorithmen in einem *imperativen* Stil. Ein sequentielles Programm beschreibt eine Abfolge von Aktionen als Befehle, die durch Kontrollstrukturen (wie z. B. Verzweigungen und Schleifen) derart verknüpft sind, daß sich die genaue Abfolge der Aktionen (auch *Kontrollfluß* genannt) im allgemeinen dynamisch aus den Zuständen von *Variablen* während der Programmausführung ergibt. Die elementare Aktion eines imperativen Programms ist die *Zuweisung* eines Wertes an eine Variable (auch *imperative Variable* genannt). Mechanismen zur Ein- und Ausgabe können entweder als Spezialfall von Variablen behandelt oder aber gesondert ergänzt werden. Unterprogramme, die Aktionssequenzen wiederverwendbar beschreiben, werden auch als *Prozeduren* bezeichnet.

*Aktionen*  
*Kontrollfluß*

Sieht man von dem Konzept der imperativen Variablen mit Zuweisung ab und betrachtet nur das Konzept, mittels elementarer Aktionen und Kontrollstrukturen eine Abfolge von Aktionen zu konstruieren, so ergibt sich eine Sicht, die auf abschnittsweise definierte Prozesse angewendet werden kann. Aktionen sind dabei Abschnitte im Verhalten eines Prozesses. Bedingungen, die den Kontrollfluß steuern, ergeben sich aus den Zuständen von Prozessen (d. h. Variablen in diesem Sinn).

### 6.6.1 Aktionen und Kontrollstrukturen

Der Bezug zu den für reaktive Prozesse definierten Phasen und Transitionen wurde in Abschnitt 6.5.1 bereits hinführend diskutiert. Phasen wurden in Abschnitt 6.5.2 formal so definiert, daß sie das Verhalten eines Prozesses ab dem Beginn eines Abschnitts beschreiben. Der Zeitpunkt des Beginns ist variabel und das beschriebene Verhalten im allgemeinen von diesem Zeitpunkt abhängig. *Terminierung*, also das Beenden eines Abschnitts für den Eintritt in einen neuen Abschnitt, wurde dem Konzept der Transition zugeordnet. Ereignisse rufen dabei Phasenwechsel im Prozeßverhalten hervor, bei denen sowohl die Vorgänger- als auch die Nachfolgerphase jeweils ein Verhalten bis auf unbestimmte Zeit beschreiben können; die „Nutzung“ der Vorgängerphase zur Beschreibung des Prozeßverhaltens wird sozusagen „abgebrochen“.

*Phasen-*  
*terminierung*

Die Terminierung ist auf diese Weise *nicht* Bestandteil der Phasendefinition. Eine *Phase* kann mit beliebigen Ereignissen kombiniert wer-

den, um ein Ende des durch die Phase definierten Abschnitts im Prozeßverhalten zu beschreiben. Im Gegensatz dazu verbindet sich mit *Aktionen* die Vorstellung, daß sie eine gewisse Zeit in Anspruch nehmen und dann enden, d. h. *von sich aus* terminieren. In einer Sequenz von Aktionen beginnt jede nachfolgende Aktion mit der Terminierung ihres Vorgängers. Eine Aktion wird nach der anderen ausgeführt.

Nun kann eine feste Kombination aus einer Phase und einem bestimmten terminierenden Ereignis als *atomare Aktion* gedeutet werden. Mit dem Ereignis ist gedanklich eine Transition zu verbinden, deren Zielphase dabei unbestimmt ist. Dieser Ansatz liegt dem Aktionsmodell des im folgenden präzisierten Konzepts sequentieller Prozesse zugrunde.

Man betrachte den Spezialfall einer Transitionsdefinition, die genau ein Ereignis beinhaltet (keine alternativen Transitionen), also

$$\text{switch}_X \varphi [\text{when}_x e \psi] \quad (6.36)$$

für eine Phase  $\varphi \in \text{Phase}_X$  und eine Fortsetzung  $\psi \in \text{Continuation}_X$  zu einer Zustandsmenge  $X$  und einem Ereignis  $e$ . Legt man sich bei  $\varphi$  und  $e$  fest und läßt  $\psi$  variabel, so entsteht ein Verhaltensmuster, das mit beliebigen Fortsetzungen  $\psi \in \text{Continuation}_X$  zu einer zusammengesetzten Phase kombiniert werden kann.

Die Kombination aus einer Phase  $\varphi$  und einem (terminierenden) Ereignis  $e$ , die nach (6.36) als Teil einer zusammengesetzten Phase, als endliche Phase in einer Phasenfolge gedeutet wird, soll als (elementare) *Aktion* bezeichnet werden (siehe Abbildung 6.18).

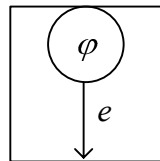


Abbildung 6.18: Elementare Aktion

Man beachte, daß die *Verkettung* zweier solcher Aktionen zu einem Objekt gleicher Art möglich ist (siehe Abbildung 6.19). Auf diese Weise entsteht aus zwei Aktionen eine zusammengesetzte Aktion. Für den Mechanismus der Verkettung ist es dabei unwesentlich, ob es sich um elementare oder bereits zusammengesetzte Aktionen handelt.

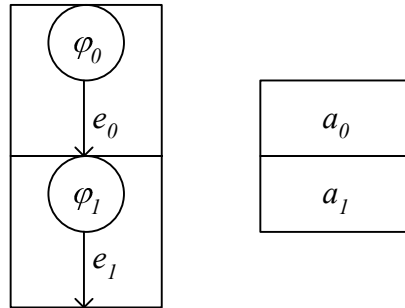


Abbildung 6.19: Aktionssequenz

Damit sind die wesentlichen Konzepte angerissen, die zur Definition von Aktionen auf Basis von Phasen auf Basis von Prozessen ausreichen: die Definition elementarer Aktionen aus Phasen und Ereignissen, die Vervollständigung von Aktionen zu Phasen (siehe auch Abbildung 6.20) und die Verknüpfung von Aktionen, vertreten durch die *Sequenzierung*. Sie wird noch um zusätzliche Kontrollstrukturen ergänzt, wobei auf das bewährte Instrumentarium sequentieller Programmierung zurückgegriffen werden kann. Im Kern handelt es sich traditionell um die Konzepte *Alternative* und *Iteration*, die in verschiedenen Variationen angeboten werden können. Als Verbindungsglied zu Variablen (d. i. Prozessen) empfehlen sich noch die *Auswertung* einer Variablen zum aktuellen Zeitpunkt für den nachfolgenden Gebrauch als Konstante (analog zu `valueInPhase`) sowie lokale Variablen (analog zu `variableInPhase`).

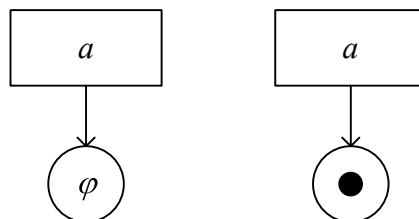


Abbildung 6.20: Vervollständigung von Aktionen zu Phasen

Es folgt nun wieder die Definition eines abstrakten Datentyps:

**Spezifikation 6.23 (Aktionen)** für eine Menge  $\text{Action}_X$  und Funktionen  $\text{behaviour}_X$ ,  $\text{termination}_X$ ,  $\text{until}_X$ ,  $\text{continue}_X$ ,  $\text{loop}_X$ ,  $\text{sequence}_X$ ,  $\text{ifElse}_X$ ,

$\text{repeatUntil}_X$ ,  $\text{valueInAction}_{Y,X}$  und  $\text{variableInAction}_{Y,X}$  zu jeder Zustandsmenge  $X$  und jeder Zustandsmenge  $Y$ :

$$\begin{aligned}
\text{behaviour}_X &\in \text{Action}_X \rightarrow \text{Phase}_X \\
\text{termination}_X &\in \text{Action}_X \rightarrow \text{Event} \\
\text{until}_X &\in \text{Event} \rightarrow \text{Phase}_X \rightarrow \text{Action}_X \\
\text{continue}_X &\in \text{Action}_X \rightarrow \text{Continuation}_X \rightarrow \text{Phase}_X \\
\text{loop}_X &\in \text{Action}_X \rightarrow \text{Phase}_X \\
\text{sequence}_X &\in \text{Action}_X \rightarrow \text{Action}_X \rightarrow \text{Action}_X \\
\text{ifElse}_X &\in \text{Process}_{\mathbb{B}} \rightarrow \text{Action}_X \rightarrow \text{Action}_X \rightarrow \text{Action}_X \\
\text{repeatUntil}_X &\in \text{Action}_X \rightarrow \text{Process}_{\mathbb{B}} \rightarrow \text{Action}_X \\
\text{valueInAction}_{Y,X} &\in \text{Process}_Y \rightarrow (Y \rightarrow \text{Action}_X) \rightarrow \text{Action}_X \\
\text{variableInAction}_{Y,X} &\in \text{Phase}_Y \rightarrow (\text{Process}_Y \rightarrow \text{Action}_X) \rightarrow \text{Action}_X
\end{aligned}$$

$$\begin{aligned}
\text{behaviour}_X (\text{until}_X e \varphi) &\underset{\text{Phase}_X}{=} \varphi \\
\text{termination}_X (\text{until}_X e \varphi) &\underset{\text{Event}}{=} e \\
\text{continue}_X a \psi &\underset{\text{Phase}_X}{=} \\
&\quad \text{switch}_X (\text{behaviour}_X a) [\text{when}_X (\text{termination}_X a) \psi] \\
\text{loop}_X a &\underset{\text{Phase}_X}{=} \\
&\quad \text{continue}_X a (\text{goto}_X (\text{loop}_X a)) \\
\text{behaviour}_X (\text{sequence}_X a b) &\underset{\text{Phase}_X}{=} \\
&\quad \text{continue}_X a (\text{goto}_X (\text{behaviour}_X b)) \\
\text{termination}_X (\text{sequence}_X a b) &\underset{\text{Event}}{=} \\
&\quad (\text{termination}_X a) + (\text{termination}_X b) \\
\text{behaviour}_X (\text{ifElse}_X c a b) &\underset{\text{Phase}_X}{=} \\
&\quad \text{ifElsePhase}_X c (\text{behaviour}_X a) (\text{behaviour}_X b) \\
\text{termination}_X (\text{ifElse}_X c a b) &\underset{\text{Event}}{=} \\
&\quad \text{ifElseEvent}_X c (\text{termination}_X a) (\text{termination}_X b)
\end{aligned}$$

$$\begin{aligned}
\text{behaviour}_X (\text{repeatUntil}_X a c) &=_{\text{Phase}_X} \text{loop}_X a \\
\text{termination}_X (\text{repeatUntil}_X a c) &=_{\text{Event}} \text{guard } c (\text{termination}_X a) \\
\text{start}_X (\text{behaviour}_X (\text{valueInAction}_{Y,X} y a_y)) t_0 &\stackrel{t_0}{\mid}=_{\text{Process}_X} \text{start}_X (\text{behaviour}_X (a_y (\text{state}_Y y t_0))) t_0 \\
\text{happened} (\text{termination}_X (\text{valueInAction}_{Y,X} y a_y)) t_0 t &= \text{happened} (\text{termination}_X (a_y (\text{state}_Y y t_0))) t_0 t \\
\text{start}_X (\text{behaviour}_X (\text{variableInAction}_{Y,X} \varphi a_\varphi)) t_0 &\stackrel{t_0}{\mid}=_{\text{Process}_X} \text{start}_X (\text{behaviour}_X (a_\varphi (\text{start}_Y \varphi t_0))) t_0 \\
\text{happened} (\text{termination}_X (\text{variableInAction}_{Y,X} \varphi a_\varphi)) t_0 t &= \text{happened} (\text{termination}_X (a_\varphi (\text{start}_Y \varphi t_0))) t_0 t
\end{aligned}$$

$\text{behaviour}_X$  und  $\text{termination}_X$  charakterisieren Aktionen ausschließlich zu Spezifikationszwecken.  $\text{until}_X$  führt als Konstruktor in die Menge der Aktionen hinein,  $\text{continue}_X$  und  $\text{loop}_X$  führen aus ihr hinaus; die übrigen Operationen dienen der Verknüpfung von Aktionen.

Neben dem bedingten Sprung in Form der  $\text{repeatUntil}_X$ -Schleife wurde in Spezifikation 6.23 auch der unbedingte Sprung in Form der  $\text{loop}_X$ -Schleife eingeführt.  $\text{loop}_X$  ist jedoch ein Konstruktor nicht für Aktionen, sondern für Phasen. An die unendliche Wiederholung der gleichen Aktion kann keine Fortsetzung angeschlossen werden.

Damit Schleifen wohldefiniert sind, muß jeder Schleifendurchlauf einen Zeitfortschritt bzw. mindestens eine atomare Aktion enthalten. Diese Bedingung wird per Konstruktion nach Spezifikation 6.23 erfüllt. Die Kontrollstrukturen sind so gewählt, daß keine Null-Aktionen entstehen können. Dies hat zur Folge, daß optionale Aktionen und Schleifen mit Anfangsbedingungen nicht in der gewohnten Form zur Verfügung stehen. Je eine Variante dieser Kontrollstrukturen, die stets die Fortsetzung durch eine weitere Aktion erfordern, wird in FSPL als syntaktischer Zucker eingeführt (siehe Abschnitt 7.3.5).

*atomare  
Aktionen und  
Prozeduren*

Eine Aktion  $\text{until}_X e \varphi$  heit *atomar*, alle anders konstruierten Aktionen heien *zusammengesetzt*. Zusammengesetzte Aktionen werden auch als *Prozeduren* bezeichnet. Es folgt ein Modell, das Aktionen stets als Paare  $[\varphi, e]$  betrachtet; fr Phasen und Ereignisse werden die Modelle 6.19.1 und 6.3.2 zugrundegelegt:

**Modell 6.23.1** *Fr jede Zustandsmenge  $X$  und jede Zustandsmenge  $Y$  werden die Menge  $\text{Action}_X$  und die Funktionen  $\text{behaviour}_X$ ,  $\text{termination}_X$ ,  $\text{until}_X$ ,  $\text{continue}_X$ ,  $\text{loop}_X$ ,  $\text{sequence}_X$ ,  $\text{ifElse}_X$ ,  $\text{repeatUntil}_X$ ,  $\text{valueInAction}_{Y,X}$  und  $\text{variableInAction}_{Y,X}$  wie folgt definiert:*

$$\begin{aligned}
\text{Action}_X &\stackrel{\text{def}}{=} \text{Phase}_X \times \text{Event} \\
\text{behaviour}_X &\stackrel{\text{def}}{=} \lambda [\varphi \in \text{Phase}_X, e \in \text{Event}] . \varphi \\
\text{termination}_X &\stackrel{\text{def}}{=} \lambda [\varphi \in \text{Phase}_X, e \in \text{Event}] . e \\
\text{until}_X &\stackrel{\text{def}}{=} \lambda e \in \text{Event} . \lambda \varphi \in \text{Phase}_X . [\varphi, e] \\
\text{continue}_X &\stackrel{\text{def}}{=} \lambda [\varphi \in \text{Phase}_X, e \in \text{Event}] . \lambda \psi \in \text{Continuation}_X . \\
&\quad \text{switch}_X \varphi [\text{when}_X e \psi] \\
\text{loop}_X &\stackrel{\text{def}}{=} \lambda [\varphi \in \text{Phase}_X, e \in \text{Event}] . l \\
&\quad \text{wobei } l \stackrel{\text{def}}{=} \text{switch}_X \varphi [\text{when}_X e (\text{goto}_X l)] \\
\text{sequence}_X &\stackrel{\text{def}}{=} \lambda [\varphi_0 \in \text{Phase}_X, e_0 \in \text{Event}] . \\
&\quad \lambda [\varphi_1 \in \text{Phase}_X, e_1 \in \text{Event}] . \\
&\quad [\text{switch}_X \varphi_0 [\text{when}_X e_0 (\text{goto}_X \varphi_1)], e_0 +_{\text{Event}} e_1] \\
\text{ifElse}_X &\stackrel{\text{def}}{=} \lambda c \in \text{Process}_{\mathbb{B}} . \\
&\quad \lambda [\varphi_0 \in \text{Phase}_X, e_0 \in \text{Event}] . \\
&\quad \lambda [\varphi_1 \in \text{Phase}_X, e_1 \in \text{Event}] . [ \\
&\quad \quad \lambda t_0 \in \text{Time} . \begin{cases} \varphi_0 t_0 & \text{falls } \text{state}_X c t_0 = \text{true} \\ \varphi_1 t_0 & \text{sonst} \end{cases} \\
&\quad , \\
&\quad \lambda t_0 \in \text{Time} . \begin{cases} e_0 t_0 & \text{falls } \text{state}_X c t_0 = \text{true} \\ e_1 t_0 & \text{sonst} \end{cases} \\
&\quad ]
\end{aligned}$$

$$\begin{aligned}
\text{repeatUntil}_X &\stackrel{\text{def}}{=} \lambda [\varphi \in \text{Phase}_X, e \in \text{Event}] . \lambda c \in \text{Process}_{\mathbb{B}} . \\
&\quad [\text{loop}_X [\varphi, e], \text{guard } c \ e] \\
\text{valueInAction}_{Y,X} &\stackrel{\text{def}}{=} \lambda y \in \text{Process}_Y . \lambda a_y \in (\text{Process}_Y \rightarrow \text{Action}_X) . \\
&\quad [\lambda t_0 \in \text{Time} . \text{behaviour}_X (a_y (\text{state}_Y y \ t_0)) \ t_0, \\
&\quad \lambda t_0 \in \text{Time} . \text{termination}_X (a_y (\text{state}_Y y \ t_0)) \ t_0] \\
\text{variableInAction}_{Y,X} &\stackrel{\text{def}}{=} \lambda \varphi \in \text{Phase}_Y . \lambda a_\varphi \in (\text{Phase}_Y \rightarrow \text{Action}_X) . \\
&\quad [\lambda t_0 \in \text{Time} . \text{behaviour}_X (a_\varphi (\text{start}_Y \varphi \ t_0)) \ t_0, \\
&\quad \lambda t_0 \in \text{Time} . \text{termination}_X (a_\varphi (\text{start}_Y \varphi \ t_0)) \ t_0] \quad \square
\end{aligned}$$

Für Aktionen wird Gleichheit wie folgt definiert:

Vergleichsoperatoren

**Definition 6.12 (Gleichheit von Aktionen)** Seien  $X$  eine Zustandsmenge und  $a, a' \in \text{Action}_X$ .

$$\begin{aligned}
a &=_{\text{Action}_X} a' \stackrel{\text{def}}{=} (\text{behaviour}_X a) =_{\text{Phase}_X} (\text{behaviour}_X a') \wedge \\
&\quad (\text{termination}_X a) =_{\text{Event}} (\text{termination}_X a') \\
a &\neq_{\text{Action}_X} a' \stackrel{\text{def}}{=} \neg(a =_{\text{Action}_X} a')
\end{aligned}$$

□

Für die Sequenzierung gilt ein Assoziativgesetz:

Assoziativität

$$\text{sequence}_X a_0 (\text{sequence}_X a_1 a_2) =_{\text{Action}_X} \text{sequence}_X (\text{sequence}_X a_0 a_1) a_2. \quad (6.37)$$

Als ein weiterer Fall von Iteration kann die  $n$ -fache Wiederholung einer Aktion ( $n \geq 1$ ) induktiv auf die Sequenzierung zurückgeführt werden. Die Anzahl der Wiederholungen steht dabei statisch fest:

**Definition 6.13 (Aktionswiederholung)** Sei  $a \in \text{Action}_X$  für eine Zustandsmenge  $X$ . Für alle  $n \in \mathbb{N} \setminus \{0\}$  sei definiert:

$$\begin{aligned}
1 \ *_{\text{Action}_X} a &\stackrel{\text{def}}{=} a \\
(n+1) \ *_{\text{Action}_X} a &\stackrel{\text{def}}{=} \text{sequence}_X (n \ *_{\text{Action}_X} a) a
\end{aligned}$$

□

*Parallelisierung*

Als Gegenstück zur Phasenparallelisierung stellt die *Parallelisierung von Aktionen* einen Mechanismus zur Synchronisation nebenläufiger sequentieller Prozesse dar. Die beiden Synchronisationspunkte einer Aktion sind Start und Terminierung. Startet man einen Satz von Aktionen zum gleichen Zeitpunkt, kann eine Synchronisation der Terminierung entweder dadurch erreicht werden, daß man alle Aktionen bis zur Terminierung der letzten verlängert (Maximierung), oder dadurch, daß die Terminierung der ersten Aktion alle Aktionen terminiert (Minimierung).

Das Verlängern einer Aktion  $a \in \text{Action}_X$  kann theoretisch dadurch erreicht werden, daß man das zugehörige Verhalten  $\text{behaviour}_X a$  über das Ereignis  $\text{termination}_X a$  hinaus beibehält. Alternativ kann auch der beim Ereignis  $\text{termination}_X a$  erreichte Zustand von  $\text{behaviour}_X a$  als Endzustand beibehalten werden. Der zweiten Alternative wird hier der Vorzug gegeben, da das Verlängern des dynamischen Verhaltens einer beliebigen Aktion in seiner Wirkung im allgemeinen nicht absehbar ist, während das Warten im Endzustand auch durch  $\text{continue}_X a \text{ wait}_X$  erreicht wird.

Die Synchronisation parallel ausgeführter Aktionen betrifft die zeitliche Dimension. Hinsichtlich des Zustandsraums unterscheidet sich die Parallelisierung von Aktionen nicht von der Parallelisierung (Orthogonalisierung) von Phasen (vgl. Spezifikation 6.20 in Abschnitt 6.5.2):

**Spezifikation 6.24 (Parallelisierung von Aktionen)** für Funktionen  $\text{parallelizeTerminating}_{\prod(\lambda i \in I. X_i)}$  und (in den Fällen wo  $I \neq \emptyset$ )  $\text{parallelizeWaiting}_{\prod(\lambda i \in I. X_i)}$  zu jeder Mengenabbildung  $\lambda i \in I. X_i$ , die jedem Index  $i$  aus einer Indexmenge  $I$  eine Zustandsmenge  $X_i$  zuordnet:

$$\begin{aligned} \text{parallelizeTerminating}_{\prod(\lambda i \in I. X_i)} &\in \prod(\lambda i \in I. \text{Action}_{X_i}) \rightarrow \text{Action}_{\prod(\lambda i \in I. X_i)} \\ \text{parallelizeWaiting}_{\prod(\lambda i \in I. X_i)} &\in \prod(\lambda i \in I. \text{Action}_{X_i}) \rightarrow \text{Action}_{\prod(\lambda i \in I. X_i)} \end{aligned}$$



$$\begin{aligned}
& \text{behaviour}_X (\text{parallelizeTerminating}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . a_i)) \stackrel{\text{Phase}_X}{=} \\
& \quad \text{orthogonalize}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . (\text{behaviour}_{X_i} a_i)) \\
& \text{termination}_X (\text{parallelizeTerminating}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . a_i)) \stackrel{\text{Event}}{=} \\
& \quad \bigvee_{\text{Event}} (\lambda i \in I . (\text{termination}_{X_i} a_i)) \\
& \text{behaviour}_X (\text{parallelizeWaiting}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . a_i)) \stackrel{\text{Phase}_X}{=} \\
& \quad \text{orthogonalize}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . (\text{continue}_{X_i} a_i \text{ wait}_X)) \\
& \text{termination}_X (\text{parallelizeWaiting}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . a_i)) \stackrel{\text{Event}}{=} \\
& \quad \bigwedge_{\text{Event}} (\lambda i \in I . (\text{termination}_{X_i} a_i)) \quad \square
\end{aligned}$$

**Modell 6.24.1** Für alle Mengenabbildungen  $\lambda i \in I . X_i$  seien die Funktionen  $\text{parallelizeTerminating}_{\prod(\lambda i \in I . X_i)}$  und (in den Fällen wo  $I \neq \emptyset$ )  $\text{parallelizeWaiting}_{\prod(\lambda i \in I . X_i)}$  wie folgt definiert:

$$\begin{aligned}
& \text{parallelizeTerminating}_{\prod(\lambda i \in I . X_i)} \stackrel{\text{def}}{=} \\
& \quad \lambda a \in \prod(\lambda i \in I . \text{Action}_{X_i}) . [ \\
& \quad \quad \text{orthogonalize}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . (\text{behaviour}_{X_i} (a i))), \\
& \quad \quad \bigvee_{\text{Event}} (\lambda i \in I . (\text{termination}_{X_i} (a i))) \\
& \quad ] \\
& \text{parallelizeWaiting}_{\prod(\lambda i \in I . X_i)} \stackrel{\text{def}}{=} \\
& \quad \lambda a \in \prod(\lambda i \in I . \text{Action}_{X_i}) . [ \\
& \quad \quad \text{orthogonalize}_{\prod(\lambda i \in I . X_i)} (\lambda i \in I . (\text{continue}_{X_i} (a i) \text{ wait}_X)), \\
& \quad \quad \bigwedge_{\text{Event}} (\lambda i \in I . (\text{termination}_{X_i} (a i))) \\
& \quad ] \quad \square
\end{aligned}$$

### 6.6.2 Ausnahmebehandlung

Moderne imperative Programmiersprachen (wie Ada, C++ oder Java) bieten Mechanismen, um die Behandlung von Ausnahmesituationen zur Laufzeit eines Programms zu vereinfachen. Zu den Ausnahmesituationen können arithmetische Überläufe oder Divisionen durch 0 genauso gehören wie Ein-/Ausgabefehler oder entsprechend der Anwendungslogik selbstdefinierte Ausnahmebedingungen. Ausnahmebehandlungsmechanismen in einer Sprache erlauben es, Kontrollflüsse einer Anwendung grundsätzlich auf die normalen Abläufe auszurichten und in einen Kontext einzubetten, der Ausnahmesituationen „abfängt“ und Gegenmaßnahmen einleitet, bevor eventuell der Kontrollfluß an geeigneter Stelle fortgesetzt wird. Fehlen solche Mechanismen, müssen Ausnahmen innerhalb des Kontrollflusses behandelt werden. Dabei besteht die Gefahr, daß sich der normale Kontrollfluß und die Ausnahmebehandlung in schwer durchschaubarer und schlecht wartbarer Weise verweben und die kombinatorischen Möglichkeiten des Auftretens solcher Situationen die Programmlogik aufblähen (vgl. [Seb04, Kapitel 14]).

Die Detektion von Ausnahmesituationen kann als *Spezialfall von Ereignissen* und Ausnahmebehandlung als *Spezialfall reaktiven Verhaltens* betrachtet werden. In Maschinensteuerungen sind Ausnahmesituationen mitunter Teil des Zustandsmodells der Maschine, das etwa zwischen verschiedenen Betriebsmodi unterscheidet, zu denen auch Diagnosemodi oder das geordnete Abschalten von Geräten gehören können. Die Trennung von Kontrollfluß und Ausnahmebehandlung findet sich grundsätzlich in der Unterscheidung und Verschränkung der Verhaltensmodelle „reaktive Prozesse“ und „sequentielle Prozesse“ wieder. Durch die Vervollständigung von Aktionen zu Phasen (siehe Abbildung 6.20) wird ein Kontrollfluß (eine zusammengesetzte Aktion) stets in einen reaktiven Prozeß einer übergeordneten Verhaltensebene eingebettet, wo auf Ausnahmeereignisse reagiert werden kann.<sup>4</sup>

Die Behandlung einer Ausnahmesituation, die innerhalb eines Kontrollflusses auftritt, hat bei den bereits angeführten Sprachen Ada, C++ und Java übereinstimmend zunächst den Abbruch des überwachten Kontrollflusses zur Folge, bevor nach Abschluß besonderer Maßnahmen ein übergeordneter Kontrollfluß fortgesetzt wird. Durch ein

<sup>4</sup>Der Zusammenhang zwischen der Verschränkung der Verhaltensmodelle und Abstraktionsebenen im Verhalten wird ausführlicher in Kapitel 3.2 und 5.3 und in Kapitel 8.2 diskutiert.

zyklisches übergeordnetes Verhalten ist nach Beendigung der Ausnahmesituation gleichwohl ein Wiedereintritt in den abgebrochenen Kontrollfluß im Sinn einer Wiederholung möglich.

Als wesentliches Element der Ausnahmebehandlung tritt somit der Abbruch einer Aktion als Reaktion auf ein Ereignis, fortgesetzt durch ein spezielles Verhalten, hervor. Um die spezifische Behandlung unterschiedlicher Ausnahmen, die in Konkurrenz zueinander auftreten können, zu ermöglichen, empfiehlt sich ferner die Verallgemeinerung auf mehrere Ausnahmeereignisse, die zu jeweils individuellen Fortsetzungen führen. Da die Ausnahmebehandlung auf einer reaktiven Verhaltenzebene erfolgt, können Fortsetzungen dieser Art als spezielle Transitionen aufgefaßt werden. Sie führen also entweder zu Phasen oder zu Endzuständen (siehe Abbildung 6.21).

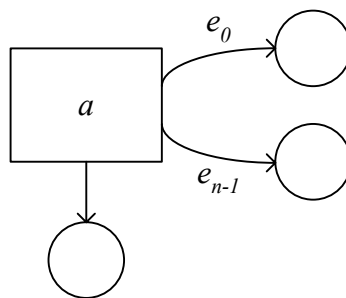


Abbildung 6.21: Ausnahmebehandlung

Die Aktion  $a$  in Abbildung 6.21 wird im Normalfall (d. h. wenn keine der Ausnahmesituationen  $e_0$  bis  $e_{n-1}$  eintritt) durch die Phase oder den Endzustand fortgesetzt, auf die bzw. den der Pfeil ohne Beschriftung zeigt. Tritt während der Ausführung der Aktion jedoch eines der Ereignisse  $e_0$  bis  $e_{n-1}$  auf, wird die Aktion unmittelbar terminiert und durch die Phase oder den Endzustand fortgesetzt, zu der bzw. dem der jeweilige Pfeil zeigt. Wie die Auflösung der Verhaltensstruktur in Abbildung 6.21 entsprechend der Aktionssemantik zeigt, verbirgt sich hinter dem Mechanismus der Ausnahmebehandlung ein gewöhnliches Phasentransitionssystem (Abbildung 6.22).

Ausgehend von dem in Abbildung 6.21 skizzierten Konstrukt erscheint die einfache Fortsetzung einer Aktion durch eine Phase oder einen Endzustand ( $\text{continue}_x$ ) als Spezialfall für  $n = 0$ . In der Tat führt die Ausnahmebehandlung auf eine Verallgemeinerung dieses Konstrukts:

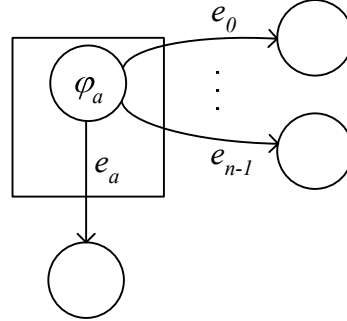


Abbildung 6.22: Semantik der Ausnahmebehandlung

**Spezifikation 6.25 (Ausnahmebehandlung)** für Funktionen  $\text{continueWithExceptions}_X$  zu jeder Zustandsmenge  $X$ :

$$\text{continueWithExceptions}_X \in \text{Action}_X \rightarrow \text{Continuation}_X \rightarrow \text{Transition}_X^* \rightarrow \text{Phase}_X$$

$$\text{continueWithExceptions}_X a \psi [\tau_0, \dots, \tau_{n-1}] \underset{\text{Phase}_X}{=} \text{switch}_X (\text{behaviour}_X a) [\text{when}_X (\text{termination}_X a) \psi, \tau_0, \dots, \tau_{n-1}]$$

□

**Modell 6.25.1** Für jede Zustandsmenge  $X$  sei  $\text{continueWithExceptions}_X$  wie folgt definiert:

$$\begin{aligned} \text{continueWithExceptions}_X &\stackrel{\text{def}}{=} \\ &\lambda a \in \text{Action}_X . \lambda \psi \in \text{Continuation}_X . \\ &\lambda [\tau_0 \in \text{Transition}_X, \dots, \tau_{n-1} \in \text{Transition}_X] . \\ &\text{switch}_X (\text{behaviour}_X a) [\text{when}_X (\text{termination}_X a) \psi, \tau_0, \dots, \tau_{n-1}] \end{aligned} \quad \square$$

## 6.7 Zusammenfassung

Die entworfenen Beschreibungsmittel konzentrieren sich auf vier miteinander verschränkte Konzepte dynamischen Verhaltens, denen gemeinsam eine diskrete Zeitbasis (synchrones Systemmodell) zugrundeliegt:

**Prozeß (Signal, Variable)** Ein Prozeß besitzt zu jedem Zeitpunkt einen Zustand innerhalb eines Zustandsraums. Alle Prozesse starten zum Nullzeitpunkt und haben unendliche Dauer.

**Ereignis** Ein Ereignis ist gekennzeichnet durch sein Stattfinden zu einzelnen Zeitpunkten. Ereignisse können durch eintretende Zustandsbedingungen von Prozessen ausgelöst werden.

**Phase** Eine Phase beschreibt den Zustandsverlauf eines Prozesses ab einem variablen Startzeitpunkt. Ein Prozeß kann mit Hilfe mehrerer Phasen, von denen eine zum Nullzeitpunkt gestartet wird, beschrieben werden. Phasenübergänge (Transitionen) werden dabei durch Ereignisse ausgelöst.

**Aktion** Eine Aktion beschreibt eine Phase bis zu einem terminierenden Ereignis. Eine Phase kann mit Hilfe mehrerer Aktionen beschrieben werden, die durch Kontrollstrukturen sequenziert werden; jedes Aufeinanderfolgen zweier Aktionen entspricht dabei einem Phasenübergang. Die für die Darstellung einer Phase notwendige Unendlichkeit des beschriebenen Verhaltens wird entweder durch eine Endlosschleife oder durch eine Vervollständigung der abschließenden Transition (Fortsetzung) erreicht.

Atomare Phasen entstehen aus Prozessen durch den Konstruktor  $\text{phase}_X$ , Prozesse generieren Ereignisse mittels  $\text{trigger}$ , und das Starten einer Phase mit  $\text{start}_X$  zum Zeitpunkt  $0_{\text{Time}}$  erzeugt einen Prozeß aus einer Phase; eine atomare Aktion wird aus einer Phase und einem Ereignis gebildet durch  $\text{until}_X$ , eine Aktion wird in eine Phase überführt durch Fortsetzung mittels  $\text{continue}_X$  oder durch eine Endlosschleife mittels  $\text{loop}_X$ .

Die Beschreibungsmittel wurden entworfen im Hinblick auf die Überdeckung dreier Verhaltensmodelle: quasi-kontinuierliches, reaktives und sequentielles Verhalten.

*Verhaltensmodelle*

**quasi-kontinuierlich** Als Diskretisierung kontinuierlicher Systeme, die durch Differentialgleichungen beschrieben werden, werden quasi-kontinuierliche Systeme mathematisch durch *Differenzengleichungen* beschrieben. Die zeitliche Diskretisierung betrachtet die Systeme als mit einem konstanten *Zeittakt* getaktet. Ein eindeutig aufgelöstes Gleichungssystem kann durch einen *Signalflußgraphen* dargestellt werden, der die funktionalen Abhängigkeiten zwischen den Variablen beschreibt.

Zeitverschiebung um einen Zeitschritt sowie algebraische Beziehungen zwischen Variablen sind die für Differenzengleichungen

wesentlichen Beschreibungselemente. Sie werden durch die Signalkonstruktoren  $\text{delay}_X$  bzw.  $\text{apply}_{X \rightarrow Y}$  bereitgestellt. Signalflußgraphen werden durch rekursive Signaldefinitionen beschrieben. Quasi-kontinuierliches Verhalten ist immer eine Eigenschaft von *Systemen*, also ein Zusammenhang zwischen Eingangs- und Ausgangssignalen.

In einem quasi-kontinuierlichen System ergibt sich der aktuelle Zustand eines Signals im allgemeinen aus aktuellen Zuständen anderer Signale sowie aus Zuständen von Signalen zum vorhergehenden Zeitpunkt (bereitgestellt durch Zeitverschiebung). Mit jedem Zeitschritt findet eine Neubestimmung des Zustands statt. Dieses Verhalten kann vergrößert werden, indem  $\text{delay}_X$  durch  $\text{sample}_X$  in Verbindung mit einem Taktereignis ersetzt wird (siehe Beispiel 6.7). Für beide Fälle charakteristisch ist jedoch, daß Zustandsänderungen durch den gleichmäßigen Fortschritt der Zeit gesteuert werden.

**reaktiv** Reaktives Verhalten ist charakterisiert durch sprunghafte Änderungen, ausgelöst durch Ereignisse. Zwischen den Ereignissen muß der Prozeß jedoch nicht in einem konstanten Zustand verharren, sondern kann ein beliebiges Verhalten aufweisen wie etwa den Zustandsverlauf eines quasi-kontinuierlichen Systems. Statt eines *Zustands* kann daher von einer *Phase* des Prozesses gesprochen werden. Phasenwechsel können als Umschalten zwischen unterschiedlichen Verhalten (d. i. Prozessen bei atomaren Phasen) verstanden werden.

Ereignisse, Phasen und Transitionen sind die wesentlichen Beschreibungselemente reaktiver Prozesse; sie werden durch Ereignis- und Phasenkonstruktoren bereitgestellt. Durch rekursive Phasendefinitionen werden *Phasenübergangssysteme* beschrieben.

**sequentiell** Sequentielle Prozesse bestehen wie reaktive Prozesse aus einer Abfolge von Phasen, gesteuert durch Ereignisse. Sie werden jedoch strukturiert in Aktionen, die selbsttätig terminieren und hintereinander ausgeführt werden. Die Ausführung einer Aktion entspricht dem Verweilen in einer Phase bis zu einem Ereignis. Die Ereignisse werden in atomaren Aktionen gekapselt und sind im Kontrollfluß nicht sichtbar.

Sequentielle Prozesse, wie sie hier modelliert wurden, sind unmittelbar auf die Elemente reaktiver Prozesse aufgesetzt: Ato-

mare Aktionen kapseln je eine Phase, die das Verhalten während der Aktionsausführung bestimmt, und ein Ereignis, das die Terminierung der Aktion bestimmt. Kontrollstrukturen, die der strukturierten Programmierung entlehnt sind, sequenzieren Aktionen, wobei Signale zu Bedingungen ausgewertet werden. Auf diese Weise zusammengesetzte Aktionen stellen *Prozeduren* dar.

Die drei Verhaltensmodelle sind durch die miteinander verschränkten Konzepte so integriert, daß sie Verhaltensabstraktion ermöglichen (siehe Kapitel 3.2 und 5.3).

Die schrittweise modellierten Konzepte bilden ein zusammenhängenden semantisches Modell. Es bildet die Grundlage für die Semantik der Programmiersprache FSPL (siehe Anhang B zur formalen Definition der Semantik). Der Entwurf der Sprache FSPL wird im folgenden Kapitel beschrieben. Sie setzt die in diesem Kapitel entworfenen Beschreibungsmittel syntaktisch um und baut sie mit allgemeinen programmiersprachlichen Konzepten zu einer vollständigen Implementierungssprache zur Anwendungsprogrammierung von eingebetteten Echtzeitsystemen aus.

## 6.8 Literatur

Die Modellierung der Zeit durch  $\mathbb{R}$  (kontinuierlich) und  $\mathbb{Z}$  (diskret) bzw.  $\mathbb{R}_+$  und  $\mathbb{N}$  und von Signalen als Funktionen der Zeit ist üblich in der Signal- und Systemtheorie (siehe z. B. [KJ02, Rup93, BSMM01]). Die Modellierung von Signalflüssen macht Anleihen bei Simulationssprachen für kontinuierliche Systeme und diskrete Systeme, nicht zuletzt Simulink [The06d], und der Programmierung mit Strömen in funktionalen Sprachen (siehe z. B. [Har01]).

Die Modellierung von Ereignissen und reaktiven Prozessen ist stark beeinflußt von FRAN [EH97] und orientiert sich an Statecharts [Har87]. In der auch in den möglichen Verknüpfungen zum Ausdruck kommenden Semantik von Ereignissen differenziert sich diese Arbeit von dem Ereignismodell in Statemate [HP99] und anderen Sprachen, die auf Basis der diskreten Zeit Ereignisse als Variante boolescher Signale behandeln und auch die Abwesenheit von Ereignissen zu einem Zeitpunkt als Ereignis zulassen. Zu den in dieser Arbeit beschriebenen Ereignis-Operatoren ähnliche wurden unabhängig auch in anderen Arbeiten (siehe z. B. [CL04]) entdeckt. Die Interpretation von Flanken auf

booleschen Signalen (der Trigger-Operator) ist beeinflusst von Ereignissen in Simulink [The06d].

Die Verbindung zwischen Zuständen eines Automaten und zugehörigem dynamischen Verhalten ist vergleichbar mit, wenn auch anders als bei, hybriden Automaten [Hen96] (siehe auch [MR03]): Hier (wie auch bei FRAN) schalten Transitionen im Automaten zwischen Signalen um, in hybriden Automaten zwischen Differentialgleichungssystemen (oder allgemeineren Prädikaten über kontinuierlichen Variablen und ihren Ableitungen nach der Zeit). Der Begriff des Phasenübergangssystems (Phase Transition System) wird auch von [MMP92] in einer ähnlichen Bedeutung wie für hybride Automaten verwendet.

Die Kontrollstrukturen für sequentielle Prozesse orientieren sich an den in imperativen Sprachen üblicherweise verfügbaren. Die Konstrukte zur Parallelität und Ausnahmebehandlung für sequentielle Prozesse sind beeinflusst von Esterel [Ber00].



# 7. Die Programmiersprache FSPL

## 7.1 Übersicht

Die in dieser Arbeit entwickelte höhere Programmiersprache FSPL (Functional Synchronous Programming Language) ist auf das spezielle Anwendungsgebiet „Eingebettete Echtzeitsysteme“ abgerichtet. Durch die Modellbildung in Kapitel 6 wurden das Anwendungsgebiet erschlossen und spezielle Beschreibungsmittel dafür entwickelt. Die Umsetzung und den Ausbau dieser Beschreibungsmittel zu einer vollständigen Implementierungssprache beschreibt das vorliegende Kapitel.

Ausgehend von der Pragmatik der Software-Entwicklung werden die speziellen semantischen Konzepte des Anwendungsgebiets um allgemeine Programmiersprachkonzepte ergänzt. Im weiteren werden die Sprachelemente entworfen. Dabei stellt dieses Kapitel nur die Konzeption und die Grundzüge des Entwurfs dar; die Notationen und ihre Bedeutung werden informell eingeführt. Die vollständige Detaillierung und die formale Definition der Sprache (Syntax und Semantik) sind in Anhang B beschrieben.

Im formalen Sprachentwurf hat FSPL eine textuelle Syntax. Eine graphische Alternative für einen Teil der Beschreibungsmittel bilden die graphischen Notationen aus der Modellbildung (Kapitel 6), die hier weitergeführt und informell neben die textuellen Notationen gestellt

werden. Werkzeugimplementierungen der Sprache können sich für eine graphische Eingabe- und Präsentationssyntax entscheiden.

Der Aufbau des Kapitels ist wie folgt: Abschnitt 7.2 stellt die Architektur von FSPL dar. Dazu gehört als Hauptmerkmal die Aufteilung in eine Objektsprache und eine Metasprache. Die Objektsprache schließt an die Modellbildung für das Anwendungsgebiet an, die Metasprache bringt programmiersprachliche allgemeine Beschreibungskonzepte hinzu. Objektsprache und Metasprache sind Gegenstand der Abschnitte 7.3 bzw. 7.4. Abschließend folgen in Abschnitt 7.5 Hinweise zur Implementierung der Sprache.

## 7.2 Spracharchitektur

### *Objektsprache*

Die entworfene Programmiersprache dient dazu, das Verhalten eingebetteter Echtzeitsysteme konstruktiv zu beschreiben. Gegenstand der Sprache sind daher Zustände (Werte), Prozesse (Signale, Variablen), Ereignisse, Phasen und Aktionen, wie sie in Kapitel 6 modelliert wurden. Sie muß daher Beschreibungsmittel für dieserart Objekte enthalten: die *Objektsprache*.

### *Metasprache*

Zur systematischen Definition von Objekten, die Gegenstand der Objektsprache sind, ist allerdings eine *Metasprache* unerlässlich. Die Lösung von Teilproblemen zur Komposition und Wiederverwendung von Teillösungen, die Bildung von Abstraktionen zur Trennung von Problemebenen (beides hierarchisch angewendet) und die Überführung der Komponenten und Abstraktionen in einen modularen Aufbau von Systemen setzen Mechanismen zur Definition, Parametrierung, Kapselung usw. voraus (siehe auch Kapitel 2.2.2). Gegenstand der Metasprache sind deshalb *Namen* und ihre Bindung an Objekte bzw. deren Beschreibungen (in der Objektsprache) sowie die Verwendung und Gültigkeit von Namen in Beschreibungen.

Namen sind Hilfsmittel zur Trennung und Zusammenführung von Beschreibungen. Bei der *Programmierung* werden Objekte getrennt beschrieben und mit gegenseitig verwendeten Namen versehen. Die *Kompilierung* eines Programms führt die Beschreibungen zusammen und löst die Namen auf. Die Kompilierung interpretiert also den metasprachlichen Anteil eines Programms. Sie ist normalerweise mit einer *Übersetzung* verbunden, die den objektsprachlichen Anteil interpretiert. Nur so können zyklisch voneinander abhängige Beschreibungen (Rekursion) überhaupt aufgelöst werden. Eine aus organisatorischen Gründen häufig gewünschte *getrennte Übersetzung von Modulen*

macht weiterhin eine Zweistufigkeit der Kompilierung erforderlich, da zunächst nur Namen innerhalb eines Moduls aufgelöst werden können. Das Resultat der Übersetzung, der *Objektcode*, enthält dann noch einen metasprachlichen Anteil. Die zweite Stufe, bei der der Objektcode der Programmmodule zusammengeführt wird, wird als *Bindung* bezeichnet. Der metasprachliche Anteil eines Programms erfüllt seinen Zweck also bei der Kompilierung (bzw. Bindung), während der objektsprachliche Anteil des Beschriebenen seine Entsprechung in *Laufzeitobjekten* hat.

Die Metasprache stellt sozusagen eine allgemeine „Infrastruktur“ zur Diskussion von speziellen Objekten bereit, für die die Objektsprache Beschreibungsmittel liefert. Sie entspricht in dieser Hinsicht der Dissertationssprache (bzw. ihrem formalen Anteil, der Sprache der Mathematik) und übernimmt daher hinsichtlich der semantischen Konzepte einige grundlegende Definitionen unmittelbar aus Anhang A. Beide Teilsprachen – Objektsprache und Metasprache – fügen sich zu einer zusammenhängenden Programmiersprache zusammen, wobei die Grenze in Einzelfällen verschwimmt, auf die dann speziell hingewiesen wird.

Die Metasprache nimmt mit  $\lambda$ -Kalkül, Typsystem und Modulsystem starke Anleihen bei *funktionalen* Programmiersprachen, die in ihrer Konzeption anerkanntermaßen sehr fundiert sind und hervorragende Abstraktionsmechanismen bieten. Angewendet auf vordefinierte oder speziell konstruierte Typen, Konstanten und Funktionen entsteht aus der Metasprache eine funktionale Programmiersprache. Auf diese Weise kommen Meta- und Objektsprache zusammen.

*Verbindung von  
Objekt- und  
Metasprache*

Die Beschreibungsmittel der Objektsprache zerfallen in zeitunabhängige und zeitbehaftete. *Daten*, die einen Systemzustand zu einem gegebenen Zeitpunkt kennzeichnen (Prozeßzustände bzw. Werte von Signalen bzw. Variablen), sowie *Berechnungen*, die bei Zustandsübergängen auftreten, werden durch Beschreibungsmittel der ersten Kategorie beschrieben; Berechnungen dieser Art finden aus Sicht der zeitbehafteten Beschreibungsmittel in Nullzeit statt. Diese ihrerseits dienen der Beschreibung des Verhaltens eines Systems über die Zeit hinweg.

Vordefinierte primitive Datentypen mit zugehörigen Operationen bieten unter Einsatz der Metasprache ausreichende Möglichkeiten zur Beschreibung von Daten und Berechnungen. Dieser Teil der Objektsprache zusammen mit der Metasprache entspricht ungefähr der Mächtigkeit einer rein funktionalen Programmiersprache herkömmli-

*funktionale  
Basissprache*

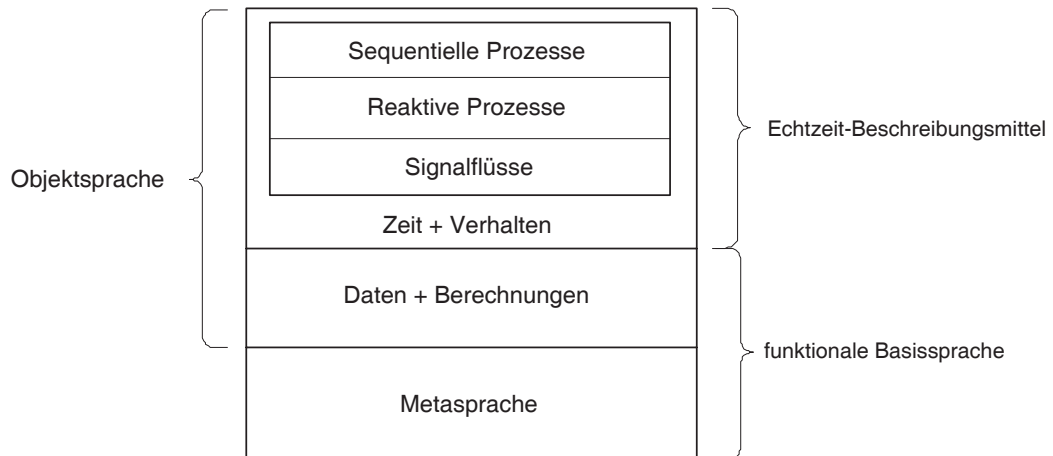


Abbildung 7.1: Struktur der Beschreibungsmittel

chen Zuschnitts. Beide zusammen bilden einen in sich abschließbaren Kern der gesamten Programmiersprache, dessen Semantik ohne einen Zeitbegriff auskommt (*funktionale Basissprache*).

#### *Echtzeit-Beschreibungsmittel*

Dem gegenüber befindet sich der zweite Teil der Objektsprache, der die Konzepte aus Kapitel 6 umsetzt und entsprechend in die Bereiche *Signalflüsse*, *reaktive Prozesse* und *sequentielle Prozesse*, umfaßt von *Zeit und Verhalten*, untergliedert werden kann. Er setzt Beschreibungsmittel für die zeitliche Dimension auf die Datenbeschreibungsmittel auf und stattet so die Programmiersprache zur Beschreibung von Echtzeitsystemen aus (*Echtzeit-Beschreibungsmittel*). Auch hier finden zur Systembildung Mechanismen der Metasprache Anwendung.

Bild 7.1 faßt die Struktur der Beschreibungsmittel zusammen. Die primäre Unterteilung in Objekt- und Metasprache liefert die Gliederung in diesem Kapitel. Die alternative Zusammenfassung in funktionale Basissprache und Echtzeit-Beschreibungsmittel liegt der Systematik in Anhang B zugrunde.

## 7.3 Objektsprache

#### *Ausdrücke und Werte*

Die Ausdrücke der Objektsprache repräsentieren sowohl Objekte statischer Natur (Daten, ohne zeitliche Dimension) als auch dynamische Objekte (Verhalten, mit Bezug zur Zeit). Zur ersten Gruppe gehören Prozeßzustände/Signalwerte/Variablenwerte sowie Zeitpunkte und

Zeitintervalle. Zur zweiten Gruppe gehören Prozesse, Ereignisse, Phasen und Aktionen. In der formalen Semantik der Sprache sind es mathematische Objekte (Werte), wobei die der zweite Gruppe stets Funktionen sind.

Die Menge aller in der Objektsprache darstellbaren Werte zerfällt in Untermengen gleichartiger Objekte, auf die bestimmte Operationen einheitlich anwendbar sind. Eine solche Menge, die mit einer oder mehreren Operationen, die einheitlich auf alle Elemente der Menge anwendbar sind, ausgestattet sind, werden als *Typen* bezeichnet. Ein Wert  $v$  oder ein Ausdruck  $E$  *hat den Typ* oder *ist vom Typ*  $T$ , wenn  $v \in T$  bzw. der durch  $E$  bezeichnete Wert in  $T$  liegt. (vgl. [Wat04])

Die auftretenden Typen können innerhalb der Programmiersprache selbst bezeichnet werden. Die Typsprache ist Teil der Metasprache (siehe Abschnitt 7.4.5). Dennoch werden bei der nachfolgenden Einführung der Konstrukte der Objektsprache auch stets die jeweiligen Typen mit eingeführt.

### 7.3.1 Daten und Berechnungen

*Datentypen* innerhalb der Menge der Typen dienen der Repräsentation von *statischer* Information, im vorliegenden Fall insbesondere von einzelnen Prozeßzuständen (Signalwerten, Variablenwerten), Konstanten in der Signalverarbeitung (wie Verstärkungsfaktoren) und Ähnlichem. *Datentypen* stellen diejenigen Mengen dar, die in der Modellbildung als *Zustandsmengen* bezeichnet wurden (siehe Abschnitt 6.3.2).

*Primitive Datentypen* fassen die für das Anwendungsgebiet elementarsten Wertemengen und ihre Verknüpfungen zusammen, im vorliegenden Fall (in Übereinstimmung mit den meisten Programmiersprachen, siehe [Wat04, Kap. 2]) Wahrheitswerte, Zahlen und Schriftzeichen; hinzu kommt der primitive Datentyp „Zeit“ in Abschnitt 7.3.2.

Das Typsystem von FSPL ermöglicht es, daraus zusammengesetzte Datentypen zu konstruieren, in denen einzelne Daten zu *Datenstrukturen* zusammengefaßt sind (siehe Abschnitt 7.4.5.2). Als Anwendung des Modulkonzepts der Metasprache (siehe Abschnitt B.2.11) wird schließlich die Definition von *abstrakten Datentypen* (ADT) ermöglicht. Sie erlauben es, Datenstrukturtypen (oder auch speziell verwendete primitive Datentypen) und dafür selbstdefinierte Operationen (Konstanten und Funktionen) zu „neuen“ Datentypen zusammenzufassen und einzukapseln, deren „interne“ Werterepräsentation bei der Verwendung der Typen verborgen bleibt.

*Berechnungen*

Ausdrücke, die als Typ einen Datentyp haben, beschreiben entweder konstante Daten, die zur Kompilierzeit feststehen, oder *Berechnungen*, die erst zur Laufzeit ausgeführt werden. Der Abstraktionsmechanismus „Funktion“, den die Metasprache bereitstellt (siehe Abschnitt 7.4.4), spielt dabei eine wesentliche Rolle; Berechnungen können damit in der aus der Mathematik bekannten Weise verallgemeinert und als Berechnungsschemata wiederverwendet werden.

**7.3.1.1 Wahrheitswerte (Booleans)**

Als erster der primitiven Datentypen wird für Wahrheitswerte ein Typ

Boolean

mit den Konstanten

true und false

sowie den Operatoren

! (Negation),  
 && (Konjunktion),  
 || (Disjunktion),  
 == (Äquivalenz) und  
 != (Exklusiv-Oder)

eingeführt (siehe Anhang B.2.4). Die Syntax der Operatoren ist von der Sprache C übernommen.

Ein für alle Typen überladener Operator für bedingte Ausdrücke kommt hinzu: Die Verknüpfung zweier Ausdrücke  $x_1$  und  $x_2$  des gleichen Typs mit einem booleschen Ausdruck  $x_0$  zu

if  $x_0$  then  $x_1$  else  $x_2$

wird zu dem Wert desjenigen Ausdrucks ausgewertet, auf den der Wert des booleschen Ausdrucks zeigt. Die entsprechende Notation in C wäre

$x_0 ? x_1 : x_2$ .

Der if-then-else-Operator ist nicht nur für Datentypen einsetzbar, sondern allgemein für beliebige Typen. Boolesche Konstanten in bedingten Ausdrücken können so z.B. auch als Schalter verwendet werden, die zur Kompilierzeit ausgewertet werden – vergleichbar mit

`#ifdef-#elsif-#endif`-Konstrukten in C, die häufig zur Variantenbildung eingesetzt werden. Der if-then-else-Operator überschreitet somit die Grenze zwischen Objekt- und Metasprache. Mit den in Abschnitt 7.4.5.2 eingeführten Variantentypen kann diese Art der Variantenbildung noch allgemeiner gehandhabt werden.

### 7.3.1.2 Zahlen

Zur Repräsentation von Quantitäten und physikalischen Größen sind *Zahlen* unumgänglich. Als Standardmodelle der Mathematik dienen die Mengen  $\mathbb{N}$  oder  $\mathbb{Z}$  der natürlichen bzw. ganzen Zahlen (oder Teilmengen davon wie z. B.  $0 \dots 9$ ) zur Modellierung *diskreter*, die Mengen  $\mathbb{R}$  oder  $\mathbb{C}$  der reellen bzw. komplexen Zahlen (oder Teilmengen davon wie  $[0, \pi]$ , wobei mit  $\pi$  die Kreiszahl gemeint sei) zur Modellierung *kontinuierlicher* Wertebereiche.

Die von eingebetteten Systemen verarbeiteten Wertebereiche unterliegen grundsätzlich der Beschränkung auf *endliche* Zahlenrepräsentationen im Rechner. Für Ganzzahlen ergibt sich dadurch eine Beschränkung auf endliche Aufzählungsbereiche. Sie liegen üblicherweise entweder als *vorzeichenlose* Zahlen, mit 0 beginnend, im Nichtnegativen oder als *vorzeichenbehaftet* je zur Hälfte im negativen und im nichtnegativen Bereich unter Einschluß der Null (bei Binärkodierung in Zweierkomplementdarstellung). Üblich sind vorzeichenlose und vorzeichenbehaftete 8-, 16- und 32-Bit-Zahlen mit insgesamt  $2^8$ ,  $2^{16}$  bzw.  $2^{32}$  Zahlenwerten.

*beschränkte  
Wertebereiche*

Für kontinuierliche Wertebereiche ergibt sich – analog zur Zeit im vorigen Abschnitt – zusätzlich zur Wertebereichseinschränkung eine Beschränkung der Auflösung und dadurch eine Diskretisierung.  $\mathbb{R}$  wird im Rechner entweder durch Gleitkommazahlen sogenannter einfacher oder doppelter Genauigkeit<sup>1</sup> oder durch Festkommazahlen approximiert. Festkommazahlen sind im wesentlichen Ganzzahlen in Verbindung mit einem festen Skalierungsfaktor, dessen Betrag kleiner als 1 sein kann.

*beschränkte  
Auflösung kon-  
tinuierlicher  
Wertebereiche*

Daß die Repräsentation im Rechner nur endlich viele (rationale) Zahlen zuläßt, verträgt sich damit, daß auch die Wertebereiche der ge-

<sup>1</sup>Nach dem IEEE-Standard 754-1985 [IEE85] sind dies Zahlen  $\pm 1.m \cdot 2^e$ , wobei für die Mantisse  $m$  23 bzw. 52 Bit (vorzeichenlos) und für den Exponenten  $e$  des Skalierungsfaktors 8 bzw. 11 Bit (vorzeichenbehaftet) zur Verfügung stehen. Mit dem Vorzeichen für die gesamte Zahl kommt man damit auf 32 bzw. 64 Bit insgesamt zur Zahlenrepräsentation.

steuerten technischen Prozesse (bzw. der Eingangssignale) in der Regel durch physikalische Gegebenheiten beschränkt sind und nur bis zu einer bestimmten Genauigkeit erfaßt werden können. Nach [Kop97, S. 203] ist die Genauigkeit eines Analogsignals ohne besondere Vorkehrungen durch elektrisches Rauschen auch unter günstigen Bedingungen derart beschränkt, daß 16-Bit-Festkommazahlen in vielen Fällen zur Repräsentation mehr als ausreichend sind.

Im Sinne der Einfachheit der Sprachdefinition reicht es aus, sich auf Ganzzahl-Arithmetik für vorzeichenlose und vorzeichenbehaftete Ganzzahlen unterschiedlicher Bitbreiten und auf Gleitkomma-Arithmetik doppelter Genauigkeit nach IEEE-Standard in Approximation reeller Zahlen zu beschränken. Festkomma-Arithmetik kann im Bedarfsfall von Hand mittels Ganzzahl-Arithmetik realisiert und in abstrakte Datentypen gekapselt werden; das gleiche gilt für die Repräsentation komplexer Zahlen mittels Paaren von reellen Zahlen. Dazu können standardisierte Modul-Bibliotheken entwickelt werden, die von Entwicklungsumgebungen zur Verfügung gestellt werden. Gleitkomma-Arithmetik wird man bei der Wahl der Datenrepräsentationen in vielen Fällen zu vermeiden suchen, da viele Mikrocontroller auf eine Gleitkomma-Recheneinheit (Floating-Point Unit, FPU) verzichten. Für Steuerungsrechner mit FPU und für Simulationsprogramme ist es jedoch vorteilhaft, wenn die Sprache Gleitkomma-Arithmetik vorsieht.

### Bitfelder

Eine Besonderheit der hardwarenahen Programmierung ist die Interpretation und Manipulation der Binärdarstellung ganzer Zahlen als Bitfelder und Bitmuster. Auf diese Weise werden z. B. die an (z. B. 8, 16 oder 32 Bit breiten) Ein-/Ausgabeports anliegenden, als Ganzzahlen interpretierbaren Werte in die binären Werte einzelner Pins zerlegt. Abstrakte Datentypen bieten einen methodischen Weg, um derartige Codierungen typsicher und wartbar einzusetzen.

Um die Verwendung von Bitfeldern in einer anwendungsgerechten Flexibilität zu ermöglichen, wurde für FSPL die Entwurfsentscheidung getroffen, Ganzzahl-Datentypen mit beliebigen, aber festen Bitbreiten und wahlweise vorzeichenlos oder -behaftet vorzusehen.<sup>2</sup> Die dabei entstehenden, abzählbar unendlich vielen Datentypen werden im Dienste der Typsicherheit (siehe Abschnitt 7.4.5) streng unterschieden:

Integer  $n$  U

<sup>2</sup>Sinnvoll sind dabei nur vorzeichenbehaftete Zahlentypen ab 2 Bit (einschließlich Vorzeichen) und vorzeichenlose ab 1 Bit.



bezeichnet einen vorzeichenlosen Ganzzahl-Datentyp mit  $n$  Bit,

Integer  $n$  S

dagegen einen vorzeichenbehafteten, ebenfalls mit  $n$  Bit, wovon eines für das Vorzeichen verwendet wird. Die Variable  $n$  steht dabei für eine Kardinalzahl. Kardinalzahlen gehören zur Metasprache (siehe Abschnitt 7.4.2) und finden hier eine Grenzüberschreitende Anwendung in der Objektsprache.

Da wie in Abschnitt 7.4.5 beschrieben, alle konstruierbaren Werte genau einen Typ besitzen, müssen auch die Ganzzahlliterale alle nötige Information enthalten, um sie eindeutig einem Datentyp zuordnen zu können. Sie setzen sich daher aus einem Literal für den Zahlwert und einer Typ- bzw. Formatangabe zusammen:

$m$  as Integer  $n$  U

oder in Kurzform

$m \# n$  U      oder       $m \# n$  u

steht für eine vorzeichenlose Zahl mit  $n$  Bit und dem Wert

$$m \bmod 2^n \in 0 \dots 2^n - 1,$$

wobei  $m$  für eine Kardinalzahl steht.

$m$  as Integer  $n$  S

oder kurz

$m \# n$  S      oder       $m \# n$  s

steht dagegen für eine vorzeichenbehaftete Zahl mit  $n$  Bit und dem Wert

$$m \bmod 2^n \in -2^{n-1} \dots 2^{n-1} - 1.$$

Für die arithmetischen Datentypen, also die Ganzzahl- oder *Integer*-Datentypen und der Datentyp für Gleitkomma- oder *Real*-Zahlen, werden die üblichen Rechenoperationen Negation, Addition, Subtraktion, Multiplikation, Division und Divisionsrest (modulo) sowie die Bit-Operationen Negation, Und, Oder und Exklusiv-Oder sowie Schiebeoperationen definiert, wobei die Syntax von C übernommen wird. Hinzu kommen die üblichen Vergleichsoperatoren, ebenfalls in C-Notation (siehe Anhang B.2.4).

*Typ-  
umwandlung*

Das Rechnen mit gemischt auftretenden arithmetischen Datentypen macht Typumwandlungen erforderlich. Diese erfolgen in FSPL stets *explizit*. Die Notation ist

$$x \text{ as } T.$$

Hier wird ein Ausdruck  $x$  zu einem Ausdruck vom Typ  $T$ , vorausgesetzt die Typumwandlung ist definiert. Alle Integer-Typen können in-einander abgebildet werden; Ganzzahlen aller Integer-Typen können in Gleitkommazahlen konvertiert werden. Beispielsweise ist

$$257\#16\mathbf{u} \text{ as Integer}8\mathbf{U}$$

gleichbedeutend mit

$$1\#8\mathbf{u}$$

und

$$1\#8\mathbf{u} \text{ as Real}$$

mit

$$1.0.$$

### 7.3.1.3 Schriftzeichen

*Zeichen*

Auch für eingebettete Systeme ist die Verarbeitung von Schriftzeichen und Zeichenketten relevant, da sie mitunter Benutzerschnittstellen ansteuern, die Text anzeigen. Zeichen bilden daher einen weiteren primitiven Datentyp. Er repräsentiert einen Zeichensatz, der implementierungsabhängig im Detail zu spezifizieren ist (z. B. den Zeichensatz ISO-Latin-1, der den ASCII-Zeichensatz als Teilmenge enthält). Die Zeichen innerhalb eines Zeichensatzes sind linear angeordnet, weshalb die üblichen Vergleichsoperationen auch für Zeichen definiert werden (siehe Anhang B.2.4).

*Zeichenketten*

*Zeichenketten* sind wie in der Sprache C Arrays (siehe Abschnitt 7.4.5.2) mit Zeichen, wobei die konkrete Syntax dafür eine spezielle Notation (Zeichenketten-Literale) vorsieht. Sie stimmt im wesentlichen mit der in C überein (siehe Anhang B.4).

## 7.3.2 Zeit und Verhalten

In diesem und den nächsten Abschnitten werden die Beschreibungsmittel für zeitbezogenes Verhalten von den semantischen Modellen aus Kapitel 6 in Sprachkonstrukte übersetzt.

### 7.3.2.1 Zeit

Das diskrete Zeitmodell aus Abschnitt 6.3.1 (siehe Spezifikation 6.1) spielt innerhalb der Modellbildung eine wesentliche Rolle als Definitionsbereich von Prozessen (Signalen) als Funktionen der Zeit. Mit der Abstraktion, die Spezifikation 6.2 leistet, werden Prozesse jedoch als abstrakte Objekte behandelt, die durch primitive Prozesse und Prozeßkonstrukturen beschrieben werden. Es ist dabei nirgends erforderlich, unmittelbar auf einen Prozeßzustand zu einem absoluten Zeitpunkt zuzugreifen. Ähnlich ist es bei der Verarbeitung von Ereignissen, in deren Interpretation ebenfalls Funktionen der Zeit auftreten (siehe Spezifikation 6.3). Die Zeitdimension muß hierfür folglich nicht an die Oberfläche der Sprache treten.

Anders sieht es aus bei dem primitiven Prozeß `timeProcess`, der die absolute Zeit als Signalwert liefert, und Ereigniskonstruktoren wie `after`, die Zeitintervalle als konstante Parameter erhalten. Hier werden absolute und relative Zeitwerte als Daten behandelt. In dieser Rolle fließt die Zeit *als Datentyp* in die Sprache ein.

Bei der Abbildung des Zeitmodells in die Sprachdefinition stellt sich die Frage, wie die Auswahl des Zeitrasters, also die Festlegung der Variablen `step` erfolgen soll, für die eine anwendungsspezifische Anpassung an die worst case execution time vorgeschlagen wurde (siehe Kapitel 6.3.1). Um größtmögliche Portierbarkeit der Programme zu gewährleisten (die WCET hängt von der gewählten Ausführungsplattform ab), wird folgendes Vorgehen gewählt:

Alle Zeitangaben erfolgen explizit in Millisekunden, Sekunden, Minuten oder Stunden<sup>3</sup>. Durch die kleinste verfügbare Zeiteinheit (hier 1 ms), wird sprachintern ein minimales Zeitraster festgelegt. Durch geeignete Wahl aller Zeitkonstanten als Vielfache eines Vielfachen von 1 ms wird innerhalb einer Anwendung ein anwendungsspezifisches Zeitraster erzeugt, das auch automatisch durch den Kompilierungsprozeß generiert (als größter gemeinsamer Teiler aller im Programm angegebenen Zeitintervalle) oder überprüft werden kann.

Gegenüber den Operationen für `Time` ergibt sich in der Sprache somit die Änderung, daß an Stelle von `step` und `toTime` ein Konstruktor tritt, der aus einer natürlichen Zahl und einer Zeiteinheit eine Zeitkonstante

<sup>3</sup>Diese Liste der Zeiteinheiten wurde pragmatisch in Anbetracht typischer Echtzeitanwendungen festgelegt. Erweiterungen nach oben oder unten ändern nichts am Prinzip.

erzeugt. Weiterhin dienen auch die Definitionen für  $\text{Time}_+$ ,  $\text{min}_{\text{Time}}$ , und  $\text{max}_{\text{Time}}$  nur internen Zwecken der Modellierung und finden sich nicht in der Sprache wieder. Die Modellierung ergänzt sich wie folgt:

$$\text{step} \stackrel{\text{def}}{=} 0.001 \in \mathbb{R}_+ \quad (7.1)$$

$$\text{step}_{\text{Double}} \stackrel{\text{def}}{=} 0.001 \in \text{Double}. \quad (7.2)$$

Ein neuer Typ

**Time**

ergänzt die primitiven Datentypen (siehe Anhang B.3.1.1). Zeitkonstanten, die stets *Zeitintervalle* quantifizieren, aber auch (relativ zum Nullzeitpunkt) zu absoluten Zeitangaben verwendet werden können, setzen sich aus einer Kardinalzahl und einer Maßangabe,

ms, s, min oder h,

zusammen. Die Funktionen *time* und *tick* des semantischen Modells erweisen sich bei der Sprachdefinition als weitere Fälle von Typumwandlungen, für die sinnrichtig der bereits existierende Operator *as* überladen werden kann. Gleiches gilt für die Rechen- und Vergleichsoperatoren.

### 7.3.2.2 Prozeß

Der nächste Schritt ist die Übertragung von Spezifikation 6.2 (siehe Kapitel 6.3.2). Ist  $X$  der Datentyp für eine Zustandsmenge, so ist

**Process  $X$**

der zugehörige Prozeßtyp (siehe Anhang B.3.1.2). Wie eingangs zu Abschnitt 7.3.2.1 angedeutet, wird  $\text{state}_X$  lediglich innerhalb der Modellbildung verwendet<sup>4</sup>, so daß die Sprache hierfür keine Erweiterung findet.

### 7.3.2.3 Ereignis

Ereignisse (siehe Kapitel 6.3.3) sind wie Prozesse erstklassige Objekte. Sie bilden einen neuen Typ

**Event,**

für den neue Operatoren eingeführt oder existierende überladen werden (siehe Anhang B.3.1.3); sie sind in den Tabellen 7.1 und 7.2 dargestellt.

Syntax	Semantik	Erklärung
<b>never</b>	never	unmögliches Ereignis
<b>after <math>\Delta t</math></b>	after $\Delta t$	relativer Timeout (z. B. Timeout für das Verweilen in einer Phase)
<b>clock <math>\Delta t</math> offset <math>t_0</math></b>	clock $t_0 \Delta t$	periodisches Ereignis (Takt)
<b>trigger <math>x</math></b>	trigger $x$	Trigger auf Zustandsbedingung
<b>if <math>x</math> then <math>e_0</math> else <math>e_1</math></b>	ifElseEvent $x e_0 e_1$	bedingtes Ereignis
<b>no <math>e</math> within <math>\Delta t</math></b>	watchdog $e \Delta t$	Timeout für das regelmäßige Eintreffen eines Ereignisses

Tabelle 7.1: Ereignisgeneratoren

Syntax	Semantik	Erklärung
$e_0 \parallel e_1$	$e_0 \vee e_1$ Event	$e_0$ <b>oder</b> $e_1$
$e_0 \&\& e_1$	$e_0 \wedge e_1$ Event	$e_0$ <b>und</b> $e_1$ in beliebiger Reihenfolge
$e_0 + e_1$	$e_0 + e_1$ Event	<b>erst</b> $e_0$ , <b>dann</b> $e_1$
$e_0 - e_1$	$e_0 - e_1$ Event	$e_0$ , <b>ohne daß</b> $e_1$
$x * e$	$x * e$ Event	$x$ <b>mal</b> $e$
$e \% x$	guard $x e$	$e$ <b>unter der Bedingung</b> $x$

Tabelle 7.2: Ereignisoperatoren

Die Variablen  $\Delta t$ ,  $t_0$ ,  $x$  usw. stehen in dieser informellen Darstellung auf der linken Seite für Ausdrücke der Sprache, auf der rechten Seite für bedeutungsentsprechende mathematische Ausdrücke. Ähnlich wird im Rest des Kapitels vorgegangen. In der formalen Beschreibung im Anhang B wird hierzu eine Semantikfunktion definiert.

*syntaktischer  
Zucker*

Als sogenannten *syntaktischen Zucker* erlaubt die konkrete Syntax von FSPL das Weglassen des offset-Teils des clock-Operators:

$$\text{clock } dt \quad (7.3)$$

wird zu

$$\text{clock } dt \text{ offset } 0 \text{ ms.} \quad (7.4)$$

### 7.3.3 Signalflüsse

Zur Beschreibung von Signalen und signalverarbeitenden Funktionen (siehe Kapitel 6.4) enthält FSPL Konstrukte für primitive Signale, statische und dynamische Systeme (siehe Anhang B.3.2).

Die primitiven Signale  $\text{const}_X x$  werden als

$$\text{const } x$$

dargestellt. Der polymorphen Funktion  $\text{const}_X$  wird somit durch Überladung eines neuen Operators  $\text{const}$  für jeden Typ entsprochen. Analog verhält es sich in den weiteren Fällen, die in Tabelle 7.3 zusammengefaßt sind.

In der Tabelle sind auch Entsprechungen zu den hier eingeführten Sprachkonstrukten für die graphische Notation der Signalflußgraphen nach Kapitel 6.4.1 angegeben. Man beachte gewisse strukturelle Abweichungen zwischen textueller und graphischer Notation: Der zu  $\text{apply } f \text{ to } x$  gehörige Block wird zwar mit der Funktion  $f$  parametrisiert, läßt sich aber auch ohne Festlegung von  $x$  betrachten; der textuelle Operator benötigt beide Argumente; ähnlich verhält es sich mit  $\text{previous } x \text{ initially } x_0$ . Weiterhin gibt es auch keine direkte Entsprechung zu  $\text{zip}$  und  $\text{unzip}$  als Blöcke; stattdessen kombinieren die in Abbildung 7.2 dargestellten Blöcke die Tupel/Record/Array-Bildung (sie-

<sup>4</sup> $X$  ist hierbei die Zustandsmenge selbst, nicht der zugehörige Datentyp. In der informellen Darstellung in diesem Kapitel wird, anders als in der formalen Definition im Anhang B, nicht streng zwischen dem Typ und der dadurch repräsentierten Menge unterschieden.


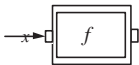
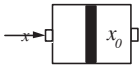
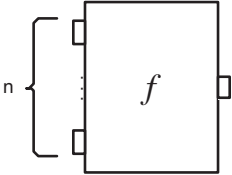
Original-Syntax	graphische Notation	Semantik	Erklärung
<code>const <math>k</math></code>		$\text{const}_X k$	konstantes Signal
<code>time</code>		$\text{timeProcess}$	Zeitsignal
<code>apply <math>f</math> to <math>x</math></code>		$\text{apply}_{X \rightarrow Y} f x$	Funktionslifting und Anwendung
<code>zip <math>x</math></code>		$\text{zip}_{\prod(\lambda i \in I. X_i)}$	Produkttransformation
<code>unzip <math>x</math></code>		$\text{unzip}_{\prod(\lambda i \in I. X_i)}$	Produkttransformation
<code>previous <math>x</math> initially <math>x_0</math></code>		$\text{delay}_X x_0 x$	Signalverschiebung um eine Zeiteinheit
$f$		$f$	$n$ -stellige signalwertige Funktion

Tabelle 7.3: Signalfluß-Beschreibungsmittel

he Abschnitt 7.4.5.2) mit der Produkttransformation und haben deshalb mehrere Ein- bzw. Ausgänge. Die textuellen Operatoren sind einstellig. `zip` und `unzip` verbinden ein Konzept der Metasprache (strukturierte Typen) mit einem Konzept der Objektsprache (Prozesse) und gehören deshalb zum Grenzbereich beider Teilsprachen.

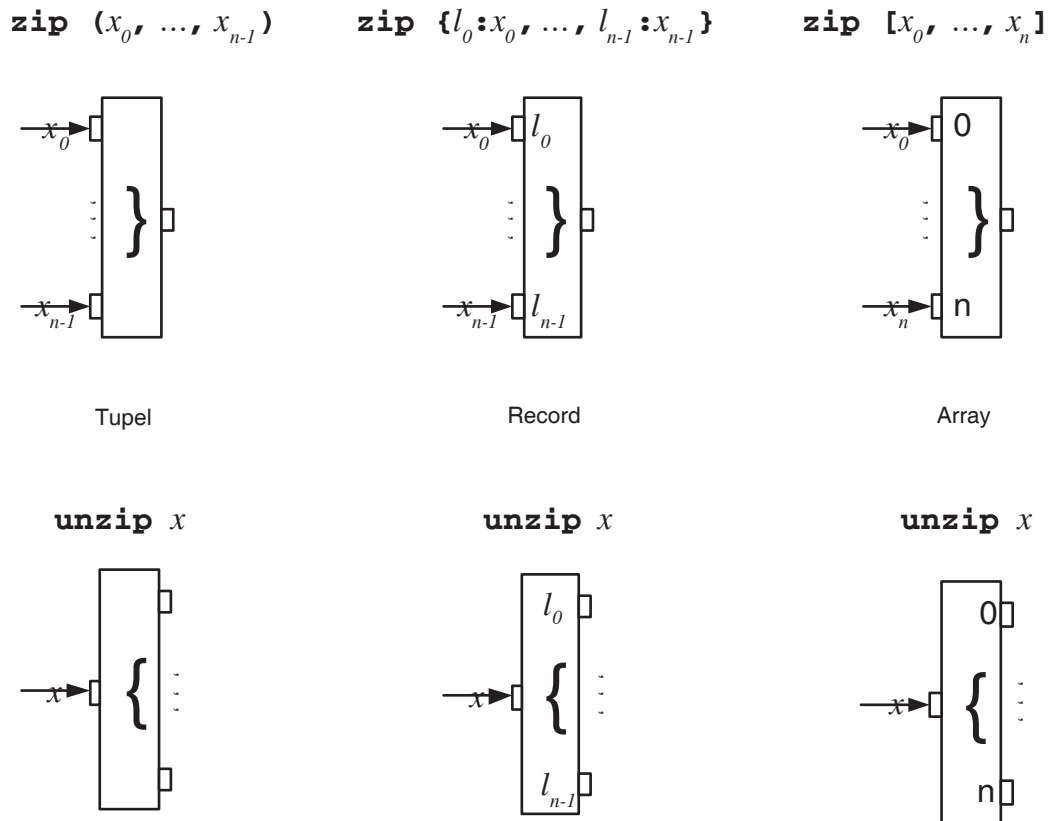


Abbildung 7.2: Graphische Notation für Transformationen von Signalstrukturen

In der letzten Zeile von Tabelle 7.3 wird der allgemeine Fall einer signalwertigen Funktion betrachtet. Im Gegensatz zu den Beispielen in Kapitel 6.4 wird für funktionswertige Ausdrücke mit mehreren Argumenten jetzt stets Currying vorausgesetzt; für Argument-Tupel sind die `zip/unzip`-Blöcke zu verwenden. `time` kann als signalwertiger Ausdruck als nullstellige Funktion betrachtet werden und fällt damit ebenfalls unter die allgemeine Darstellung, bei der der Block mit dem Ausdruck beschriftet wird.



Über die Grundmechanismen zu Bildung statischer Systeme hinaus werden in der textuellen Syntax alle arithmetisch-logischen Operationen und die Typumwandlung auf Signalebene angehoben: In

$$x_0 + x_1$$

steht beispielsweise  $+$  für den mittels  $\text{apply}_{X \rightarrow Y}$  zu einer signalverarbeitenden Operation angehobenen Addition, wenn  $x_0$  und  $x_1$  Signale sind. Auch wenn nur einer der  $x_0, x_1$  ein Signal ist findet ein Lifting statt, wobei ein Signal mit einer Konstanten verknüpft wird. Der gemischte Fall eignet sich besonders für Vergleiche eines Signals mit einem konstanten Schwellwert oder auch für Linearkombinationen von Signalen/Variablen. Die  $\text{gain}_{\mathbb{R}}$ -Funktion entsteht als Spezialfall davon.

Der ebenfalls überladene `if then else`-Operator fungiert als Multiplexer: Zu jedem Zeitpunkt wird abhängig vom booleschen Wert des ersten Signals der aktuelle Wert des zweiten oder dritten Signals „durchgeschaltet“.

### 7.3.4 Reaktive Prozesse

Die Übertragung der semantischen Modelle für Phasenübergangssysteme und reaktive Prozesse (Kapitel 6.5) in die Sprache ist in Tabelle 7.4 dargestellt (siehe auch Anhang B.3.3). Hinzu kommt noch ein Phasentyp

Phase  $X$

für jeden Datentyp  $X$ .

Man beachte, daß `start` als Sprachkonstrukt im Gegensatz zu  $\text{start}_X$  im semantischen Modell lediglich einen Operanden erhält. Der Startzeitpunkt wird auf den Nullzeitpunkt festgelegt. Für alle anderen Zeitpunkte wird  $\text{start}_X$  nur innerhalb von Semantik-Definitionen bzw. der Modellbildung verwendet.

Die beiden `local`-Konstrukte weisen metasprachliche Elemente auf, indem sie in einen abgegrenzten Gültigkeitsbereich neue Bezeichner einführen. Wie die angegebene Semantik zeigt, wird implizit eine  $\lambda$ -Abstraktion gebildet. Die Syntax ist an `let` und `letrec` angelehnt (siehe Abschnitt 7.4.3).

Auch `orthogonalize` verbindet (wie `zip` und `unzip`) ein Konzept der Metasprache (strukturierte Typen) mit einem Konzept der Objektsprache

Original-Syntax	graphische Notation	Semantik	Erklärung
<b>keep</b> $x$		$\text{phase}_X x$	atomare Phase
<b>do</b> $\varphi$ <b>when</b> $e_0$ <b>then</b> ... $\vdots$ <b>when</b> $e_n$ <b>then</b> ...		$\text{switch}_X \varphi [$ $\text{when}_X e_0 (\dots),$ $\vdots$ $\text{when}_X e_n (\dots)$ $]$	Transitionen
<b>when</b> $e$ <b>then</b> $\varphi$		$\text{when}_X e (\text{goto}_X \varphi)$	Transition zu Phase
<b>when</b> $e$ <b>then wait</b>		$\text{when}_X e \text{ wait}_X$	Transition zu Endzustand
<b>local</b> <b>value</b> $x_0 : X := x$ <b>in</b> $\varphi$		$\text{valueInPhase}_{X,Y}$ $x$ $(\lambda x_0 \in X . \varphi)$	Signal- auswertung zu Phasen- beginn
<b>local</b> <b>variable</b> $x : X := \varphi_x$ <b>in</b> $\varphi$		$\text{variableInPhase}_{X,Y}$ $\varphi_x$ $(\lambda x \in \text{Process}_X . \varphi)$	lokale Va- riable
<b>if</b> $x$ <b>then</b> $\varphi_0$ <b>else</b> $\varphi_1$		$\text{ifElsePhase}_X x \varphi_0 \varphi_1$	bedingte Phase
<b>orthogonalize</b> $\varphi$		$\text{orthogonalize}_X \varphi$	Phasen- paralleli- sierung
<b>start</b> $\varphi$		$\text{start}_X \varphi 0_{\text{Time}}$	Starten als Prozeß

Tabelle 7.4: Beschreibungsmittel für reaktive Prozesse

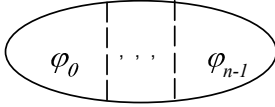
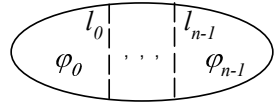
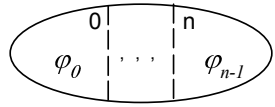
Original-Syntax	graphische Notation	Erklärung
$\text{orthogonalize } (\varphi_0, \dots, \varphi_{n-1})$		Tupel
$\text{orthogonalize } \{ l_0 : \varphi_0, \dots, l_{n-1} : \varphi_{n-1} \}$		Record
$\text{orthogonalize } [\varphi_0, \dots, \varphi_n]$		Array

Tabelle 7.5: Notationen für Phasenparallelisierung

(Phasen). Analog zu `zip` und `unzip` gibt es eine spezielle graphische Notation, die Tupel/Record/Array-Bildung und `orthogonalize` kombiniert (siehe Tabelle 7.5).

Ergänzend zu den Beschreibungsmitteln für Ereignisse aus Abschnitt 7.3.2.3 wird der `local`-Operator auch zur Implementierung von `valueInEventX` und `variableInEventX` überladen (siehe Tabelle 7.6).

Original-Syntax	Semantik	Erklärung
<code>local</code> <code>  value <math>x_0 : X := x</math></code> <code>in</code> <code>  <math>e</math></code>	<code>valueInEvent<sub>X</sub></code> $x$ $(\lambda x_0 \in X . e)$	ereignislokale Auswertung
<code>local</code> <code>  variable <math>x : X := \varphi_x</math></code> <code>in</code> <code>  <math>e</math></code>	<code>variableInEvent<sub>X</sub></code> $\varphi_x$ $(\lambda x \in \text{Process}_X . e)$	ereignislokale Variable

Tabelle 7.6: Syntax für ereignislokale Auswertung und Variable

### 7.3.5 Sequentielle Prozesse

Sequentielle Programmierung (nach Kapitel 6.6) wird durch die in Tabellen 7.7 und 7.8 dargestellten Sprachelemente unterstützt (siehe auch Anhang B.3.4). Hinzu kommt ein Aktionstyp

Action  $X$

für jeden Datentyp  $X$ .

In der textuellen Original-Syntax werden das schon mehrfach überladene `if then else`-Konstrukt, das `local`-Konstrukt und der Multiplikationsoperator auch hier wiederverwendet. Man beachte, daß `terminationX` und `behaviourX` aus Spezifikation 6.23 keine Entsprechung in der Sprache erhalten; sie dienen ausschließlich der Modellbildung und somit der Spezifikation der Semantik.

Analog zu `zip`, `unzip` und `orthogonalize` verbinden `parallelize terminating` und `parallelize waiting` strukturierte Typen aus der Metasprache mit dem Konzept der Aktionen aus der Objektsprache. Analog gibt es auch eine spezielle graphische Notation, die Tupel/Record/Array-Bildung und Parallelisierung kombiniert (siehe Tabelle 7.9).

In Übereinstimmung mit dem Aktionsmodell, das keine Null-Aktionen zuläßt (siehe Kapitel 6.6) stehen die Kontrollstrukturen *Alternative* und *Iteration mit Endbedingung*, nicht aber *Option* und *Iteration mit Anfangsbedingung* zur Verfügung. Als syntaktischer Zucker werden allerdings je eine Variante dieser Kontrollstrukturen derart, daß sie stets die Fortsetzung durch eine weitere Aktion erfordern, eingeführt und wie folgt definiert:

$$\text{if } x \text{ then } a_0 \text{ always } a_1 \quad (7.5)$$

ist eine Abkürzung für

$$\text{if } x \text{ then } (a_0; a_1) \text{ else } a_1 \quad (7.6)$$

und

$$\text{while } x \text{ repeat } a_0 \text{ finally } a_1 \quad (7.7)$$

für

$$\text{if } x \text{ then } ((\text{repeat } a_0 \text{ until } !x); a_1) \text{ else } a_1. \quad (7.8)$$

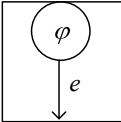
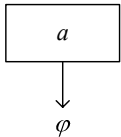
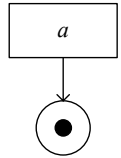
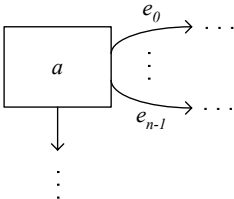
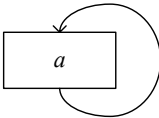
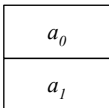
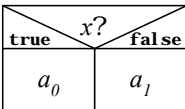
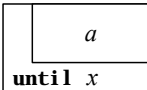
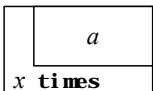
Original-Syntax	graphische Notation	Semantik	Erklärung
<b>do</b> $\varphi$ <b>until</b> $e$		$\text{until}_X e \varphi$	elementare Aktion
<b>complete</b> $a$ <b>then</b> $\varphi$		$\text{continue}_X a (\text{goto}_X \varphi)$	Aktionsvervollständigung
<b>complete</b> $a$ <b>then</b> $\varphi$		$\text{continue}_X a \text{ wait}_X$	Aktionsvervollständigung
<b>complete</b> $a$ <b>then</b> ... <b>except</b> <b>when</b> $e_0$ <b>then</b> ... ... <b>when</b> $e_n$ <b>then</b> ...		$\text{continueWithExceptions}_X$ $a \dots$ $[\text{when}_X e_0 (\dots),$ $\dots$ $\text{when}_X e_n (\dots)]$	Ausnahmebehandlung
<b>loop</b> $a$		$\text{loop}_X a$	Endlosschleife
$a_0 ; a_1$		$\text{sequence}_X a_0 a_1$	Sequenz
<b>if</b> $x$ <b>then</b> $a_0$ <b>else</b> $a_1$		$\text{ifElse}_x x a_0 a_1$	Alternative
<b>repeat</b> $a$ <b>until</b> $x$		$\text{repeatUntil}_X a x$	Iteration mit Endbedingung
$x * a$		$x \quad * \quad a$ $\text{Action}_X$	$x$ -fache Wiederholung

Tabelle 7.7: Beschreibungsmittel für sequentielle Prozesse

Original-Syntax	Semantik	Erklärung
<code>parallelize a</code> <code>terminating</code>	$\text{parallelizeTerminating}_X a$	Parallelisierung, Synchronisation auf erste Terminierung
<code>parallelize a</code> <code>waiting</code>	$\text{parallelizeWaiting}_X a$	Parallelisierung, Synchronisation auf letzte Terminierung
<code>local</code> <code>value <math>x_0 : X := x</math></code> <code>in</code> <code>a</code>	$\text{valueInAction}_{X,Y}$ $x$ $(\lambda x_0 \in X . a)$	aktionslokale Auswertung
<code>local</code> <code>variable <math>x : X := \varphi_x</math></code> <code>in</code> <code>a</code>	$\text{variableInAction}_{X,Y}$ $\varphi_x$ $(\lambda x \in \text{Process}_X . a)$	aktionslokale Variable

Tabelle 7.8: Beschreibungsmittel für sequentielle Prozesse (Forts.)

## 7.4 Metasprache

Die Objektsprache dient zur Beschreibung von Objekten des Anwendungsgebiets. Gegenstände der Objektsprache sind diese Objekte. Die Metasprache hingegen dient zur Handhabung von Objektbeschreibungen. Gegenstände der Metasprache sind solche Beschreibungen. In der Metasprache werden für Beschreibungen bzw. die beschriebenen Objekte Namen vergeben, Beschreibungen werden aufgeteilt oder zusammengefügt, Objekte werden klassifiziert usw.

Die Metasprache ist weitgehend unabhängig von der speziellen Objektsprache. Beide zusammen bilden die spezielle Programmiersprache, deren Kern die Objektsprache ist. Die Metasprache macht die Objektsprache, die einen bestimmten Modellbildungsansatz umsetzt, zu einer vollwertigen Beschreibungssprache.

### 7.4.1 Ausdrücke, Bezeichner und Typen

#### Ausdrücke

Ausdrücke der Objektsprache sind die primitiven Elemente, mit denen die Metasprache operiert – sozusagen Literale für Konstanten aus Sicht der Metasprache. Sie bilden den Kern einer Menge von Ausdrücken der Metasprache (und damit der Gesamtsprache), die in einem


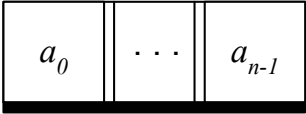
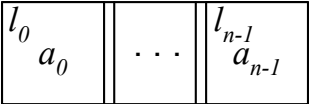
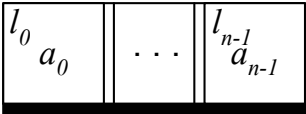
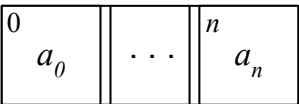
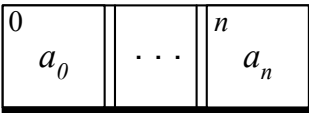
Original-Syntax	graphische Notation	Erklärung
<b>parallelize</b> $(a_0, \dots, a_{n-1})$ terminating		Tupel
<b>parallelize</b> $(a_0, \dots, a_{n-1})$ waiting		Tupel
<b>parallelize</b> $\{l_0 : a_0, \dots, l_{n-1} : a_{n-1}\}$ terminating		Record
<b>parallelize</b> $\{l_0 : a_0, \dots, l_{n-1} : a_{n-1}\}$ waiting		Record
<b>parallelize</b> $[a_0, \dots, a_n]$ terminating		Array
<b>parallelize</b> $[a_0, \dots, a_n]$ waiting		Array

Tabelle 7.9: Notationen für Aktionsparallelisierung

engeren Sinn als *Ausdrücke* verstanden werden. Ausdrücke in diesem Sinn beschreiben Werte; Werte aus Sicht der Metasprache sind Objekte, objektwertige Funktionen, Module usw. Ausdrücke in diesem Sinn beschreiben also Objekte oder enthalten solche Beschreibungen.

#### *Bezeichner*

Eine zweite wichtige Kategorie innerhalb der Objektsprache sind *Bezeichner*, Namen für Ausdrücke. Bezeichner werden vereinbart und verwendet. Die Vereinbarung von Bezeichnern ist immer mit einem bestimmten *Gültigkeitsbereich* verbunden, innerhalb dessen der Bezeichner mit der vereinbarten Bedeutung verwendet werden kann. Ein verwendeter Bezeichner (auch als *Variable* – im Sinn der Metasprache – bezeichnet) ist selbst ein Ausdruck. Bezeichner sind in der konkreten Syntax (siehe Anhang B.4) Zeichenfolgen, bestehend aus Buchstaben, Ziffern und Unterstrichen, beginnend mit einem Kleinbuchstaben; ausgenommen (reserviert) sind alle Wörter, die in objekt- oder metasprachlichen Operatoren auftreten (wie etwa *after* oder *no*).

#### *Typen*

Die Menge der Ausdrücke enthält undifferenziert die gesamte Objektsprache und darauf aufbauende Konstruktionen. Semantische Unterscheidungen zwischen unterschiedlichen Kategorien von Ausdrücken macht die Metasprache durch ein *Typsystem*. Ausdrücken werden dabei eindeutige *Typen* zugeordnet. Ein Bezeichner (aus einer ebenfalls undifferenzierten Mengen von Namen) wird stets mit einem bestimmten Typ vereinbart; er repräsentiert dann einen Ausdruck dieses Typs. Operatoren, die Unterausdrücke zu zusammengesetzten Ausdrücken verknüpfen, müssen stets *typkorrekt* verwendet werden, d. h. die jeweils dargestellte Operation muß auf Elemente der Typen der Unterausdrücke anwendbar sein (vgl. die Einleitung zu Abschnitt 7.3).

Eine ausführlichere Diskussion des Zusammenhangs zwischen Objektsprache, Metasprache und Typisierung sowie syntaktischen und semantischen Kategorien findet sich in Anhang B.1.3 im Rahmen der Beschreibung des Formalisierungskonzepts für die Sprachdefinition. Zur formalen Definition von Ausdrücken und Bezeichnern siehe Anhang B.2.1 und B.4. Typen werden in Abschnitt 7.4.5 und Anhang B.2.2 vertieft.

### **7.4.2 Kardinalzahlen**

Kardinalzahlen bilden eine eigene syntaktische und semantische Kategorie der Metasprachebene mit Anwendungen in der Objektsprache. Sie dienen der Dimensionierung von Arrays und der Projektion aus Tupeln (siehe Abschnitt B.2.8) und werden auch für die sowohl in den



Typnamen für Ganzzahlen als auch in Ganzzahlliteralen oder Schiebeoperationen auftretenden Zahlenwerte verwendet (siehe Abschnitt 7.3.1.2).

Die Kardinalzahlen werden syntaktisch in gewohnter Weise als

0, 1, 2 usw.

dargestellt. In der konkreten Syntax können Kardinalzahlen zusätzlich auch in Hexadezimalschreibweise (in der Notation von C) angegeben werden (siehe Anhang B.4). Man beachte, daß Kardinalzahlen keinen eigenen Datentyp bilden, sondern lediglich die natürlichen Zahlen in der Sprache repräsentieren. Datentypen mit Ganzzahlen sind in FSPL immer endlich und werden mit Hilfe der Kardinalzahlen konstruiert (siehe Abschnitt 7.3.1.2).

### 7.4.3 Deklarationen und Definitionen

Deklarationen und Definitionen führen Bezeichner in bestimmte Kontexte ein. Eine Deklaration vereinbart einen Typ für einen Bezeichner, eine Definition (zusätzlich) einen Wert. Deklarationen werden für die Parameter von Funktionen (siehe Abschnitt 7.4.4) und in Modulschnittstellen (siehe Abschnitt 7.4.7), Definitionen entweder als lokale Definitionen (siehe unten) oder in Modulimplementierungen (siehe Abschnitt 7.4.7) verwendet.

Deklarationen und Definitionen haben eine bestimmte Grundform: Eine Deklaration

$$\text{value } id : T$$

vereinbart einen Bezeichner  $id$  unter dem Typ  $T$ . Eine Definition

$$\text{value } id : T := x$$

vereinbart den Bezeichner unter dem Typ  $T$  und mit dem Wert  $x$ ; die Definition ist dabei nur gültig, wenn  $x$  tatsächlich ein Ausdruck vom Typ  $T$  ist. Da der Typ eines Ausdrucks eindeutig bestimmbar ist, ist die in einer gültigen Definition enthaltene Typangabe redundant, wird aber zum Selbstschutz des Programmierers und zu Dokumentationszwecken von der Sprache erzwungen. Die Typübereinstimmung wird während der Kompilierung des Programms geprüft.

Als syntaktischer Zucker erlaubt FSPL abgekürzte Deklarationen und *syntaktischer Zucker*

Syntax-Variante	Original-Syntax
<b>process</b> $id : T$	<b>value</b> $id : \text{Process } (T)$
<b>signal</b> $id : T$	
<b>variable</b> $id : T$	
<b>event</b> $id$	<b>value</b> $id : \text{Event}$
<b>phase</b> $id : T$	<b>value</b> $id : \text{Phase } (T)$
<b>action</b> $id : T$	<b>value</b> $id : \text{Action } (T)$
<b>process</b> $id : T = x$	<b>value</b> $id : \text{Process } (T) = x$
<b>signal</b> $id : T = x$	
<b>variable</b> $id : T = x$	
<b>event</b> $id = x$	<b>value</b> $id : \text{Event} = x$
<b>phase</b> $id : T = x$	<b>value</b> $id : \text{Phase } (T) = x$
<b>action</b> $id : T = x$	<b>value</b> $id : \text{Action } (T) = x$

Tabelle 7.10: Syntax-Varianten für Deklarationen und Definitionen

Definitionen für die Typen der Echtzeit-Beschreibungsmittel der Objektsprache (Tabelle 7.10).

Die alternativen Schlüsselwörter `process`, `signal` und `variable` besitzen keinen semantischen Unterschied, geben aber dem Programmierer eine zusätzliche Möglichkeit zur Klassifikation im Sinne der Pragmatik (vgl. Kapitel 6.3.2). Die Wahlmöglichkeit zwischen `process`, `signal` und `variable` dient also zu Dokumentationszwecken.

Außer den in Tabelle 7.10 dargestellten, werden für Funktionsdeklarationen und -definitionen in Abschnitt 7.4.4 weitere syntaktische Varianten eingeführt. Formal definiert werden Deklarationen und Definitionen in Anhang B.2.5 bzw. B.2.7.

#### lokale Definitionen

FSPL kennt zwei Arten, wie Definitionen eingeführt werden: gekapselt in Modulen (siehe Abschnitt 7.4.7) oder aber lokal zu einzelnen Ausdrücken. Lokale Definitionen sind hinsichtlich ihres Gültigkeitsbereiches vergleichbar mit lokalen Variablen in einem Anweisungsblock in der Sprache C. Sie sind Sätzen wie

„Sei  $pi \in \mathbb{R}$  mit  $pi = 3.141$ . ...“

in mathematischen Ausführungen nachempfunden und lesen sich dann im Vergleich dazu so:

`let value pi : Real = 3.141 in ...`

Das `let`-Konstrukt wird in einer verallgemeinerten Form eingeführt, die mehrere gleichzeitige Definitionen, parallel oder wechselseitig rekursiv, erlaubt. Die beiden Varianten – mit und ohne Rekursion – werden syntaktisch dadurch unterschieden, daß im Fall von Rekursion das Schlüsselwort `let` durch `letrec` ersetzt wird (siehe Tabelle 7.11 und Anhang B.2.7). Ohne Rekursion ist der Gültigkeitsbereich der Definitionen bzw. der durch sie eingeführten Bezeichner auf den Ausdruck beschränkt, der auf `in` folgt; mit Rekursion erstreckt sich der Gültigkeitsbereich auf den gesamten `letrec`-Ausdruck, so daß ein Bezeichner sowohl in seiner eigenen Definition als auch in anderen der Definitionen als Variable verwendet werden kann.

Syntax	Erklärung
<code>let</code> Definition : Definition <code>in</code> Ausdruck	parallele lokale Definitionen
<code>letrec</code> Definition : Definition <code>in</code> Ausdruck	rekursive lokale Definitionen

Tabelle 7.11: Syntax lokaler Definitionen

Rekursive Definitionen sind grundsätzlich nur auf Funktionen anwendbar. Neben Funktionen allgemeiner Art, wie sie nach Abschnitt 7.4.4 in FSPL beschrieben werden können, gehören dazu auch alle speziellen Objekte wie Signale und Phasen, die in Kapitel 6 als Funktionen modelliert wurden. Rückkopplungen in Signalflüssen und Zyklen in Phasenübergangssystemen werden durch rekursive Definitionen darstellbar. Siehe dazu auch die Diskussion der Rekursionsproblematik in Anhang B.2.7.

Als Beispiel kann etwa die rekursive Definition nach Abbildung 6.8 des Phasenübergangssystems nach Abbildung 6.9 in der Syntax von FSPL wie folgt ausgedrückt werden:

```

letrec
  phase  $node_0 : X =$ 
    do  $\varphi_0$ 
      when  $e_1$  then  $node_1$ 
      when  $e_2$  then  $node_2$ 
      when  $e_0$  then  $node_0$ 
  phase  $node_1 : X =$ 
    do  $\varphi_1$ 
      when  $e_3$  then  $node_0$ 
  phase  $node_2 : X =$ 
    do  $\varphi_2$ 
      when  $e_4$  then  $node_0$ 
in
   $node_0$ 

```

In Abbildung 6.8 ist die initiale Phase des Phasenübergangssystems durch einen besonderen Pfeil gekennzeichnet. Dem entspricht im letrec-Ausdruck das

```

in
   $node_0$ .

```

Durch das Phasenübergangssystem wird insgesamt eine Phase beschrieben. Sie zerfällt in drei Unterphasen mit gegenseitigen Transitionen, die in lokalen Definitionen beschrieben werden. Die Gesamtphase beginnt mit der ausgewählten Unterphase.

So kann also die graphische Notation um einen Spezialfall für Phasen in Verbindung mit letrec erweitert werden (siehe Tabelle 7.12); hierbei liegt eine weitere Grenzüberschreitung zwischen Meta- und Objektsprache vor, die eine Spezialität der graphischen Notation ist. Man beachte auch, daß die Notation in Tabelle 7.12 im Gegensatz zu Abbildung 6.8 zusätzlich die Knotennamen enthält (hier  $l$ ).

#### 7.4.4 Funktionen

*Funktionen als  
Werte*

Funktionen sind in funktionalen Programmiersprachen „erstklassige“ Werte, d. h. sie können als Werte in Ausdrücken und so auch als Parameter oder als Rückgabewerte von Funktionen auftreten. Funktionen,

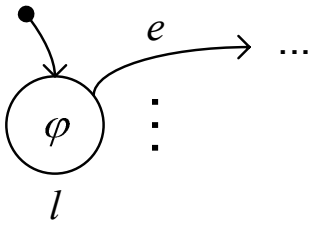
graphische Notation	Original-Syntax
	<pre> letrec   ⋮   phase l : X =     do φ       when e then ...     ⋮   in     l </pre>

Tabelle 7.12: Notation für Phasenübergangssysteme

die Funktionen verarbeiten, heißen auch *Funktionen höherer Ordnung* (oder *Funktionale*). Funktionszeiger in C bilden eine gewisse Annäherung an dieses Konzept, da diese ebenfalls als Parameter an Funktionen übergeben werden können. Eine wesentliche Einschränkung in ihrer Mächtigkeit besteht jedoch darin, daß in C Funktionen stets mit Namen definiert werden müssen, was lokal in einem Ausdruck in C nicht möglich ist; Funktionszeiger zeigen dann auf solche, an globaler Stelle definierten Funktionen. Die freie, namenlose Beschreibung einer Funktion in einem lokalen Kontext, etwa als Ergebnis einer Berechnung und Rückgabewert einer anderen Funktion, ist nicht möglich.

Genau solche namenlosen Funktionsausdrücke ermöglicht die  $\lambda$ -Abstraktion (siehe Anhang A.1.4). So bezeichnet etwa der Funktionsausdruck

 $\lambda$ -Abstraktion

$$\lambda x \in \mathbb{Z}. x^2 \quad (7.9)$$

diejenige Funktion, die ganze Zahlen quadriert und in der Schulmathematik üblicherweise mit

$$x \mapsto x^2 \quad (7.10)$$

notiert oder als benannte Funktion

$$f(x) = x^2 \quad (7.11)$$

in Gleichungsform vereinbart würde. In den beiden zuletzt genannten Notationen fehlt eine Spezifikation des Definitionsbereichs, die üblicherweise durch Angabe einer Funktionalität, hier z. B.

$$f : \mathbb{Z} \rightarrow \mathbb{Z}, \quad (7.12)$$

ergänzt würde. Im Gegensatz zu getypten  $\lambda$ -Ausdrücken wie (7.9) kennt die Mathematik auch ungetypte wie

$$\lambda x. x^2,$$

der (7.10) entspricht. Die getypte Variante entspricht dem hier gewählten Typkonzept.

Die Variable  $x$  in den Beispielen wird auch *Abstraktionsvariable* genannt. Sie ist ein Bezeichner, der durch den  $\lambda$ -Operator innerhalb des  $\lambda$ -Ausdrucks gebunden wird. An dieser Stelle wird das Konzept der Deklaration (siehe Abschnitt 7.4.3) verwendet, um den Bezeichner unter einem bestimmten Typ einzuführen; der Gültigkeitsbereich dieser Deklaration ist der Funktionsrumpf, d. h. der Ausdruck, der in der Notation von (7.9) dem Punkt folgt.

*Abstraktion  
und  
Applikation*

Die in Anhang A.1.4 definierte getypte  $\lambda$ -Abstraktion wird nun direkt in die Metasprache übernommen. Die Syntax verwendet als Notation eine dem Funktionsrumpf vorangestellte geklammerte Parameterdeklaration. Das Gegenstück zur Abstraktion ist die *Applikation*, die Funktionsanwendung auf ein Argument, wodurch der Parameter (die Abstraktionsvariable) an einen Wert bzw. Ausdruck gebunden wird. Die Notation folgt Anhang A.1.4. Syntax und Semantik sind informell in Tabelle 7.13 dargestellt; zur formalen Definition siehe Anhang B.2.6.

Syntax	Semantik	Erklärung
$(\text{value } id : T) \ x$	$\lambda id \in T. x$	Abstraktion
$f \ x$	$f \ x$	Applikation

Tabelle 7.13: Syntax für Funktionen und Funktionsanwendung

*Funktionstypen*

Zur Typisierung von  $\lambda$ -Ausdrücken werden *Funktionstypen* eingeführt, die den Funktionsmengen aus Anhang A.1.8 entsprechen. Funktionstypen haben die Form

$$T_D \rightarrow T_R,$$

wobei  $T_D$  der Typ für den Definitionsbereich,  $T_R$  der für den Wertebereich ist.

Funktionen der so beschreibbaren Art sind einstellig, d. h. sie erhalten bei der Anwendung genau ein Argument. Wie in Anhang A.1.9 für die Sprache der Mathematik beschrieben, können mehrstellige

Funktionen als Funktionen höherer Ordnung oder mehrere Argumente als Tupel-Argument dargestellt werden. Im ersten Fall entsteht eine Funktion

$$f \in X_0 \rightarrow \cdots \rightarrow X_n \rightarrow X,$$

die nach dem Schema

$$f \ x_0 \ \dots \ x_n$$

appliziert wird; dies ist hier bereits unmittelbar anwendbar. Im zweiten Fall entsteht eine Funktion

$$f \in X_0 \times \cdots \times X_n \rightarrow X,$$

die nach dem Schema

$$f \ [x_0, \dots, x_n]$$

angewendet wird; die hierzu nötige Einführung von Produkttypen und Tupeln folgt in Abschnitt 7.4.5.2.

Bei der Definition von Funktionen führt die explizite Typisierung syntaktisch zu einer Redundanz: In einer Definition

*syntaktischer  
Zucker*

$$\text{value } id_1 : T_0 \rightarrow T_1 = (\text{value } id_0 : T_0) \ x$$

tritt der Typ  $T_0$  für den Definitionsbereich der Funktion doppelt auf. Die Definitionen werden schreib- und lesbarer, wenn man diese Redundanz vermeidet. Als syntaktischer Zucker werden deshalb implizite Definitionsgleichungen der Form

$$\text{value } id_1 \ (\text{value } id_0 : T_0) : T_1 = x$$

erlaubt (siehe Tabelle 7.14)<sup>5</sup>. Gleiches gilt für reine Deklarationen, wo auch Funktionsdeklarationen der Form

$$\text{value } id_1 \ (\text{value } id_0 : T_0) : T_1$$

erlaubt werden (der Bezeichner  $id_0$  für die Abstraktionsvariable hat dabei dokumentierenden Charakter ohne formale Bedeutung). Diese syntaktischen Variationen lassen sich rekursiv auf Funktionen höherer Ordnung anwenden und sind frei kombinierbar mit den Syntax-Varianten aus Tabelle 7.10. Die allgemeine Form ist in der Spezifikation der konkreten Syntax in Anhang B.4 angegeben.

<sup>5</sup>Diese Darstellung entspricht in der mathematischen Standardnotation impliziten Definitionsgleichungen der Form  $f(x) = y$  oder  $f \ x = y$  statt der expliziten Form  $f = \lambda x. y$ . In der impliziten Form steht auf der linken Seite nicht lediglich ein Bezeichner, sondern ein Ausdruck, der den zu definierenden Bezeichner enthält.

Syntax-Variante	Original-Syntax
<code>value id<sub>1</sub> (value id<sub>0</sub> : T<sub>0</sub>) : T<sub>1</sub></code>	<code>value id<sub>1</sub> : T<sub>0</sub>-&gt;T<sub>1</sub></code>
<code>value id<sub>1</sub> (value id<sub>0</sub> : T<sub>0</sub>) : T<sub>1</sub> = x</code>	<code>value id<sub>1</sub> : T<sub>0</sub>-&gt;T<sub>1</sub> = (value id<sub>0</sub> : T<sub>0</sub>) x</code>

Tabelle 7.14: Syntax-Varianten für Deklarationen und Definitionen von Funktionen

**Beispiel 7.1** Als Beispiel für die Definition einer rekursiven Funktion diene das klassische Beispiel der Fakultätsfunktion, die hier für vorzeichenlose 16-Bit-Zahlen definiert und auf den Wert 3 angewendet wird:

```

letrec
  value fac (value n : Integer16U) : Integer16U
    = if n == 0#16U then 1#16U else n * fac (n - 1#16U)
in
  fac 3#16U

```

Nach Eliminierung des syntaktischen Zuckers:

```

letrec
  value fac : Integer16U -> Integer16U
    = (value n : Integer16U)
      if n == 0#16U then 1#16U else n * fac (n - 1#16U)
in
  fac 3#16U

```

□

### 7.4.5 Typsystem

Wie eingangs zu Abschnitt 7.3 beschrieben, wird als *Typ* eine Menge bezeichnet, die mit Operationen ausgestattet ist, die einheitlich auf alle Elemente der Menge anwendbar sind. Verallgemeinert können dabei auch Operationen auftreten, die mehrere Typen involvieren. Das Typkonzept von FSPL ordnet Ausdrücken im engeren Sinn bzw. den durch sie bezeichneten Werten eindeutige Typen zu (vgl. Abschnitt 7.4.1).

#### *Typsicherheit*

FSPL wird als eine *typsichere* Sprache entworfen. Eine Programmiersprache heißt typsicher, wenn die Sprache dem Programmierer nicht erlaubt, einen bestimmten Wert wie einen Wert eines anderen Typs zu behandeln als zu dem er gehört [Wik05d]. Ferner ist FSPL eine *statisch*



*typisierte* Sprache, d. h. jeder Bezeichner und jeder Ausdruck hat einen festen Typ (siehe z. B. [Wat04]); somit können Typfehler zur Kompilierzeit festgestellt werden<sup>6</sup>.

Typsicherheit schließt nicht aus, daß ein Wert in einen äquivalenten Wert eines anderen Typs *umgewandelt* werden kann (z. B. ein ganzzahliger in einen reellen Wert). Die Umwandlung erfolgt jedoch stets explizit und wird durch die Typprüfung nur in den dafür definierten Fällen zugelassen.

Die Typsicherheit wird einerseits dadurch erreicht, daß die Typen streng unterschieden werden, indem alle konstruierbaren Werte genau einen Typ besitzen. Dies gilt insbesondere für ganzzahlige Werte unterschiedlicher Wertebereiche (z. B. 8, 16 oder 32 Bit, mit oder ohne Vorzeichen), für die es keine gemeinsamen Literale gibt (z. B. ist die 1 als 8-Bit-Zahl verschieden von der 1 als 16-Bit-Zahl und als vorzeichenbehaftete Zahl verschieden von einer vorzeichenlosen; siehe Abschnitt 7.3.1.2). Umwandlungen zwischen den verschiedenen Typen müssen stets explizit und damit bewußt vorgenommen werden.

Die Typsicherheit wird zum anderen dadurch erreicht, daß immer dann, wenn Bezeichner vereinbart werden, die explizite Angabe eines Typs durch den Programmierer erforderlich ist (siehe Abschnitt 7.4.3). Als Folge davon finden sich Ausdrücke im Programmtext, die Typen bezeichnen. Durch Typliterale und -konstruktoren können beliebig umfangreich konstruierte Typen beschrieben werden. Hinzu kommen Typvariablen (Typbezeichner), Typdefinitionen und Typfunktionen (siehe Abschnitt 7.4.5.3). Typausdrücke bilden eine eigene Teilsprache innerhalb der Programmiersprache, die *Typsprache*. Innerhalb der Typsprache können mehrere Typausdrücke den gleichen Typ (die gleiche Menge) bezeichnen; die Typausdrücke gelten dann als *äquivalent*.

*Typausdrücke*

*Typsprache*

Zur formalen Betrachtung von Typen und des Typsystems von FSPL siehe Anhang B.1.3 und B.2.2. Grundzüge des Typsystems werden nachfolgend beschrieben. Die fortgeschrittenen Typkonzepte der generischen Programmierung und des Modulsystems folgen in den Abschnitten 7.4.6 und 7.4.7.

#### 7.4.5.1 Primitive Typen

Primitive Typen bilden die Basis des Typsystems. Zu ihnen gehören sowohl die *primitiven Datentypen*

<sup>6</sup>Wie in Anhang B.1.4 ausgeführt, wird einem Programm nur dann eine Bedeutung zuerkannt, wenn es als Ausdruck der Sprache typkorrekt ist.

- Boolean
- Integer  $n$  U und Integer  $n$  S für alle Kardinalzahlen  $n$
- Real
- Character

nach Abschnitt 7.3.1 und der primitive Datentyp

- Time

nach Abschnitt 7.3.2.1 als auch der Typ

- Event

nach Abschnitt 7.3.2.3.

#### 7.4.5.2 Strukturierte Typen

Auf Basis der primitiven Typen können weitere Typen konstruiert werden. *Typkonstruktoren* erlauben die Konstruktion komplexerer Typen aus vorhandenen. So wurde in Abschnitt 7.4.4 ein Konstruktor für Funktionstypen eingeführt. *Strukturierte* Typen nun entstehen durch Typkonstruktoren für additive und multiplikative Verknüpfungen sowie Potenzierung.

*Daten-  
strukturen*

Werden strukturierte Typen aus primitiven *Datentypen* konstruiert, so entstehen *strukturierte Datentypen*, die ihrerseits weiter zu strukturierten Datentypen verknüpft werden können. Strukturierte Datentypen sind Typen für *Datenstrukturen*. Strukturierte Typen sind jedoch nicht auf Datenstrukturen beschränkt. Es können beispielsweise auch Strukturen, die aus Prozessen oder Ereignissen bestehen, beschrieben werden.

*Summentypen*

*Summentypen* (oder auch *Variantentypen*) repräsentieren Mengensummen (siehe Anhang A.1.7) der Form

$$\sum \{l_0 : X_0, \dots, l_n : X_n\} \quad (7.13)$$

mit  $n \in \mathbb{N}$ , also endliche, nichtleere Summen mit expliziten *Beschriftungen* für die *Varianten* (beschriftete Summen). Die Mengensumme enthält die Varianten als Paare der Form  $[l_i, x_i]$  mit  $x_i \in X_i$  und  $i \in 0 \dots n$ .

Durch *Produkttypen* werden Mengenprodukte (siehe Anhang A.1.6) der Form *Produkttypen*

$$\prod \{l_0 : X_0, \dots, l_{n-1} : X_{n-1}\} \quad (7.14)$$

und

$$X_0 \times \dots \times X_{n-1} \quad (7.15)$$

mit  $n \in \mathbb{N}$  unterstützt, also endliche, leere oder nichtleere Produkte mit oder ohne explizite *Beschriftungen* für die *Felder* (beschriftete bzw. unbeschriftete Produkte). Im ersten Fall enthält das Mengenprodukt *Verbunde (Records)* der Form

$$\{l_0 : x_0, \dots, l_{n-1} : x_{n-1}\}, \quad (7.16)$$

im zweiten Fall *Tupel* der Form

$$[x_0, \dots, x_{n-1}], \quad (7.17)$$

jeweils mit  $x_i \in X_i$  für  $i \in 0..n-1$ .

*Potenztypen* stellen Mengenpotenzen der Form

*Potenztypen*

$$X^n \quad (7.18)$$

dar (siehe Anhang A.1.8). Sie bestehen aus endlichen, nichtleeren Folgen (oder *Arrays*) der Form

$$[x_0, \dots, x_{n-1}] \quad (7.19)$$

mit  $x_i \in X$  für  $i \in 0..n-1$ . Es handelt sich jeweils um Arrays einer beliebigen, aber festen Länge  $n \in \mathbb{N} \setminus \{0\}$ , die auch als Spezialfall für  $n$ -Tupel gedeutet werden können. Der Fall  $n = 0$  wird ausgenommen, weil für ein leeres Array kein eindeutiger Potenztyp bestimmt werden kann, denn  $\square \in X^0$  für alle Typen  $X$ .

Die Verallgemeinerung zu Mengen  $X^*$  von endlichen Folgen variabler Länge oder sogar zu Mengen  $X^\omega$  von endlichen oder unendlichen Folgen (siehe Anhang A.1.8), die in funktionalen Programmiersprachen als Listen bzw. Ströme bekannt sind, ist für eingebettete Systeme von geringer Bedeutung. In den betrachteten Steuerungssystemen liegen die Strukturen typischerweise statisch fest, so daß die Array-Längen zur Kompilierzeit bekannt sind. Würde man Typen für  $X^*$  an Stelle von solchen für  $X^n$  in das Typsystem einführen und alle Arrays unabhängig von ihrer Länge so typisieren, fehlte der Typinformation Wesentliches an Aussagekraft. Zugriffe auf einzelne Elemente eines Arrays sind

unsicher, da eine Bereichsüberprüfung in der Typprüfung fehlt. Auch die Array-Typen in der Sprache C weisen genau diese fehlerträchtige Eigenschaft auf.

*Konstruktoren  
und Selektoren*

Die zu strukturierten Typen gehörigen primitiven Operationen sind *Konstruktoren* für Varianten, Records, Tupel und Arrays aus vorhandenen Objekten sowie *Selektoren* für eine Zerlegung in umgekehrter Richtung.

Records, Tupel und Arrays wie in (7.16), (7.17) bzw. (7.19) sind per definitionem Funktionen mit endlichem Definitionsbereich; sie werden in aufzählender Form dargestellt. Die Extraktion von Werten aus Records, Tupeln und Arrays ist dementsprechend ein Sonderfall der Funktionsanwendung. So gilt für  $i \in 0..n-1$ :

$$\{l_0:x_0, \dots, l_{n-1}:x_{n-1}\} l_i = x_i \quad (7.20)$$

bzw.

$$[x_0, \dots, x_{n-1}] i = x_i. \quad (7.21)$$

Sind die Typen der zusammengestellten Einzelobjekte bekannt, kann aus dem Record bzw. Tupel der zugehörige Produkttyp eindeutig bestimmt werden. Ist umgekehrt für die Selektion aus einem Record oder Tupel dessen Produkttyp bekannt, kann bei Anwendung auf einen gültigen Index der Ergebnistyp eindeutig bestimmt werden. Arrays haben die zusätzliche Bedingung bzw. Eigenschaft, daß alle Elemente den gleichen Typ haben.

Schwieriger verhält es sich bei Varianten. Ein Paar  $[l_i, x_i]$  mit  $x_i \in X_i$  und  $i \in 0..n$  ist zwar Element der Mengensumme  $\sum\{l_0:X_0, \dots, l_n:X_n\}$ , kann jedoch genauso gut auch Element einer Summe  $\sum\{l_0:X_0, \dots, l_{n+1}:X_{n+1}\}$  oder einer anderen Summe sein, die  $X_i$  an der Stelle  $l_i$  enthält. Das Paar  $[l_i, x_i]$  enthält als Konstruktor daher zu wenig Information, um aus dem Typ von  $x_i$  auf einen eindeutigen Summentyp schließen zu können. Der Wertkonstruktor für Varianten muß dazu mit zusätzlicher Information zur Identifikation des Summentyps angereichert werden.

Das Gegenstück zur Variantenkonstruktion ist die Fallunterscheidung nach der Variantenbeschriftung bzw. der Zugehörigkeit zu einem der Summanden der Mengensumme. Eine aufzählende Form liegt hier nicht bei der Variantenkonstruktion, sondern bei der Selektion vor. Für jeden der Fälle individuell wird ein Ergebnis bestimmt, das im allgemeinen von dem in der Variante enthaltenen Wert abhängt. Dieser

Wert kann einen von Fall zu Fall unterschiedlichen Typ haben, doch der Fallunterscheidungsausdruck insgesamt muß einen eindeutigen Typ erhalten. Die Fallunterscheidung setzt sich deshalb aus Funktionen zusammen, die je einen der im Summentyp verknüpften Typen auf einen gemeinsamen Ergebnistyp abbilden. Eine Fallunterscheidung für eine Variante  $x \in \sum \{l_0 : X_0, \dots, l_n : X_n\}$  und eine Ergebnismenge  $Y$  hat dann die Form

$$\begin{cases} f_0 x' & \text{falls } x = [l_0, x'] \\ \vdots & \vdots \\ f_n x' & \text{falls } x = [l_n, x'] \end{cases} \quad (7.22)$$

mit  $f_i \in X_i \rightarrow Y$  für  $i \in 0..n$ .

Fallunterscheidungen, die nur einen Teil der Fälle separat auswerten und alle übrigen Fälle gemeinsam abhandeln, können mit einem „sonst“-Zweig versehen werden, der statt einer Funktion direkt einen Wert des Ergebnistyps benennt.

Mit beschrifteter Mengensumme, beschriftetem sowie unbeschriftetem Mengenprodukt und Mengenpotenz als Modellen für Typkonstruktoren, mit Varianten, Records, Tupeln und Folgen als Modellen für Wertkonstruktoren sowie mit Fallunterscheidung für Varianten und mit der Anwendung von Records, Tupeln und Folgen (als Funktionen) auf Indizes als Modellen für Wertselektoren sind Konzepte vorhanden, um die die Definition der Programmiersprache erweitert werden kann (siehe Tabelle 7.15 und Anhang B.2.8). Da die Konstruktoren und Selektoren zueinander komplementär sind, kann von der in den Modellen enthaltenen mengentheoretischen Deutung, etwa von Records und Tupeln als Funktionen, abstrahiert werden. Die Konstruktoren und Selektoren bilden primitive Operationen auf eigenständigen Strukturen und werden syntaktisch von den bisher beschriebenen Sprachkonstrukten unterschieden. Insbesondere werden auch Tupel und Arrays syntaktisch voneinander unterschieden. Ein Potenztyp und ein äquivalenter Produkttyp werden in FSPL wohlunterschieden. Allerdings wird die explizite Typumwandlung von Arrays zu Tupeln (und umgekehrt) unterstützt (siehe Typisierung B.2.8.1 in Anhang B.2.8).

Für die Tupel-Syntax wurde die weit verbreitete<sup>7</sup> Variante mit runden Klammern gewählt (vgl. (A.61) in Anhang A.1.5), während für Arrays die Schreibweise mit eckigen Klammern beibehalten wurde. Arrays

<sup>7</sup>Sie wird auch in C an vergleichbarer Stelle (nämlich bei Funktionen mit mehreren Argumenten) verwendet.

Syntax	Semantik	Erklärung
$l_0 : T_0 \mid \dots \mid l_n : T_n$	$\sum \{l_0 : T_0, \dots, l_n : T_n\}$	Varianten-Typ
$\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}$	$\prod \{l_0 : T_0, \dots, l_{n-1} : T_{n-1}\}$	Record-Typ
$( T_0, \dots, T_{n-1} )$	$\prod [T_0, \dots, T_{n-1}]$	Tupel-Typ
$T [ n ]$	$T^n$	Array-Typ
$l : x \text{ as } T$	$[l, x]$	Variante
$\{ l_0 : x_0, \dots, l_{n-1} : x_{n-1} \}$	$\{l_0 : x_0, \dots, l_{n-1} : x_{n-1}\}$	Record
$( x_0, \dots, x_{n-1} )$	$[x_0, \dots, x_{n-1}]$	Tupel
$[ x_0, \dots, x_n ]$	$[x_0, \dots, x_n]$	Array
<b>case</b> $x$ <b>of</b> $l_0 : f_0$ $\vdots$ $l_n : f_n$ <b>case</b> $x$ <b>of</b> $l_0 : f_0$ $\vdots$ $l_n : f_n$ <b>otherwise</b> $y$ $x . l$ $x . n$ $x \text{ at } y \text{ default } x'$	$\begin{cases} f_0 x' & \text{falls } x = [l_0, x'] \\ \vdots & \vdots \\ f_n x' & \text{falls } x = [l_n, x'] \end{cases}$ $\begin{cases} f_0 x' & \text{falls } x = [l_0, x'] \\ \vdots & \vdots \\ f_n x' & \text{falls } x = [l_n, x'] \\ y & \text{sonst} \end{cases}$ $x \ l$ $x \ n$ $\begin{cases} x \ y & \text{falls } y \in 0 .. \#x - 1 \\ x' & \text{sonst} \end{cases}$	Varianten-Selektor Varianten-Selektor Record-Selektor Tupel-Selektor und statischer Array-Selektor dynamischer Array-Selektor
$x_0 + x_1$	$x_0 \frown x_1$	Array-Verkettung

Tabelle 7.15: Strukturierte Typen

und Tupel werden somit syntaktisch unterschieden. Allerdings entfallen (wie in der Sprache Haskell) einelementige Tupel, da sie nicht von geklammerten Ausdrücken unterscheidbar sind. Nullelementige Tupel sind wie auch leere Records zulässig; die zugehörigen Typen besitzen je genau ein Element und entsprechen so beide der Einheitsmenge  $\mathbb{U} = \{\text{unit}\}$ , die in C als der Datentyp `void` bekannt ist.

Da bei Arrays im Gegensatz zu allgemeinen Tupeln alle Elemente den gleichen Typ besitzen, kann die Selektion aus einem Array dynamisch, d. h. mit einem zur Laufzeit berechneten Index erfolgen, während die Selektion aus Tupeln statische, d. h. zur Kompilierzeit bekannte Indizes erfordert. Der Exponent bei einem Potenztyp ist eine positive natürliche Zahl, die ebenfalls statisch bestimmt sein muß, um statische Typprüfung (zur Kompilierzeit) vornehmen zu können.

Statischen Zahlenangaben dienen die natürlichen Zahlen der Metasprache (siehe Abschnitt 7.4.2), während für dynamische Indizes Datentypen der Objektsprache eingesetzt werden müssen. Vorzeichenlose Integer-Datentypen sind dafür geeignet, können allerdings im allgemeinen nicht sicherstellen, daß der erlaubte Indexbereich eingehalten wird (nur in den Fällen, wo der Exponent  $n$  eine Zweierpotenz ist, existieren Integer-Datentypen, die genau den Indexbereich umfassen). Um dennoch eine sichere Array-Indizierung zu erhalten, wird sie mit einem „Default“-Wert für den Fall, daß der Index außerhalb des Bereichs liegt, kombiniert.

Die Verknüpfung von Zeichenketten (d. i. Character-Arrays) motiviert die Verkettung von Arrays gleichen Basistyps als eine zusätzliche Operation. Wie in Java wird dazu der Additionsoperator überladen.

Zur Vereinfachung der Definition von Funktionen mit Tupel-Argumenten (mehrstellige Funktionen) läßt sich die Notation aus (A.79) in Anhang A.1.9 leicht auf die Programmiersprache übertragen (siehe Tabelle 7.16). Die Notation kann als syntaktischer Zucker erklärt werden (siehe Anhang B.2.8).

*Definition  
mehrstelliger  
Funktionen*

Syntax	Semantik
$(\text{value } id_0 : T_0, \dots, \text{value } id_n : T_n) \ x$	$\lambda[id_0 \in T_0, \dots, id_n \in T_n]. x$

Tabelle 7.16: Syntax-Variante zur Definition mehrstelliger Funktionen

Die Notationen für Definition und Deklaration nach Tabelle 7.14 werden entsprechend um Mehrstelligkeit erweitert. Die allgemeinen For-

men sind in der Spezifikation der konkreten Syntax in Anhang B.4 angegeben.

### 7.4.5.3 Typdefinitionen und -funktionen

#### Typbezeichner

Die Typkonstruktoren für Funktionstypen und strukturierte Typen ermöglichen es, beliebig tief geschachtelte Typausdrücke zu bilden. Zur anwendungsspezifischen Begriffsbildung, aber auch zur Wiederverwendung erscheint es zweckmäßig, wie auch bei normalen Ausdrücken die Vereinbarung von Namen (Bezeichnern) für Typausdrücke und eine Form von Parametrierung zu unterstützen. Definitionen und Funktionen werden also auf Typen übertragen, wenngleich nur in eingeschränkter Form.

Die erste Einschränkung besteht darin, daß keine *rekursiven* Typdefinitionen unterstützt werden. Rekursive Typen allgemein sind ein Konzept, das von vielen Programmiersprachen – zumindest in eingeschränkter Form – unterstützt wird (siehe z. B. [Wat04]). Zum Beispiel werden Datenstrukturen wie verkettete Listen oder Bäume gewöhnlich mit rekursiven Typen definiert. Bei rekursiven Typen steht die (maximale) Größe der Objekte nicht statisch fest. Ähnlich wie bei Arrays dynamischer Größe (vgl. Abschnitt 7.4.5.2) ist diese Situation untypisch für eingebettete Systeme, die Steuerungsaufgaben wahrnehmen und normalerweise feste oder (ressourcenbedingt) strikt limitierte Strukturen aufweisen.

Die zweite Einschränkung betrifft insbesondere die funktionale Abstraktion ( $\lambda$ -Abstraktion) für Typen. Im Gegensatz zu gewöhnlichen Funktionen nach Abschnitt 7.4.4 ist die  $\lambda$ -Abstraktion für Typen untypisiert. Es werden also Ausdrücke

$$\lambda T . T'$$

verwendet. Auf Metatypen (Typen für Typen), die typisierte  $\lambda$ -Abstraktionen der Art

$$\lambda T \in K . T'$$

ermöglichen würden, wird aus Gründen der Einfachheit verzichtet. Die Typsprache enthält den untypisierten  $\lambda$ -Kalkül mit der Menge der Typen als „Universum“.

Die durch Typdefinitionen und Parameter von Typfunktionen eingeführten Typbezeichner sind als Typvariablen selbst wieder Typausdrücke. In der konkreten Syntax unterscheiden sich Typbezeichner



von Bezeichnern darin, daß sie mit *großen* Anfangsbuchstaben beginnen. Typdeklarationen haben die Form

$$\text{type } tid$$

und Typdefinitionen die Form

$$\text{type } tid = T,$$

wobei *tid* für einen Typbezeichner steht. Lokale Typdefinition in Analogie zu *let* sowie die  $\lambda$ -Abstraktion und Applikation für Typfunktionen sind wie in den Tabellen 7.17 bzw. 7.18 definiert.

Syntax	Erklärung
<code>lettype</code>	parallele lokale
Typdefinition	Typdefinitionen
<code>:</code>	
Typdefinition	
<code>in</code>	
Ausdruck	

Tabelle 7.17: Syntax lokaler Typdefinitionen

Syntax	Semantik	Erklärung
$(\text{type } tid) T$	$\lambda tid. T$	Abstraktion
$T T'$	$T T'$	Applikation

Tabelle 7.18: Syntax für Typfunktionen und ihre Anwendung

Die in den Abschnitten 7.3.2.2, 7.3.4 und 7.3.5 eingeführten Typen *Process X*, *Phase X* und *Action X* sind nichts anderes als Anwendungen vordefinierter Typfunktionen *Process*, *Phase* und *Action*.

Typdefinitionen mit Parametern (siehe Tabelle 7.19) sind eine Syntaxvariante analog zu den vereinfachten Funktionsdefinitionen in Abschnitt 7.4.4. *syntaktischer Zucker*

Syntax-Variante	Original-Syntax
$\text{type } tid (\text{type } tid_0) = T$	$\text{type } tid = (\text{type } tid_0) T$

Tabelle 7.19: Syntax-Varianten für Definitionen von Typfunktionen

Typdefinitionen und -funktionen werden in Anhang B.2.9 formalisiert.

### 7.4.6 Generische Programmierung

#### *Polymorphie*

In der Sprachdefinition wurden bisher an verschiedenen Stellen Operatoren für unterschiedliche Typen *überladen*. So wurden z. B. syntaktisch gleiche arithmetische Operatoren für Integer-Datentypen und den Real-Datentyp eingeführt. Die Typen der jeweiligen Operanden entscheiden, welche arithmetische Operation angewendet wird. Ein ähnliches Vorgehen ist in objektorientierten Sprachen wie C++ oder Java auch für Funktionen (Methoden) bekannt: Der gleiche Funktionsname wird im gleichen Kontext für unterschiedliche Funktionen verwendet, die an Hand der Typen der Funktionsargumente unterschieden werden. Der Hauptzweck dieser Überladungen, die auch noch mit automatischen Typumwandlungen kombiniert werden können, ist die syntaktische Vereinheitlichung semantisch irgendwie verwandter Operationen bzw. Funktionen. Man spricht hier von *Polymorphie*, und zwar von *Ad-hoc*-Polymorphie [CW85]. Charakteristisch für Ad-hoc-Polymorphie ist, daß die Überladung nur für gewisse Typen definiert wird und die jeweiligen Varianten der Funktion völlig verschieden implementiert sein können; ihr Zusammenhang ist lose, im wesentlichen nur syntaktisch.

#### *universelle, parametrische Polymorphie*

Im Gegensatz dazu arbeitet *universelle Polymorphie* mit Funktionen oder anderen Objekten, die für alle Typen (oder zumindest für alle mit einer bestimmten Struktur) einheitlich definiert sind. Da im Rahmen dieser Sprachdefinition auf Untertypen verzichtet wird, wird hier ausschließlich mit dem Untertypus der *parametrischen Polymorphie* gearbeitet, bei dem Typen als Parameter zu Objekten auftreten. Handelt es sich dabei um Funktionen, werden sie auch als *generische Funktionen* bezeichnet. (Siehe [CW85].)

#### *$\Lambda$ -Abstraktion*

Generische Funktionen sind etwa solche, die rein strukturell arbeiten und deshalb für alle Typen bzw. Typkombinationen gleich definiert werden können. Man betrachte z. B. die Funktion

$$\text{swap}_{T_0, T_1} = \lambda [x_0 \in T_0, x_1 \in T_1] . [x_1, x_0],$$

die die zwei Elemente eines Paares vertauscht. Sie sei für beliebige Mengen (Typen)  $T_0$  und  $T_1$  definiert, wobei dann gilt:

$$\text{swap}_{T_0, T_1} \in (T_0 \times T_1) \rightarrow (T_1 \times T_0).$$

Mit parametrischer Polymorphie kann nun eine verallgemeinerte Funktion *swap* definiert werden, die die Typen  $T_0$  und  $T_1$  als Typparameter erhält. In einer Schreibweise nach [Rey98] (angelehnt an die

$\lambda$ -Notation für Funktionen) würde die generische Funktion geschrieben als

$$\text{swap} = \Lambda T_0 . \Lambda T_1 . \lambda [x_0 \in T_0, x_1 \in T_1] . [x_1, x_0]$$

und

$$\text{swap} \in \forall T_0 . \forall T_1 . (T_0 \times T_1) \rightarrow (T_1 \times T_0).$$

Man beachte, daß der  $\Lambda$ -Operator einen Typparameter zu einem Wertobjekt deklariert, während Typfunktionen Typparameter zu Typobjekten und gewöhnliche Funktionen Wertparameter zu Wertobjekten deklarieren. Der Typ eines polymorphen (generischen) Objekts  $x \in \forall T . T'$  besagt, daß  $x$  „für alle Typen  $T$ “ den Typ  $T'$  hat (*All-Quantifizierung*), wobei  $T'$  im allgemeinen ein Ausdruck ist, der von der Typvariablen  $T$  abhängt. Parametrische Polymorphie entspricht dem Template-Konzept in C++, soweit es auf Typen angewendet wird.

*All-Quantifizierung*

Das Gegenstück zur  $\Lambda$ -Abstraktion zur Beschreibung eines generischen Objekts ist die *Instantiierung* für einen konkreten Typ. Sie wird in Anlehnung an die Notation in C++ für Template-Instantiierungen als

*Instantiierung*

$$x < T_0 >$$

geschrieben, wobei hier für ein polymorphes Objekt  $x \in \forall T . T'$  der Typparameter  $T$  an den Typausdruck  $T_0$  gebunden wird.

In der Syntax der Programmiersprache werden der  $\Lambda$ -Operator und der Typkonstruktor für die All-Quantifizierung an den  $\lambda$ -Operator und die Funktionstypen angeglichen (siehe Tabelle 7.20).

Syntax	Semantik	Erklärung
$< \text{type } tid > \rightarrow T$	$\forall tid . T$	Typkonstruktor (All-Quantifizierung)
$< \text{type } tid > x$	$\Lambda tid . x$	generisches Objekt
$x < T >$	$x < T >$	Instanz

Tabelle 7.20: Syntax für generische Programmierung

Abschließend kann eine Variante der Schreibweise für Deklarationen und Definitionen generischer Objekte (kombinierbar mit den bisher definierten Varianten) als syntaktischer Zucker eingeführt werden (siehe Tabelle 7.21).

*syntaktischer Zucker*

Die Sprachelemente der generischen Programmierung werden in Anhang B.2.10 formalisiert.

Syntax-Variante	Original-Syntax
$\text{value } id < tdecl > : T$	$\text{value } id : < tdecl > -> (T)$
$\text{value } id < tdecl > : T = x$	$\text{value } id : < tdecl > -> (T) = < tdecl > x$

Tabelle 7.21: Syntax-Varianten für Deklarationen und Definitionen generischer Objekte

### 7.4.7 Module

#### *Module*

Ein *Modul* enthält eine Ansammlung von *Definitionen* (Typdefinitionen und Wertdefinitionen) und verbindet sie zu einer neuen, wiederverwendbaren Einheit. Im Gegensatz zu lokalen Definitionen (und Typdefinitionen) werden die Definitionen eines Moduls prinzipiell von den sie verwendenden Programmteilen getrennt, so daß sie eigenständig entwickelt und gewartet sowie in unterschiedlichen Programmteilen und Programmen mehrfach verwendet werden können. Module sind ein metasprachliches Beschreibungsmittel, das das Entwurfsprinzip der *Modularisierung* (siehe Kapitel 1.1.3) unterstützt. Für die Argumentation und die Ausgestaltung dieser Sprachkonzepte im Rahmen dieser Arbeit bildet [Rey98, Kapitel 18 „Module Specification“] eine wesentliche Grundlage.

#### *Schnittstellen*

Jedes Modul hat eine *Schnittstelle*, die *Deklarationen* zusammenfaßt, die den Definitionen, die das Modul bereitstellt, entsprechen. Man sagt auch, daß die durch ein Modul definierten Objekte (Typen und Werte) von dem Modul *exportiert* werden. Schnittstellen auf Basis von Deklarationen sind eine einfache Form von *Modulspezifikationen*, die automatisch (im Rahmen der Typprüfung) überprüft werden können: Module sind innerhalb von FSPL erstklassige (Wert-)Objekte; Modulschnittstellen sind die Typen dieser Objekte. Unterschiedliche Module gleichen Typs sind dabei alternative Implementierungen der gleichen Schnittstelle.

#### *Kapselung*

Neben der wiederverwendbaren Zusammenfassung von Definitionen erfüllen Module einen zweiten Zweck, für den sie auch unter dem Namen *Kapselungen* beschrieben werden (so z. B. in [Seb04, Kapitel 11]). Sie stellen darin eine Verallgemeinerung Abstrakter Datentypen (in einer Ausprägung für eine funktionale Sprache) dar.

#### *abstrakte Typen*

Die Besonderheit der Kapselung besteht darin, daß ein Modul *abstrakte Typen* definiert: Während lokale Typdefinitionen Abkürzungen für Typausdrücke einführen, die bei der Typprüfung durch die Typausdrücke selbst ersetzt werden, sind die von einem Modul exportier-

ten Typen *opak*. Die Verwendung von Objekten eines abstrakten Typs ist nur mit Operationen möglich, die innerhalb des gleichen Moduls (in Kenntnis der Typdefinition) definiert wurden. Ein Modul kapselt die Repräsentation der Typen zusammen mit relevanten primitiven Operationen.

Zur Verwendung der Definitionen eines Moduls wird das Modul *geöffnet* oder *importiert* (genaugenommen werden die vom Modul definierten Objekte importiert). Die Definitionen ergänzen dann den lokalen Kontext (ähnlich lokalen Definitionen und Typdefinitionen), die Typvariablen werden allerdings nicht durch ihre Definitionen ersetzt (diese sind nur innerhalb des Moduls bekannt). Folglich darf der Typ eines Gesamtausdrucks, in den ein Modul importiert wird, nicht von den darin definierten abstrakten Typen abhängen. Ein abstrakter Typ ohne zugehörige Operationen, die in den Typ hinein- und aus dem Typ herausführen, ist daher unbrauchbar.

Module können als Werte an Funktionen übergeben werden; der Typ des Funktionsarguments ist dabei der Schnittstellentyp. Es liegt hier eine neue Art von Polymorphie vor, bei der nicht Typen, sondern Module (mit darin potentiell enthaltenen Typdefinitionen) als Parameter auftreten. Tatsächlich kann für Module und Schnittstellen ein Grundtypus angegeben werden, der ein Gegenstück zur parametrischen Polymorphie aus Abschnitt B.2.10 darstellt. All-Quantifizierung für Typen wird hierbei um Existenz-Quantifizierung ergänzt. (Siehe auch [Pie02, Teil V „Polymorphism“] und [Mit90].)

*Polymorphie*

Existenz-quantifizierte Typen werden in der Typtheorie als

*Existenz-Quantifizierung*

$$\exists T . T'$$

(oder ähnlich) geschrieben, wobei  $T'$  in der Regel ein Produkttyp ist;  $T'$  hängt als Typausdruck im allgemeinen von der Typvariablen  $T$  ab. Ein Modul, das die abstrakten Typen  $T_0, \dots, T_n$  und die Operationen  $f_0 \in T'_0, \dots, f_m \in T'_m$  exportiert, hat dann etwa den Typ

$$\exists T_0 . \dots \exists T_n . \prod \{f_0 : T'_0, \dots, f_m : T'_m\}.$$

So erhält man beim Öffnen des Moduls einen Record mit Operationen, wobei die Beschriftungen im Record als Namen der Operationen angesehen werden können. Die Existenz-Quantifizierung besagt in etwa, daß zu einem Objekt dieses Typs (d. h. einem Modul mit dieser Schnittstelle) Typen  $T_0, \dots, T_n$  existieren, so daß das Objekt eine Struktur mit



Ein Modul kann beliebig viele Typen mit beliebig vielen Operationen zusammenfassen. Falls die Typen entfallen, dient ein Modul lediglich der Wiederverwendung von Definitionen und ist äquivalent zu einem Record. Entfallen die Operationen, ist ein Modul äquivalent zum Einheitswert `unit` und praktisch bedeutungslos (auch etwa vorhandene Typen sind nicht nutzbar).

Das `import`-Konstrukt übernimmt alle Definitionen des Moduls in den Kontext und verdeckt damit andere Objekte gleichen Namens. Will man qualifiziert auf ein einzelnes im Modul definiertes Objekt zugreifen, kann der überladene Record-Selektor verwendet werden (siehe Tabelle 7.23). Wegen der Opakheit der Typen kann in einem solchen Fall das selektierte Objekt nicht von einem abstrakten Typ des Moduls abhängig sein.

*syntaktischer  
Zucker*

Syntax-Variante	Original-Syntax	Bemerkung
$x.id$	<code>import x in id</code>	falls $x$ Modul

Tabelle 7.23: Syntax-Variante für Einzelimport aus Modulen

## 7.5 Hinweise zur Implementierung

Um von dem bisher beschriebenen Sprachentwurf zu einer anwendbaren Sprache zu gelangen, ist zunächst eine *vollständige, möglichst formale Definition* sowohl der Syntax als auch der Semantik notwendig. Sie wird in Anhang B gegeben, wobei die Syntax auf die *textuelle* Notation beschränkt wird.

Da FSPL als seiteneffektfreie funktionale Programmiersprache keine eingebauten Mechanismen zur Ein- und Ausgabe enthält, ist ferner als Schnittstelle zur Umgebung ein *Laufzeitsystem* erforderlich, das ein FSPL-Programm ausführt (d. h. als Systemfunktion anwendet). Abschnitt 7.5.1 verfolgt diesen Gedanken.

Für die Programmentwicklung ist ferner ein *Bibliothekskonzept* erforderlich, um große Anwendungen strukturiert verwalten und programübergreifende Wiederverwendung betreiben zu können. Ein solches Bibliothekskonzept mit *Paketen* (Packages) als Strukturierungsmittel ist in Abschnitt 7.5.2 beschrieben.

Schließlich sind *Entwicklungswerkzeuge* notwendig, um Bibliotheken und Programme, die in FSPL geschrieben werden, zu verarbeiten (siehe Abschnitt 7.5.4).

### 7.5.1 Laufzeitsystem

#### *Systemtakt*

Ein Laufzeitsystem für FSPL stellt zum einen die Basismechanismen bereit, auf die ein Compiler Elemente eines FSPL-Programms abbildet. Dazu gehört insbesondere eine Implementierung des synchronen Ausführungsmodells. Spezifisch für eine konkrete Implementierung (für eine bestimmte Zielplattform) und Konfiguration ist für den FSPL-Programmierer dabei die minimale Zeitauflösung (maximaler Systemtakt), die als Konstante verfügbar sein muß.<sup>8</sup>

#### *Ports, Interrupts, nichtflüchtiger Speicher*

Zum anderen abstrahiert das Laufzeitsystem von den Schnittstellen zur Ein- und Ausgabe und von nichtflüchtigem Speicher. I/O-Ports können zu Signalen (Variablen) abstrahiert werden, Interrupts zu Ereignissen. Zu jedem Zeitpunkt kann von einem Eingabe-Port ein Wert gelesen werden, und jeder Ausgabe-Port hält zu jedem Zeitpunkt einen Wert (Ports entsprechen also Signalen/Variablen). Ein Interrupt kann nacheinander zu verschiedenen Zeitpunkten angefordert werden (Interrupts entsprechen also Ereignissen). Eine Zelle nichtflüchtigen Speichers kann charakterisiert werden durch ihren Initialwert bei Systemstart (als Konstante) und durch ihren veränderlichen Inhalt während der Laufzeit (als Variable).<sup>9</sup>

#### *Haupt- programm*

Ein FSPL-Programm ist dann eine Funktion, die Eingaben auf Ausgaben abbildet (Systemfunktion). Zu den Eingaben können Signale (von Eingangs-Ports), Ereignisse (von externen Interrupts) und Konstanten (Initialwerte von persistentem Speicher) gehören, die Ausgaben bestehen stets aus Signalen bzw. Variablen (Ausgangs-Ports und nichtflüchtiger Speicher). Das Laufzeitsystem muß für einen bestimmten Funktionstyp konfiguriert werden. Die Konfiguration bildet die Elemente des Definition- und Wertebereichs auf konkrete Hardware-Adressen ab. Ein FSPL-Programm dieses Funktionstyps kann dann als Hauptprogramm von dem Laufzeitsystem ausgeführt werden.

Durch Eigenschaften der Hardware werden auch die bei den Ein- und Ausgabesignalen und den persistenten Variablen bzw. Konstanten

<sup>8</sup>Die Implementierung des Anwendungsbeispiels sieht diese Konstante als Element einer Basisbibliothek vor (siehe Anhang C.3.2).

<sup>9</sup>Der Wert der Variable zum Zeitpunkt 0 muß dabei nicht dem Initialwert entsprechen. Nach dem synchronen Ausführungsmodell liegen Ein- und Ausgabe logisch zwar auf dem selben Zeitpunkt. Der Initialwert ist jedoch der Wert der Speicherzelle, der zu Beginn des ersten Taktzyklus gelesen wird, während der Wert der Zustandsvariable zum Zeitpunkt 0 der Wert ist, der zum Ende des Taktzyklus (falls eine Veränderung vorliegt) geschrieben wird.



verwendbaren Datentypen bestimmt. Ports haben bestimmte Bitbreiten; ihre Daten können stets durch (vorzeichenlose) Ganzzahlen entsprechender Bitbreite dargestellt werden; durch Bitoperationen können einzelne zweiwertige Signale extrahiert werden (siehe Abschnitt 7.3.1.2). Es reicht also aus, die Ein-/Ausgabe mit Integer-wertigen Signalen zu handhaben. Signale abstrakterer Datentypen sind mit den dafür vorgesehenen Mechanismen programmintern zu bilden. Interrupts müssen entsprechend dem gewählten Zeitmodell auf den Zeittakt synchronisiert werden; ihre maximale Frequenz wird dadurch limitiert (siehe auch Abschnitt 7.3.2.1).

Ein Laufzeitsystem für FSPL kann entweder direkt auf der Hardware oder auf einem Echtzeitbetriebssystem aufgesetzt und z. B. in C programmiert werden.

## 7.5.2 Bibliothekskonzept

Unter einer Bibliothek soll eine Sammlung von Definitionen verstanden werden, die in Programmen oder anderen Bibliotheken als gegeben verwendet werden können. Bisher wurden als Bestandteil der (Meta-)Sprache lediglich *lokale* Definitionen vorgesehen (siehe Abschnitte 7.4.3 und 7.4.5.3). Die prototypische Teilimplementierung von FSPL, mit der das Anwendungsbeispiel in Anhang C entwickelt wurde, arbeitet mit einer Spracherweiterung, die es erlaubt, globale Definitionen in *Paketen* (Packages) zu organisieren.

*Pakete*

Ein Paket enthält eine Folge von Definitionen, wobei Typdefinitionen und Definitionen erstklassiger Objekte in beliebiger Reihenfolge auftreten dürfen. Nachfolgende Definitionen kennen die vorangehenden Definitionen, aber nicht umgekehrt. Definitionen erstklassiger Objekte kennen auch sich selbst, d. h. sind rekursiv (wie in *letrec* und in Modulen). Pakete können beliebig viele andere Pakete einbinden, deren Definitionen dann (in der Reihenfolge ihrer Einbindung) den eigenen Definitionen vorangestellt werden. Die dadurch entstehende Referenzstruktur zwischen Paketen muß azyklisch sein.

Tabelle 7.24 zeigt die Syntax für Pakete. Paket-Bezeichner bilden dabei einen eigenen Namensraum.

Ein Hauptprogramm ist stets eine global definierte Funktion, die z. B. als letzte Definition in einem Paket abgelegt wird. Durch die Sequenzierungsemantik geht jeder Definition in einem Package eine (im Grenzfall leere) Folge von anderen Definitionen voraus. Redefinitionen

Syntax	Erklärung
<pre> package <i>packageId</i> where include <i>packageId</i><sub>0</sub> ... include <i>packageId</i><sub><i>n</i>-1</sub>  type   <i>tid</i><sub>0</sub> ... value  <i>id</i><sub><i>m</i>-1</sub> : <i>T</i>'<sub><i>m</i>-1</sub> </pre>	type- und value-Definitionen (auch in Syntaxvarianten) in beliebiger Folge

Tabelle 7.24: Syntax für Pakete

sind dabei möglich, aber nicht empfehlenswert; alle in ein Hauptprogramm eingehenden globalen Definitionen sollten eindeutig sein. Eine Weiterentwicklung des Konzepts kann Pakete mit einem hierarchischen Namensraumkonzept (für Typ- und Wertbezeichner) verbinden.

Pakete empfehlen sich (eher als die einzelnen Definitionen) als Quellcode-Verwaltungseinheiten für Versions- und Konfigurationsmanagement. In dateiorientierten Entwicklungsumgebungen kann ein Package z.B. Inhalt genau einer Datei sein. Verzeichnisse im Dateisystem können Pakete und ihren Namensraum weiter organisieren.

### 7.5.3 Standard-Bibliothek

Neben Bibliotheken, die von den Anwendern selbst entwickelt werden, kann eine Sprachimplementierung stets auch eine vorgegebene Basisbibliothek mitliefern. Hilfreich sind Standardlösungen für wiederkehrende Probleme aus dem Anwendungsgebiet, z.B. Grundfunktionen der Signalverarbeitung.

FSPL bietet eine Reihe eingebauter Operationen zur Verarbeitung von Signalen an, so z.B. arithmetische Verknüpfungen oder den Zeitverschiebungsoperator `previous initially`. In Beispiel 6.7 wurde eine Standard-Funktion zur Signalabtastung angegeben, die `previous initially` verallgemeinert und mit den Mitteln der Sprache implementiert werden kann:

**Beispiel 7.2 (Ereignisgetriebene Signal-Abtastung)** *Der folgende Programmcode implementiert die Funktion  $\text{sample}_X$  aus Beispiel 6.7*

```
signal sample <type X> (event e) (value x0 : X) (signal x : X) : X
= letrec
  phase sampleCycle (value x1 : X) : X
  = do
    keep (const x1)
  when e then
    local
      value x2 : X := previous x initially x0
    in
      sampleCycle x2
in
  start (sampleCycle x0)
```

□

Diese und ähnliche Funktionen bieten sich als Elemente einer Standard-Bibliothek an.

Das MSR-Konsortium (*Manufacturer Supplier Relationship*) deutscher Automobilhersteller und -zulieferer [MSR] hat im Projekt MSR-MEGMA eine Bibliothek-Standard für den modellbasierten Entwurf von Steuergerätfunktionen für Kraftfahrzeuge entwickelt [WMB01]. Die werkzeugunabhängige Spezifikation [MSR01] definiert einen Satz von zeitdiskreten, signalverarbeitenden Funktionen, der sich wie folgt gliedert:

- Arithmetische Operatoren
- Logische Operatoren
- Vergleichsoperatoren
- Mathematische Funktionen (Sinus, Kosinus, Exponentialfunktion usw.)
- Zähler und Timer
- Verzögerungsblöcke
- Speicherblöcke

- Nichtlineare Blöcke
- Integratoren
- Hoch- und Tiefpässe
- Parameter und Konstanten (Suchtabellen)
- Signalpfadschalter

Ein Teil der spezifizierten Funktionen ist bereits durch die eingebauten Operatoren der Sprache abgedeckt (wie etwa die arithmetischen und logischen Operatoren); die übrigen lassen sich mit minimalen Einschränkungen auch in FSPL implementieren. Als Vorteil des Standards darf gewertet werden, daß er von Anwenderseite aus erstellt wurde; nach [MSR01] konzentriert er sich auf typische Funktionalität von Steuergeräten.

#### 7.5.4 Entwicklungswerkzeuge

##### *Editor*

Die textuelle Original-Syntax von FSPL erlaubt die Programmerstellung mit einem beliebigen Texteditor. Gleichwohl wird man auf die Sprache abgestimmte Eingabeunterstützungen wie Syntaxhervorhebung und Auto-Formatierung bevorzugen. Dedizierte Programmiereditoren (wie z. B. Emacs [GNU] oder jEdit [P<sup>+</sup>]) sind in der Regel dahingehend konfigurierbar.

Für die graphischen Notationen ist eine Editierumgebung mit einer Kombination aus graphischen und Text-Editoren erforderlich. In Anlehnung an Implementierungen von SDL [SDL00] ist dabei anzuraten, daß auch graphisch eingegebene Programmteile in textueller Form gespeichert werden oder zumindest so darstellbar sind, so daß ein Wechsel der Syntax und die Weiterverarbeitung mit Drittwerkzeugen möglich sind. Die Graphik-Information kann im Programmtext in speziellen Kommentaren (ähnlich den JavaDoc-Komentaren bei Implementierungen von Java [Sun]), die nicht zur Programmsemantik beitragen, abgelegt werden.

##### *Compiler*

Das wichtigste Werkzeug ist zweifelsohne ein *Compiler*, um aus den Programmen Code für die Zielplattform zu generieren. Bei aller Vielfalt an Mikroprozessoren und -controllern für eingebettete Systeme steht in der Regel für jede Plattform ein Cross-Compiler für ANSI-C zur Verfügung. Eine *zweistufige Codegenerierung*, bei der der Compiler der speziellen Hochsprache zunächst C-Code erzeugt, der dann erst

zu Objektcode kompiliert wird, ist gängige Praxis bei heutigen CASE-Werkzeugen. Zur Zielplattform kann auch ein Echtzeitbetriebssystem gerechnet werden, dessen Funktionalität als Basis bei der Codegenerierung und/oder bei der Implementierung des Laufzeitsystems zu berücksichtigen ist.

Bei der Codegenerierung besteht sehr viel Raum für *Optimierungen*. Wenn z.B. kein Ausgangssignal direkt (d.h. ohne Zeitverzögerung) von einem Eingangssignal abhängt und wenn alle Zeitverzögerungen durch Ereignisse getaktet werden (etwa mit der Bibliotheksfunktion `sample` aus Beispiel 7.2), nie jedoch direkt durch den Zeittakt, so liegt ein rein ereignisgesteuertes System vor. Je nachdem wie hart die Echtzeitbedingungen sind und welche zu erwartende Frequenz der externen Ereignisse vorliegt, kann ggf. auf Synchronisation mit einem Zeittakt verzichtet und direkt mit Interrupts gearbeitet werden (ereignisgesteuertes Laufzeitsystem).

Nicht für die Ausführung in Echtzeit auf dem Zielsystem, aber als Bestandteil von Test- und Debugging-Werkzeugen kann an die Stelle des Compilers auch ein *Interpreter* (oder *Simulator*) treten, der Programme direkt ausführt. Zur Validierung der Sprachspezifikation und der Beispiele wurde im Rahmen dieser Arbeit ein Interpreter implementiert, der die Sprachspezifikation ausführbar macht. Der Parser ist dabei in *ASF+SDF* [BK02], die Typprüfung und die Auswertung sind in *Haskell* [Has] implementiert. Als Entwicklungswerkzeug kann jedoch nur eine effizientere Implementierung dienen, die annähernd in Echtzeit arbeitet.

*Interpreter,  
Test- und  
Debugging-  
Werkzeuge*

*Testfälle* können weitgehend in FSPL selbst beschrieben werden, wenn eine konstruktive Form gewählt wird, bei der sowohl Eingabesignale und -ereignisse als auch eine Auswertungsfunktion für die Ausgabesignale ausprogrammiert werden. Der genannte Interpreter implementiert lediglich eine Spracherweiterung (entsprechend der Funktion `stateX` aus Spezifikation 6.2 in Kapitel 6.3.2), um Signale (Prozesse, Variablen) zu einzelnen Zeitpunkten abzutasten.

*Debugging* ist in funktionalen Sprachen nicht unmittelbar vergleichbar mit dem bei imperativen Sprachen: Ein Fortschreiten von Code-Zeile zu Code-Zeile ist im allgemeinen nicht möglich. Prinzipiell ist aber eine schrittweise Ausführung und Inspektion von Prozessen/Signalen/Variablen darstellbar, wobei die Schritte entweder durch den Zeittakt oder bestimmte ausgewählte Ereignisse festgelegt werden.

*automatische  
Verifikation*

Mit einer ergänzenden Spezifikationssprache, die Verhalten nichtkonstruktiv (basierend etwa auf temporaler Logik) beschreiben kann, können in weiterführenden Arbeiten Methoden zur Prüfung konstruierten Verhaltens gegenüber einer Spezifikation entwickelt werden. Die formale, mathematische Definition der Semantik der Programmiersprache bildet die Basis dazu. Der Hauptunterschied zwischen der (funktionalen) Programmiersprache und der (logischen) Spezifikationssprache wird darin bestehen, daß das Verhalten einerseits rein konstruktiv mittels Funktionen und andererseits deklarativ in Form von Prädikaten (und somit abstrakter) beschrieben wird. Das längerfristige Ziel wäre die Entwicklung von Werkzeugen zur automatischen Verifikation.

## 7.6 Zusammenfassung

FSPL zerfällt im wesentlichen orthogonal in eine *Objektsprache* und eine *Metasprache*. Die Objektsprache schließt an die Modellbildung in Kapitel 6 an. *Konstanten*, *Prozesse* (*Signale*, *Variablen*), *Ereignisse*, *Phasen*, und *Aktionen* werden als erstklassige Objekte behandelt und durch Literale und Konstruktoren beschrieben.

Die Metasprache stellt, beginnend mit *Bezeichnern*, *Definitionen* und *Gültigkeitsbereichen* (einschließlich rekursiver Definitionen als generischem Mechanismus) eine Infrastruktur zur Manipulation solcher Objekte und Abstraktionsmechanismen bereit, die durch ein *statisches Typsystem* abgesichert werden. *Funktionale Abstraktion* wird in allgemeiner Form und syntaktisch sowohl durch Parameter in Definitionen als auch durch unbenannte  $\lambda$ -Ausdrücke unterstützt. Funktionen sind erstklassige Objekte, können also auch als Parameter zu Funktionen höherer Ordnung auftreten. Datenabstraktion und Kapselung werden durch *Module* mit abstrakten Typen unterstützt. Auch Module sind erstklassige Objekte mit Modulschnittstellen als Typen. Typ-Parameter (*parametrische Polymorphie*) unterstützen als ergänzender Mechanismus die generische Abstraktion und somit die generische Programmierung.

FSPL verbindet so Konzepte höherer Programmiersprachen (vgl. Kapitel 2.2.2) mit den Beschreibungsmitteln für das Anwendungsgebiet eingebetteter Echtzeitsysteme und macht sie durchgängig verfügbar.

## 7.7 Literatur

Den größten Einfluß auf den Entwurf der Metasprache und die formale Definition von FSPL hatte [Rey98]. Darüber hinaus sind zahlreiche Ideen von anderen Sprachen in die Entwicklung von FSPL eingeflossen. Syntaktische Anleihen wurden vor allem bei C [KR88], Haskell [Has] und Pascal [Wir71] gemacht. Haskell und Standard ML [MTHM97] waren allgemeine Vorbilder als funktionale Sprachen, Standard ML auch mit seinem Modulsystem. Cayenne [Aug98] lieferte die Idee für die Typ-Kennzeichnung für Varianten ( $l : x \text{ as } T$ ). Die Bit-Operationen auf Ganzzahlen sind C entnommen. Diese Quellen von Ideen seien stellvertretend für alle genannt. Eine Trennung in Objektsprache und Metasprache findet sich unabhängig von dieser Arbeit auch bei Hume [HM03].





## 8. Das Entwurfsmuster „Abstrakte Maschine“

### 8.1 Übersicht

Erst der *methodische Einsatz* der durch FSPL bereitgestellten Sprachmittel kann die Vorteile realisieren, auf die der Sprachentwurf hinsichtlich der Belange des Software Engineering abzielt. Dieses Kapitel zeigt auf, wie das methodische Konzept zur Modularisierung von Prozeßsteuerungen aus Kapitel 3 mit den Mitteln von FSPL umgesetzt werden kann. Kernelement ist die Anwendung des Abstraktionsprinzips („information hiding“).

Abschnitt 8.2 beschreibt die Möglichkeiten der Verhaltensabstraktion (siehe Kapitel 3.2), die durch die spezifischen Sprachmittel von FSPL zur Beschreibung dynamischen Verhaltens vorgesehen sind. Abschnitt 8.3 beschreibt als Entwurfsmuster auf Ebene der Programmiersprache, wie abstrakte Maschinen (siehe Kapitel 3.3) als Module in FSPL dargestellt werden können. In dem in Anhang C beschriebenen Beispiel wird das Entwurfsmuster in dieser Form durchgängig für den Entwurf einer Kaffeemaschinensteuerung verwendet (siehe auch Kapitel 9.5).

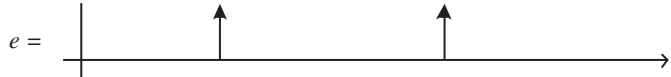
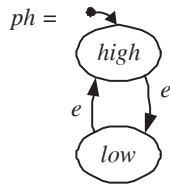
### 8.2 Möglichkeiten der Verhaltensabstraktion

Die Ausgangsbasis für Verhaltensabstraktion zur *Prozeßsteuerung* im

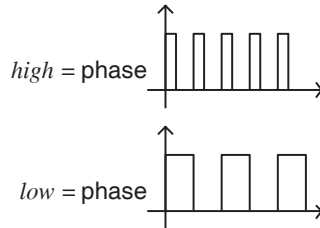
*Abstraktionen  
zur Prozeß-  
steuerung*

Rahmen der in Kapitel 6 modellierten und durch die Sprache FSPL implementierten Beschreibungsmittel ist der allgemeine Prozeß, der zu jedem Zeitpunkt der diskreten Zeitmenge (Time) genau einen Zustand innerhalb eines definierten Zustandsraums ( $X$ ) annimmt. Durch Lifting ( $\text{apply}_{\text{Time} \rightarrow X} f_x \text{ timeProcess}$ ) einer Funktion der Zeit ( $f_x \in \text{Time} \rightarrow X$ ) zu einem Prozeß ( $x \in \text{Process}_X$ ) können prinzipiell beliebige Prozesse auf dieser Abstraktionsebene beschrieben werden. Die gesamte Komplexität wird dabei in die definierende Funktion verlagert.

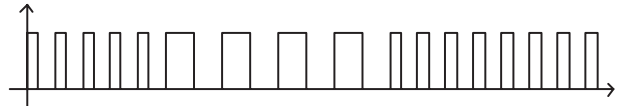
#### Reaktiver Prozeß



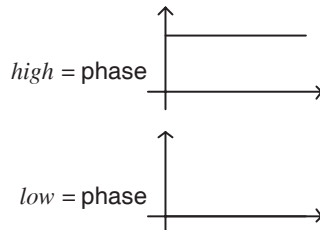
#### Realisierung 1:



$\text{start } ph \ 0_{\text{Time}} =$



#### Realisierung 2:



$\text{start } ph \ 0_{\text{Time}} =$



Abbildung 8.1: Verhaltensabstraktion mit Alternativen der Konkretisierung

Ein Phasenübergangssystem, das (gestartet zum Nullzeitpunkt) einen Prozeß darstellt, vergrößert die Sicht wesentlich: An die Stelle des Zustandsraums und der Zeitmenge treten ein „Phasenraum“ und Ereignisse; an die Stelle des Zustandsverlaufs entlang der Zeit treten Phasenübergänge bei Ereignissen. Erst durch die Definition der atomaren

Phasen über Prozesse (Abstraktionsebenenübergang) wird der gleiche Detaillierungsgrad erreicht. Die Trennung des abstrakten Phasenübergangssystems von der Realisierung der atomaren Phasen innerhalb der Gesamtprozeßbeschreibung entspricht einer Trennung von Entwurfsentscheidungen. Abbildung 8.1 zeigt ein Beispiel, in dem zwei alternative Realisierungen der atomaren Phasen eines Phasenübergangssystems zu unterschiedlichen Konkretisierungen eines (reaktiven) Prozesses führen.

Eine in ähnlicher Weise vergrößerte Prozeßsicht liefert eine Prozedur: Ein Vorrat an atomaren Aktionen und ein Algorithmus beschreiben abstrakt das Verhalten. Die Realisierung der atomaren Aktionen auf der reaktiven Beschreibungsebene führt zu einer Konkretisierung. Eine Prozedur abstrahiert von einem (sequentiellen) Phasenübergangssystem und so indirekt von einem (sequentiellen) Prozeß. Beide Abstraktionen von einem Prozeß, nämlich

- Prozeß  $\rightarrow$  Phasenübergangssystem
- Prozeß  $\rightarrow$  Prozedur

dienen der strukturierten Konstruktion von Prozeßverhalten.

Zur Prozeßbeobachtung, die nur mittelbar zur Prozeßsteuerung beiträgt, sind Prozeßabstraktionen entweder Signale oder Ereignisse. Ein Signal ist naturgemäß von gleicher Art wie ein Prozeß. Die Unterdrückung von Details findet hier durch Reduktion des Zustandsraumes statt. Ein abstraktes Signal kann etwa eine statische Projektion, aber auch eine dynamische Auswertung eines Prozesses oder im einfachsten Fall eine Zustandsbedingung sein. Sei beispielsweise der Prozeß die Bewegung eines Pendels; eine Abstraktion davon könnte die Bewegungsrichtung als zweiwertiges Signal sein. Eine Abstraktion zu einem Ereignis könnte in diesem Beispiel das Erreichen eines Wendepunktes sein. Dieses Ereignis kann weiter abstrahiert werden zu einem Ereignis, das nur jeden zweiten Wendepunkt angibt, also volle Perioden der Bewegung abzählt. Möglich sind also drei Arten von Abstraktionen:

*Abstraktionen  
zur Prozeßbeob-  
achtung*

- Prozeß/Signal  $\rightarrow$  Signal
- Prozeß/Signal  $\rightarrow$  Ereignis
- Ereignis  $\rightarrow$  Ereignis

So wie die Aktorik eine letzte Stufe der Konkretisierung in der Abstraktionshierarchie zur Prozeßsteuerung leistet, bildet die Sensorik auf der Seite der Prozeßbeobachtung die ersten Abstraktionen. Sie kann sowohl Signale (über Eingangs-Ports) als auch bereits Ereignisse (über Interrupts) liefern (siehe auch Kapitel 7.5.1).

### 8.3 Modulschnittstellen für abstrakte Maschinen

Eine abstrakte Maschine wird repräsentiert durch eine oder mehrere konkretere Maschinen (jeweils entweder eine konkrete, physikalische Maschine oder eine andere abstrakte Maschine) und ein Stück Steuerungssoftware, das die Schnittstelle(n) der konkreteren Maschine(n) verwendet und eine abstraktere Schnittstelle bereitstellt. Das Stück Steuerungssoftware ist zweckmäßigerweise ein *Software-Modul*. Dies kann auch für den Modulbegriff gelten, wie er in FSPL verwendet wird.

Module in FSPL sind mit dem Konzept *abstrakter Typen* verbunden. Analog zu abstrakten Datentypen [Gut77] lassen sich abstrakte Prozeßtypen mit Zustandsmenge und primitiven Operationen (elementare Prozesse, Phasen und/oder Aktionen sowie Generatoren für Signale und Ereignisse) definieren. Module sind (als erstklassige Objekte) durch *Schnittstellen* typisiert, die aus Deklarationen von abstrakten Typen und zugehörigen Operationen bestehen. Im folgenden wird eine Gliederung für die Modulschnittstelle einer abstrakten Maschine vorgeschlagen.

#### *Maschine und Programm*

Zunächst wird ein Steuerungsprozeß, der das Soll-Verhalten der Maschine beschreibt, von dem Verhalten der Maschine selbst unterschieden. Es werden damit zwei Abstraktionen nebeneinander betrachtet: Die *Maschine* und ihr *Programm*. Für beide Abstraktionen wird je ein abstrakter Typ deklariert, etwa ein Typ  $X$  für die Maschine und ein Typ  $XControl$  für den Zustandsraum des Steuerungsprozesses. Ein Modell der Maschine ist ein Objekt (Wert) vom Typ  $X$ , wobei  $X$  typischerweise ein Produkt aus Prozeß- (Signal-) und Ereignistypen ist. Das Programm (der Steuerungsprozeß) ist ein Objekt vom Typ  $Process_{XControl}$ .

#### *Konstruktions-schnittstelle*

In das Modell der Maschine gehen insbesondere die von der Sensorik kommenden Eingangssignale und -ereignisse ein, die über das Ist-Verhalten der Maschine Aufschluß geben. Weiterhin geht in die zur Aktorik gehenden Ausgangssignale insbesondere das Programm ein, bzw. das Programm zerfällt in die Aktorsignale. Je nach Art der Sensorik

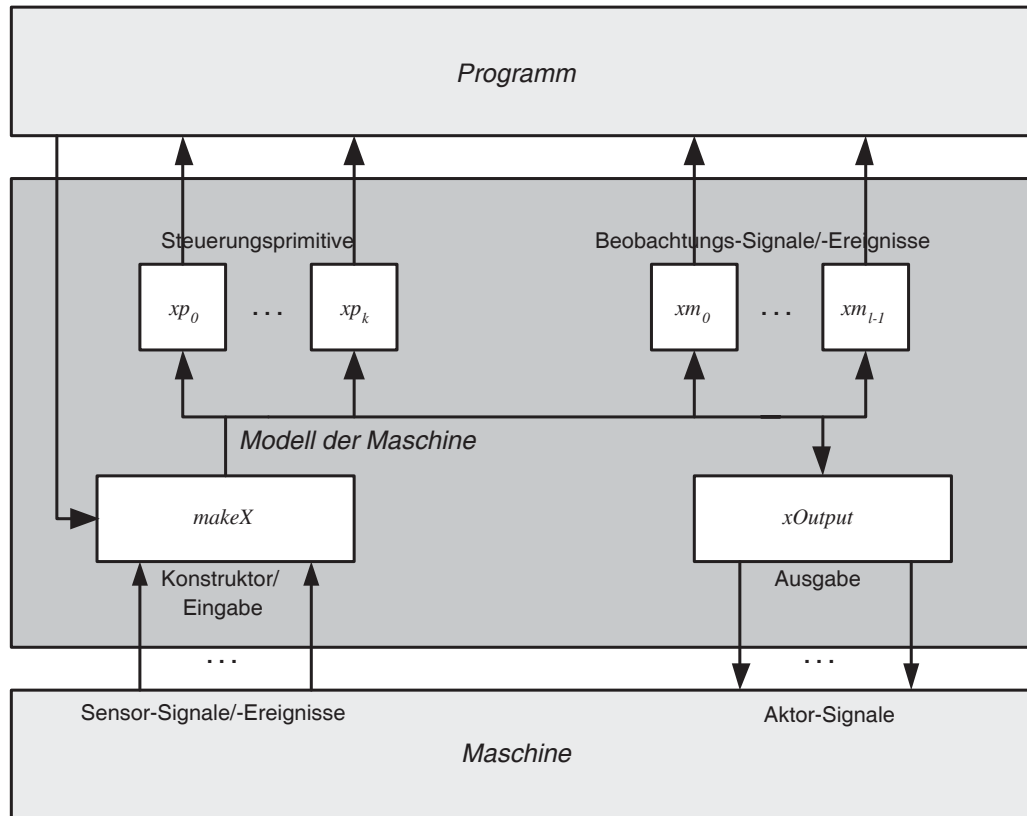


Abbildung 8.2: Abstrakte Maschine als Modul

und Aktorik kann es aber sinnvoll sein, auch die Ansteuerung der Aktorik bzw. das Programm selbst als Zustandsinformation in das Modell mit einfließen zu lassen, etwa um differenzielles Verhalten darstellen oder auswerten zu können. Umgekehrt kann es sinnvoll sein, für die Repräsentation des Programms nicht 1:1 die Aktor-Signale zu verwenden, sondern die Ansteuerung aus dem Programm und Modellinformation zu bilden. Man kann daher allgemein fordern, daß das Modell der Maschine das Programm mit einbezieht und somit auch alle Information für die Ansteuerung der Aktorik enthält.

Für das Modell der Maschine stellt das Modul daher zusätzlich zum abstrakten Typ  $X$  einen *Konstruktor*

$$makeX \in \text{Process}_{XControl} \rightarrow \text{Input} \rightarrow X$$

bereit, wobei *Input* eine Signalmenge oder Event oder ein Produkt aus Signalmengen und/oder Event und/oder solchen Produkten ist. *Input* ist

der Typ der Eingabe, die von der Sensorik stammt. Für die Ausgabe an die Aktorik stellt das Modul eine Funktion

$$xOutput \in X \rightarrow Output$$

bereit, wobei *Output* eine Signalmenge oder ein Produkt aus Signalmengen und/oder solchen Produkten ist (siehe auch Abbildung 8.2). Eine Signalmenge ist die Menge  $Process_S$  für eine Zustandsmenge *S*.

*Programmierschnittstelle*

Zur Konstruktion des Programms stellt das Modul elementare Prozesse, Phasen und/oder Aktionen als Verhaltensprimitive bereit, nämlich einen Satz von Funktionen

$$\begin{aligned} xp_0 &\in (X \times \dots) \rightarrow C_0 \\ &\vdots \\ xp_k &\in (X \times \dots) \rightarrow C_k \\ \text{wobei} \quad C_i &\in \{Process_{XControl}, Phase_{XControl}, Action_{XControl}\}, \end{aligned}$$

die auf ein Maschinen-Objekt (vom Typ *X*), ggf. mit weiteren Parametern, angewendet werden können.<sup>1</sup> Man beachte, daß die Verhaltensprimitive im allgemeinen vom Zustand der Maschine abhängig sind (z. B. eine Aktion, die bei Erreichen eines bestimmten Zustandes terminiert). Man beachte ferner in Abbildung 8.2, daß die von  $xp_0, \dots, xp_m$  ausgehenden Pfeile keine Signale darstellen, sondern Prozesse, Phasen und/oder Aktionen.

Um das abstrakte Ist-Verhalten der Maschine für Transitionen bzw. den Kontrollfluß im Programm auswerten zu können, liefert ein zweiter Satz von Funktionen

$$\begin{aligned} xm_0 &\in (X \times \dots) \rightarrow M_0 \\ &\vdots \\ xm_{l-1} &\in (X \times \dots) \rightarrow M_{l-1} \\ \text{wobei} \quad M_i &\text{ Signalmenge oder } M_i = \text{Event} \end{aligned}$$

Signale (z. B. Zustandsbedingungen) und/oder Ereignisse zu einer Maschine (siehe auch Abbildung 8.2). Die Verhaltensprimitive zur Steuerung zusammen mit diesen Primitiven zur Beobachtung stellen eine

<sup>1</sup>Diese Funktionen sind vergleichbar mit Methoden im Sinn der objektorientierten Programmierung.

Programmierschnittstelle für die Maschine dar. Aus primitiven Prozessen, Phasen, Aktionen, Signalen und Ereignissen können Programme für die Maschine konstruiert werden. Das Programm selbst wird außerhalb des Moduls erstellt, ggf. als Verhaltensprimitiv einer nächsthöheren abstrakten Maschine. Die in Abbildung 8.2 dargestellten Zusammenhänge illustrieren die Instantiierung einer abstrakten Maschine über einer konkreten Maschine.

Wie aus der Abbildung ersichtlich, stellt die (zweigeteilte) Programmierschnittstelle die eigentliche Abstraktion für den Programmierer bereit, während der Konstruktor und die Ausgabefunktion die konkrete Instantiierung herstellen. *X* und *XControl* bleiben gegenüber dem Anwender des Moduls abstrakt, sie treten als abstrakte Typen in der Modulschnittstelle auf. *Input* und *Output* sind abhängig von der Konkretisierung, sie treten als formale (Typ-)Parameter zur Schnittstelle auf. Ein Modul, das die Schnittstelle implementiert, kann konkrete Implementierungen für *X* und *XControl* in Abhängigkeit von ausgewählten *Input* und *Output* finden. Beispiele finden sich in Kapitel 9.5 und Anhang C.

In FSPL stellt sich dann eine Modulschnittstelle der beschriebenen Form für eine abstrakte Maschine wie in Abbildung 8.3 dar.

## 8.4 Diskussion

Die Definition abstrakter Modulschnittstellen steht in engem Zusammenhang mit der Abstraktion von Verhalten. Verhaltensabstraktion wurde in Abschnitt 8.2 *prozeßorientiert* begriffen; Verhalten ist dabei ein Zustandsverlauf, auf den mit Phasenübergangssystemen und Prozeduren vergrößerte Sichten geschaffen werden. Demgegenüber steht die bei signalflußorientierten Beschreibungen vorliegende *systemorientierte*, funktionale Sicht, bei der Verhalten als Ein-/Ausgabeverhalten verstanden wird (Übertragungsfunktion). Die systemorientierte Sicht führt zu Blockstrukturen als Architekturen mit Blöcken als Modulen (vgl. die Kritik dazu in Kapitel 3.1). Die prozeßorientierte Sicht konstruktiver Verhaltensabstraktion führt zu Hierarchien abstrakter Maschinen als Architekturen.

Auf die historische Verwendung des Begriffs „Abstrakte Maschine“ bei Dijkstra [Dij68, Dij69] wurde in Kapitel 3.3 bereits hingewiesen. Der Begriff ist in der Literatur weit verbreitet und wird unter anderem

```

type XModule (type Input)(type Output) =
interface {
  type X
  type XControl

  // Construction and I/O
  value makeX (process control : XControl) : Input -> X
  value xOutput : X -> Output

  // Control
  action  xp0 (value x : X,...) : XControl
  phase  xp1 (value x : X,...) : XControl
  :
  process xpk (value x : X,...) : XControl

  // Monitoring
  event  xm0 (value x : X,...)
  :
  signal xml (value x : X,...) : Y
}

```

Abbildung 8.3: Modulschnittstelle einer abstrakten Maschine in FSPL

auch in formalen Methoden (z. B. ASM [BS03] oder B [Abr96]) verwendet. Die Abstrakten Maschinen in dieser Arbeit entsprechen als Architektorentwurfsmuster den Abstrakten Geräten aus [FSMG04].

Entwurfsmuster für Software allgemein wurden populär durch das Buch von Gamma et al. [GHJV95], und zahlreiche Arbeiten widmen sich dem Sammeln von Entwurfsmustern und der Art und Weise sie darzustellen. Das in diesem Kapitel dargestellte Entwurfsmuster folgt in seiner Darstellung keinem bestimmten Meta-Muster.



## 9. Anwendungsbeispiele

Zur Validierung und Demonstration der entwickelten und in der Sprache FSPL umgesetzten Konzepte wurde ein zusammenhängendes Programmbeispiel in FSPL (Teilfunktionalität einer Kaffeemaschinensteuerung) ausgearbeitet. Das vollständige Beispiel findet sich in Anhang C. Im folgenden werden die wesentlichen Konzepte, mit denen der Ansatz von FSPL auf die in Kapitel 3.4 formulierten Anforderungen antwortet, mit einer Reihe von Programmfragmenten illustriert. Mit Ausnahme des quasi-kontinuierlichen Reglers in Abschnitt 9.2 sind sie alle der genannten Kaffeemaschinensteuerung entnommen.

### 9.1 Übersicht

Die Beispiele in den Abschnitten 9.2 bis 9.4 überdecken die drei Verhaltensmodelle, mit denen FSPL das *Anwendungsgebiet* „eingebettete Echtzeitsysteme“ adressiert:

- quasi-kontinuierliches Verhalten (siehe Kapitel 2.4.1, 6.4 und 7.3.3)
- reaktives Verhalten (siehe Kapitel 2.4.2, 6.5 und 7.3.4)
- sequentielles Verhalten (siehe Kapitel 2.4.3, 6.6 und 7.3.5).

Die Abschnitte 9.5 und 9.6 illustrieren den methodischen Ansatz und die Unterstützung durch die Sprache im Hinblick auf essentielle Belange des *Software Engineering*:

- Anwendung des Abstraktionsprinzips („information hiding“) auf Daten und Verhalten (siehe Kapitel 3 und 8)
- Abstraktions- und Variantenmechanismen (siehe Kapitel 7.4).

An einzelnen Stellen erfolgt exemplarisch der Vergleich mit alternativen Lösungen in anderen Sprachen. Abschnitt 9.7 faßt die an den Beispielen erkennbaren besonderen Merkmale von FSPL zusammen.

## 9.2 Quasi-kontinuierliches Verhalten und Signalflüsse

Abbildung 9.1 zeigt einen zeitdiskreten PI-Regler (Proportional-Integral-Regler) in der Sprache von MATLAB Simulink. Man erkennt die Bildung der Differenz zwischen der Führungsgröße  $w$  und der Regel- oder Rückführgröße  $u$ , die proportional mit dem Faktor  $k_P$  und aufintegriert mit dem Faktor  $k_I$  in die Stellgröße  $v$  eingeht.

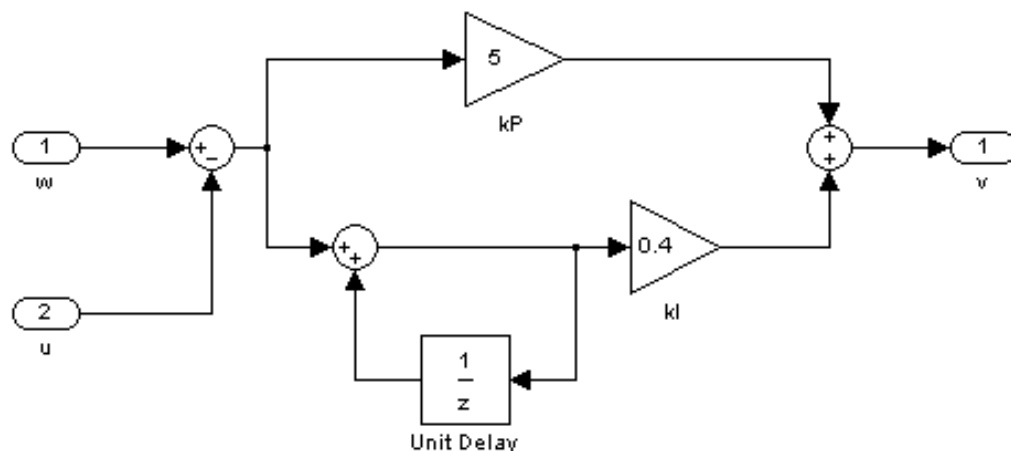


Abbildung 9.1: PI-Regler in MATLAB Simulink

Abbildung 9.2 zeigt den Regler als Funktion in FSPL. Im FSPL-Code werden zwei Hilfsgrößen  $e$  und  $x$  explizit benannt, die im Simulink-Modell unbenannt bleiben können. Dies ist erforderlich wegen der Mehrfachverwendung von  $e$  und der rekursiven Definition von  $x$ , die im graphischen Modell als Signalverzweigung bzw. -rückkopplung dargestellt werden. Die Schachtelung von `let` und `letrec` in dieser Form ist nicht zwingend. Lediglich die Definition von  $x$  benötigt die Rekursion; andererseits können auch Konstanten und nichtrekursive Signale

```

signal myPIController (signal w : Real, signal u : Real) : Real =
let
  value kP : Real = 5.0
  value kI : Real = 0.4
in letrec
  signal e : Real = w - u
  signal x : Real = e + (previous x initially 0.0)
in
  kP * e + kI * x

```

Abbildung 9.2: PI-Regler in FSPL

in einer letrec-Umgebung definiert werden. Die gewählte Aufteilung trennt die Konstanten von dem Signalflußgraphen, der durch im allgemeinen wechselseitig rekursive Signaldefinitionen<sup>1</sup> dargestellt wird. Im übrigen stimmen Modell und Programm strukturell überein, wobei auffällt, daß die Ausgangsgröße  $v$  im Programm (als „Rückgabewert“ der Funktion `piController`) nicht benannt wird.

Eine verallgemeinerte Bibliotheksfunktion für einen PI-Regler könnte wie in Abbildung 9.3 definiert werden. Die Differenzbildung wird dabei dem Anwender überlassen; die Verstärkungsfaktoren für den P- bzw. I-Anteil werden zu Parametern.

```

signal piControl (value kP : Real, value kI : Real) (signal e : Real) : Real =
letrec
  signal x : Real = e + (previous x initially 0.0)
in
  kP * e + kI * x

```

Abbildung 9.3: PI-Regler in FSPL als Bibliotheksfunktion

---

<sup>1</sup>im einfachsten Fall wie hier ein Differenzgleichungssystem mit `previous` als Zeitverschiebungsoperator

Der Regler aus Abbildung 9.2 lässt sich damit umschreiben zu:

```

signal myPILController (signal w : Real, signal u : Real) : Real =
let
  value kP : Real = 5.0
  value kI : Real = 0.4
in
  piControl (kP, kI) (w - u)

```

*Regelungs-  
algorithmus  
als Parameter*

Die mit der Funktion aus Abbildung 9.3 gefundene Abstraktion lässt sich ohne Schwierigkeiten in Simulink nachbilden (Subsystemblock mit Parametern). Die Mächtigkeit der funktionalen Metasprache von FSPL kommt erst dort zum Tragen, wo nicht nur Konstanten, sondern komplexere Objekte als Parameter dienen können, z. B. der Regelungsalgorithmus selbst. Für bestimmte Konstanten  $k_P, k_I$  ist

$$\text{piControl}(k_P, k_I)$$

eine Funktion vom Typ  $\text{Process Real} \rightarrow \text{Process Real}$ , die auch durch eine andere Funktion gleichen Typs, etwa für einen PID-Regler (Proportional-Integral-Differential-Regler), ersetzt werden könnte. So lässt sich beispielsweise ein generischer Regler definieren, der (als eine Funktion höherer Ordnung) den Regelungsalgorithmus als Parameter erhält (Abbildung 9.4).

```

signal genericController
  (signal controlAlgorithm (signal deviation : Real) : Real)
  (signal u : Real, signal w : Real)
: Real
= controlAlgorithm (w - u)

```

Abbildung 9.4: Generischer Regler

Der Regler aus Abbildung 9.2 lässt sich damit erneut umschreiben zu:

```

signal myPILController (signal w : Real, signal u : Real) : Real
= genericController (piControl (5.0, 0.4)) (w, u)

```

## 9.3 Reaktives Verhalten und Phasenübergangssysteme

Abbildung 9.5 zeigt einen reaktiven Algorithmus zur Positionsbestimmung für die Presse in der Kaffeemaschine (siehe Anhang C.5.2). Ausgehend von einer festen Position `referencePosition` zum Startzeitpunkt wird eine Bewegung der Presse um eine Positionseinheit durch ein Ereignis `motionSensorEventDistanceUnit` detektiert, wobei die aktuelle Bewegungsrichtung aus einem zweiwertigen Signal `direction` abgelesen werden kann. Der Gesamtausdruck in Abbildung 9.5 bezeichnet ein positionswertiges Signal, das mit einer rekursiv definierten und durch den aktuellen Positionswert parametrisierten Phase konstruiert wird.

*reaktiver  
Algorithmus*

```

letrec
  phase position (value p0 : PressPosition) : PressPosition
  = do
    keep const p0
    when motionSensorEventDistanceUnit then
      if (previous direction initially down) == down then
        position (oneDown p0)
      else
        position (oneUp p0)
in
  start (position referencePosition)

```

Abbildung 9.5: Positionsbestimmung der Presse

Ein typisches Phasenübergangssystem im Stil eines Zustandsautomaten ist in Abbildung 9.6 dargestellt (zum Kontext siehe Anhang C.5.9). Die Phasen `waitReady`, `makeProduct` und `clean` werden wechselseitig betreten. Eine vierte Phase `resumeCleaning` kann nur alternativ zu `waitReady` als Startphase betreten werden, je nachdem welche Anfangsbedingungen herrschen. Von der Phase `waitReady` aus gibt es zwei alternative Transitionen zu `makeProduct` bzw. `clean`, die durch unterschiedliche Ereignisse ausgelöst werden. Die drei anderen Phasen führen jeweils eine Aktion aus und kehren nach ihrer Terminierung selbsttätig zu `waitReady` zurück. Die Phase `waitReady` ist parametrisiert und wird bei der Transition von `waitReady` aus in Abhängigkeit vom gewählten Produkt instantiiert.

*Phasen-  
übergangssystem*

Das gesamte Phasenübergangssystem dient zur Definition einer Pha-

*Hierarchie*

```

phase serve : CoffeeMachineApplicationControl = (
  letrec
    phase waitReady : CoffeeMachineApplicationControl = (
      do
        orthogonalize {
          machine          : noCoffeeMachineActivity (machine),
          user             : requestSelection (user, allProducts, false),
          cleaningState    : keep const neutral,
          powerDownRequest : keep const false
        }
      when productSelected (user) then (
        local
          value product : Product := selectedProduct (user)
        in
          makeProduct product
      )
      when cleaningSelected (user) then
        clean
    )
    phase makeProduct (value product : Product)
    : CoffeeMachineApplicationControl = (
      complete (
        ...
      )
      then waitReady
    )
    phase clean : CoffeeMachineApplicationControl = (
      complete (
        ...
      )
      then waitReady
    )
    phase resumeCleaning : CoffeeMachineApplicationControl = (
      complete (
        ...
      )
      then waitReady
    )
  in (
    if previousCleaningState != neutral then
      resumeCleaning
    else
      waitReady
  )
)

```

Abbildung 9.6: Dienstleistung der Kaffeemaschine

se `serve`, die hierarchisch in Unterphasen zerfällt. Die Unterphase `waitReady` zerfällt in Unterphasen zweiter Stufe für vier orthogonale Zustandsräume:

*Orthogonalität*

```
orthogonalize {
  machine          : noCoffeeMachineActivity (machine),
  user             : requestSelection (user, allProducts, false),
  cleaningState    : keep const neutral,
  powerDownRequest : keep const false
}
```

Phasen sind erstklassige Objekte, die als Parameter übergeben werden können. Abbildung 9.7 zeigt die Definition einer Ausnahmebehandlungsroutine (Aktion), die neben der zu behandelnden Ausnahme auch eine bestimmte Benutzermeldung (Phase) als Parameter erhält (zum Kontext siehe Anhang C.5.9). Im vorliegenden Fall ist die Benutzermeldung die statische Anzeige eines Textes, könnte im allgemeinen Fall aber auch ein dynamisches Verhalten (z. B. Blinken oder Signalton) beinhalten. Die Ausnahmebehandlung besteht hier darin, bei angehaltener Maschine unter Anzeige der Benutzermeldung auf das Ende der durch den Benutzer zu behebenden Ausnahmesituation zu warten.

*Phasen als  
erstklassige  
Objekte*

```
action handleException (value exception : Exception,
                        phase alarmUser : UserControl)
: CoffeeMachineApplicationControl = (
  do
    orthogonalize {
      machine          : noCoffeeMachineActivity (machine),
      user             : alarmUser,
      cleaningState    : keep previousCleaningState,
      powerDownRequest : keep const false
    }
  until exception.cleared
)
```

Abbildung 9.7: Generische Ausnahmebehandlung

Auch Ereignisse sind erstklassige Objekte, wie z. B. die Konstruktion des im Beispiel aus Abbildung 9.7 auftretenden Typs `Exception` zeigt (Abbildung 9.8, zum Kontext siehe Anhang C.4.1). Hier treten Ereig-

*Ereignisse als  
erstklassige  
Objekte*

nisse als Elemente einer Datenstruktur und als Parameter einer Konstruktorfunktion auf.

```

type Exception = {
  raised  : Event,
  cleared : Event,
  holds   : Process Boolean
}

value exception (signal exceptionCondition : Boolean) : Exception = {
  raised  : trigger exceptionCondition,
  cleared : trigger (!exceptionCondition),
  holds   : exceptionCondition
}

value failure (event failureEvent) : Exception = {
  raised  : failureEvent,
  cleared : never,
  holds   : happened failureEvent
}

```

Abbildung 9.8: Ereignistyp mit Konstruktoren

### Ereignis-arithmetik

Eine sprachlich-syntaktische Besonderheit von FSPL stellt die „Ereignisarithmetik“ dar: Für den Typ *Event* überladene arithmetische Operatoren dienen zur Verknüpfung von Ereignissen in ereigniswertigen Ausdrücken. Abbildung 9.9 zeigt die Definition einer Aktion zum Abmessen einer Wassermenge mit Hilfe eines Durchflußmessers, der den Durchfluß einer konstanten Wassermenge (einer Wassereinheit, entsprechend der Umdrehung eines Schaufelrades) als Ereignis anzeigt (zum Kontext siehe Anhang C.5.1 oder das Beispiel in Abschnitt 9.5). Bis zum Erreichen einer diskreten Anzahl von Wassereinheiten wird ein Ventil offen gehalten. Der Ausdruck

amount \* flow.waterUnit

bezeichnet das Ereignis, das das Abmessen beendet: das Ereignis für den Durchfluß einer Wassereinheit multipliziert mit der Anzahl der zu zählenden Wassereinheiten.

Abbildung 9.10 zeigt die Definition eines Ereignisses aus der Benutzerschnittstelle der Kaffeemaschine. Ein Ereignis `powerOff` tritt ein, sobald die Standby-Taste eine Sekunde lang gedrückt wurde. Dazu muß



```

action measureOff (value flow : Flow,
                  value amount : FlowWaterAmount) : FlowControl = (
  do
    keep open
  until
    amount * flow.waterUnit
)

```

Abbildung 9.9: Abzählen einer Wassermenge

eine Sequenz (+) aus zwei Ereignissen stattfinden, nämlich zuerst das Drücken der Standby-Taste und dann das Vergehen einer Sekunde, ohne daß (–) die Taste losgelassen wird.

```

event powerOff (value user : User) = (
  keyPressed (user.keyboard, standbyKey) +
  (after 1s – keyPressed (user.keyboard, standbyKey))
)

```

Abbildung 9.10: Halten der Standby-Taste zum Ausschalten

Abbildung 9.11 zeigt noch die Definition eines Ereignisses für den Stillstand der Presse beim Fahren gegen Widerstand. Sie verwendet zum einen ein Nicht-Ereignis (no within), das durch ein Ereignis und einen Timeout gegeben ist, und zum anderen die Bewachung durch eine Bedingung (%). Innerhalb einer Zeile läßt sich so der Sachverhalt lesbar ausdrücken: „Keine Bewegung der Presse innerhalb einer bestimmten Frist unter der Bedingung, daß der Motor eingeschaltet ist.“

```

event stalled (value press : Press) =
let
  signal motorOn : Boolean =
    previous (press.motor.power != 0 as MotorPower)
    initially false
in
  (no press.motion within stallDetectTimeout) % motorOn

```

Abbildung 9.11: Stillstandsdetektion der Presse

Auch *interaktives* Verhalten mit wechselseitiger Triggerung durch Er- *Interaktion*

eignisse läßt sich rekursiv in FSPL programmieren. Ein Beispiel dafür findet sich in dem in Anhang C.5.5 beschriebenen Protokoll zur manuellen Reinigungsmitteldosierung.

## 9.4 Sequentielles Verhalten und Prozeduren

Abbildung 9.12 zeigt ein Kernstück der Reinigungsprozedur für die Kaffeemaschine, bestehend aus zwei Zyklen von Spülungen. Im ersten Zyklus werden die Spülungen durch Wartezeiten unterbrochen, abschließend erfolgt ein „Auspressen“. Im zweiten Zyklus wechseln sich Spülung und Auspressen ab. Die Anzahl der Durchläufe (bestehend aus je einer Sequenz von Aktionen) liegt für jeden Zyklus fest. Diese primitive Art von Iteration ist in FSPL durch den Multiplikationsoperator für Aktionen sehr elegant darstellbar.

*primitive  
Iteration*

```

let
  action pressOut : BrewingUnitControl = (
    movePressToPhysicalLimit (machine.brewingUnit);
    movePressTo (machine.brewingUnit, closingPosition)
  )
in (
  // First rinsing cycle
  rinsingCyclesWithWaiting * (
    injectWater (machine.brewingUnit, rinsingWater);
    do
      noBrewingUnitActivity (machine.brewingUnit)
    until (after rinsingWaitingTime)
  );
  pressOut;

  // Second rinsing cycle
  rinsingCyclesWithPressOut * (
    injectWater (machine.brewingUnit, rinsingWater);
    pressOut
  )
)

```

Abbildung 9.12: Spülzyklen

Abbildung 9.13 zeigt zum Vergleich eine Implementierung in UML 1.4 mit Hilfe einer State Machine. Eine State Machine wird verwendet,

weil die einzelnen Aktionen (wie in FSPL) reaktiv realisiert sind, so daß sequentielles Verhalten auf reaktives Verhalten aufgesetzt wird. Die einzelnen Aktionen werden mit Unterzuständen realisiert, unter denen sich jeweils ein Terminierungszustand befindet, über den der Zustand insgesamt verlassen wird (die `MovingTo*`-Zustände verfeinern sich beispielsweise wie in Abbildung 9.14). Da die Aktionszustände selbst terminieren, kann durch eine einzelne Ausgangstransition ohne Trigger die Sequenzierung dargestellt werden. Iteration wird über bedingte Sprünge (Transitionen) nachgebildet; daß dabei eine Zähler-Variable verwendet wird, entspricht dem Standard imperativer Sprachen.

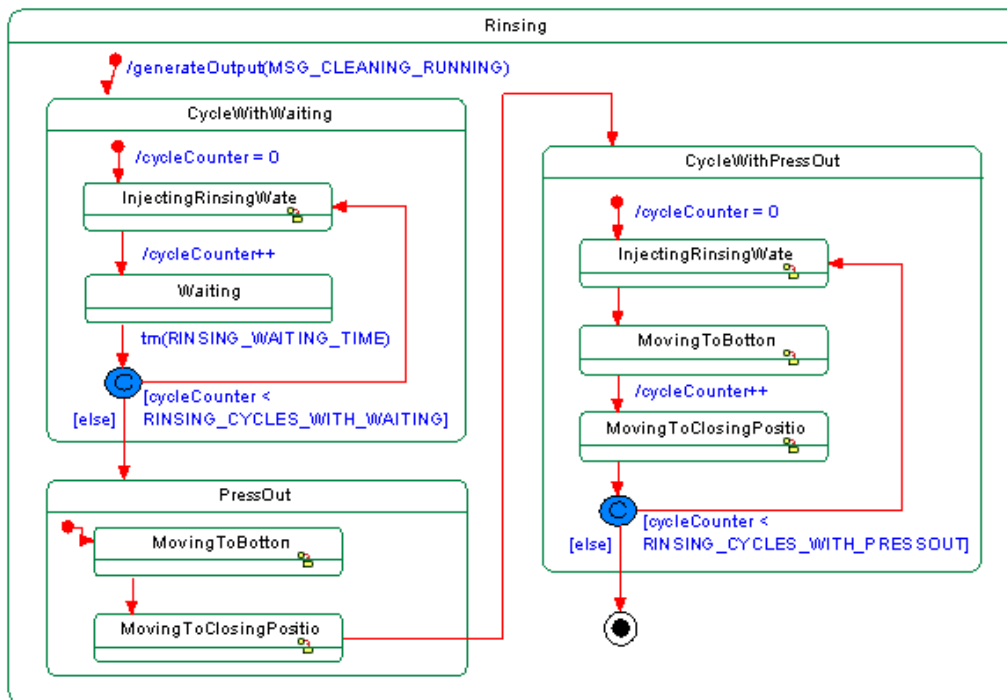


Abbildung 9.13: Spülzyklen in UML 1.4

Es fällt auf, daß im Gegensatz zur FSPL-Implementierung die Sequenz „Auspressen“ nicht als Prozedur herausfaktoriert ist. Doch bereits die Verfeinerungen der wiederkehrenden Bewegungs- und Spülaktionen, teilweise mit unterschiedlichen Parametern, werden kopiert (in jedem Einzelfall muß ein Bild wie Abbildung 9.14 erstellt werden). Eine Mehrfachverwendung von verfeinerten Zuständen ist inzwischen in UML 2.0 über das Konzept der „Submachine States“ zwar möglich, es fehlt dabei allerdings weiterhin die Möglichkeit der Parametrierung.

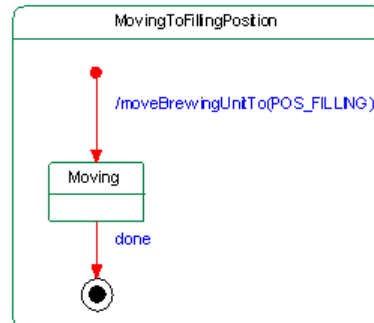


Abbildung 9.14: Aktion mit reaktiver Realisierung in UML

*Parallelität*

Die Definition der im Programmfragment aus Abbildung 9.12 verwendeten Aktion `injectWater` ist in Abbildung 9.15 dargestellt. Als Aktion der Brühgruppe verwendet sie die Wasserdosierung nach Abbildung 9.9 und schaltet parallel dazu die Druckpumpe ein. Der Wasserstrahl drückt gegen den oberen Brühkolben und würde die Presse anschieben. Aus diesem Grund wird zusätzlich die Presse entgegen der Strahlrichtung gefahren; ihre Leistung dabei ist so abgestimmt, daß sich in Summe keine Bewegung ergibt. Man erkennt, daß sowohl die Presse als auch die Pumpe selbst nichtterminierend angesteuert werden (`do ...until never`). Die drei parallelen Vorgänge zusammen terminieren mit dem Ende der Wasserdosierung (`parallelize ...terminating`).

```

action injectWater (value brewingUnit : BrewingUnit,
                    value amount      : FlowWaterAmount)
: BrewingUnitControl = (
  parallelize {
    press : do
      move (brewingUnit.press,
            directionTowardsFlow,
            injectionResistancePower)
      until never,
    flow  : measureOff (brewingUnit.flow, amount),
    pump  : do keep on until never
  }
  terminating
)
  
```

Abbildung 9.15: Wasserstrahl mit Gegendruck

Eine weitere Spezialität für sequentielle Kontrollflüsse ist das Konzept der Ausnahmebehandlung, das für FSPL gemäß Kapitel 6.6.2 umgesetzt wurde. Ein interessantes Ergebnis der Programmierung des Kaffeemaschinen-Beispiels ist, daß die entsprechenden Sprachkonstrukte nicht verwendet wurden, obwohl die Anwendung sehr wohl Ausnahmebehandlungen einschließt (siehe z. B. Abbildung 9.7). Die einfache Einbettung sequentieller Abläufe in reaktives Verhalten reicht in der Praxis möglicherweise aus, um unterbrechende Ereignisse auf der reaktiven Ebene zu behandeln.

*Ausnahme-  
behandlung*

```

type FlowWaterAmount = Integer16U

type FlowModule(type Input)(type Output) =
interface {
    type Flow
    type FlowControl

    // Construction and I/O
    value makeFlow (process control : FlowControl) : Input -> Flow
    value flowOutput : Flow -> Output

    // Control
    action measureOff (value flow    : Flow,
                     value amount : FlowWaterAmount) : FlowControl
    phase noFlow      (value flow    : Flow)              : FlowControl

    // Monitoring
    event flowFailure (value flow : Flow)
}

```

Abbildung 9.16: Durchflußkontrolle: Schnittstelle abstrakter Maschine in FSPL

## 9.5 Daten- und Verhaltensabstraktion

Die Konzepte von Daten- und Verhaltensabstraktion hinterliegen dem Entwurfsmuster der Abstrakten Maschine (siehe Kapitel 8), nach welchem die meisten Module im Anwendungsbeispiel Kaffeemaschinensteuerung entworfen sind.

*Abstrakten  
Maschinen*

Abbildung 9.16 zeigt die Schnittstelle einer abstrakten Maschine für einen (Wasser-)Fluß. In dem sehr einfachen Befehlssatz gibt es nur

eine Aktion für das Abmessen einer Wassermenge `measureOff` und als Nicht-Operation eine Phase `noFlow`. Für die Dauer einer Ausführung von `measureOff` wird der Fluß (z. B. durch ein Ventil) geöffnet. In der Phase `noFlow` ist der Fluß gestoppt. Um einen (unendlichen) Prozeß zur Steuerung eines Flusses zu konstruieren, stehen nur diese beiden Elemente von einem `FlowControl`-Typ zur Verfügung. Im Prinzip werden durch die beiden Konstruktoren zwei mögliche Zustände des Flusses (offen und geschlossen) repräsentiert. Das Öffnen ist jedoch klar limitiert, nämlich durch eine anzugebende Wassermenge, aus der sich die Dauer einer Öffnung ergibt. Das zusätzliche Ereignis `flowFailure` dient der Überwachung und Fehlerbehandlung. So kann bei unterbrochener oder unzureichender Wasserzufuhr eine nichtterminierende oder überlange Ausführung von `measureOff` erkannt und im Programm behandelt werden.

Abbildung 9.17 zeigt eine Implementierung der abstrakten Maschine nach Abbildung 9.16. Sie basiert auf einem Schaltventil und einem Durchflußmesser auf Basis eines Schaufelrades, der bei jeder Umdrehung des Schaufelrades ein Ereignis liefert, das dem Durchfluß einer bestimmten, konstanten Wassermenge entspricht. Der Integer-Datentyp `FlowWaterAmount` ist auf diese Wassermenge als „Wassereinheit“ normiert<sup>2</sup>. Das Abmessen einer in „Wassereinheiten“ bemessenen Wassermenge besteht somit im Abzählen der entsprechenden Anzahl von Schaufelrad-Ereignissen, während das Schaltventil in dieser Zeit geöffnet wird.

Die beiden Typ-Parameter des Modul-Typs (`Input` und `Output`), werden dementsprechend mit einem Ereignis-Typ für die Eingabe (Schaufelrad) und einem booleschen Signal-Typ für die Ausgabe (Schaltventil) appliziert. Für das Feststellen des Ereignisses `flowFailure` deklariert die Modul-Implementierung einen Parameter `flowFailureTimeout`, der bei der Verwendung des Moduls frei appliziert werden kann. Er bestimmt die Zeitspanne, innerhalb der bei geöffnetem Ventil eine Schaufelrad-Umdrehung erkannt werden muß.

Die Repräsentation eines `Flow`-Objektes besteht aus dem Eingabe-Ereignis und dem Ausgabe-Signal. Das Ausgabesignal entspricht direkt dem Steuerungsprogramm; der Zustandsraum der Flußsteuerung

---

<sup>2</sup>An dieser Stelle ist ein gewisser Abstraktionsverlust zu beklagen, da der schnittstellenrelevante Datentyp `FlowWaterAmount` auf das Meßverfahren der konkreten Implementierung zugeschnitten ist.

```

type FlowSensing      = Event                // Flow meter turbine revolution
type ValveActuation = Process Boolean // Valve open

value flowMeterValveFlowModule (value flowFailureTimeout : Time)
: FlowModule(FlowSensing)(ValveActuation) =
let
  process open   : Boolean = const true
  process closed : Boolean = const false
in
module {
  type Flow = {
    waterUnit : FlowSensing,
    valve      : ValveActuation
  }
  type FlowControl = Boolean

  value makeFlow (process control : FlowControl)
                (event flowMeterEventWaterUnit) : Flow
  = {
    waterUnit : flowMeterEventWaterUnit,
    valve      : control
  }

  signal flowOutput (value flow : Flow) : Boolean
  = flow.valve

  action measureOff(value flow   : Flow,
                  value amount : FlowWaterAmount) : FlowControl
  = do
    keep open
    until
      amount * flow.waterUnit

  phase noFlow(value flow : Flow) : FlowControl
  = keep closed

  event flowFailure(value flow : Flow)
  = (no flow.waterUnit within flowFailureTimeout) %
    (previous (flow.valve == open) initially false)
}

```

Abbildung 9.17: Durchflußkontrolle: Implementierung abstrakter Maschine in FSPL

entspricht genau den beiden Zuständen „geöffnet“ und „geschlossen“ des Schaltventils.

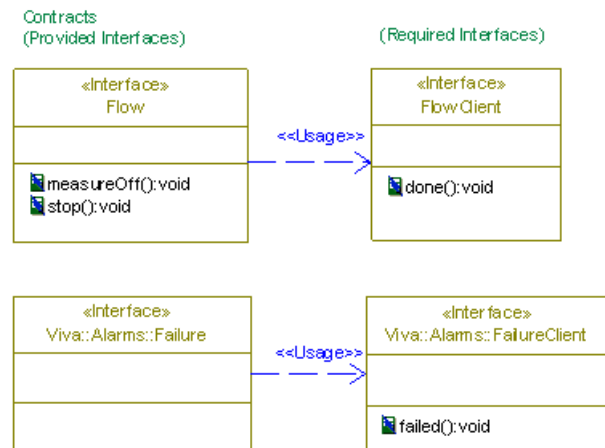


Abbildung 9.18: Durchflußkontrolle: Schnittstellen in UML

Zum Vergleich zeigt Abbildung 9.18 die Schnittstellen einer äquivalenten Implementierung der abstrakten Maschine in UML nach der in [FSMG04] beschriebenen Methodik mit dem Werkzeug *Rhapsody* von *I-Logix*. Alle dargestellten Schnittstellen-Elemente sind Ereignis-Rezeptoren, wobei `measureOff` einen in der graphischen Darstellung nicht sichtbaren Parameter für die Wassermenge hat (Ereignis mit Parameter). Die Gesamt-Schnittstelle der abstrakten Maschine zerfällt in vier Teile. Die Steuerungsschnittstelle (obere Zeile) und die Beobachtungsschnittstelle (untere Zeile) sind separiert und zerfallen jeweils nochmals in eine Aufrufschnittstelle (linke Spalte) und eine Rückruffschnittstelle (rechte Spalte).

Die Auftrennung in Steuerungs- und Beobachtungsschnittstelle wäre in FSPL nicht möglich, da in FSPL eine Schnittstelle der Typ eines Moduls und somit eindeutig ist. Die Paarbildung von Aufruf- und Rückruffschnittstelle ist jedoch ein Artefakt: Zum einen können in UML (in der Interpretation von *Rhapsody*) nicht Ereignisse (Ereignis-Generatoren), sondern nur Ereignis-Rezeptoren (ein Spezialfall von Operationen) als Schnittstellen-Elemente exportiert werden (der Empfänger muß bei Eintreten des Ereignisses aufgerufen werden); so muß für den Export eines Ereignisses eine entsprechende Rezeptionsschnittstelle eines möglichen Gegenübers importiert werden. Zum anderen entspricht der Aktion `measureOff` in FSPL hier ein Paar (`measureOff`, `done`) von Ereignissen zum Starten der Aktion und zum Rückmelden ihrer



Terminierung. Eine zu dem Aktionskonzept von FSPL analoge Abstraktion zu einer blockierenden Operation, die als normale Methode aufzurufen wäre, gelingt in der verwendeten UML-Implementierung von Rhapsody (Version 5.0) nicht, da innerhalb einer Operation nicht auf ein Ereignis gewartet werden kann. Der endgültige Standard von UML 2.0 [UML05a] sieht prinzipiell auch solche Aktionen vor. Ohne Äquivalent in UML bleibt allerdings nach wie vor der Export von Phasen als Bausteine eines reaktiven Transitionssystems.

Im Beispiel der Kaffeemaschine wurden FSPL-Module mit einer Ausnahme nur für abstrakte Maschinen eingesetzt. Ein abstrakter Datentyp für Konstanten wurde im Bereich der Benutzerschnittstelle eingeführt (siehe Anhang C.5.7), wo die gleichen abstrakten Benutzernachrichten in unterschiedlichen Anwendersprachen (Deutsch, Englisch, ...) und mit unterschiedlichen Anzeigetypen (z. B. Text- oder Graphikdisplay unterschiedlicher Größen) realisierbar sein sollen. Als Konstanten traten im übrigen vorwiegend skalare Größen auf, die auch als solche behandelt werden (z. B. `FlowWaterAmount` in der Multiplikation mit einem Ereignis). Hierfür wurden nur Typ-Aliase verwendet, die die intensionalen Typen bezeichnen. Ein Grenzfall, wo auch hier abstrakte Datentypen wünschenswert gewesen wären, ist die (effizient aufeinander abgestimmte) Codierung der Produkte und Tasten (siehe Anhang C.5.4 und C.5.6), die hier auch als Bitkodierung voll durchscheint.

## 9.6 Modularität und Variabilität

Austauschbare Module mit abstrakten Schnittstellen, komplexe Parameter (z. B. Funktionen oder Module als erstklassige Objekte) und parametrische Polymorphie (Typen als Parameter) sind Abstraktionsmechanismen, mit denen FSPL die Trennung orthogonaler Entwurfsentscheidungen und Variabilitäten und die Wiederverwendung und Generalisierung von Lösungen stützt.

Das Beispiel einer abstrakten Maschine, die auf Basis zweier anderer abstrakter Maschinen implementiert wird, verwendet alle drei Elemente. Die Schnittstelle `BrewingUnitModule` für die Brühgruppe (siehe Anhang C.5.3) wird sowohl durch das Modul `brewingUnitModule` (siehe Anhang C.5.3) als auch durch ein Ersatzmodul `testBrewingUnitModule` (siehe Anhang C.6) implementiert (letzteres wird als „Stub“ zum Testen des Moduls

*austauschbare  
Module mit  
abstrakten  
Schnittstellen*

CoffeeMachineModule der darüberliegenden abstrakten Maschine verwendet).

Das Modul brewingUnitModule mit der „echten“ Implementierung der abstrakten Brühgruppe auf Basis der abstrakten Maschinen für Presse und Wasserzufluß ist wie folgt deklariert:

```
value brewingUnitModule
  < type PressInput > < type PressOutput >
    (value pressModule : PressModule(PressInput)(PressOutput))
  < type FlowInput > < type FlowOutput >
    (value flowModule : FlowModule(FlowInput)(FlowOutput))
  (value directionToPhysicalLimit : VerticalDirection,
   value directionTowardsFlow      : VerticalDirection,
   value injectionResistancePower : MotorPower)
: BrewingUnitModule (BrewingUnitInput(PressInput)(FlowInput))
                    (BrewingUnitOutput(PressOutput)(FlowOutput))
```

*komplexe  
Parameter*

Die beiden wichtigsten Parameter sind pressModule und flowModule: Beliebige Implementierungen der beiden darunterliegenden abstrakten Maschinen Presse und Wasserzufluß als Module werden dem Modul, das die Brühgruppe implementiert, als Parameter übergeben. So wird eine Abstraktion in Termini zweier anderer Abstraktionen implementiert, ohne die jeweiligen Implementierungen voneinander abhängig zu machen. Die Schnittstellen dieser Module sind variabel in den jeweiligen, implementierungsabhängigen Ein- und Ausgabe-Typen (PressInput/PressOutput bzw. FlowInput/FlowOutput). Diese Typen werden zu Typ-Parametern von brewingUnitModule, die es zu einem universell polymorphen Objekt machen.

*universelle  
Poly-  
morphie*

*einfache  
Parameter*

Es folgen noch weitere Parameter einfacher Datentypen, die das Verhalten der Brühgruppe applizierbar machen. Die beiden Richtungsparameter entkoppeln die Logik zur Ansteuerung der Presse innerhalb der Brühgruppe von der Codierung der geometrischen Anordnung der Bauteile. injectionResistancePower ist ein experimentell zu bestimmender, physikalischer Parameter.

## 9.7 Diskussion

Wie das Beispiel in Abschnitt 9.2 zeigt, können zur Beschreibung *quasi-kontinuierlichen* Verhaltens – etwa für Regelungsalgorithmen –

bekannte Konzepte (hierarchische Signalflußgraphen, Differenzengleichungssysteme) übertragen werden. Es steht jedoch die volle Mächtigkeit der Metasprache (etwa Funktionen höherer Ordnung) zur Verfügung, um generische Lösungen zu entwickeln und wiederzuverwenden.

Die rekursive *reaktive* Programmierung mit Phasen und Ereignissen enthält hierarchische Phasentransitionssysteme (in Analogie zu Statecharts) und interaktives Verhalten als Spezialfälle. Phasen und Ereignisse als erstklassige Objekte zu behandeln, ermöglicht eine elegante Wiederverwendung und Parametrierung innerhalb von Phasenübergangssystemen, spielt aber auch eine entscheidende Rolle bei der Verhaltensabstraktion auf dem Weg zur Modularisierung einer Steuerung (Phasen und Ereignisse als Schnittstellenobjekte). Darüber hinaus läßt es die Ereignissprache von FSPL zu, mehrere Ereignisse ohne den Umweg über explizite Zwischenzustände eines Automaten miteinander zu zusammengesetzten Ereignissen zu kombinieren, wodurch sich sehr prägnante Beschreibungen reaktiven Verhaltens geben lassen (siehe Abschnitt 9.3).

Bei der Beschreibung *sequentieller* Abläufe gelingt die Abstraktion von reaktivem Verhalten derart, dass zwar primitive Aktionen in Termini reaktiven Verhaltens implementiert, aber mit Kontrollstrukturen prozeduraler Programmierung verknüpft werden. Die umgekehrte Einbettung von sequentiell in reaktives Verhalten macht ein explizites Ausnahmebehandlungskonzept innerhalb sequentiellen Verhaltens möglicherweise verzichtbar, wie das Beispiel nahelegt. Nicht überflüssig sind jedoch Parallelisierungskonstrukte, um die Nebenläufigkeit im physikalischen System abzubilden. Der für Aktionen überladene Multiplikationsoperator sorgt für Eleganz im Detail bei einer bestimmten Art von Iterationen (siehe Abschnitt 9.4).

*Daten- und Verhaltensabstraktion* nach dem Entwurfsmuster der Abstrakten Maschinen (siehe Kapitel 8) weicht stark von der gängigen Praxis des Entwurfs eingebetteter Systemen ab. FSPL ermöglicht abstrakte Modulschnittstellen derart, daß Verhalten in Termini von (zeitverbrauchenden) Aktionen und Phasen – als exportiertem „Befehlssatz“ – gesteuert wird, anstatt die Kommunikation zwischen Systemkomponenten auf Signale und Ereignisse zu beschränken oder zu reduzieren (wie z. B. in MATLAB Simulink oder in C mit globalen Variablen und `void f(void)`-Funktionen in einem Echtzeitbetriebssystem-Kontext). Das unauffällige Zusammenspiel von Aktionen, Phasen, Ereignissen und Signalen belegt die

Durchgängigkeit der verschiedenen Paradigmen in FSPL (siehe Abschnitt 9.5).

Die Abstraktionsmechanismen zur Unterstützung von *Modularität und Variabilität* (erstklassige, typisierte Module mit abstrakten Typen,  $\lambda$ -Abstraktion und parametrische Polymorphie) werden im Beispiel vollzählig beim hierarchischen Entwurf abstrakter Maschinen angewendet, um die Implementierungen der einzelnen Module voneinander zu entkoppeln. Dabei treten Module höherer Ordnung (Module mit Modulen als Parameter) auf (siehe Abschnitt 9.6). Die oben bereits erwähnten Funktionen höherer Ordnung können darüber hinaus im regelungstechnischen Algorithmenentwurf bei der Trennung von Belangen und der Entwicklung generischer Lösungen hilfreich sein (siehe Abschnitt 9.2).

## 10. Zusammenfassung

Die in dieser Arbeit entwickelte *höhere Programmiersprache FSPL für eingebettete Echtzeitsysteme* setzt sich zum Ziel, den Belangen des Anwendungsgebiets und des Software Engineering gleichermaßen gerecht zu werden.

Für das Anwendungsgebiet der Steuerungs- und Regelungstechnik wurden zur Präzisierung der Anforderungen drei Grundtypen von Prozeßsteuerungen und zugehörige Modelle dynamischen Verhaltens identifiziert: *Quasi-kontinuierliche Regelung*, *reaktive* und *sequentielle Steuerung*. Software Engineering aus der Perspektive der Programmierung wurde vornehmlich mit dem Modulkonzept in Verbindung gebracht und fordert die Einführung und Absicherung von *Abstraktionen* als wesentliches Element einer *Modularisierung*. Für die Anwendung auf das Gebiet der Prozeßsteuerung wurde ein methodisches Konzept beschrieben und der Entwicklung der Sprache zugrundegelegt: Steuerung mit konstruktiver Verhaltensabstraktion und Modularität durch Abstrakte Maschinen.

Zur Sprachunterstützung war einerseits die *Integration dreier verschiedener Verhaltensmodelle* zu einem durchgängigen Satz von Beschreibungsmitteln für quasi-kontinuierliches, reaktives und sequentielles Verhalten erforderlich. Die gefundene *Verschränkung der Beschreibungsmittel* unterstützt konstruktive Verhaltensabstraktion unmittelbar. Das integrierte Verhaltensmodell basiert dabei auf dem synchronen Systemmodell und verzichtet zudem vollständig auf imperative Variablen mit Zuweisung.

Andererseits bieten vorhandene *Konzepte höherer Programmiersprachen* eine adäquate Infrastruktur, um die Abstraktionen auszuprägen und abzusichern: Funktionale Abstraktion, Datenabstraktion und Kapselung sowie generische Abstraktion neben sehr grundlegenden Konzepten wie Bezeichnen, Definitionen und einem Typsystem. Der Sprachentwurf zur funktionalen, synchronen Sprache FSPL verbindet die zuvor entwickelten spezifischen Beschreibungsmittel für dynamisches Verhalten als *Objektsprache* mit einer allgemeinen *Metasprache* als im wesentlichen zueinander orthogonale Teile einer Programmiersprache. Die Anwendung des methodischen Konzepts Abstrakter Maschinen mit diesen Sprachmitteln beschreibt ein zugehöriges *Entwurfsmuster*.

Zur Validierung von Methodik und Sprachentwurf und zur Beleuchtung der Ergebnisse diente ein *ausgearbeitetes Programmbeispiel*, das von existierender Steuerungssoftware eines Kaffeeautomaten abgeleitet ist. Die Architektur der Steuerung konnte durchgängig nach dem methodischen Konzept entwickelt und die Anwendung in der rein funktionalen Sprache mit den gefundenen Beschreibungsmitteln dargestellt werden.

Die vorliegende Arbeit erbringt den Nachweis, daß ein Sprachentwurf möglich ist, der das Anwendungsgebiet der Steuerungs- und Regelungstechnik hinreichend vollständig und genau in problemnahen, konstruktiven Beschreibungsmitteln abbildet und zu gleicher Zeit wesentliche Anforderungen aus Software-Engineering-Prinzipien an eine Programmiersprache erfüllt und diese Prinzipien auf das Anwendungsgebiet anwendbar macht.

# 11. Ausblick

Die prototypische Implementierung von FSPL durch einen Interpreter und die Entwicklung eines größeren Anwendungsbeispiels erbringt auch einen ersten Nachweis für die Tragfähigkeit der Sprachkonzepte selbst. Eine endgültige Validierung des Sprachentwurfs kann jedoch erst nach Implementierung eines optimierenden Compilers und Codegenerierung für eine Zielformatplattform erfolgen. Vorbereitend dazu sind auch noch einige Restriktionen an den Einsatz von Sprachkonstrukten (wie z. B. zulässige Prozedertypen, rekursive Definitionen) zu präzisieren, um sie von einem Compiler überprüfbar zu machen.

Zur Einbindung in den Entwurfsablauf für Software eingebetteter Systeme sind in unterschiedliche Richtungen erweiternde Arbeiten denkbar. Zur Unterstützung des Software-Architecturentwurfs erweisen sich graphische Darstellungsmittel oft als hilfreich. Die in Anhang C.2 gegebenen Darstellungen befriedigen noch nicht. Zu untersuchen wäre auch, inwieweit sich etwa eine objektorientiert-imperative Modellierungssprache wie UML als Architekturbeschreibungssprache zusammen mit einer funktionalen Programmiersprache, die sich zum Teil leistungsfähigerer Abstraktionsmechanismen bedient, einsetzen läßt.

Regelungstechnischer Algorithmenentwurf wird sehr häufig mit Unterstützung eines Werkzeugs wie MATLAB Simulink durchgeführt, das nicht zuletzt auch zur zeitkontinuierlichen Modellierung und Simulation der Umgebung (Regelstrecke) eingesetzt werden kann. Ein Codegenerator, der FSPL-Funktionen aus Simulink-Modellen erzeugen kann, wäre eine denkbare Unterstützung für den Entwurf regelungstechnischer Module.

Zur Unterstützung der Qualitätssicherung kann zunächst an Werkzeuge für Debugging und Testautomatisierung gedacht werden. Die formal definierte Semantik von FSPL bietet darüber hinaus einen Ansatzpunkt für Methoden zur formalen Verifikation. Die rein konstruktiven Sprachelemente der Implementierungssprache FSPL reichen für Spezifikationen im allgemeinen nicht aus. Eine auf FSPL abgestimmte Spezifikationssprache oder entsprechende nicht-konstruktive Erweiterungen wären zu entwickeln.



# A. Mathematische Grundlagen

## A.1 Mengen und Funktionen

Nach einer Einführung des Mengenbegriffs werden Relationen und Funktionen als Mengen geordneter Paare eingeführt. Tupel und Folgen erscheinen daraufhin als spezielle Funktionen.

Die Darstellung folgt weitgehend [Rey98], jedoch teilweise mit abweichender Notation. Siehe auch [GS93].

### A.1.1 Mengen

Der Mengenbegriff ist für die Mathematik grundlegend. Nach Georg Cantor, dem Begründer der Mengenlehre, ist eine *Menge* „eine Zusammenfassung bestimmter wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens, welche die *Elemente* der Menge genannt werden, zu einem Ganzen“ (zitiert nach [NS62]). Für den Mengenbegriff wesentlich ist die Elementbeziehung. Sie wird mit

$$e \in M \tag{A.1}$$

notiert, wodurch ausgedrückt wird, daß das durch  $e$  bezeichnete Objekt Element der durch  $M$  bezeichneten Menge ist;

$$e \notin M \tag{A.2}$$

besagt das Gegenteil. Zwei Mengen sind gleich, wenn sie genau die gleichen Elemente enthalten. Weiterhin ist eine Menge selbst ein Objekt, das seinerseits Element einer Menge sein kann.

*aufzählende  
Mengendarstellung*

Zur Notation von Mengen sind zwei verschiedene Darstellungen gebräuchlich. In der *aufzählenden* Mengendarstellung bezeichnet

$$\{e_1, \dots, e_n\} \quad (\text{A.3})$$

eine Menge mit genau den durch  $e_1, \dots, e_n$  bezeichneten Elementen. Die Reihenfolge oder Wiederholungen der Elemente in der Aufzählung sind dabei irrelevant, da lediglich über die Elementeneigenschaft entschieden wird. So bezeichnen etwa  $\{1, 2, 3\}$  und  $\{1, 3, 2, 1\}$  die gleiche Menge. Für die leere Menge  $\{\}$  wird auch das Symbol  $\emptyset$  verwendet:

$$\emptyset \stackrel{\text{def}}{=} \{\}. \quad (\text{A.4})$$

Wird durch eine Aufzählung der Art

$$e_1, e_2, \dots$$

erkennbar eine unendliche Folge beschrieben, so kann mit

$$\{e_1, e_2, \dots\} \quad (\text{A.5})$$

(semi-formal) eine unendliche Menge dargestellt werden. So wird beispielsweise durch  $\{0, 1, 2, \dots\}$  die Menge der natürlichen Zahlen angegeben.

*beschreibende  
Mengendarstellung*

Die *beschreibende* Mengendarstellung

$$\{E \mid P\} \quad (\text{A.6})$$

definiert eine Menge durch eine gemeinsame Eigenschaft aller Elemente dieser Menge. Dabei ist  $E$  ein Ausdruck und  $P$  ein Prädikat, also ein Ausdruck, der zu einem Wahrheitswert ausgewertet wird. Mit

$$M = \{E \mid P\}$$

gilt dann

$$E \in M \Leftrightarrow P,$$

d. h.  $x \in M$  genau dann, wenn es Werte für die (freien) Variablen in  $E$  und  $P$  gibt, so daß  $E = x$  und  $P$  wahr wird. So bezeichnet beispielsweise

$\{2 \cdot n \mid n \in \mathbb{N}\}$  die Teilmenge der natürlichen Zahlen, die genau die geraden Zahlen enthält.

Welche Variablen in  $E$  und  $P$  frei sind, entscheidet sich kontextabhängig. So ist z.B. bei einer Mengenangabe  $\{i \mid i \in \mathbb{N} \wedge i \leq n\}$  davon auszugehen, daß  $n$  durch den Kontext gebunden,  $i$  jedoch frei bzw. durch die Mengenbeschreibung gebunden ist. Mehrdeutigkeiten können durch eine geänderte Notation wie

$$\{\bar{x} \mid P . E\} \quad (\text{A.7})$$

vermieden werden (siehe [GS93]), wobei  $\bar{x}$  eine Liste mit Variablen,  $P$  ein Prädikat und  $E$  ein Ausdruck ist.  $\bar{x}$  enthält dabei genau die durch die Mengenaufzählung gebundenen Variablen. Die obigen Beispiele würden dann  $\{n \mid n \in \mathbb{N} . 2 \cdot n\}$  bzw.  $\{i \mid i \in \mathbb{N} \wedge i \leq n . i\}$  geschrieben. Im folgenden wird die traditionelle Schreibweise beibehalten.

Bei Verwendung der Notation (A.6) ist Vorsicht geboten, da sich damit durch Selbstreferenz auch nicht existierende Mengen beschreiben lassen (d. h. der entsprechende Ausdruck ist bedeutungslos). Ein bekanntes Beispiel ist die Menge aller Mengen, die sich nicht selbst enthalten (Russelsches Paradoxon). Die Existenz dieser Menge führt zum Widerspruch ((A.9) folgt aus (A.8)):

$$M = \{x \mid x \notin x\} \quad (\text{A.8})$$

$$M \in M \Leftrightarrow M \notin M. \quad (\text{A.9})$$

Die Mengenaufzählung läßt sich als abgekürzte Schreibweise für einen Spezialfall der Mengenbeschreibung auffassen (vgl. [GS93]):

$$\{e_1, \dots, e_n\} \stackrel{\text{def}}{=} \{x \mid x = e_1 \vee \dots \vee x = e_n\}. \quad (\text{A.10})$$

Weiterhin kann

$$\{x \in M \mid P\} \stackrel{\text{def}}{=} \{x \mid x \in M \wedge P\} \quad (\text{A.11})$$

für eine übersichtlichere Beschreibung von Teilmengen verwendet werden.

Bestimmte Mengen sollen als bekannt und definiert vorausgesetzt werden:  $\mathbb{N}$  bezeichnet die Menge der natürlichen Zahlen unter Einschluß der 0,  $\mathbb{Z}$  die ganzen Zahlen,  $\mathbb{R}$  die reellen Zahlen und  $\mathbb{C}$  die komplexen

*wohlbekannte  
Mengen*

Zahlen. Für einige häufig verwendete Teilmengen werden besondere Bezeichnungen eingeführt (für beliebige  $m, n \in \mathbb{Z}$ ,  $a, b \in \mathbb{R}$ ):

$$m .. n \stackrel{\text{def}}{=} \{i \in \mathbb{Z} \mid m \leq i \leq n\} \quad (\text{A.12})$$

$$m .. \infty \stackrel{\text{def}}{=} \{i \in \mathbb{Z} \mid m \leq i\} \quad (\text{A.13})$$

$$-\infty .. n \stackrel{\text{def}}{=} \{i \in \mathbb{Z} \mid i \leq n\} \quad (\text{A.14})$$

$$-\infty .. \infty \stackrel{\text{def}}{=} \mathbb{Z} \quad (\text{A.15})$$

$$[a \ b] \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a \leq x \leq b\} \quad (\text{A.16})$$

$$[a \ b[ \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a \leq x < b\} \quad (\text{A.17})$$

$$]a \ b] \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a < x \leq b\} \quad (\text{A.18})$$

$$]a \ b[ \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a < x < b\} \quad (\text{A.19})$$

$$[a \ \infty[ \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a \leq x\} \quad (\text{A.20})$$

$$]a \ \infty[ \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a < x\} \quad (\text{A.21})$$

$$]-\infty \ b] \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid x \leq b\} \quad (\text{A.22})$$

$$]-\infty \ b[ \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid x < b\} \quad (\text{A.23})$$

$$]-\infty \ \infty[ \stackrel{\text{def}}{=} \mathbb{R} \quad (\text{A.24})$$

$$\mathbb{R}_+ \stackrel{\text{def}}{=} [0 \ \infty[. \quad (\text{A.25})$$

Weiterhin werden mit

$$\mathbb{B} \stackrel{\text{def}}{=} \{\text{true}, \text{false}\} \quad (\text{A.26})$$

$$\mathbb{U} \stackrel{\text{def}}{=} \{\text{unit}\} \quad (\text{A.27})$$

die Menge der Wahrheitswerte (Booleans) und eine einelementige Einheitsmenge definiert.

*Mengen-  
operationen*

Ausgehend von der beschreibenden Mengendarstellung lassen sich auf Basis der Prädikatenlogik die bekannten Mengenoperationen (*Vereinigung*, *Schnitt* und *Differenz*) und -relationen (*Teilmenge*, *Übermenge*) sowie die *Potenzmenge* einer Menge definieren:

$$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \vee x \in B\} \quad (\text{A.28})$$

$$A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \in B\} \quad (\text{A.29})$$

$$A \setminus B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \notin B\} \quad (\text{A.30})$$

$$A \subseteq B \stackrel{\text{def}}{=} (x \in A \Rightarrow x \in B) \quad (\text{A.31})$$

$$A \supseteq B \stackrel{\text{def}}{=} B \subseteq A \quad (\text{A.32})$$

$$\mathcal{P} M \stackrel{\text{def}}{=} \{A \mid A \subseteq M\}. \quad (\text{A.33})$$

Vereinigung und Schnitt lassen sich verallgemeinert definieren. Sei  $\mathcal{M}$  eine Menge, deren Elemente Mengen sind; dann seien

$$\bigcup \mathcal{M} \stackrel{\text{def}}{=} \{x \mid \exists M \in \mathcal{M}. x \in M\} \quad (\text{A.34})$$

$$\bigcap \mathcal{M} \stackrel{\text{def}}{=} \{x \mid \forall M \in \mathcal{M}. x \in M\} \quad \text{für } \mathcal{M} \neq \emptyset. \quad (\text{A.35})$$

Seien weiterhin  $I$  eine Menge und  $M$  ein Ausdruck, der eine Menge bezeichnet, wenn  $i$  ein Element von  $I$  bezeichnet; dann sind

$$\bigcup_{i \in I} M \stackrel{\text{def}}{=} \bigcup \{M \mid i \in I\} \quad (\text{A.36})$$

$$\bigcap_{i \in I} M \stackrel{\text{def}}{=} \bigcap \{M \mid i \in I\} \quad \text{für } I \neq \emptyset. \quad (\text{A.37})$$

Seien ferner  $a \in \mathbb{Z} \cup \{-\infty\}$  und  $b \in \mathbb{Z} \cup \{\infty\}$ ; dann sind

$$\bigcup_{i=a}^b M \stackrel{\text{def}}{=} \bigcup_{i \in a..b} M \quad (\text{A.38})$$

$$\bigcap_{i=a}^b M \stackrel{\text{def}}{=} \bigcap_{i \in a..b} M \quad \text{für } a \leq b. \quad (\text{A.39})$$

Seien schließlich  $M_1, \dots, M_n$  Mengen ( $n \geq 2$ ); dann sind

$$M_1 \cup \dots \cup M_n \stackrel{\text{def}}{=} \bigcup \{M_1, \dots, M_n\} \quad (\text{A.40})$$

$$M_1 \cap \dots \cap M_n \stackrel{\text{def}}{=} \bigcap \{M_1, \dots, M_n\}. \quad (\text{A.41})$$

(A.28) und (A.29) erscheinen damit als Spezialfälle.

Die *Kardinalität* einer Menge  $M$ , also die Anzahl ihrer Elemente, wird *Kardinalität* mit

$$\#M \quad (\text{A.42})$$

bezeichnet und ist entweder eine natürliche Zahl ( $\#M \in \mathbb{N}$ ) oder unendlich ( $\#M = \infty$ ).

### A.1.2 Geordnete Paare

*Geordnetes  
Paar*

Zur Definition von Relationen und Funktionen und darauf aufbauender Konzepte soll zunächst das *geordnete Paar* als ein weiteres primitives Konzept vorausgesetzt werden: Mit

$$(x:y) \quad (\text{A.43})$$

werde das geordnete Paar zweier *Individuen* (Objekte)  $x$  und  $y$  bezeichnet, und es gelte  $(x:y) = (x':y')$  genau dann, wenn  $x = x'$  und  $y = y'$  (für eine mengentheoretische Definition siehe z. B. [NS62]).

### A.1.3 Relationen

*Relation*

Eine *Relation* ist eine Menge, deren Elemente geordnete Paare sind. Die Schreibweisen

$$x \xrightarrow{\rho} y \quad \text{und} \quad x \rho y \quad (\text{A.44})$$

sind Synonyme für  $(x:y) \in \rho$ .

Für endliche Relationen (d. h. Relationen endlicher Kardinalität) kann die Schreibweise der Mengenaufzählung verwendet und wie folgt vereinfacht werden:

$$\{x_0:y_0, \dots, x_{n-1}:y_{n-1}\} \stackrel{\text{def}}{=} \{(x_0:y_0), \dots, (x_{n-1}:y_{n-1})\}. \quad (\text{A.45})$$

*leere Relation,  
Einschränkung,  
Erweiterung*

Auf Relationen können die üblichen Mengenoperationen angewendet werden. Die leere Menge ist zugleich die *leere Relation*. Bei zwei Relationen  $\rho$  und  $\sigma$  mit  $\rho \subseteq \sigma$  heißt  $\rho$  eine *Einschränkung* von  $\sigma$  und  $\sigma$  eine *Erweiterung* von  $\rho$ .

*Identitäts-  
relation*

Eine Relation  $\rho$  heißt *Identitätsrelation*, wenn für jedes Paar  $x:y \in \rho$  gilt:  $x = y$ . Zu jeder Menge  $M$  gibt es eine eindeutige *Identitätsrelation auf  $M$* :

$$\text{id}_M \stackrel{\text{def}}{=} \{(x:x) \mid x \in M\}. \quad (\text{A.46})$$

*Vorbereich,  
Nachbereich,  
Komposition,  
Spiegelung*

Weiterhin sind der *Vorbereich* (engl. *domain*), der *Nachbereich* (engl. *range*) und die *Spiegelung* einer Relation  $\rho$  sowie die *Komposition* zweier Relationen  $\rho$  und  $\sigma$  definiert:

$$\text{dom } \rho \stackrel{\text{def}}{=} \{x \mid \exists y. (x:y) \in \rho\} \quad (\text{A.47})$$

$$\text{ran } \rho \stackrel{\text{def}}{=} \{y \mid \exists x. (x:y) \in \rho\} \quad (\text{A.48})$$

$$\sigma \circ \rho \stackrel{\text{def}}{=} \{(x:z) \mid \exists y. (x:y) \in \rho \wedge (y:z) \in \sigma\} \quad (\text{A.49})$$

$$\rho^\dagger \stackrel{\text{def}}{=} \{(y:x) \mid (x:y) \in \rho\}. \quad (\text{A.50})$$

Die *Exponentiation* einer Relation  $\rho$  auf einer Menge  $M$  ist definiert durch *Exponentiation*

$$\rho^{(0)} \stackrel{\text{def}}{=} I_M \quad \rho^{(n+1)} \stackrel{\text{def}}{=} \rho \circ \rho^{(n)}. \quad (\text{A.51})$$

Für eine Relation  $\rho$  und eine Menge  $M$  sei

$$\rho \upharpoonright M \stackrel{\text{def}}{=} \rho \circ \text{id}_M \quad (\text{A.52})$$

*Einschränkung  
auf eine Menge*

die *Einschränkung von  $\rho$  auf  $M$*  und

$$\rho \lceil M \stackrel{\text{def}}{=} \text{id}_M \circ \rho \quad (\text{A.53})$$

*Gegen-  
einschränkung*

die *Gegeneinschränkung von  $\rho$  auf  $M$* . Die erste Art der Einschränkung *auf eine Menge* (bei beiden handelt es sich um Spezialfälle von Einschränkungen; nicht jede Einschränkung ist eine Einschränkung oder Gegeneinschränkung *auf eine Menge*) betrifft den Vorbereich, die zweite den Nachbereich.

### A.1.4 Funktionen

Eine Relation  $\rho$ , bei der aus  $(x : y) \in \rho$  und  $(x : y') \in \rho$  stets  $y = y'$  folgt, heißt *Funktion* (oder *Abbildung*). Der Vorbereich einer Funktion heißt auch *Definitionsbereich*, der Nachbereich *Wertebereich*. *Funktion,  
Definitions-  
und Werte-  
bereich*

Die leere Menge und Identitätsrelationen sind Funktionen, ebenso jede Einschränkung einer Funktion sowie die Komposition zweier Funktionen. Andererseits ist die Spiegelung einer Funktion nicht notwendigerweise eine Funktion; sind sowohl  $f$  als auch  $f^\dagger$  Funktionen so heißt  $f$  auch *Injektion*. *Injektion*

Ist  $f$  eine Funktion, so gibt es zu jedem  $x \in \text{dom } f$  genau ein  $y$  mit  $x \xrightarrow{f} y$  (man sagt:  $f$  bildet  $x$  auf  $y$  ab). Man nennt  $y$  das *Ergebnis der Anwendung von  $f$  auf  $x$*  und schreibt dafür auch *Funktions-  
anwendung*

$$f x \quad (\text{A.54})$$

(als eine unter vielen gebräuchlichen Schreibweisen wie z. B. auch  $f(x)$  oder  $x.f$ ). Syntaktisch soll die Funktionsanwendung als linksassoziativ angenommen werden, so daß

$$f x y = (f x) y.$$

Sind  $f$  und  $g$  Funktionen, so gilt für  $x \in \text{dom } g \circ f$ :

$$(g \circ f) x = g (f x).$$

*Gleichungsform* Funktionen werden oft in Gleichungsform beschrieben:

$$f x = E \tag{A.55}$$

definiert eine Funktion  $f$ , sofern zusätzlich ein Definitionsbereich  $D = \text{dom } f$  festgelegt wird und  $E$  ein Ausdruck ist, der stets definiert ist, wenn  $x$  einen Wert aus  $D$  bezeichnet. Es gilt dann:

$$f = \{(x : E) \mid x \in D\}.$$

*$\lambda$ -Notation*

Die gleiche Funktion kann, ohne sie zu benennen, vollständig als sogenannter (getypter)  $\lambda$ -Ausdruck

$$\lambda x \in D . E \tag{A.56}$$

notiert werden. Die Verknüpfung wird auch  *$\lambda$ -Abstraktion* genannt,  $x$  auch *Abstraktionsvariable*. So bezeichnet etwa  $\lambda x \in \mathbb{N} . x^2$  die Funktion  $\{(x : x^2) \mid x \in \mathbb{N}\}$ , die nichtnegative ganze Zahlen quadriert. Da  $\mathbb{N} \subseteq \mathbb{Z}$ , stellt z. B.  $\lambda x \in \mathbb{Z} . x^2$  eine Erweiterung dieser Funktion im Sinn der Erweiterung zweier Relationen dar.

Syntaktisch sei die  $\lambda$ -Abstraktion rechtsassoziativ, also

$$\lambda x \in X . \lambda y \in Y . E = \lambda x \in X . (\lambda y \in Y . E).$$

Für Definitionsbereiche  $n..m$  mit  $n \in \mathbb{N}$  und  $m \in \mathbb{N} \cup \{\infty\}$  kann auch geschrieben werden:

$$[E]_{i=n}^m \stackrel{\text{def}}{=} \lambda i \in n..m . E. \tag{A.57}$$

*Variation*

Zur *Variation* einer Funktion  $f$  an einer einzelnen Stelle soll folgende Notation verwendet werden:

$$f[x_0 := y_0] \stackrel{\text{def}}{=} \{(x_0 : y_0)\} \cup \{(x : f x) \mid x \in \text{dom } f \setminus \{x_0\}\}. \tag{A.58}$$



### A.1.5 Tupel und Folgen

Ein  $n$ -Tupel ( $n \in \mathbb{N}$ ) ist eine Funktion  $a$  mit Definitionsbereich  $0..n-1$   $n$ -Tupel und kann als

$$[a_0, \dots, a_{n-1}] \stackrel{\text{def}}{=} \{0:a_0, \dots, n-1:a_{n-1}\} \quad (\text{A.59})$$

mit  $a_i = a\ i$  für  $0 \leq i \leq n-1$  geschrieben werden. Es gilt dann:

$$[a_0, \dots, a_{n-1}] = [a\ i]_{i=0}^{n-1}.$$

Ein  $n$ -Tupel heißt auch *endliche Folge* der Länge  $n$  (man beachte, daß  $\#a = n$ ). Funktionen mit Definitionsbereich  $0..\infty = \mathbb{N}$  heißen *unendliche Folgen*. Die  $a_i$  werden auch als *Komponenten* des Tupels bzw. als *Elemente* der Folge bezeichnet. Sind alle Elemente einer Folge einer Menge  $M$  entnommen, heißt sie auch *Folge über  $M$* . Die leere Folge  $[]$  ist identisch mit der leeren Menge  $\{\}$ . *Folge*

Ein 2-Tupel wird auch als *Paar*, ein 3-Tupel als *Tripel* usw. bezeichnet. Ein Paar  $[x, y]$  ist nicht identisch mit dem geordneten Paar  $(x : y)$ , das auch als *primitives Paar* bezeichnet werden kann. Die Verwendung der primitiven Paare soll im allgemeinen auf die Elemente von Relationen und Funktionen beschränkt werden. Man beachte ferner, daß ein 1-Tupel  $[x] = \{0:x\}$  von  $x$  zu unterscheiden ist. *Paar, Tripel*

Zwecks Kompatibilität mit gebräuchlicher mathematischer und programmiersprachlicher Notation seien ferner

$$() \stackrel{\text{def}}{=} [] \quad (\text{A.60})$$

und für  $n \geq 2$

$$(a_0, \dots, a_{n-1}) \stackrel{\text{def}}{=} [a_0, \dots, a_{n-1}]. \quad (\text{A.61})$$

Runde Klammern um ein einzelnes Element hingegen dienen stets nur der Gruppierung, d. h. der Sicherung korrekter Assoziation innerhalb von Ausdrücken. So ist

$$(a) = a \quad (\text{A.62})$$

und somit

$$(a) \neq [a].$$

### A.1.6 Mengenprodukt

*Produkt*

Im folgenden sollen weitere wichtige Notationen für Mengen von Funktionen eingeführt werden. Sei  $\theta$  eine Funktion, deren Wertebereich aus Mengen besteht;  $\theta$  heißt dann *Mengenabbildung*. Das (*kartesische*) *Produkt* von  $\theta$  ist die Menge der Funktionen

$$\prod \theta \stackrel{\text{def}}{=} \{f \mid \text{dom } f = \text{dom } \theta \wedge \forall i \in \text{dom } \theta . f i \in \theta i\}. \quad (\text{A.63})$$

Man beachte, daß im Unterschied zu (A.34, A.35) das Produkt nicht über einer (per se ungeordneten) Menge von Mengen definiert ist, sondern daß die Mengen gemäß dem Definitionsbereich von  $\theta$  „angeordnet“ sind.

*Records*

Bei Mengenabbildungen endlicher Kardinalität, etwa

$$\theta = \{l_0 : M_0, \dots, l_{n-1} : M_{n-1}\}$$

mit dem Produkt

$$\prod \theta = \{\{l_0 : x_0, \dots, l_{n-1} : x_{n-1}\} \mid x_0 \in M_0 \wedge \dots \wedge x_{n-1} \in M_{n-1}\},$$

*Felder*

*Beschriftungen*

werden die  $\{l_0 : x_0, \dots, l_{n-1} : x_{n-1}\}$  auch als Verbunde oder *Records* bezeichnet und dabei die  $(l_i : x_i)$  als *Felder* und die  $l_i$  als (Feld-)Beschriftungen.

Für weitere Spezialfälle sollen eigene Notationen eingeführt werden: Seien  $I$  eine Menge und  $M$  ein Ausdruck, der eine Menge bezeichnet, wenn  $i$  ein Element von  $I$  bezeichnet; dann ist

$$\prod_{i \in I} M \stackrel{\text{def}}{=} \prod \lambda i \in I . M, \quad (\text{A.64})$$

so daß

$$\prod \theta = \prod_{i \in \text{dom } \theta} \theta i.$$

Seien ferner  $a \in \mathbb{Z} \cup \{-\infty\}$  und  $b \in \mathbb{Z} \cup \{\infty\}$ ; dann ist

$$\prod_{i=a}^b M \stackrel{\text{def}}{=} \prod_{i \in a..b} M. \quad (\text{A.65})$$

Seien schließlich  $M_1, \dots, M_n$  Mengen ( $n \geq 2$ ); dann ist

$$\begin{aligned} M_0 \times \dots \times M_{n-1} &\stackrel{\text{def}}{=} \prod [M_0, \dots, M_{n-1}] \\ &= \{[x_0, \dots, x_{n-1}] \mid x_0 \in M_0 \wedge \dots \wedge x_{n-1} \in M_{n-1}\}. \end{aligned} \quad (\text{A.66})$$

So ist  $M_0 \times \dots \times M_{n-1}$  die Menge der  $n$ -Tupel  $[x_0, \dots, x_{n-1}]$ , so daß alle  $x_i \in M_i$ .

### A.1.7 Mengensumme

Für eine Mengenabbildung  $\theta$  sei die *Summe* (oder *disjunkte Vereinigung*) wie folgt definiert: *Summe*

$$\begin{aligned} \sum \theta &\stackrel{\text{def}}{=} \{[l, x] \mid l \in \text{dom } \theta \wedge x \in \theta l\} \\ &= \bigcup_{l \in \text{dom } \theta} \{[l, x] \mid x \in \theta l\}. \end{aligned} \quad (\text{A.67})$$

Bei Mengenabbildungen endlicher Kardinalität heißen die Paare  $[l, x]$  auch *Varianten* und die  $l$  (Varianten-)Beschriftungen. *Varianten*

Analog zum Produkt können Spezialfälle definiert werden:

$$\sum_{i \in I} M \stackrel{\text{def}}{=} \sum_{i \in I} \lambda i . M \quad (\text{A.68})$$

$$\sum \theta = \sum_{i \in \text{dom } \theta} \theta i \quad (\text{A.69})$$

$$\sum_{i=a}^b M \stackrel{\text{def}}{=} \sum_{i=a..b} M \quad (\text{A.70})$$

$$\begin{aligned} M_0 + \dots + M_{n-1} &\stackrel{\text{def}}{=} \sum [M_0, \dots, M_{n-1}] \\ &= \{[0, x] \mid x \in M_0\} \cup \dots \cup \{[n-1, x] \mid x \in M_{n-1}\}. \end{aligned} \quad (\text{A.71})$$

Bei der Summenbildung werden die Mengen also zunächst durch Paarung ihrer Elemente mit dem jeweiligen Mengenindex disjunkt gemacht und dann vereinigt.

### A.1.8 Mengenpotenz

Seien  $A$  und  $M$  Ausdrücke für Mengen und die Variable  $i$  nicht frei in  $M$ . Dann ist die *Mengenpotenz*  $M^A$  wie folgt definiert: *Potenz*

$$M^A \stackrel{\text{def}}{=} \prod_{i \in A} M = \{f \mid \text{dom } f = A \wedge \text{ran } f \subseteq M\}. \quad (\text{A.72})$$

$M^A$  ist also die Menge der Funktionen mit Definitionsbereich  $A$  und Werten in  $M$ . Synonym kann auch die sehr gebräuchliche Notation  $A \rightarrow B$  für Funktionsmengen verwendet werden: *Funktionsmenge*

$$A \rightarrow B \stackrel{\text{def}}{=} B^A. \quad (\text{A.73})$$

Dabei sei  $\rightarrow$  rechtsassoziativ, also

$$A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C).$$

Mit

$$M^n \stackrel{\text{def}}{=} M^{0..n-1} \quad (\text{A.74})$$

bildet die Menge der endlichen Folgen der Länge  $n \in \mathbb{N}$  mit Elementen aus  $M$  einen Spezialfall. Für  $n \geq 2$  gilt dabei

$$M^n = \underbrace{M \times \cdots \times M}_n.$$

$M^0$  ist die einelementige Menge  $\{[]\} = \{\emptyset\}$  und damit isomorph zu  $\mathbb{U}$ .  $M^1 = \{(0:x) \mid x \in M\}$  ist isomorph zu  $M$ .

Die Notation  $M^n$  läßt sich auf den Fall unendlicher Folgen über  $M$  ausdehnen:

$$M^\infty \stackrel{\text{def}}{=} M^{0..\infty} = M^{\mathbb{N}}. \quad (\text{A.75})$$

Damit können weiter mit

$$M^* \stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} M^n \quad M^+ \stackrel{\text{def}}{=} \bigcup_{n=1}^{\infty} M^n \quad M^\omega \stackrel{\text{def}}{=} M^* \cup M^\infty \quad (\text{A.76})$$

die Menge aller endlichen Folgen, aller nichtleeren endlichen Folgen bzw. aller Folgen über  $M$  definiert werden.

### A.1.9 Mehrstellige Relationen und Funktionen

*Tupel*

Auf Basis primitiver Tupel wurden zweistellige Relationen und Funktionen mit einem Argument definiert. Auf Basis von  $n$ -Tupeln können ohne zusätzliche Primitive mehrstellige Relationen und Funktionen dargestellt werden:

Eine  $n$ -stellige Relation ist eine Teilmenge eines Mengenprodukts  $M_0 \times \cdots \times M_{n-1}$ . Eine  $n$ -stellige Funktion mit Argumenten aus  $M_0, \dots, M_{n-1}$  und Ergebnissen aus  $M$  läßt sich als einstellige Funktion aus  $M^{M_0 \times \cdots \times M_{n-1}}$  darstellen, die als Argument ein  $n$ -Tupel akzeptiert. Man beachte, daß die gewählte Tupel-Schreibweise (siehe (A.59) und (A.61)) zu Funktionsanwendungen der Form

$$f [x_0, \dots, x_{n-1}] \quad (\text{A.77})$$

oder

$$f(x_0, \dots, x_{n-1}) \quad (\text{A.78})$$

führt. Mit (A.78) ist es möglich, der weit verbreiteten Syntax für die Funktionsanwendung mit runden Klammern und mehreren Argumenten zu folgen, wobei nach (A.62) auch

$$f(x) = f x.$$

Um bei dieser Vorgehensweise die  $\lambda$ -Abstraktion zu vereinfachen, sei *Patterns* folgende Erweiterung definiert:

$$\begin{aligned} \lambda[x_0 \in M_0, \dots, x_{n-1} \in M_{n-1}]. E &\stackrel{\text{def}}{=} \\ \lambda x \in M_0 \times \dots \times M_{n-1}. (\lambda x_0 \in M_0. \dots \lambda x_{n-1} \in M_{n-1}. E) (x\ 0) \dots (x\ (n-1)). \end{aligned} \quad (\text{A.79})$$

Zum Beispiel kann die binäre Addition in  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  geschrieben werden als

$$\lambda[x \in \mathbb{R}, y \in \mathbb{R}]. x + y.$$

$[x_0 \in M_0, \dots, x_{n-1} \in M_{n-1}]$  in (A.79) heißt auch *Muster* (engl. *pattern*).

Eine alternative Darstellungsweise für  $n$ -stellige Funktionen ist das *Currying* im englischen Sprachraum so genannte *Currying*<sup>1</sup>. Hierbei ist eine  $(n+1)$ -stellige Funktion mit Argumenten aus  $M_0, \dots, M_n$  und Ergebnissen aus  $M$  eine Funktion, die Argumente aus  $M_0$  auf  $n$ -stellige Funktionen mit Argumenten aus  $M_1, \dots, M_n$  und Ergebnissen aus  $M$  abbildet. Eine einstellige Funktion eine gewöhnliche Funktion. Insgesamt ist eine  $n$ -stellige Funktion mit Argumenten aus  $M_0, \dots, M_{n-1}$  und Ergebnissen aus  $M$  also ein Element von  $(\dots (M)^{M_{n-1}} \dots)^{M_0}$ . Funktionsanwendungen haben dann die Form

$$f\ x_0 \dots x_{n-1} \quad (\text{A.80})$$

wobei die Linksassoziativität der Funktionsanwendung zum Tragen kommt.

Beide Darstellungsweisen sind äquivalent, da

$$M_0 \times \dots \times M_{n-1} \rightarrow M = M^{M_0 \times \dots \times M_{n-1}}$$

und

$$M_0 \rightarrow \dots \rightarrow M_{n-1} \rightarrow M = (\dots (M)^{M_{n-1}} \dots)^{M_0}$$

zueinander isomorph sind.

<sup>1</sup>benannt nach dem Logiker *Haskell B. Curry*

### A.1.10 Operationen auf Tupeln und Folgen

Tupel und Folgen lassen sich direkt als Funktionen konstruieren und auswerten. Zusätzliche Operationen sind jedoch zweckmäßig, um ihrem Charakter als *Listen* von Elementen einer Menge zu entsprechen. Neben der aufzählenden Schreibweise für Tupel (endliche Folgen) nach (A.59) ist die *induktive* Darstellung von großer Bedeutung. Folgen der Länge 0 sind identisch mit der leeren Folge []. Zur Konstruktion einer Folge der Länge  $n + 1$  aus einem neuen Element und einer Folge der Länge  $n$  dient der folgende Operator (*Listenkonstruktor*):

*Listen-  
konstruktor*

$$a \triangleleft x \stackrel{\text{def}}{=} \lambda i \in 0..n. \begin{cases} a & \text{falls } i = 0 \\ x(i-1) & \text{falls } 1 \leq i \leq n. \end{cases} \quad (\text{A.81})$$

Dabei seien  $M$  eine beliebige nichtleere Menge,  $a \in M$ ,  $x \in M^n$  und  $n \in \mathbb{N}$ .

Syntaktisch gilt:

$$a \triangleleft b \triangleleft x = a \triangleleft (b \triangleleft x).$$

Damit gilt:

$$[a_0, \dots, a_{n-1}] = a_0 \triangleleft \dots \triangleleft a_{n-1} \triangleleft []. \quad (\text{A.82})$$

Der Operator  $\triangleleft$  läßt sich auf unendliche Folgen übertragen ( $x \in M^\infty$ ):

$$a \triangleleft x \stackrel{\text{def}}{=} \lambda i \in \mathbb{N}. \begin{cases} a & \text{falls } i = 0 \\ x(i-1) & \text{falls } i > 0. \end{cases} \quad (\text{A.83})$$

*erstes Element,  
Rest*

Das Gegenstück zum Listenkonstruktor bilden die beiden folgenden Operatoren zur Zerlegung einer (nichtleeren) Folge in das erste Element und den Rest ( $x \in M^+ \cup M^\infty$ ):

$$\begin{aligned} \text{fst } x &\stackrel{\text{def}}{=} x 0 \\ \text{rst } x &\stackrel{\text{def}}{=} \lambda i \in D. x(i+1), \end{aligned} \quad (\text{A.84})$$

wobei  $D = 0..n-2$ , falls  $x \in M^n$  für ein  $n \in \mathbb{N}$ , und  $D = \mathbb{N}$ , falls  $x \in M^\infty$ . Es gilt:

$$\text{fst } (a \triangleleft x) = a \quad (\text{A.85})$$

$$\text{rst } (a \triangleleft x) = x. \quad (\text{A.86})$$

Und für nichtleere Folgen  $x$  gilt:

$$x = (\text{fst } x) \triangleleft (\text{rst } x). \quad (\text{A.87})$$

Speziell für Paare gedacht, aber auch für alle Folgen mit mindestens *zweites Element* zwei Elementen definiert, ist der folgende Operator ( $x \in M^n$ ,  $n \geq 2$ ,  $n \in \mathbb{N}$  oder  $x \in M^\infty$ ):

$$\text{snd } x \stackrel{\text{def}}{=} x \cdot 1. \quad (\text{A.88})$$

Damit gilt:

$$\text{fst } [a, b] = a \quad (\text{A.89})$$

$$\text{snd } [a, b] = b. \quad (\text{A.90})$$

Die Verkettung  $x \frown y$  zweier Folgen  $x$  und  $y$  ist rekursiv wie folgt *Verkettung* definiert:

$$x \frown y \stackrel{\text{def}}{=} \begin{cases} x & \text{falls } \#x = \infty \\ y & \text{falls } \#x = 0 \\ (\text{fst } x) \triangleleft ((\text{rst } x) \frown y) & \text{sonst.} \end{cases} \quad (\text{A.91})$$

So gilt etwa ( $a_i, b_j \in M$ ,  $b \in M^\omega$ ):

$$\begin{aligned} [a_0, \dots, a_{n-1}] \frown [b_0, \dots, b_{m-1}] &= [a_0, \dots, a_{n-1}, b_0, \dots, b_{m-1}] \\ [a_0, a_1, \dots] \frown b &= [a_0, a_1, \dots]. \end{aligned}$$

## A.2 Aussagen und Prädikate

Aussagen- und Prädikatenlogik sind grundlegende Teilgebiete der Logik. Einen Überblick zu den hier verwendeten Begriffen geben [BSMM01] und [Wik05a, Wik05b, Wik05c].

Eine *Aussage* ist ein Satz, der entweder wahr oder falsch ist. In dieser Arbeit wird im allgemeinen nicht zwischen Aussagen und  $\mathbb{B}$ -wertigen (booleschen) Ausdrücken unterschieden (zur Definition der Menge  $\mathbb{B}$  siehe Abschnitt A.1.1). Zu den atomaren Aussagen gehören deshalb die aussagenlogischen Konstanten

true

als eine wahre und

false

als eine falsche Aussage. Weiterhin ist für ein Objekt  $e$  und eine Menge  $M$

$$e \in M$$

eine Aussage, die genau dann wahr ist, wenn  $e$  Element der Menge  $M$  ist. Damit sind nach (A.44) auch

$$x \xrightarrow{\rho} y \quad \text{und} \quad x \rho y$$

für Objekte  $x$  und  $y$  und eine Relation  $\rho$  Aussagen.

*Negation*  
*Konjunktion*  
*Disjunktion*

Sind  $a$  und  $b$  Aussagen, so sind auch  $\neg a$ ,  $a \wedge b$  und  $a \vee b$  Aussagen, wobei gilt:

$\neg a$  ist wahr genau dann, wenn  $a$  falsch ist.  
(Negation) (A.92)

$a \wedge b$  ist wahr genau dann, wenn sowohl  $a$  als auch  $b$  wahr ist.  
(Konjunktion) (A.93)

$a \vee b$  ist wahr genau dann, wenn mindestens eine der Aussagen  $a$  oder  $b$  wahr ist (Disjunktion). (A.94)

$\neg$  bindet stärker als  $\wedge$  und  $\wedge$  stärker als  $\vee$ . So ist z. B.

$$\neg a \vee b \wedge c$$

als

$$(\neg a) \vee (b \wedge c)$$

zu lesen.

*Implikation*  
*Äquivalenz*

Ferner wird definiert:

$$a \Rightarrow b \stackrel{\text{def}}{=} \neg a \vee b \quad (\text{Implikation}) \quad (\text{A.95})$$

$$a \Leftrightarrow b \stackrel{\text{def}}{=} a = b \quad (\text{Äquivalenz}). \quad (\text{A.96})$$

*Prädikat*

*Prädikate* sind  $\mathbb{B}$ -wertige Funktionen, also Funktionen

$$P \in X \rightarrow \mathbb{B}$$

für beliebige Mengen  $X$ . Für ein Prädikat  $P \in X \rightarrow \mathbb{B}$  und ein Objekt  $x \in X$  ist also

$$P x$$



eine Aussage.

Weiterhin werden *Quantoren* definiert, die Aussagen aus Prädikaten gewinnen. Ist  $P \in X \rightarrow \mathbb{B}$  ein Prädikat, so sind  $\bigwedge P$  und  $\bigvee P$  Aussagen, für die gilt:

*All-Quantor*  
*Existenz-Quantor*

$\bigwedge P$  ist wahr genau dann, wenn  $P x$  für alle  $x \in X$  wahr ist  
(*All-Quantor, Generalisierung*). (A.97)

$\bigvee P$  ist wahr genau dann, wenn ein  $x \in X$  existiert, für das  
 $P x$  wahr ist (*Existenz-Quantor, Partikularisierung*). (A.98)

Weiterhin werden folgende abkürzende Schreibweisen verwendet:

$$\forall x \in X. E \stackrel{\text{def}}{=} \bigwedge (\lambda x \in X. E) \quad (\text{A.99})$$

$$\exists x \in X. E \stackrel{\text{def}}{=} \bigvee (\lambda x \in X. E). \quad (\text{A.100})$$

Dabei sei  $E$  ein Ausdruck, der definiert ist und einen Wahrheitswert bezeichnet, wenn  $x$  einen Wert aus  $X$  bezeichnet.



## B. Definition von FSPL

Während die Hauptkapitel 5, 6 und 7 die Konzeption der im Rahmen dieser Arbeit entwickelten Sprache und die wesentlichen Entwurfsentscheidungen beschreiben, stellt dieses Anhang-Kapitel logisch gesehen die *Implementierung* dar. Es beschreibt in formaler Weise die Definition der Syntax und der Semantik von FSPL und diskutiert dabei Detail-Entwurfsentscheidungen. Die jeweils hinführenden Erläuterungen zu den einzelnen Konzepten, die zusammen die Pragmatik der Sprache erläutern, werden hier im allgemeinen nicht wiederholt, sind aber zum Verständnis wichtig. Es wird jeweils auf die zugehörigen Abschnitte aus Kapitel 7 verwiesen.

Die Gliederung ist wie folgt: Zunächst wird der verwendete Ansatz zur formalen Definition der Sprache dargestellt (Abschnitt B.1). Anschließend beschreiben die Abschnitte B.2 und B.3 die abstrakte Syntax und die Semantik der Sprache. Die Aufteilung in die beiden Hauptteile „Funktionale Basissprache“ und „Echtzeitbeschreibungsmittel“ ist nicht identisch mit der Aufteilung in „Objektsprache“ und „Metasprache“ in Kapitel 7, bietet aber eine für den Zweck der *inkrementellen* formalen Definition entlang der einzelnen Konzepte besser geeignete Sequenzierung (vgl. Abschnitt 7.2 („Spracharchitektur“) in Kapitel 7). Als vierter Teil folgt in Abschnitt B.4 die Definition der konkreten Syntax in einer Aufteilung, die den syntaktischen Kategorien (Ausdrücke, Typen usw.) entspricht. Während die konkrete Syntax vollständig angegeben wird, wird in der Definition der abstrakten Syntax und der Semantik der „syntaktische Zucker“ weggelassen, soweit er in Kapitel

7 mit der jeweiligen Auflösung der Varianten in die Original-Syntax beschrieben ist.

Als konkrete Syntax wird ausschließlich die textuelle Notation formal definiert. Die in Kapitel 7 angegebenen graphischen Varianten können aus formaler Sicht als zusätzlicher syntaktischer Zucker gewertet werden. Ihre Präzisierung bleibt dedizierten Entwicklungswerkzeugen für die Sprache (vgl. Kapitel 7.5) vorbehalten.

## B.1 Formalisierungskonzept

### B.1.1 Ausdrücke

Die mathematische Modellbildung macht sich die (historisch gewachsene) Sprache der Mathematik zunutze, um abstrakte (nicht-gegenständliche) Objekte zu beschreiben. Abstrakte Werte erfahren dabei eine konkrete Notation durch sprachliche *Ausdrücke*. Die beiden Ausdrücke „ $2 + 1$ “ und „ $3$ “ bezeichnen beispielsweise den gleichen Wert, nämlich eine natürliche Zahl, während „ $12$ “ eine davon verschiedene, zweite natürliche Zahl bezeichnet. Die Zahlzeichen „ $1$ “, „ $2$ “ und „ $3$ “, das Operationssymbol „ $+$ “ sowie eine Verknüpfung von zwei der Zahlzeichen miteinander durch Hintereinanderschreiben ohne Zwischenraum einerseits und mit dem Operationssymbol durch Hintereinanderschreiben mit Zwischenstellung des Operationssymbols („Infix“-Notation) andererseits bilden sprachliche Elemente, eingesetzt zur Konstruktion von drei Ausdrücken.

Während Ausdrücke dieser Art im mathematischen Diskurs gedanklich mit den durch sie bezeichneten abstrakten Werten identifiziert werden sollen, ist das Prinzip selbst, daß durch Ausdrücke Werte (Objekte) beschrieben werden, elementar für die Definition einer formalen Sprache. Die *Syntax* der Sprache entscheidet darüber, von welcher Art die Ausdrücke sind (z. B. Zeichenketten oder Graphiken) und welche derartigen Ausdrücke gültig sind (d. h. zur Sprache gehören) und welche nicht. Die *Semantik* der Sprache gibt jedem gültigen Ausdruck eine Bedeutung, d. h. ordnet ihm einen Wert zu.

### B.1.2 Syntax

Die für Systembeschreibungssprachen relevanten Syntaxformen sind

*textuelle vs.  
graphische  
Syntax*

entweder textuell oder eine Mischung aus graphischen und textuellen Elementen, wobei für Programmiersprachen die textuelle Form typisch ist. Der in dieser Arbeit vorgelegte Sprachentwurf ist vollständig und formal mit einer textuellen (Primär-)Syntax ausgeführt; die Darstellung zeigt aber an geeigneten Stellen alternative, graphische Notationsmöglichkeiten auf, ohne sie jedoch formal zu beschreiben. Davon ausgehend kann eine semi-graphische Sekundär-Syntax entwickelt werden; bei geeigneter Werkzeugunterstützung wäre ein „Umschalten“ zwischen beiden Notationen möglich. Als existierendes Vorbild für eine derartige Doppel-Syntax kann die Sprache SDL [SDL00] gelten.

Bei der Definition der Syntax einer Sprache ist zunächst zwischen der *konkreten Syntax*, die die Notation im Detail definiert, und einer von der konkreten Notation abstrahierenden, *abstrakten Syntax* zu unterscheiden, die sich auf die logische Struktur der Ausdrücke konzentriert. Die Entscheidung zwischen textueller und graphischer Notation ist eine Frage der konkreten Syntax, wobei die abstrakte Syntax gleich bleiben kann.

*konkrete und  
abstrakte  
Syntax*

Konkrete Syntax faßt Ausdrücke einer textuellen Sprache als lineare *Zeichenketten*, einer graphischen Sprache als flächige Zeichnungen (etwa als Vektor-Graphiken) auf. In diesem Sinn ist die Syntax einer textuellen Sprache definiert durch einen Zeichenvorrat  $C$  und eine Menge  $L \subseteq C^*$ . Alle Zeichenketten in  $L$  sind gültige Ausdrücke der Sprache. Die Menge  $L$  wird üblicherweise durch eine *Grammatik* definiert.

*Zeichenketten*

Abstrakte Syntax hingegen faßt Ausdrücke (einer textuellen Sprache) als *Terme* über einer *Signatur* auf [Bro98]. Eine Signatur kann definiert werden als ein Tripel

*Terme*

$$\Sigma = [Sort, Op, fct] \quad (B.1)$$

aus einer Menge *Sort* von Mengensymbolen (genannt *Sorten*), einer Menge *Op* von Operationssymbolen (genannt *Operatoren*) und einer Abbildung  $fct \in Op \rightarrow Sort^+$ .  $fct$  ordnet jedem  $op \in Op$  eine *Funktionalität*  $fct\ op$  zu. Die Menge *Term*  $\Sigma$  der (Grund-)Terme über  $\Sigma$  ist dann die Menge  $T = \bigcup_{S \in Sort} T_S$  für die jeweils kleinsten Mengen  $T_S$ , für die gilt (mit  $op \in Op$ ;  $S, S_i \in Sort$ ;  $t_i \in T$ ;  $n \geq 1$ ):

1.  $fct\ op = [S] \Rightarrow op \in T_S$
2.  $fct\ op = [S_0, \dots, S_{n-1}, S] \wedge \forall i \in 0..n-1. t_i \in T_{S_i}$   
 $\Rightarrow (op\ t_0 \dots t_{n-1}) \in T_S.$

Die Terme in  $T_S$  heißen Grundterme *der Sorte*  $S$ . Die  $t_i$  in  $(op\ t_0 \ \dots\ t_{n-1})$  heißen *Operanden*.

**Beispiel B.1** Sei  $\Sigma = [Sort, Op, fct]$  mit

$$\begin{aligned} Sort &= \{\text{Boolean}\} \\ Op &= \{\text{true, false, not, and, or}\} \\ fct &= \{\text{true: [Boolean],} \\ &\quad \text{false: [Boolean],} \\ &\quad \text{not: [Boolean, Boolean],} \\ &\quad \text{and: [Boolean, Boolean, Boolean],} \\ &\quad \text{or: [Boolean, Boolean, Boolean]}\} \end{aligned}$$

*eine einsortige Signatur. Dann gilt beispielsweise*

$$\begin{aligned} \text{true} &\in \text{Term } \Sigma \\ (\text{not false}) &\in \text{Term } \Sigma \\ (\text{not (and true false)}) &\in \text{Term } \Sigma. \end{aligned}$$

□

Für eine weitergehende Diskussion von Signaturen und Termen, siehe z. B. [Kre91, Bro98].

Wo die konkrete Syntax die Ausdrücke einer Sprache als „flache“, eindimensionale Zeichenketten betrachtet, sind sie für die abstrakte Syntax baumartige Strukturen, zumindest bei textuellen Sprachen. Nicht alle graphischen Ausdrücke lassen sich auf natürliche Weise ebenfalls zu Termen abstrahieren, so daß man zur Definition graphischer Sprachen die abstrakte Syntax auch mittels Graphen oder anderer Strukturen definiert. Die abstrakte Syntax der Sprache UML etwa ist als objektorientiertes Datenmodell in MOF [OMG02] spezifiziert (siehe [UML03, UML05a]).

*lexikalische  
und grammati-  
kalische  
Syntax*

Die Extraktion eines *abstrakten Syntax-Baums* aus einer Zeichenkette erfolgt typischerweise zweistufig. Man unterscheidet deshalb innerhalb der konkreten Syntax noch zwischen einem *lexikalischen* und einem *grammatikalischen* Teil der Syntax als zwei aufeinander aufbauenden Definitionsschichten. Die lexikalische Syntax betrifft die Assoziation der Zeichenkette mit einer Kette von Symbolen (engl. *Tokens*) durch das Erkennen der den Symbolen entsprechenden Zeichen oder Zeichenfolgen und das „Überlesen“ von Leerräumen (engl. *Whitespace*), die sich aus als solchen spezifizierten Leerzeichen und „Kommentaren“

ergeben; die Symbole entsprechen den Wörtern und Satzzeichen in einer natürlichen Sprache.

Die grammatikalische Syntax betrifft die Assoziation der Token-Kette mit einer geschachtelten Struktur, entsprechend der Satzbildung in einer natürlichen Sprache. Analog zum Whitespace bei der lexikalischen Analyse werden hierbei Klammerungssymbole unterdrückt und Operator-Stellungen (Präfix, Postfix, Infix) und -Prioritäten (wie die bekannte Regel „Punkt vor Strich“ in der Schulmathematik) bei der Assoziation von Operanden zu Operatoren ausgewertet.

**Beispiel B.2** *Die Zeichenkette  $1+2\_*\_(3+\_4)$  wird in vielen Programmiersprachen gemäß ihrer lexikalischen Syntax als Token-Folge  $[1, +, 2, *, (, 3, +, 4, )]$  und gemäß ihrer grammatikalischen Syntax als abstrakter Syntax-Baum  $(+ 1 (* 2 (+ 3 4)))$  gelesen, wobei hier die Operationssymbole der abstrakten Syntax als identisch mit den zugehörigen Tokens gewählt wurden (jede andere eindeutige Codierung wäre ebenfalls geeignet).* □

Der abstrakte Syntax-Baum eines Programms (seine Termstruktur) enthält die für die Semantik wesentlichen Informationen. Für die Darstellung des Sprachentwurfs in diesem Kapitel wird daher auch die abstrakte Syntax der Sprache verwendet, und zwar in einer Variante der Term-Schreibweise, die der konkreten Syntax sehr nahe kommt und nachfolgend noch erläutert wird. Eine vollständige Definition der lexikalischen, grammatikalischen und abstrakten Syntax findet sich in Anhang B.4, angegeben in dem Formalismus SDF (Syntax Definition Formalism) in der in [BK02] beschriebenen Version.<sup>1</sup> In den Programmbeispielen wird eine formatierte Variante der konkreten Syntax verwendet.

Zur Notation der abstrakten Syntax werden die Operationssymbole von der konkreten Syntax übernommen und wie auch dort teilweise „überladen“, d. h. für mehrere Operationen unterschiedlicher Funktionalität benutzt (wie etwa  $-$  als einstelliger und als zweistelliger Operator); die Stellung (wie etwa Infix bei  $+$  mit Termen der Form  $x + y$  statt  $(+ x y)$ ) sowie ggf. die Mehrteiligkeit von Operatoren (wie z. B. bei  $\text{if } x \text{ then } y \text{ else } z$ ) wird beibehalten. Die äußeren Klammern bei Termen werden meist weggelassen, wo sie nicht zur Festlegung oder Klärung

<sup>1</sup>SDF vereinigt Konzepte *kontextfreier* Grammatiken (die traditionell in Notationen wie EBNF [Wir77] beschrieben werden) für konkrete Syntax und Signaturen für abstrakte Syntax zu einem durchgängigen Formalismus.

der Assoziierung erforderlich sind. Für Sorten wird die Schreibweise  $\langle \text{Name} \rangle$  verwendet. Die Angabe der Operator-Funktionalitäten erfolgt in Anlehnung an SDF. So ist etwa

if  $\langle \text{Expression} \rangle$  then  $\langle \text{Expression} \rangle$  else  $\langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

oder allgemeiner

if  $S_0$  then  $S_1$  else  $S_2 \rightarrow S_3$

als Deklaration zu lesen, die besagt, daß (im Sinn von (B.1))

$fct(\text{if then else}) = [\langle \text{Expression} \rangle, \langle \text{Expression} \rangle, \langle \text{Expression} \rangle, \langle \text{Expression} \rangle]$

bzw.

$fct(\text{if then else}) = [S_0, S_1, S_2, S_3]$ .

Die Überladung von Operatoren kann formal so verstanden werden, daß das Konzept der Signatur nach (B.1) dahingehend abgeändert wird, daß für  $\Sigma = [Sort, Op, fct]$  gilt:

$$fct \in \mathcal{P}(Sort^+) \setminus \emptyset$$

mit der Einschränkung, daß (für alle  $op \in Op; S_i, S'_i \in Sort; n \geq 0$ ) gilt:

$$\begin{aligned} [S_0, \dots, S_{n-1}, S_n] \in fct\ op \wedge [S_0, \dots, S_{n-1}, S'_n] \in fct\ op \\ \Rightarrow S_n = S'_n. \end{aligned} \quad (B.2)$$

Die Menge  $Term\ \Sigma$  der (Grund-)Terme über  $\Sigma$  ist in dieser Variante die Menge  $T = \bigcup_{S \in Sort} T_S$  für die jeweils kleinsten Mengen  $T_S$ , für die gilt (mit  $op \in Op; S, S_i \in Sort; t_i \in T; n \geq 1$ ):

1.  $[S] \in fct\ op \Rightarrow op \in T_S$
2.  $[S_0, \dots, S_n] \in fct\ op \wedge \forall i \in 0..n-1. t_i \in T_{S_i}$   
 $\Rightarrow (op\ t_0 \dots t_{n-1}) \in T_{S_n}.$

Die Einschränkung aus (B.2) stellt sicher, daß die Überladungen eines Operators bei Auftreten in Termen unterscheidbar sind und so jeder Term eindeutig einer Sorte zuzuordnen ist.

Analog zur Modellbildung erfolgt die Sprachdefinition inkrementell, wobei Sorten und Operatoren mit ihren Funktionalitäten nach und nach eingeführt werden und sich implizit zu einer Signatur addieren. Einzelne Inkremente werden wie im folgenden Demonstrationsbeispiel dargestellt:



### Syntax B.1.2.0 (Boolesche Ausdrücke)

#### Sorten

⟨Expression⟩

#### Operatoren

true	→	⟨Expression⟩	// <i>Beispielkommentar</i>
false	→	⟨Expression⟩	
not ⟨Expression⟩	→	⟨Expression⟩	
⟨Expression⟩ and ⟨Expression⟩	→	⟨Expression⟩	
⟨Expression⟩ or ⟨Expression⟩	→	⟨Expression⟩	

□

Wiederholungen einer Deklaration in späteren Inkrementen sind zulässig und werden zum besseren Verständnis im Zusammenhang mit neuen Deklarationen an mehreren Stellen verwendet.

Die Wahl einer allgemeinen Sorte ⟨Expression⟩ anstatt einer speziellen wie ⟨Boolean⟩ oder ⟨Assertion⟩ in diesem Beispiel ist ein Vorgriff auf den Sprachentwurf und begründet sich dort aus der Integration aus Meta- und Objektsprache. Sie wird kompensiert durch die Einführung eines *Typsystems*.

### B.1.3 Typisierung

Mit der Objektsprache zur Beschreibung von Objekten des Anwendungsgebiets und der Metasprache zur Organisation von Beschreibungen überdeckt die Gesamtheit der entworfenen Sprache zwei semantische Ebenen. Durch die mathematische Modellbildung entstehen auf jeder Ebene *semantische Kategorien* in Form von Mengen, auf denen Operationen definiert werden, die innerhalb und zwischen den Mengen abbilden. Auf Ebene der Metasprache können dies z. B. die Menge der Bezeichner, der Ausdrücke, der Definitionen usw. sein. Auf Ebene der Objektsprache finden sich z. B. Mengen für Daten, Prozesse oder Ereignisse.

Wie in Abschnitt B.1.2 eingeführt, erlaubt zur Definition einer abstrakten Syntax das Konzept der Signatur die Definition *syntaktischer Kategorien* in Form von Sorten, zu denen Operatoren definiert werden. So stellt sich zwangsläufig die Frage der Abbildbarkeit der semantischen

Kategorien auf syntaktische sowie die des Verhältnisses der semantischen Kategorien der beiden Sprachebenen zueinander.

Aus dem Entwurfsprinzip *Trennung von Belangen* (oder auch *orthogonaler Entwurf*), angewendet auf den Sprachentwurf, läßt sich die Forderung ableiten, daß die Konstrukte der Metasprache im allgemeinen auf beliebige Mengen der Objektsprache anwendbar sein müssen, also in diesem Sinne *polymorph* zu definieren sind. Innerhalb der Semantik der Objektsprache tritt nach dem gleichen Prinzip ein weiterer Fall von Polymorphie auf, daß nämlich Prozesse für beliebige Zustandsräume (im Rahmen der definierbaren Datenstrukturen) definierbar sein müssen. Eine weitere Erschwernis bildet der Umstand, daß die Konstruktion beliebig tief geschachtelter Datenstrukturen bereits zu unendlich vielen, innerhalb der Objektsprache unterscheidbaren Mengen führt.

Eine syntaktische Entsprechung zur Polymorphie ist die Überladung von Operatoren für unterschiedliche Sorten. Sie weist jedoch die natürliche und in (B.2) formulierte Einschränkung auf, daß die Eindeutigkeit von Termen an Hand von Anzahl und Sorten der Operanden sichergestellt sein muß. Daher können nullstellige Operatoren (wie z. B. Bezeichner als Operatoren der Metasprache) nicht überladen werden. Gravierender ist jedoch die Existenz einer unendlich großen Anzahl von Mengen in der Semantik der Objektsprache. Definitionsformalismen für Grammatiken können jedoch im allgemeinen keine Entsprechung zu unendlich vielen Sorten bilden (z. B. unendlich viele Nicht-terminalsymbole). Eine eindeutige Extraktion der abstrakten aus der konkreten Syntax ist somit nicht ohne weiteres möglich.

Gelöst wird das Problem durch Einführung eines *Typsystems* und die Trennung der Ebenen bei der Kategorisierung von Ausdrücken. Die Kategorien der Metasprache werden auf syntaktische Kategorien (Sorten) abgebildet, während die gesamte Objektsprache in eine einzige Sorte (mit  $\langle \text{Expression} \rangle$  bezeichnet) gelegt wird. Termen der Sorte  $\langle \text{Expression} \rangle$  werden durch Typisierungsregeln *Typen* zugeordnet. Sorten repräsentieren die Kategorien der Metasprache, Typen die der Objektsprache. Das Typsystem selbst und seine Einbettung als Teilsprache in die getypte Metasprache wird in Abschnitt B.2 dargestellt.

Als Nebenwirkung dieses Konzepts können syntaktisch korrekte Ausdrücke bedeutungslos sein (wie dies z. B. bei dem Ausdruck `1 + true` der Fall sein wird; auch die deutsche Sprache weist diese Eigenschaft auf, wie als Beispiel der Satz „Der Vogel läuft grüner als darüber.“

zeigt). Der Umfang einer Sprache wird dementsprechend nicht mehr allein durch die Syntax bestimmt, sondern auch durch den Definitionsbereich der Semantik.

Eine Besonderheit im Zusammenspiel von Metasprache und Objektsprache bilden *Bezeichner* (engl. identifier). Bezeichner sind Namen für Objekte bzw. deren Beschreibungen in der Objektsprache, die durch Mechanismen der Metasprache gebunden werden (vgl. die Abschnitte 7.2 und 7.4.1). Zur Wiederverwendung von Ausdrücken oder zur Abstraktion innerhalb komplexer Beschreibungen führen *Definitionen* neue Namen für existierende (d. i. beschriebene) Objekte ein. *Parameter* zu Beschreibungen führen Namen stellvertretend für unbekannte Objekte (Variable) ein. Bezeichner können an die Stelle von Ausdrücken der Objektsprache treten und sind damit selbst Ausdrücke (wie z. B. die Variable  $x$  in dem Ausdruck  $x + 1$  in der Sprache der Mathematik). Syntaktisch betten sie sich daher in die Sorte  $\langle \text{Expression} \rangle$  ein, bilden aber eine eigene Untersorte (bezeichnet mit  $\langle \text{Id} \rangle$ ), da sie innerhalb der Metasprache an anderen Stellen (etwa in Definitionen und Parameter-Deklarationen) eigenständig auftreten (wie z. B.  $x$  im ersten Vorkommen im Ausdruck  $\lambda x \in \mathbb{N}. x + 1$  in der Sprache der Mathematik).

*Bezeichner*

### Syntax B.1.3.1 (Einbettung Bezeichner $\rightarrow$ Ausdrücke)

#### Sorten

$\langle \text{Id} \rangle, \langle \text{Expression} \rangle$

#### Operatoren

$\langle \text{Id} \rangle \rightarrow \langle \text{Expression} \rangle$

□

Untersortendeklarationen wie in Syntax B.1.3.1 sind ein Hilfsmechanismus, um die Darstellung der abstrakten Syntax weitgehend an die konkrete Syntax anzugleichen. Formal betrachtet wird ein neuer Operator mit Funktionalität  $[\langle \text{Id} \rangle, \langle \text{Expression} \rangle]$  deklariert, der aber in der Darstellung unbenannt bleibt, weil ihm kein Symbol in der konkreten Syntax entspricht. Im weiteren seien Expression die Menge der Terme der Sorte  $\langle \text{Expression} \rangle$  und Id die Menge der Terme der Sorte  $\langle \text{Id} \rangle$ .

Das Beispiel  $\lambda x \in \mathbb{N}. x + 1$ , bei dem der Bezeichner  $x$  beim ersten Vorkommen durch die  $\lambda$ -Abstraktion gebunden wird (*bindendes* Vorkommen) und beim zweiten Vorkommen durch sie gebunden ist (*gebundenes* Vorkommen), während er im Ausdruck  $(\lambda x \in \mathbb{N}. x + 1) y + x$  beim

*Kontext*

dritten Vorkommen genauso wie der Bezeichner  $y$  frei ist (*freies* Vorkommen), macht deutlich, daß Bezeichner innerhalb von Ausdrücken *kontextabhängig* sind. Freiheit und Gebundenheit von Variablen sind relativ zur Wahl des Ausschnitts: Innerhalb von  $x + 1$  ist  $x$  frei, innerhalb von  $\lambda x \in \mathbb{N}.x + 1$  tritt  $x$  jedoch nicht mehr frei auf. Mit freien Variablen innerhalb eines Ausdrucks ist stets die Vorstellung verbunden, daß sie im Kontext gebunden werden.

Um die Typbestimmung für einen Term der Sorte  $\langle \text{Expression} \rangle$  auf die Typen der darin vorkommenden Teilterme der Sorte  $\langle \text{Expression} \rangle$  in Form von Ableitungsregeln zurückführen zu können, ist daher die Bezugnahme auf den Kontext erforderlich, um die Typen frei vorkommender Variablen bestimmen zu können. Dazu werden (in Anlehnung an [Rey98]) Kontexte wie folgt formal definiert und zur Typbestimmung verwendet:

Sei  $\text{Type}$  die Menge der Typen. Dann ist ein *Kontext* eine Funktion

$$\kappa \subseteq \text{Type}^{\text{Id}}. \quad (\text{B.3})$$

Sie ordnet ausgewählten Bezeichnern Typen zu, so etwa der Kontext  $\{id_0:T_0, \dots, id_{n-1}:T_{n-1}\}$  dem Bezeichner  $id_0$  den Typ  $T_0$  zuordnet usw.

*Typurteil*

Ein *Typurteil* (engl. *typing judgement*) ist dann eine Deklaration

$$\kappa \vdash x : T, \quad (\text{B.4})$$

*Typkorrektheit*

die besagt, daß für einen Term  $x \in \text{Expression}$  im Kontext  $\kappa$  der Typ  $T$  hergeleitet werden kann. Ein Term  $x$  heißt *typkorrekt* im Kontext  $\kappa$ , wenn ein  $T \in \text{Type}$  existiert, so daß  $\kappa \vdash x : T$ . Ein Term  $x$  heißt *typkorrekt*, wenn ein  $T \in \text{Type}$  existiert, so daß  $\emptyset \vdash x : T$ . Ein Ausdruck (z. B. ein Programm) heißt *typkorrekt*, wenn er als Term (d. h. in abstrakter Syntax) typkorrekt ist.

*Typisierungsregeln*

*Typisierungsregeln* sind dann Inferenzregeln der Form

$$\frac{\kappa_0 \vdash x_0 : T_0 \quad \dots \quad \kappa_{n-1} \vdash x_{n-1} : T_{n-1}}{\kappa \vdash x : T} \quad (\text{B.5})$$

oder

$$\frac{\kappa_0 \vdash x_0 : T_0 \quad \dots \quad \kappa_{n-1} \vdash x_{n-1} : T_{n-1}}{\kappa \vdash x : T} \quad \text{falls } P, \quad (\text{B.6})$$

wobei  $n \in \mathbb{N}$  und  $P$  eine Bedingung formuliert. Die Typurteile oberhalb der waagerechten Linie heißen die *Prämissen*, das darunter die *Konklusion* der Inferenzregel. Regeln der Form

$$\frac{}{\kappa \vdash x : T}$$

heißen *Axiome*; Regeln der Form

$$\frac{\kappa_0 \vdash x_0 : T_0 \quad \dots \quad \kappa_{n-1} \vdash x_{n-1} : T_{n-1}}{\kappa \vdash x : T} \quad \text{falls } P$$

heißen *bedingte* Regeln.

Regeln werden nachfolgend im allgemeinen als Regelschemata dargestellt, d. h. sie können freie Metavariablen enthalten, die beliebig belegt werden können. Ein Regelschema repräsentiert so eine Menge von Regeln. Eine Regel gilt dann als definiert, wenn eine Belegung der freien Variablen eines definierten Regelschemas existiert, so daß bei Einsetzung die Regel entsteht.

Ein Typurteil  $\kappa \vdash x : T$  ist dann *gültig* (und besagt damit, daß für  $x$  im Kontext  $\kappa$  der Typ  $T$  hergeleitet werden kann), wenn entweder eine Regel

$$\frac{}{\kappa \vdash x : T}$$

definiert ist oder eine Regel

$$\frac{\kappa_0 \vdash x_0 : T_0 \quad \dots \quad \kappa_{n-1} \vdash x_{n-1} : T_{n-1}}{\kappa \vdash x : T}$$

definiert ist, deren Prämissen gültig sind, oder eine Regel

$$\frac{\kappa_0 \vdash x_0 : T_0 \quad \dots \quad \kappa_{n-1} \vdash x_{n-1} : T_{n-1}}{\kappa \vdash x : T} \quad \text{falls } P$$

definiert ist, deren Prämissen gültig sind, und die Bedingung erfüllt ist.

Typisierungsregeln werden für ein Inkrement der Sprachdefinition wie im folgenden Demonstrationsbeispiel dargestellt:

**Typisierung B.1.3.0 (Boolesche Ausdrücke)**

$$\begin{array}{c}
\frac{}{\kappa \vdash \mathbf{true} : \mathbf{Boolean}} \qquad \frac{}{\kappa \vdash \mathbf{false} : \mathbf{Boolean}} \\
\\
\frac{\kappa \vdash x : \mathbf{Boolean}}{\kappa \vdash \mathbf{not} \, x : \mathbf{Boolean}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Boolean} \quad \kappa \vdash x_1 : \mathbf{Boolean}}{\kappa \vdash x_0 \mathbf{and} \, x_1 : \mathbf{Boolean}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Boolean} \quad \kappa \vdash x_1 : \mathbf{Boolean}}{\kappa \vdash x_0 \mathbf{or} \, x_1 : \mathbf{Boolean}}
\end{array}$$

□

Dabei wurde  $\mathbf{Boolean} \in \mathbf{Type}$  vorausgesetzt. Die Metavariablen  $\kappa$  und  $x$ , auch in indizierter oder gestrichener Form (wie in  $\kappa_0$  bzw.  $x'$ ) stehen in den Regelschemata stets für Kontexte bzw. Terme der Sorte  $\langle \mathbf{Expression} \rangle$ .

*Kontext-  
bedingungen*

Die in Anlehnung an [Rey98] vorgenommene Unterscheidung von (abstrakter) Syntax, Typisierung und (denotationaler) Semantik schiebt eine Zwischenbetrachtung in den Grenzbereich von Syntax und Semantik, die nicht nur eine saubere Zusammenfügung von Meta- und Objektsprache ermöglicht, sondern auch die Überprüfung von *Kontextbedingungen* regelt. Der erwähnte Umstand, daß syntaktisch korrekte Ausdrücke ohne Bedeutung bleiben, kann sich unter anderem aus der inkonsistenten Verwendung von Bezeichnern ergeben. Nicht nur die typkorrekte Verwendung sondern bereits die Definiertheit von Bezeichnern in einem Ausdruck kann von kontextfreien Grammatiken oder äquivalenten Formalismen nicht sichergestellt werden, weshalb solche Konsistenzbedingungen innerhalb von Ausdrücken gerne der Semantik zugerechnet werden. Die Einführung von Typisierungsregeln ermöglicht auf elegante Weise ihre Überprüfung als Kontextbedingungen. Die einfache Regel

$$\frac{}{\kappa \vdash id : T} \quad \text{falls } (id : T) \in \kappa$$

zur Typisierung von Bezeichnern als Ausdrücken stellt (bei geeigneten Regeln für Sprachkonstrukte, die Bezeichner binden) sicher, daß für den Ausdruck nur dann ein Typ bestimmt werden kann, wenn der Kontext den Bezeichner einführt; zugleich liefert der Kontext den Typ des Bezeichners als Typ des Ausdrucks. Regeln wie im Typisierungs-Beispiel B.1.3.0 stellen dann sicher, daß für zusammengesetzte Ausdrücke nur dann ein Typ bestimmt werden kann, wenn die Teilausdrücke zulässige Typen aufweisen. Zur Abgrenzung des Sprachumfangs kann also zusätzlich zur syntaktischen Korrektheit der Ausdrücke verlangt werden, daß (Gesamt-)Ausdrücke typkorrekt sind. Der Definitionsbereich der Semantik liegt also innerhalb der typkorrekten Ausdrücke.

#### B.1.4 Denotationale Semantik

Der hier verfolgte Ansatz einer *denotationalen* Semantik assoziiert jeden gültigen (d. h. syntaktisch korrekten, im jeweiligen Kontext typkorrekten und semantisch definierten) Ausdruck direkt mit einem mathematischen Objekt, wobei die Bedeutung von Unterausdrücken (analog zur Typbestimmung) unmittelbar zur Bedeutung eines Gesamtausdrucks beiträgt [Mos90]. Ein Programm wird dann z. B. als eine Funktion interpretiert, die die Programm-Eingabe auf die Programmausgabe abbildet, während etwa arithmetische Ausdrücke innerhalb eines Programms Zahlen repräsentieren. Im Gegensatz dazu stehen *axiomatische* und *operationale* Ansätze. Ein Programm wird dabei als eine Abfolge von Schritten (Operationen) verstanden, mit denen Zustandstransformationen einhergehen, die z. B. über Vor- und Nachbedingungen und ein Regelwerk (axiomatisch) oder auf Basis eines Maschinenmodells beschrieben werden (vgl. [E<sup>+</sup>88]).

Der denotationale Ansatz, der Programme mit mathematischen Modellen hinterlegt, weist die größte Nähe zur mathematischen Modellbildung in den Ingenieurwissenschaften auf und empfiehlt sich schon allein dadurch für den hier vorgenommenen Sprachentwurf. Er ermöglicht es ferner, ohne Umwege den Aspekt „Zeit“ in die Semantik der Sprache einzufassen.

Wie die Typbestimmung Termen der Sorte  $\langle \text{Expression} \rangle$  *Typen* zuweist, *Auswertung* weist die denotationale Semantik solchen Termen *Werte* zu. Jeder Term aus Expression, für den ein Typ bestimmt werden kann, ist entweder ohne Bedeutung (wie etwa bei einer Division durch 0) oder wird auf

einen eindeutigen Wert aus einer dem Typ entsprechenden semantischen Kategorie abgebildet. Man sagt auch: Der Ausdruck wird *interpretiert*, bzw. der Term wird *ausgewertet*. Der in Abschnitt B.1.3 beschriebene regelbasierte Ansatz zur Typbestimmung erlaubt in seiner Allgemeinheit, daß für einen Ausdruck bei gleichem Kontext mehrere Typen bestimmt werden können (z. B. bei Untertypbeziehungen, bei dem ein Term außer einem speziellen Typ auch noch übergeordnete Typen besitzen kann). Diese Option wird im vorliegenden Sprachentwurf nur so ausgewählt genutzt, daß, wann immer mehrere Typen abgeleitet werden können, diese äquivalent sind, d. h. semantisch gleich. Die nachfolgend beschriebene Darstellungsweise ist mit Ausnahme der Typäquivalenz wiederum an [Rey98] angelehnt.

*Typäquivalenz* Die Äquivalenz zweier Typen  $T$  und  $T'$  soll wie folgt notiert werden:

$$T \equiv T'. \quad (\text{B.7})$$

Zur Bestimmung von Typäquivalenz werden Regeln in einer Form analog zu den Typisierungsregeln angegeben (mit Typäquivalenzen an Stelle von Typurteilen). Das folgende Beispiel ist zugleich die Initialisierung für das Regelsystem:

#### Typäquivalenz B.1.4.1 (Äquivalenz durch Gleichheit)

$$\frac{}{T \equiv T'} \quad \text{falls } T = T'$$

□

*Typsemantik* Die Beziehung zwischen Typen und semantischen Kategorien wird beschrieben durch eine Mengenabbildung  $\mathcal{D}$  mit dem Definitionsbereich Type.  $\mathcal{D}$  wird im Lauf der Sprachdefinition inkrementell definiert mittels Definitionsgleichungen der Form

$$\mathcal{D}(T) = M. \quad (\text{B.8})$$

$\mathcal{D}(T)$  heißt auch die *Bedeutung* des Typs  $T$ . Die semantischen Kategorien sind häufig die im Rahmen der Modellbildung spezifizierten Mengen (siehe Abschnitt 6.2). Als Beispiel mag eine Fortsetzung des laufenden Beispiels dienen<sup>2</sup>:

<sup>2</sup>Hierzu sei angemerkt, daß die Modellierung von Boolean ein künstliches, einfaches Beispiel ist; in der Sprachdefinition selbst wird als semantische Domain für Booleans direkt  $\mathbb{B}$  verwendet.



**Typsemantik B.1.4.0**

$$\mathcal{D}(\text{Boolean}) = \text{Boolean}$$

□

Für die Definition von Typäquivalenzen wird, wie oben angedeutet, als Konsistenzbedingung verlangt, daß aus

$$T \equiv T'$$

stets folgt:

$$\mathcal{D}(T) = \mathcal{D}(T').$$

Für Typäquivalenz B.1.4.1 ist dies trivialerweise der Fall. Weiterhin soll gelten:

**Typisierung B.1.4.1 (Typäquivalenz)**

$$\frac{\kappa \vdash x : T}{\kappa \vdash x : T'} \quad \text{falls } T \equiv T'$$

□

Wie die Typbestimmung für einen Term von den Typen der darin enthaltenen freien Variablen abhängig ist, die diese durch den Kontext erhalten, ist auch die Auswertung eines Terms von Belegungen (Wertzuweisungen) der freien Variablen abhängig, die sich aus dem Kontext ergeben. Formal wird hierfür der Begriff *Umgebung* verwendet. Eine Umgebung ist eine Funktion  $\eta$ , die Bezeichner auf Werte aus semantischen Kategorien abbildet (analog zur Abbildung von Bezeichnern auf Typen durch einen Kontext). Der Definitions- und Wertebereich einer Umgebung  $\eta$  ergibt sich aus dem zugehörigen Kontext  $\kappa$ , wobei die gleichen Bezeichner abgebildet werden, und zwar auf Werte aus den den zugehörigen Typen entsprechenden Mengen. Dazu wird die Bedeutung  $\mathcal{D}^*(\kappa)$  eines Kontexts  $\kappa$  definiert als

*Umgebung*

$$\mathcal{D}^*(\kappa) \stackrel{\text{def}}{=} \prod_{id \in \text{dom } \kappa} \mathcal{D}(\kappa \text{ id}).$$

Für eine Umgebung  $\eta$  für einen Kontext  $\kappa$  gilt dann

$$\eta \in \mathcal{D}^*(\kappa).$$

So ist z. B.

$$\eta = \{x : \text{true}, y : \text{false}\}$$

eine Umgebung und

$$\kappa = \{x : \text{Boolean}, y : \text{Boolean}\}$$

ein Kontext, für die gilt:

$$\eta \in \mathcal{D}^*(\kappa).$$

*Bedeutung  
eines Terms*

Die Definition der Semantik für typkorrekte Terme nimmt nun Bezug auf deren Typurteile. Für jedes gültige Typurteil  $\kappa \vdash x : T$  gibt es eine *Bedeutung*

$$\llbracket \kappa \vdash x : T \rrbracket \in \mathcal{D}^*(\kappa) \rightarrow \mathcal{D}(T),$$

die dem Term  $x$  unter dem Typurteil  $\kappa \vdash x : T$  einen Wert zuordnet. Diese Art der Formalisierung der Semantik wird auch als *intrinsische Sicht* bezeichnet. Die *extrinsische* Sicht bewertet Ausdrücke unabhängig von ihrer Typisierung und muß dabei Typfehler eigenständig behandeln.

*semantische  
Gleichungen*

Die Semantik-Funktion  $\llbracket \cdot \rrbracket$  wird im Lauf der Sprachdefinition inkrementell wie im folgenden Beispiel definiert, indem für jede Typisierungsregel eine Gleichung angegeben wird (*semantische Gleichung*):

#### **Semantik B.1.4.0 (Boolesche Ausdrücke)**

$$\llbracket \kappa \vdash \text{true} : \text{Boolean} \rrbracket \eta = \text{true}$$

$$\llbracket \kappa \vdash \text{false} : \text{Boolean} \rrbracket \eta = \text{false}$$

$$\llbracket \kappa \vdash \text{not } x : \text{Boolean} \rrbracket \eta = \text{not } \llbracket \kappa \vdash x : \text{Boolean} \rrbracket \eta$$

$$\begin{aligned} \llbracket \kappa \vdash x_0 \text{ and } x_1 : \text{Boolean} \rrbracket \eta &= \text{and } [\llbracket \kappa \vdash x_0 : \text{Boolean} \rrbracket \eta, \\ &\quad \llbracket \kappa \vdash x_1 : \text{Boolean} \rrbracket \eta] \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash x_0 \text{ or } x_1 : \text{Boolean} \rrbracket \eta &= \text{or } [\llbracket \kappa \vdash x_0 : \text{Boolean} \rrbracket \eta, \\ &\quad \llbracket \kappa \vdash x_1 : \text{Boolean} \rrbracket \eta] \end{aligned}$$

□

Mittels der semantischen Gleichungen sind Auswertungen von Termen möglich durch Induktion über die Beweise der Typurteile, die im allgemeinen die gleiche Struktur haben wie die Terme selbst.

*Metavariablen*

Wenn nicht anders angegeben, werden in semantischen Gleichungen wie auch in Typisierungsregeln Metavariablen wie folgt benutzt (auch in indizierter oder gestrichener Form):

- Terme der Sorte  $\langle \text{Expression} \rangle$ :  $a, dt, e, f, x, y, ph, t$
- Terme der Sorte  $\langle \text{Id} \rangle$ :  $id$
- Typen:  $T$
- Kontexte:  $\kappa$
- Umgebungen:  $\eta$
- natürliche Zahlen in Indizes von Metavariablen:  $i, m, n$ .

Metavariablen für Terme weiterer Sorten werden an gegebener Stelle hinzugegeben.

## B.2 Funktionale Basissprache

Die funktionale Basissprache umfaßt die Metasprache und den Teil der Objektsprache, der Beschreibungsmittel für Daten und Berechnungen zur Verfügung stellt (vgl. Abschnitt 7.2). Sie kann als eigenständige und allgemeine funktionale Programmiersprache verstanden werden, wenngleich sich die Ausgestaltung der Basisdatentypen am Anwendungsgebiet orientiert. Die darüber hinaus definierten Echtzeitbeschreibungsmittel machen FSPL zu einer fachspezifischen (engl. domain-specific) Programmiersprache. Aus Sicht der funktionalen Programmierung können sie als eine Erweiterung des Satzes von Basistypen mit zugehörigen Operationen verstanden werden. Mit der Basissprache werden bereits alle wesentlichen allgemeinen Eigenschaften der Sprache (wie Typsystem, Definitions- und Abstraktionsmechanismen) festgelegt, die für zeitunabhängige Daten und Berechnungen gleichermaßen wie für Modelle zeitbehafteten Verhaltens gelten.

### B.2.1 Ausdrücke und Bezeichner

Die in Abschnitt B.1 vorweg eingeführten Sorten  $\langle \text{Expression} \rangle$  und  $\langle \text{Id} \rangle$  für Ausdrücke und Bezeichner sind elementar für die Konzeption der Metasprache. Innerhalb der *Ausdrücke* liegt die gesamte Objektsprache; ferner enthält die Metasprache Verknüpfungen von Ausdrücken zu Ausdrücken. Dabei spielen *Bezeichner* eine tragende Rolle, also Namen für Ausdrücke.

### Syntax B.2.1.1 (Ausdrücke und Bezeichner)

#### Sorten

$\langle \text{Expression} \rangle, \langle \text{Id} \rangle$

#### Operatoren

$\langle \text{Id} \rangle \rightarrow \langle \text{Expression} \rangle$

□

Die Literale und Konstruktoren für  $\langle \text{Expression} \rangle$  werden im Verlauf der Sprachdefinition inkrementell eingeführt. Die Bezeichner, die in der abstrakten Syntax durch die Sorte  $\langle \text{Id} \rangle$  repräsentiert werden, sind in der konkreten Syntax (siehe Anhang B.4) Zeichenfolgen, bestehend aus Buchstaben, Ziffern und Unterstrichen, beginnend mit einem Kleinbuchstaben. Bezeichner können als Variable in Ausdrücken auftreten, weshalb, wie in B.1.3 beschrieben, die Sorte  $\langle \text{Id} \rangle$  in die Sorte  $\langle \text{Expression} \rangle$  eingebettet wird. Bezeichner bilden den Basisfall für die Nutzung von Kontextinformation in der Typisierung und analog für Umgebungen in der Semantik:

### Typisierung B.2.1.1 (Bezeichner als Ausdrücke)

$$\frac{}{\kappa \vdash id : T} \quad \text{falls } (id : T) \in \kappa$$

□

### Semantik B.2.1.1 (Bezeichner als Ausdrücke)

$$\llbracket \kappa \vdash id : T \rrbracket \eta = \eta id$$

□

## B.2.2 Typsystem

Infolge des Typsystems (siehe Kapitel 7.4.5) enthält die Programmiersprache die Teilsprache der Typausdrücke, die durch eine eigene Sorte in der abstrakten Syntax repräsentiert wird:

**Syntax B.2.2.1 (Typausdrücke)****Sorten** $\langle \text{Type} \rangle$ 

□

Die in Abschnitt B.1.3 eingeführte Menge der Typen, *Type*, sei nun definiert als die Menge aller Terme der Sorte  $\langle \text{Type} \rangle$ . Damit werden Typen innerhalb der Typprüfung rein symbolisch verarbeitet. *Typäquivalenz* nach Abschnitt B.1.4 ist damit gleichbedeutend mit der Äquivalenz von Typausdrücken nach Kapitel 7.4.5. Das in Kapitel 7 gegebene Verständnis von Typen als Mengen bleibt durch das Konzept der *Typsemantik* aus Abschnitt B.1.4 erhalten: Äquivalente Typen (d. h. Typausdrücke) haben die gleiche Typsemantik, bezeichnen also die gleiche Menge.

Die Elemente der Typsprache werden im weiteren, begleitend zur übrigen Sprachdefinition, inkrementell eingeführt.

**B.2.3 Kardinalzahlen**

Kardinalzahlen bilden eine eigene syntaktische und semantische Kategorie der Metasprachebene. Sie dienen der Dimensionierung von Arrays und der Projektion aus Tupeln (siehe Abschnitt B.2.8) und werden auch für die sowohl in den Typnamen für Ganzzahlen als auch in Ganzzahlliteralen oder Schiebeoperationen auftretenden Zahlenwerte verwendet (siehe Abschnitt 7.3.1.2). Sie können für den letztgenannten Zweck auf nichtnegative Zahlen beschränkt werden, da negative Zahlen als Ausdrücke mit einem Negationsoperator dargestellt werden können.

**Syntax B.2.3.1 (Kardinalzahlen)****Sorten** $\langle \text{Cardinal} \rangle$ **Operatoren** $0 \rightarrow \langle \text{Cardinal} \rangle$  $1 \rightarrow \langle \text{Cardinal} \rangle$  $2 \rightarrow \langle \text{Cardinal} \rangle$  $\vdots$ 

□

In der konkreten Syntax können Kardinalzahlen zusätzlich auch in Hexadezimalschreibweise (in der Notation von C) angegeben werden. In der abstrakten Syntax besteht diese Unterscheidung nicht mehr.

In Typisierungsregeln und semantischen Gleichungen wird für Terme der Sorte  $\langle \text{Cardinal} \rangle$  die Metavariablen  $c$ , ggf. mit Indizes oder gestrichen, verwendet. Zur Verwendung in Semantik-Definitionen wird eine eigene Semantik-Funktion

$$\llbracket \_ \rrbracket_c \in \text{Cardinal} \rightarrow \mathbb{N}$$

für Terme der Sorte  $\langle \text{Cardinal} \rangle$  definiert:

#### Semantik B.2.3.1 (Kardinalzahlen)

$$\begin{aligned} \llbracket 0 \rrbracket_c &= 0 \\ \llbracket 1 \rrbracket_c &= 1 \\ \llbracket 2 \rrbracket_c &= 2 \\ &\vdots \end{aligned}$$

□

### B.2.4 Primitive Datentypen

Die Konzeption der in Kapitel 7.3.1 eingeführten primitiven Datentypen (siehe auch Kapitel 7.4.5.1) für Wahrheitswerte (Booleans), Zahlen (Ganzzahlen und Gleitkommazahlen) und Schriftzeichen (Characters) werden nachfolgend getrennt ausgeführt.

#### Syntax B.2.4.1 (Datentyp Boolean)

##### Sorten

$\langle \text{Type} \rangle, \langle \text{Expression} \rangle$

##### Operatoren

Boolean  $\rightarrow \langle \text{Type} \rangle$

true  $\rightarrow \langle \text{Expression} \rangle$

false  $\rightarrow \langle \text{Expression} \rangle$

$! \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle$
$\langle \text{Expression} \rangle \&\& \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle$
$\langle \text{Expression} \rangle    \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle$
$\langle \text{Expression} \rangle == \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle$
$\langle \text{Expression} \rangle != \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle$
$\text{if } \langle \text{Expression} \rangle \text{ then } \langle \text{Expression} \rangle \text{ else } \langle \text{Expression} \rangle$	$\rightarrow \langle \text{Expression} \rangle \quad \square$

Die zugehörige Typisierung enthält die eine Besonderheit, daß der zweite und dritte Operand eines bedingten Ausdrucks den gleichen Typ aufweisen müssen, damit der Ausdruck typkorrekt ist:

#### Typisierung B.2.4.1 (Datentyp Boolean)

$\kappa \vdash \text{true} : \text{Boolean}$	$\kappa \vdash \text{false} : \text{Boolean}$
$\kappa \vdash x : \text{Boolean}$	
$\kappa \vdash !x : \text{Boolean}$	
$\kappa \vdash x_0 : \text{Boolean} \quad \kappa \vdash x_1 : \text{Boolean}$	
$\kappa \vdash x_0 \&\& x_1 : \text{Boolean}$	
<i>analog für   , == und !=</i>	
$\kappa \vdash x_0 : \text{Boolean} \quad \kappa \vdash x_1 : T \quad \kappa \vdash x_2 : T$	
$\kappa \vdash \text{if } x_0 \text{ then } x_1 \text{ else } x_2 : T$	

□

Die Modellierung der Wahrheitswerte und ihrer Operationen – hier noch ergänzt um Gleichheit (Äquivalenz) und Ungleichheit (Exklusiv-Oder) – war bereits hinreichend Gegenstand der Beispiele in den Abschnitten 6.2 und B.1. Statt der abstrakten Menge Boolean wird als Semantik für einen Datentyp Boolean jedoch unmittelbar  $\mathbb{B}$  mit den darauf definierten Operationen (siehe Anhang A.2) verwendet.

#### Typsemantik B.2.4.1 (Datentyp Boolean)

$$\mathcal{D}(\text{Boolean}) = \mathbb{B}$$

□

**Semantik B.2.4.1 (Datentyp Boolean)**

$$\begin{aligned}
\llbracket \kappa \vdash \mathbf{true} : \mathbf{Boolean} \rrbracket \eta &= \mathbf{true} \\
\llbracket \kappa \vdash \mathbf{false} : \mathbf{Boolean} \rrbracket \eta &= \mathbf{false} \\
\llbracket \kappa \vdash ! x : \mathbf{Boolean} \rrbracket \eta &= \neg \llbracket \kappa \vdash x : \mathbf{Boolean} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 \ \&\& \ x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Boolean} \rrbracket \eta \wedge \\
&\quad \llbracket \kappa \vdash x_1 : \mathbf{Boolean} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 \ || \ x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Boolean} \rrbracket \eta \vee \\
&\quad \llbracket \kappa \vdash x_1 : \mathbf{Boolean} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 == x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Boolean} \rrbracket \eta = \\
&\quad \llbracket \kappa \vdash x_1 : \mathbf{Boolean} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 != x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Boolean} \rrbracket \eta \neq \\
&\quad \llbracket \kappa \vdash x_1 : \mathbf{Boolean} \rrbracket \eta \\
\llbracket \kappa \vdash \mathbf{if} \ x_0 \ \mathbf{then} \ x_1 \ \mathbf{else} \ x_2 : T \rrbracket \eta &= \begin{cases} \llbracket \kappa \vdash x_1 : T \rrbracket \eta & \text{falls} \\ & \llbracket \kappa \vdash x_0 : \mathbf{Boolean} \rrbracket \eta = \mathbf{true} \\ \llbracket \kappa \vdash x_2 : T \rrbracket \eta & \text{sonst} \end{cases}
\end{aligned}$$

□

Auf den Datentyp der Booleans für Wahrheitswerte und logische Berechnungen folgen die arithmetischen Datentypen für Zahlen und arithmetische sowie arithmetisch-logische Berechnungen:

**Syntax B.2.4.2 (Arithmetische Datentypen)****Sorten**

$\langle \text{IntegerFormat} \rangle$

**Operatoren**

$\langle \text{Cardinal} \rangle \ \mathbf{U}$	$\rightarrow$	$\langle \text{IntegerFormat} \rangle$	<i>//unsigned</i>
$\langle \text{Cardinal} \rangle \ \mathbf{S}$	$\rightarrow$	$\langle \text{IntegerFormat} \rangle$	<i>//signed</i>
$\mathbf{Integer} \ \langle \text{IntegerFormat} \rangle$	$\rightarrow$	$\langle \text{Type} \rangle$	
$\mathbf{Real}$	$\rightarrow$	$\langle \text{Type} \rangle$	



## //Integer-Literale

$$\langle \text{Cardinal} \rangle \text{ as Integer } \langle \text{IntegerFormat} \rangle \rightarrow \langle \text{Expression} \rangle$$

## //Real-Literale

$$0.0 \rightarrow \langle \text{Expression} \rangle$$

$$\vdots$$

## //Rechenoperationen

$$- \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle + \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle - \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle * \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle / \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle \% \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

## //Bit-Operationen

$$\sim \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle \& \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle | \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle ^ \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle << \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle >> \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

## //Vergleiche

$$\langle \text{Expression} \rangle < \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle \leq \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle \geq \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle > \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle == \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

$$\langle \text{Expression} \rangle != \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$$

□

Für die neu eingeführte Sorte  $\langle \text{IntegerFormat} \rangle$  werden die Metavariablen *fmt* oder indizierte oder gestrichene Formen davon verwendet.

Nicht alle Operatoren der Rechen- und Bit-Operationen sind auf alle arithmetischen Datentypen anwendbar (Negation nicht auf vorzeichenlose Ganzzahlen, Bit-Operationen nicht auf vorzeichenbehaftete Ganzzahlen, Divisionsrest und Bit-Operationen nicht auf Gleitkommazahlen). Soweit anwendbar, sind die Operatoren überladen und rechnen stets innerhalb genau eines Typs:

**Typisierung B.2.4.2 (Arithmetische Datentypen)**

$\frac{}{\kappa \vdash c \text{ as Integerfmt} : \text{Integerfmt}}$	$\frac{}{\kappa \vdash 0.0 : \text{Real}} \quad \dots$
$\frac{\kappa \vdash x : \text{Integer } c \text{ S}}{\kappa \vdash -x : \text{Integer } c \text{ S}}$	$\frac{\kappa \vdash x : \text{Real}}{\kappa \vdash -x : \text{Real}}$
$\frac{\kappa \vdash x_0 : \text{Integerfmt} \quad \kappa \vdash x_1 : \text{Integerfmt}}{\kappa \vdash x_0 + x_1 : \text{Integerfmt}}$	$\frac{\kappa \vdash x_0 : \text{Real} \quad \kappa \vdash x_1 : \text{Real}}{\kappa \vdash x_0 + x_1 : \text{Real}}$
<i>analog für <math>-</math>, <math>*</math> und <math>/</math></i>	
$\frac{\kappa \vdash x_0 : \text{Integerfmt} \quad \kappa \vdash x_1 : \text{Integerfmt}}{\kappa \vdash x_0 \% x_1 : \text{Integerfmt}}$	$\frac{\kappa \vdash x : \text{Integer } c \text{ U}}{\kappa \vdash \sim x : \text{Integer } c \text{ U}}$
$\frac{\kappa \vdash x_0 : \text{Integer } c \text{ U} \quad \kappa \vdash x_1 : \text{Integer } c \text{ U}}{\kappa \vdash x_0 \& x_1 : \text{Integer } c \text{ U}}$	<i>analog für <math> </math> und <math>\wedge</math></i>
$\frac{\kappa \vdash x_0 : \text{Integer } c_0 \text{ U} \quad \kappa \vdash x_1 : \text{Integer } c_1 \text{ U}}{\kappa \vdash x_0 << x_1 : \text{Integer } c_0 \text{ U}}$	<i>analog für <math>&gt;&gt;</math></i>
$\frac{\kappa \vdash x_0 : \text{Integerfmt} \quad \kappa \vdash x_1 : \text{Integerfmt}}{\kappa \vdash x_0 < x_1 : \text{Boolean}}$	$\frac{\kappa \vdash x_0 : \text{Real} \quad \kappa \vdash x_1 : \text{Real}}{\kappa \vdash x_0 < x_1 : \text{Boolean}}$

*analog für  $\leq$ ,  $\geq$ ,  $>$ ,  $=$  und  $!$*

□

Die Bedeutung der Integer-Datentypen ist wie folgt definiert:

**Typsemantik B.2.4.2 (Integer-Datentypen)**

$$\begin{aligned} \mathcal{D}(\text{Integer } c \text{ U}) &= 0 \dots 2^{\llbracket c \rrbracket_C} - 1 \\ \mathcal{D}(\text{Integer } c \text{ S}) &= -2^{\llbracket c \rrbracket_C - 1} \dots 2^{\llbracket c \rrbracket_C - 1} - 1 \quad \text{falls } \llbracket c \rrbracket_C \geq 1 \end{aligned}$$

□

Die Ganzzahl-Arithmetik erfolgt modulo  $2^{\text{Bitbreite}}$ , wobei vorzeichenbehaftete Zahlen die Hälfte der Repräsentanten für die Äquivalenzklassen in den negativen Bereich legen. Dazu wird die folgende Hilfsfunktion definiert:

**Definition B.1 (Modulo-Rechnung für Ganzzahlen)**

$$\begin{aligned}
\text{int} & \in \mathbb{Z} \rightarrow \langle \text{IntegerFormat} \rangle \rightarrow \mathbb{Z} \\
\text{int } z \text{ (c U)} & \stackrel{\text{def}}{=} z \bmod 2^{\llbracket c \rrbracket c} \\
\text{int } z \text{ (c S)} & \stackrel{\text{def}}{=} \begin{cases} z \bmod 2^{\llbracket c \rrbracket c} & \text{falls } z \bmod 2^{\llbracket c \rrbracket c} < 2^{\llbracket c \rrbracket c - 1} \\ z \bmod 2^{\llbracket c \rrbracket c} - 2^{\llbracket c \rrbracket c} & \text{sonst} \end{cases}
\end{aligned}$$

□

Für die Bit-Operationen werden weitere Hilfsoperationen auf ganzen Zahlen definiert:

**Definition B.2 (Bit-Operationen)** Sei  $n \in \mathbb{N}$ . Für jedes  $x \in 0..2^{n-1}$  sei für alle  $i \in 0..n-1$

$$x_i \stackrel{\text{def}}{=} (x \text{ div } 2^i) \bmod 2.$$

$x_i$  stellt das  $i$ -te Bit der Zahl  $x$  in Binärdarstellung dar; es gilt:

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i.$$

Damit kann für alle  $x, x' \in 0..2^{n-1}$  definiert werden:

$$\begin{aligned}
\neg_n x & \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} (1 - x_i) \cdot 2^i \\
x \wedge_n x' & \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} (x_i \cdot x'_i) \cdot 2^i \\
x \vee_n x' & \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} (\min\{1, x_i + x'_i\}) \cdot 2^i \\
x \oplus_n x' & \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} ((x_i + x'_i) \bmod 2) \cdot 2^i.
\end{aligned}$$

□

Für die Semantik von Real und der Gleitkomma-Operationen wird die Existenz einer Menge Double und darauf definierter arithmetischer Standardoperationen angenommen, für deren Spezifikation auf den

IEEE-Standard für Gleitkomma-Arithmetik [IEE85] verwiesen wird. Ersatzweise wird hier die mathematische Notation für  $\mathbb{R}$  verwendet, wobei allerdings die Zahlenwerte in Punkt- statt Kommaschreibweise (wie sonst im deutschen Sprachraum üblich) notiert werden, um Double syntaktisch von Real zu unterscheiden. Es folgen die Semantik-Gleichungen:

#### Semantik B.2.4.2 (Arithmetische Datentypen)

$$\llbracket \kappa \vdash c \text{ as Integerfmt : Integerfmt} \rrbracket \eta = \text{int } \llbracket c \rrbracket_c \text{fmt}$$

$$\llbracket \kappa \vdash -x : \text{Integer } c \mathbf{S} \rrbracket \eta = \text{int } (-\llbracket \kappa \vdash x : \text{Integer } c \mathbf{S} \rrbracket \eta) (c \mathbf{S})$$

$$\llbracket \kappa \vdash x_0 + x_1 : \text{Integerfmt} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integerfmt} \rrbracket \eta) + (\llbracket \kappa \vdash x_1 : \text{Integerfmt} \rrbracket \eta)) \text{fmt}$$

$$\llbracket \kappa \vdash x_0 - x_1 : \text{Integerfmt} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integerfmt} \rrbracket \eta) - (\llbracket \kappa \vdash x_1 : \text{Integerfmt} \rrbracket \eta)) \text{fmt}$$

$$\llbracket \kappa \vdash x_0 * x_1 : \text{Integerfmt} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integerfmt} \rrbracket \eta) \cdot (\llbracket \kappa \vdash x_1 : \text{Integerfmt} \rrbracket \eta)) \text{fmt}$$

$$\llbracket \kappa \vdash x_0 / x_1 : \text{Integerfmt} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integerfmt} \rrbracket \eta) \text{div} (\llbracket \kappa \vdash x_1 : \text{Integerfmt} \rrbracket \eta)) \text{fmt}$$

$$\llbracket \kappa \vdash x_0 \% x_1 : \text{Integerfmt} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integerfmt} \rrbracket \eta) \text{mod} (\llbracket \kappa \vdash x_1 : \text{Integerfmt} \rrbracket \eta)) \text{fmt}$$

$$\llbracket \kappa \vdash \sim x : \text{Integer } c \mathbf{U} \rrbracket \eta = \neg_{\llbracket c \rrbracket_c} (\llbracket \kappa \vdash x : \text{Integer } c \mathbf{U} \rrbracket \eta)$$

$$\llbracket \kappa \vdash x_0 \& x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta = (\llbracket \kappa \vdash x_0 : \text{Integer } c \mathbf{U} \rrbracket \eta) \wedge_{\llbracket c \rrbracket_c} (\llbracket \kappa \vdash x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta)$$

$$\llbracket \kappa \vdash x_0 | x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta = (\llbracket \kappa \vdash x_0 : \text{Integer } c \mathbf{U} \rrbracket \eta) \vee_{\llbracket c \rrbracket_c} (\llbracket \kappa \vdash x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta)$$

$$\llbracket \kappa \vdash x_0 \wedge x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta = (\llbracket \kappa \vdash x_0 : \text{Integer } c \mathbf{U} \rrbracket \eta) \oplus_{\llbracket c \rrbracket_c} (\llbracket \kappa \vdash x_1 : \text{Integer } c \mathbf{U} \rrbracket \eta)$$

$$\llbracket \kappa \vdash x_0 << x_1 : \text{Integer } c_0 \mathbf{U} \rrbracket \eta = \text{int } ((\llbracket \kappa \vdash x_0 : \text{Integer } c_0 \mathbf{U} \rrbracket \eta) \cdot 2^{\llbracket \kappa \vdash x_1 : \text{Integer } c_1 \mathbf{U} \rrbracket \eta}) (c_0 \mathbf{U})$$

$$\llbracket \kappa \vdash x_0 >> x_1 : \text{Integer } c_0 \mathbf{U} \rrbracket \eta = (\llbracket \kappa \vdash x : \text{Integer } c_0 \mathbf{U} \rrbracket \eta) \text{div } 2^{\llbracket \kappa \vdash x_1 : \text{Integer } c_1 \mathbf{U} \rrbracket \eta}$$

$$\begin{aligned} \llbracket \kappa \vdash x_0 < x_1 : \mathbf{Boolean} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Integerfmt} \rrbracket \eta) < \\ &\quad (\llbracket \kappa \vdash x_1 : \mathbf{Integerfmt} \rrbracket \eta) \end{aligned}$$

*analog für  $<=$ ,  $>=$ ,  $>$ ,  $=$  und  $!=$*

$$\begin{aligned} \llbracket \kappa \vdash 0.0 : \mathbf{Real} \rrbracket \eta &= 0.0 \\ &\vdots \\ \llbracket \kappa \vdash -x : \mathbf{Real} \rrbracket \eta &= -(\llbracket \kappa \vdash x : \mathbf{Real} \rrbracket \eta) \\ \llbracket \kappa \vdash x_0 + x_1 : \mathbf{Real} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Real} \rrbracket \eta) + (\llbracket \kappa \vdash x_1 : \mathbf{Real} \rrbracket \eta) \\ \llbracket \kappa \vdash x_0 - x_1 : \mathbf{Real} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Real} \rrbracket \eta) - (\llbracket \kappa \vdash x_1 : \mathbf{Real} \rrbracket \eta) \\ \llbracket \kappa \vdash x_0 * x_1 : \mathbf{Real} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Real} \rrbracket \eta) \cdot (\llbracket \kappa \vdash x_1 : \mathbf{Real} \rrbracket \eta) \\ \llbracket \kappa \vdash x_0 / x_1 : \mathbf{Real} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Real} \rrbracket \eta) / (\llbracket \kappa \vdash x_1 : \mathbf{Real} \rrbracket \eta) \\ \llbracket \kappa \vdash x_0 < x_1 : \mathbf{Boolean} \rrbracket \eta &= (\llbracket \kappa \vdash x_0 : \mathbf{Real} \rrbracket \eta) < (\llbracket \kappa \vdash x_1 : \mathbf{Real} \rrbracket \eta) \end{aligned}$$

*analog für  $<=$ ,  $>=$ ,  $>$ ,  $=$  und  $!=$*  □

Dabei ist mit dem Operator  $\text{div}$  die Ganzzahldivision gemeint, für die gilt (für  $x, y \in \mathbb{Z}$ ):

$$x = (x \text{ div } y) \cdot y + (x \text{ mod } y).$$

Man beachte ferner, daß implizit das Ergebnis einer Division durch 0 bei Ganzzahlen durch die Semantik-Funktion auf einen undefinierten Wert abgebildet wird, da die Division durch 0 in  $\mathbb{Z}$  undefiniert ist. Für Double gilt die Semantik des IEEE-Standards, der eine Repräsentation für illegale Zahlen (NaN, not a number) definiert.

Zur typsicheren Umwandlung von Zahlen zwischen den verschiedenen arithmetischen Datentypen wird ein Typumwandlungskonstrukt definiert:

### Syntax B.2.4.3 (Typumwandlung)

#### Operatoren

$$\langle \text{Expression} \rangle \text{ as } \langle \text{Type} \rangle \rightarrow \langle \text{Expression} \rangle$$

□

Der Operator ist für ausgewählte Typkombinationen überladen (weitere folgen in späteren Abschnitten), wozu trivialerweise auch die Nicht-Umwandlung bei identischem Typ gehört:

**Typisierung B.2.4.3 (Typumwandlung)**

$$\begin{array}{c}
\frac{\kappa \vdash x : T}{\kappa \vdash x \text{ as } T : T} \\
\\
\frac{\kappa \vdash x : \text{Integerfmt}_0}{\kappa \vdash x \text{ as Integerfmt}_1 : \text{Integerfmt}_1} \\
\\
\frac{\kappa \vdash x : \text{Integerfmt}}{\kappa \vdash x \text{ as Real : Real}}
\end{array}$$

□

**Semantik B.2.4.3 (Typumwandlung)**

$$\begin{aligned}
\llbracket \kappa \vdash x \text{ as } T : T \rrbracket \eta &= \llbracket \kappa \vdash x : T \rrbracket \eta \\
\llbracket \kappa \vdash x \text{ as Integerfmt}_1 : \text{Integerfmt}_1 \rrbracket \eta &= \text{int} (\llbracket \kappa \vdash x : \text{Integerfmt}_0 \rrbracket \eta) \text{fmt}_1 \\
\llbracket \kappa \vdash x \text{ as Real : Real} \rrbracket \eta &= (\llbracket \kappa \vdash x : \text{Integerfmt} \rrbracket \eta) \cdot 1.0
\end{aligned}$$

□

Auch für eingebettete Systeme ist die Verarbeitung von Schriftzeichen und Zeichenketten relevant, da sie mitunter Benutzerschnittstellen ansteuern, die Text anzeigen. Zeichen bilden daher einen weiteren primitiven Datentyp.

**Syntax B.2.4.4 (Zeichen)****Operatoren**

Character  $\rightarrow$   $\langle \text{Type} \rangle$

'a'  $\rightarrow$   $\langle \text{Expression} \rangle$

⋮

□

Das Zeichen „a“ steht hier stellvertretend für einen ganzen Zeichensatz, der implementierungsabhängig im Detail zu spezifizieren ist. Die Zeichen innerhalb eines Zeichensatzes sind linear angeordnet, weshalb die üblichen Vergleichsoperationen auch für Zeichen definiert werden können.

**Typisierung B.2.4.4 (Zeichen)**

$$\begin{array}{c}
\frac{}{\kappa \vdash 'a' : \text{Character}} \quad \dots \\
\\
\frac{\kappa \vdash x_0 : \text{Character} \quad \kappa \vdash x_1 : \text{Character}}{\kappa \vdash x_0 < x_1 : \text{Boolean}} \\
\\
\text{analog für } <=, >=, >, == \text{ und } !=
\end{array}$$

□

Die Bedeutung der Zeichen wird beispielhaft für den Zeichensatz ISO-Latin-1 mit 8 Bit angegeben, der den ASCII-Zeichensatz als Teilmenge enthält:

**Typsemantik B.2.4.3 (Zeichen)**

$$\mathcal{D}(\text{Character}) = 0..255$$

□

Obwohl formal Zahlen als Bedeutung der Zeichen angegeben werden (Zeichencode), liegt ihre eigentliche Bedeutung darin, Buchstaben eines Alphabets, Zahlzeichen sowie weitere Symbole oder auch Steuerbefehle (wie etwas Zeilenumbruch) zu symbolisieren. Der Zeichencode bildet die Grundlage für die Anordnung der Zeichen innerhalb des Zeichensatzes und somit für ihren Vergleich.

**Semantik B.2.4.4 (Zeichen)**

$$\begin{array}{c}
\llbracket 'a' \rrbracket \eta = 97 \\
\vdots \\
\\
\llbracket \kappa \vdash x_0 < x_1 : \text{Boolean} \rrbracket \eta = (\llbracket \kappa \vdash x_0 : \text{Character} \rrbracket \eta) < \\
\quad (\llbracket \kappa \vdash x_1 : \text{Character} \rrbracket \eta) \\
\\
\text{analog für } <=, >=, >, == \text{ und } !=
\end{array}$$

□

**B.2.5 Deklarationen**

Die Vereinbarung eines Bezeichners unter einem bestimmten Typ (siehe Kapitel 7.4.3) stellt sich syntaktisch wie folgt dar:

### Syntax B.2.5.1 (Deklarationen)

#### Sorten

$\langle \text{Declaration} \rangle$

#### Operatoren

$\text{value } \langle \text{Id} \rangle : \langle \text{Type} \rangle \rightarrow \langle \text{Declaration} \rangle$

□

Zusätzlich sind die Syntax-Varianten nach Tabelle 7.10 zulässig; zur Auswertung erfolgt jeweils eine Reduktion auf die Original-Syntax. Für Deklarationen wird die Metavariablen *decl*, auch in indizierter oder gestrichener Form vereinbart.

Die Bedeutung einer Deklaration liegt darin, daß einem bestehenden Kontext eine Bindung eines Bezeichners an einen Typ hinzugefügt wird. Falls der Bezeichner identisch bereits in dem Kontext auftritt, wird er durch die neue Deklaration verdeckt. Hierzu wird eine Semantik-Funktion  $\llbracket \cdot \rrbracket_{\text{Decl}}$  definiert, die einen Kontext auf einen modifizierten Kontext abbildet:

### Semantik B.2.5.1 (Deklarationen)

$$\llbracket \text{value } id : T \rrbracket_{\text{Decl}} \kappa = \kappa[id := T]$$

□

Die rechte Seite der Gleichung in Semantik B.2.5.1 verwendet die in Anhang A.1.4 definierte Notation zur Funktionsvariation.

## B.2.6 Funktionen

*Abstraktion  
und  
Applikation*

Die in Anhang A.1.4 definierte getypte  $\lambda$ -Abstraktion wird direkt in die funktionale Basissprache übernommen (siehe Kapitel 7.4.4).

### Syntax B.2.6.1 (Funktionen)

#### Operatoren

$\langle \langle \text{Declaration} \rangle \rangle \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle \quad // \text{Abstraktion}$

$\langle \text{Expression} \rangle \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle \quad // \text{Applikation}$

$\langle \text{Type} \rangle \rightarrow \langle \text{Type} \rangle \rightarrow \langle \text{Type} \rangle$

□



Der Definitionsbereich einer Funktion ergibt sich aus der Deklaration der Abstraktionsvariablen, der Wertebereich aus dem Typ des Funktionsrumpfes im durch die Abstraktionsvariable modifizierten Kontext. Der Typ eines Arguments muß mit dem Definitionsbereichstyp der angewendeten Funktion übereinstimmen:

#### Typisierung B.2.6.1 (Funktionen)

$$\frac{\kappa[id := T_0] \vdash x : T_1}{\kappa \vdash (\text{value } id : T_0) x : T_0 \multimap T_1}$$

$$\frac{\kappa \vdash f : T_0 \multimap T_1 \quad \kappa \vdash x : T_0}{\kappa \vdash f x : T_1}$$

□

Deklarationen bilden syntaktisch eine eigene Sorte, für die Hilfsfunktionen zur syntaktischen Zerlegung in Bezeichner und Typ definiert werden könnten. Der direkte Durchgriff auf die Syntax der Deklarationen (d. h. die Verwendung von `value id : T0` anstelle von `decl`) erhöht hier aber die Lesbarkeit der Typisierungsregel.

Funktionstypen entsprechen Funktionsmengen und sind äquivalent, wenn die Typen für Definitions- und Wertebereich äquivalent sind:

#### Typsemantik B.2.6.1 (Funktionstypen)

$$\mathcal{D}(T_0 \multimap T_1) = \mathcal{D}(T_0) \rightarrow \mathcal{D}(T_1)$$

□

#### Typäquivalenz B.2.6.1 (Funktionstypen)

$$\frac{T_0 \equiv T'_0 \quad T_1 \equiv T'_1}{T_0 \multimap T_1 \equiv T'_0 \multimap T'_1}$$

□

Die Semantik von  $\lambda$ -Abstraktion und Funktionsanwendung wird auf die direkt entsprechenden mathematischen Konzepte abgestützt. Wie der Kontext durch die Abstraktionsvariable für den Funktionsrumpf modifiziert wird, so wird es auch die entsprechende Umgebung: Die Abstraktionsvariable wird innerhalb des Funktionsrumpfes an den Wert gebunden, zu dem das Argument ausgewertet wird:

### Semantik B.2.6.1 (Funktionen)

$$\begin{aligned}
\llbracket \kappa \vdash (\text{value } id : T_0) x : T_0 \multimap T_1 \rrbracket \eta &= \\
\lambda y \in \mathcal{D}(T_0) . \llbracket \kappa[id := T_0] \vdash x : T_1 \rrbracket (\eta[id := y]) \\
\llbracket \kappa \vdash f x : T_1 \rrbracket \eta &= \\
(\llbracket \kappa \vdash f : T_0 \multimap T_1 \rrbracket \eta) (\llbracket \kappa \vdash x : T_0 \rrbracket \eta)
\end{aligned}$$

□

## B.2.7 Definitionen

Ein zweites Sprachkonstrukt nach der  $\lambda$ -Abstraktion, das Bezeichner in Kontexte und Umgebungen einführt, sind lokale Definitionen, die innerhalb eines Ausdrucks gültig sind (siehe Kapitel 7.4.3). Das `let`-Konstrukt kann zunächst als syntaktischer Zucker verstanden werden mit:

$$\text{let value } id : T = x \text{ in } x' \quad \equiv \quad ((\text{value } id : T) x') x.$$

Es wird hier im weiteren aber eigenständig und in einer verallgemeinerten Form eingeführt; sie erlaubt mehrere gleichzeitige Definitionen, wahlweise parallel oder wechselseitig rekursiv. Auf eine mehr der konzeptionellen Ästhetik als der Einfachheit der Sprachdefinition dienende Rückführung auf  $\lambda$ -Ausdrücke und, im Fall von Rekursion, auf einen Fixpunkt-Operator (vgl. die Erläuterungen zur Semantik unten) wird verzichtet.

### Syntax B.2.7.1 (Lokale Definitionen)

#### Sorten

$\langle \text{Definition} \rangle$

#### Operatoren

$\langle \text{Declaration} \rangle = \langle \text{Expression} \rangle \quad \rightarrow \quad \langle \text{Definition} \rangle$

`let`     $\langle \text{Definition} \rangle^+ \text{ in } \langle \text{Expression} \rangle \quad \rightarrow \quad \langle \text{Expression} \rangle$

`letrec`  $\langle \text{Definition} \rangle^+ \text{ in } \langle \text{Expression} \rangle \quad \rightarrow \quad \langle \text{Expression} \rangle$

□

$\langle \text{Definition} \rangle_+$  steht wie in SDF (siehe [BK02], vgl. Anhang B.4) für eine mindestens einmalige Wiederholung von  $\langle \text{Definition} \rangle$ .

Die beiden Varianten unterscheiden sich voneinander in dem Gültigkeitsbereich der in den lokalen Definitionen eingeführten Bezeichner. Bei `let` beschränkt er sich auf den Rumpf des Ausdrucks (den Teil, der in folgt), während er sich bei `letrec` auch auf die in den Definitionen auftretenden Ausdrücke erstreckt. So kann der einen Bezeichner definierende Ausdruck jenen selbst oder andere Bezeichner, die durch den `letrec`-Ausdruck eingeführt werden, als Variablen verwenden.

### Typisierung B.2.7.1 (Lokale Definitionen)

$$\frac{\kappa \vdash x_0 : T_0 \quad \dots \quad \kappa \vdash x_n : T_n \quad \kappa' \vdash x : T}{\kappa \vdash \text{let value } id_0 : T_0 = x_0 \dots \text{value } id_n : T_n = x_n \text{ in } x : T}$$

*falls die  $id_0, \dots, id_n$  paarweise verschieden sind*

$$\frac{\kappa' \vdash x_0 : T_0 \quad \dots \quad \kappa' \vdash x_n : T_n \quad \kappa' \vdash x : T}{\kappa \vdash \text{letrec value } id_0 : T_0 = x_0 \dots \text{value } id_n : T_n = x_n \text{ in } x : T}$$

*falls die  $id_0, \dots, id_n$  paarweise verschieden sind*

wobei

$$\kappa' \stackrel{\text{def}}{=} \kappa[id_0 := T_0] \dots [id_n := T_n]$$

□

Entsprechend verhält es sich bei der Semantik. `let` wertet die definierenden Ausdrücke für die eingeführten Bezeichner in der Umgebung des `let`-Ausdrucks aus; `letrec` wertet sie in der Umgebung aus, die bereits die definierten Werte enthält:

### Semantik B.2.7.1 (Lokale Definitionen)

$$\begin{aligned} \llbracket \kappa \vdash \text{let } id_0 : T_0 = x_0 \dots id_n : T_n = x_n \text{ in } x : T \rrbracket \eta &= \llbracket \kappa' \vdash x : T \rrbracket \eta' \\ \llbracket \kappa \vdash \text{letrec } id_0 : T_0 = x_0 \dots id_n : T_n = x_n \text{ in } x : T \rrbracket \eta &= \llbracket \kappa' \vdash x : T \rrbracket \eta'' \end{aligned}$$

wobei

$$\begin{aligned} \kappa' &\stackrel{\text{def}}{=} \kappa[id_0 := T_0] \dots [id_n := T_n] \\ \eta' &\stackrel{\text{def}}{=} \eta[id_0 := \llbracket \kappa \vdash x_0 : T_0 \rrbracket \eta] \dots [id_n := \llbracket \kappa \vdash x_n : T_n \rrbracket \eta] \\ \eta'' &= \eta[id_0 := \llbracket \kappa' \vdash x_0 : T_0 \rrbracket \eta''] \dots [id_n := \llbracket \kappa' \vdash x_n : T_n \rrbracket \eta''] \end{aligned}$$

□

Man beachte, daß die Umgebung  $\eta''$  in Semantik B.2.7.1 durch eine Fixpunktgleichung spezifiziert ist. Die Definition eines „kleinsten“ Fixpunktes, der eindeutig ist, falls eine Lösung der Gleichung existiert, kann auf Basis der Domänen-Theorie [GS90] formal dargestellt werden. Da sie nicht wesentlich zum Verständnis des Sprachentwurfs beiträgt und andererseits einen umfangreichen theoretischen Exkurs erforderlich macht, wird im Rahmen dieser Arbeit darauf verzichtet.

### Rekursion

Operational entspricht die Auswertung rekursiver Definitionen einer bedarfsgesteuerten Rückverfolgung von Abhängigkeiten („lazy evaluation“), wobei eine zyklische Abhängigkeit von Funktionen, nicht aber von einzelnen Werten, zulässig ist. Ein Spezialfall davon sind Rückkopplungen bei Signalflüssen, wobei eine sogenannte *algebraische Schleife* die zyklische Abhängigkeit einzelner Signalwerte beinhaltet (vgl. Abschnitt B.3.2). Undefiniertheiten entstehen außerdem bei nichtterminierenden Ketten von Abhängigkeiten, also *Endlosrekursion*.

Undefiniertheiten pflanzen sich innerhalb einer Auswertung fort. Die Auswertungsreihenfolge ist dabei bedeutsam, denn ein Funktionswert, zu dessen Berechnung das Argument der Funktion nicht benötigt wird, kann als definiert gelten, auch wenn das Argument selbst nicht definiert ist. Das Prinzip der „lazy evaluation“ beinhaltet eine Auswertung in *Normalordnung*, bei der zunächst die Funktion und erst dann (im Bedarfsfall) das Argument ausgewertet wird. Wird stets zuerst das Argument ausgewertet („eager evaluation“, wie bei Funktionsaufrufen in der Programmiersprache C) werden in solchen Fällen existierende Lösungen der semantischen Gleichungen nicht gefunden. Zu Normalordnung, „lazy“ und „eager evaluation“ siehe z. B. [Rey98].

Die auftretenden Undefiniertheiten der Semantik (wie schon bei der Division durch 0) sind als Ausnahmen von der Regel zu sehen, daß jedem typkorrekten Ausdruck eine Bedeutung zugeordnet wird. Sie sind stets als Programmierfehler zu werten. Endlosrekursion, also Nichtterminierung in einer Programmdefinition ist zu unterscheiden von Nichtterminierung in der Programmausführung. Endlosschleifen in der Ausführung mit definiertem Zeitfortschritt sind für eingebettete Systeme genau das typische Verhalten und dem gewählten Prozeßmodell entsprechend (siehe Abschnitt B.3.1).

Da Definitionen zur Metasprache gehören, bedeutet das dem Grundsatz nach, daß fehlerhafte rekursive Definitionen sich zur Kompilierzeit auswirken. Jedoch werden für Berechnungen auch Funktionen

erst zur Laufzeit ausgewertet (siehe dazu z. B. das `apply`-Konstrukt in Abschnitt 6.4.3). Im Rahmen der Programmiermethodik ist es empfehlenswert und für Steuerungen im allgemeinen ausreichend, sich bei Berechnungen zur Laufzeit auf primitiv rekursive Funktionen zu beschränken und diese Beschränkungen durch statisch durchführbare Programmprüfungen durchzusetzen. Dafür spricht einerseits die Absicherung gegen Programmierfehler. Andererseits sind laufzeitbeschränkte Berechnungen nötig, um maximale Ausführungszeiten (worst-case execution times, siehe Kapitel 2.3.1.1) abschätzen zu können.

Nichtterminierende Definitionen können wegen der Unberechenbarkeit des Halteproblems [Sch01] im allgemeinen nicht aus dem Programmtext allein erkannt werden. Andererseits würde der Ausschluß von Rekursion insgesamt die Echtzeitbeschreibungsmittel im wesentlichen unbrauchbar machen, da sie dazu dienen, Zyklen in Signalflußgraphen oder Phasenübergangssystemen zu beschreiben.

## B.2.8 Strukturierte Typen

Dem in Kapitel 7.4.5.2 beschriebenen Entwurf entsprechend, sind strukturierte Typen wie folgt definiert:

### Syntax B.2.8.1 (Strukturierte Typen)

#### Operatoren

*//Varianten-Typ*

$$\underbrace{\langle \text{Id} \rangle : \langle \text{Type} \rangle \mid \dots \mid \langle \text{Id} \rangle : \langle \text{Type} \rangle}_{\geq 1} \rightarrow \langle \text{Type} \rangle$$

*//Record-Typ*

$$\{ \underbrace{\langle \text{Id} \rangle : \langle \text{Type} \rangle, \dots, \langle \text{Id} \rangle : \langle \text{Type} \rangle}_{\geq 0} \} \rightarrow \langle \text{Type} \rangle$$

*//Tupel-Typ*

$$(\underbrace{\langle \text{Type} \rangle, \dots, \langle \text{Type} \rangle}_{0, \geq 2}) \rightarrow \langle \text{Type} \rangle$$

*//Array-Typ*

$$\langle \text{Type} \rangle [ \langle \text{Cardinal} \rangle ] \rightarrow \langle \text{Type} \rangle$$

*// Variante*

$\langle \text{Id} \rangle : \langle \text{Expression} \rangle \text{ as } \langle \text{Type} \rangle \rightarrow \langle \text{Expression} \rangle$

*// Record*

$\{ \underbrace{\langle \text{Id} \rangle : \langle \text{Expression} \rangle, \dots, \langle \text{Id} \rangle : \langle \text{Expression} \rangle}_{\geq 0} \} \rightarrow \langle \text{Expression} \rangle$

*// Tupel*

$( \underbrace{\langle \text{Expression} \rangle, \dots, \langle \text{Expression} \rangle}_{0, \geq 2} ) \rightarrow \langle \text{Expression} \rangle$

*// Array*

$[ \underbrace{\langle \text{Expression} \rangle, \dots, \langle \text{Expression} \rangle}_{\geq 1} ] \rightarrow \langle \text{Expression} \rangle$

*// Varianten-Selektoren*

case  $\langle \text{Expression} \rangle$  of

$\left. \begin{array}{c} \langle \text{Id} \rangle : \langle \text{Expression} \rangle \\ \vdots \\ \langle \text{Id} \rangle : \langle \text{Expression} \rangle \end{array} \right\} \geq 1 \rightarrow \langle \text{Expression} \rangle$

case  $\langle \text{Expression} \rangle$  of

$\left. \begin{array}{c} \langle \text{Id} \rangle : \langle \text{Expression} \rangle \\ \vdots \\ \langle \text{Id} \rangle : \langle \text{Expression} \rangle \end{array} \right\} \geq 1$

otherwise  $\langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

*// Record-Selektor*

$\langle \text{Expression} \rangle . \langle \text{Id} \rangle \rightarrow \langle \text{Expression} \rangle$

*// Tupel-Selektor und statischer Array-Selektor*

$\langle \text{Expression} \rangle . \langle \text{Cardinal} \rangle \rightarrow \langle \text{Expression} \rangle$

*// dynamischer Array-Selektor*

$\langle \text{Expression} \rangle \text{ at } \langle \text{Expression} \rangle \text{ default } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle \quad \square$

Man beachte, daß die Syntax der Typkonstruktoren für die Produkttypen an die der Wertkonstruktoren angeglichen wurde. Im Vergleich mit

der mathematischen Notation entfällt das vorangestellte Produktsymbol. Im Summenkonstruktor folgt das ODER-Symbol als Trennzeichen der Notation in Haskell und Standard ML. Der Konstruktor für Varianten verlangt die Angabe eines Typs (nämlich eines Summentyps) und ähnelt dabei syntaktisch dem Operator für Typumwandlungen.

Die Beschriftungen in den Summen- und Produkttypen bzw. Varianten und Records sind Bezeichner, die jedoch nicht direkt Objekte bezeichnen, sondern Indizes in Aufzählungen darstellen. Zur Unterscheidung dieser Rolle dient eine zusätzliche Metavariablen  $l$  („label“), auch in indizierter oder gestrichener Form.

### Typisierung B.2.8.1 (Strukturierte Typen)

$$\begin{array}{c}
\frac{\kappa \vdash x_i : T_i}{\kappa \vdash l_i : x_i \text{ as } \{ l_0 : T_0 \mid \dots \mid l_n : T_n \} : \{ l_0 : T_0 \mid \dots \mid l_n : T_n \}} \\
\text{für } i \in 0..n \\
\text{falls die } l_0, \dots, l_n \text{ paarweise verschieden sind} \\
\\
\frac{\kappa \vdash x_0 : T_0 \quad \dots \quad \kappa \vdash x_{n-1} : T_{n-1}}{\kappa \vdash \{ l_0 : x_0, \dots, l_{n-1} : x_{n-1} \} : \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}} \\
\text{falls die } l_0, \dots, l_{n-1} \text{ paarweise verschieden sind} \\
\\
\frac{\kappa \vdash x_0 : T_0 \quad \dots \quad \kappa \vdash x_{n-1} : T_{n-1}}{\kappa \vdash (x_0, \dots, x_{n-1}) : (T_0, \dots, T_{n-1})} \quad \text{für } n \in \mathbb{N} \setminus \{1\} \\
\\
\frac{\kappa \vdash x_0 : T \quad \dots \quad \kappa \vdash x_n : T}{\kappa \vdash [x_0, \dots, x_n] : T[c]} \quad \text{falls } \llbracket c \rrbracket_{\mathbb{C}} = n + 1 \\
\\
\frac{\kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_n : T_n \} \quad \kappa \vdash f_0 : T_0 \rightarrow T \quad \dots \quad \kappa \vdash f_n : T_n \rightarrow T}{\kappa \vdash \text{case } x \text{ of } l_0 : f_0 \quad \dots \quad l_n : f_n : T} \\
\\
\frac{\kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_{n+m} : T_{n+m} \} \quad \kappa \vdash f_0 : T_0 \rightarrow T \quad \dots \quad \kappa \vdash f_n : T_n \rightarrow T \quad \kappa \vdash x' : T}{\kappa \vdash \text{case } x \text{ of } l_0 : f_0 \quad \dots \quad l_n : f_n \text{ otherwise } x' : T} \quad \text{für } m \geq 0
\end{array}$$

$$\begin{array}{c}
\frac{\kappa \vdash x : \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}}{\kappa \vdash x . l_i : T_i} \quad \text{für } i \in 0..n-1 \\
\\
\frac{\kappa \vdash x : (T_0, \dots, T_{n-1})}{\kappa \vdash x . c : T_i} \quad \text{falls } \llbracket c \rrbracket_{\mathcal{C}} = i, \text{ für } i \in 0..n-1 \\
\\
\frac{\kappa \vdash x : T[c]}{\kappa \vdash x . c_0 : T} \quad \text{falls } \llbracket c_0 \rrbracket_{\mathcal{C}} \in 0.. \llbracket c \rrbracket_{\mathcal{C}} - 1 \\
\\
\frac{\kappa \vdash x : T[c] \quad \kappa \vdash y : \mathbf{Integer} \, c' \, \mathbf{U} \quad \kappa \vdash x' : T}{\kappa \vdash x \, \mathbf{at} \, y \, \mathbf{default} \, x' : T} \\
\\
\frac{\kappa \vdash x : T[c]}{\kappa \vdash x \, \mathbf{as} \, ( \underbrace{T, \dots, T}_n ) : ( \underbrace{T, \dots, T}_n )} \quad \text{falls } \llbracket c \rrbracket_{\mathcal{C}} = n, n \neq 1 \\
\\
\frac{\kappa \vdash x : ( \overbrace{T, \dots, T}^n )}{\kappa \vdash x \, \mathbf{as} \, T[c] : T[c]} \quad \text{falls } \llbracket c \rrbracket_{\mathcal{C}} = n, n \geq 2
\end{array}$$

□

### Typsemantik B.2.8.1 (Strukturierte Typen)

$$\begin{aligned}
\mathcal{D}(\{ l_0 : T_0 \mid \dots \mid l_n : T_n \}) &= \sum \{ l_0 : \mathcal{D}(T_0), \dots, l_n : \mathcal{D}(T_n) \} \\
\mathcal{D}(\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}) &= \prod \{ l_0 : \mathcal{D}(T_0), \dots, l_{n-1} : \mathcal{D}(T_{n-1}) \} \\
\mathcal{D}((T_0, \dots, T_{n-1})) &= \prod [\mathcal{D}(T_0), \dots, \mathcal{D}(T_{n-1})] \quad \text{für } n \in \mathbb{N} \setminus \{1\} \\
\mathcal{D}(T[c]) &= \mathcal{D}(T)^{\llbracket c \rrbracket_{\mathcal{C}}}
\end{aligned}$$

□

Typäquivalenzen pflanzen sich in strukturierten Typen fort. Man beachte auch, daß Record- bzw. Variantentypen, die sich nur in der Reihenfolge der Aufzählung unterscheiden, äquivalent sind:



**Typäquivalenz B.2.8.1 (Strukturierte Typen)**

$$\begin{array}{c}
\frac{T_0 \equiv T'_0 \quad \dots \quad T_n \equiv T'_n}{\{ l_0 : T_0 \mid \dots \mid l_n : T_n \} \equiv \{ l_0 : T'_0 \mid \dots \mid l_n : T'_n \}} \\
\\
\frac{T_0 \equiv T'_0 \quad \dots \quad T_{n-1} \equiv T'_{n-1}}{\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \equiv \{ l_0 : T'_0, \dots, l_{n-1} : T'_{n-1} \}} \\
\\
\frac{T_0 \equiv T'_0 \quad \dots \quad T_{n-1} \equiv T'_{n-1}}{(T_0, \dots, T_{n-1}) \equiv (T'_0, \dots, T'_{n-1})} \\
\\
\frac{\{ l_0 : T_0 \mid \dots \mid l_n : T_n \} \equiv \{ l_{i_0} : T_{i_0} \mid \dots \mid l_{i_n} : T_{i_n} \}}{\text{falls } \{i_0, \dots, i_n\} = 0..n} \\
\\
\frac{\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \equiv \{ l_{i_0} : T_{i_0}, \dots, l_{i_{n-1}} : T_{i_{n-1}} \}}{\text{falls } \{i_0, \dots, i_{n-1}\} = 0..n-1}
\end{array}$$

□

Die Semantik der Wertkonstruktoren- und Selektoren folgt entsprechend der in Kapitel 7.4.5.2 dargestellten Modellierung:

**Semantik B.2.8.1 (Strukturierte Typen)**

$$\begin{aligned}
& \llbracket \kappa \vdash l_i : x_i \text{ as } \{ l_0 : T_0 \mid \dots \mid l_{n-1} : T_n \} : \{ l_0 : T_0 \mid \dots \mid l_{n-1} : T_n \} \rrbracket \eta = \\
& \quad [l_i, \llbracket \kappa \vdash x_i : T_i \rrbracket \eta] \quad \text{für } i \in 0..n \\
& \llbracket \kappa \vdash \{ l_0 : x_0, \dots, l_{n-1} : x_{n-1} \} : \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta = \\
& \quad \{ l_0 : (\llbracket \kappa \vdash x_0 : T_0 \rrbracket \eta), \dots, l_{n-1} : (\llbracket \kappa \vdash x_{n-1} : T_{n-1} \rrbracket \eta) \} \\
& \llbracket \kappa \vdash (x_0, \dots, x_{n-1}) : (T_0, \dots, T_{n-1}) \rrbracket \eta = \\
& \quad [\llbracket \kappa \vdash x_0 : T_0 \rrbracket \eta, \dots, \llbracket \kappa \vdash x_{n-1} : T_{n-1} \rrbracket \eta] \\
& \llbracket \kappa \vdash [x_0, \dots, x_n] : T[c] \rrbracket \eta = \\
& \quad [\llbracket \kappa \vdash x_0 : T \rrbracket \eta, \dots, \llbracket \kappa \vdash x_n : T \rrbracket \eta] \quad \text{wobei } \llbracket c \rrbracket_{\mathbb{C}} = n + 1
\end{aligned}$$

$$\begin{aligned}
\llbracket \kappa \vdash \mathbf{case} \ x \ \mathbf{of} \ l_0 : f_0 \ \dots \ l_n : f_n : T \rrbracket \eta &= \\
\begin{cases} (\llbracket \kappa \vdash f_0 : T_0 \rightarrow T \rrbracket \eta) \ y & \mathbf{falls} \ \llbracket \kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_n : T_n \} \rrbracket \eta = [l_0, y] \\ \vdots \\ (\llbracket \kappa \vdash f_n : T_n \rightarrow T \rrbracket \eta) \ y & \mathbf{falls} \ \llbracket \kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_n : T_n \} \rrbracket \eta = [l_n, y] \end{cases} \\
\llbracket \kappa \vdash \mathbf{case} \ x \ \mathbf{of} \ l_0 : f_0 \ \dots \ l_n : f_n \ \mathbf{otherwise} \ x' : T \rrbracket \eta &= \\
\begin{cases} (\llbracket \kappa \vdash f_0 : T_0 \rightarrow T \rrbracket \eta) \ y & \mathbf{falls} \ \llbracket \kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_{n+m} : T_{n+m} \} \rrbracket \eta = [l_0, y] \\ \vdots \\ (\llbracket \kappa \vdash f_n : T_n \rightarrow T \rrbracket \eta) \ y & \mathbf{falls} \ \llbracket \kappa \vdash x : \{ l_0 : T_0 \mid \dots \mid l_{n+m} : T_{n+m} \} \rrbracket \eta = [l_n, y] \\ \llbracket \kappa \vdash x' : T \rrbracket \eta & \mathbf{sonst} \end{cases} \\
\llbracket \kappa \vdash x . l_i : T_i \rrbracket \eta &= \\
(\llbracket \kappa \vdash x : \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta) \ l_i & \quad \mathbf{für} \ i \in 0..n-1 \\
\llbracket \kappa \vdash x . c : T_i \rrbracket \eta &= \\
(\llbracket \kappa \vdash x : (T_0, \dots, T_{n-1}) \rrbracket \eta) \llbracket c \rrbracket_{\mathbf{C}} & \quad \mathbf{falls} \ \llbracket c \rrbracket_{\mathbf{C}} = i, \mathbf{für} \ i \in 0..n-1 \\
\llbracket \kappa \vdash x . c_0 : T \rrbracket \eta &= \\
(\llbracket \kappa \vdash x : T[c] \rrbracket \eta) \llbracket c_0 \rrbracket_{\mathbf{C}} & \quad \mathbf{falls} \ \llbracket c_0 \rrbracket_{\mathbf{C}} \in 0..\llbracket c \rrbracket_{\mathbf{C}} - 1 \\
\llbracket \kappa \vdash x \mathbf{at} \ y \ \mathbf{default} \ x' : T \rrbracket \eta &= \\
\begin{cases} (\llbracket \kappa \vdash x : T[c] \rrbracket \eta) (\llbracket \kappa \vdash y : \mathbf{Integer} \ c' \ \mathbf{U} \rrbracket \eta) \\ \quad \mathbf{falls} \ \llbracket \kappa \vdash y : \mathbf{Integer} \ c' \ \mathbf{U} \rrbracket \eta \in 0..\llbracket c \rrbracket_{\mathbf{C}} - 1 \\ \llbracket \kappa \vdash x' : T \rrbracket \eta \\ \quad \mathbf{sonst} \end{cases} \\
\llbracket \kappa \vdash x \mathbf{as} \ (T, \dots, T) : (T, \dots, T) \rrbracket \eta &= \\
\llbracket \kappa \vdash x : T[c] \rrbracket \eta & \\
\llbracket \kappa \vdash x \mathbf{as} \ T[c] : T[c] \rrbracket \eta &= \\
\llbracket \kappa \vdash x : (T, \dots, T) \rrbracket \eta &
\end{aligned}$$

□

Für die Verkettung von Arrays gleichen Basistyps wird wie in Java der Additionsoperator überladen:

*Verkettung von Arrays*

### Typisierung B.2.8.2 (Verkettung von Arrays)

$$\frac{\kappa \vdash x_0 : T[c_0] \quad \kappa \vdash x_1 : T[c_1]}{\kappa \vdash x_0 + x_1 : T[c]} \quad \text{falls } \llbracket c_0 \rrbracket_C + \llbracket c_1 \rrbracket_C = \llbracket c \rrbracket_C$$

□

Die Semantikdefinition verwendet den Verkettungsoperator aus Anhang A.1.10:

### Semantik B.2.8.2 (Verkettung von Arrays)

$$\llbracket \kappa \vdash x_0 + x_1 : T[c] \rrbracket \eta = (\llbracket \kappa \vdash x_0 : T[c_0] \rrbracket \eta) \frown (\llbracket \kappa \vdash x_1 : T[c_1] \rrbracket \eta)$$

□

Zur Vereinfachung der Definition von Funktionen mit Tupel-Argumenten (mehrstellige Funktionen) läßt sich die Notation aus (A.79) in Anhang A.1.9 leicht auf die Programmiersprache übertragen. Als „syntaktischer Zucker“ ergibt sich die Notation

*syntaktischer Zucker*

$$(\text{value } id_0 : T_0, \dots, \text{value } id_n : T_n) x, \quad (\text{B.9})$$

die sich zu

$$\begin{aligned} &(\text{value } id : (T_0, \dots, T_n)) \\ &(((\text{value } id_0 : T_0) \dots (\text{value } id_n : T_n) x) (id.0) \dots (id.n)) \end{aligned} \quad (\text{B.10})$$

reduziert, wobei **n** für das  $\langle \text{Cardinal} \rangle$ -Literal steht, für das gilt:

$$\llbracket \mathbf{n} \rrbracket_C = n.$$

Die Notationen für Definition und Deklaration aus Tabelle 7.14 werden entsprechend um Mehrstelligkeit erweitert. Die allgemeinen Formen sind in der Spezifikation der konkreten Syntax in Anhang B.4 angegeben.

Alternativ zur Reduktion nach (B.9) und (B.10) mit Generierung von  $\langle \text{Cardinal} \rangle$ -Literalen können Typisierung und Semantik (unter Verwendung von (A.79)) auch direkt angegeben werden:

**Typisierung B.2.8.3 (Mehrstellige Funktionen)**

$$\frac{\kappa[id_0 := T_0, \dots, id_n := T_n] \vdash x : T}{\kappa \vdash (\text{value } id_0 : T_0, \dots, \text{value } id_n : T_n) x : (T_0, \dots, T_n) \rightarrow T}$$

□

**Semantik B.2.8.3 (Mehrstellige Funktionen)**

$$\begin{aligned} \llbracket \kappa \vdash (\text{value } id_0 : T_0, \dots, \text{value } id_n : T_n) x : (T_0, \dots, T_n) \rightarrow T \rrbracket \eta = \\ \lambda[y_0 \in \mathcal{D}(T_0), \dots, y_n \in \mathcal{D}(T_n)]. \\ \llbracket \kappa[id_0 := T_0, \dots, id_n := T_n] \vdash x : T \rrbracket (\eta[id_0 := y_0, \dots, id_n := y_n]) \end{aligned}$$

□

**B.2.9 Typdefinitionen und -funktionen***Typbezeichner*

Typdefinitionen und -funktionen (siehe Kapitel 7.4.5.3) vereinbaren Namen für Typausdrücke. Hierzu wird zunächst eine eigene Sorte  $\langle \text{Typeld} \rangle$  für Typbezeichner eingeführt; Typbezeichner sind als Typvariablen selbst wieder Typausdrücke:

**Syntax B.2.9.1 (Typbezeichner)****Sorten** $\langle \text{Typeld} \rangle$ **Operatoren** $\langle \text{Typeld} \rangle \rightarrow \langle \text{Type} \rangle$ 

□

In der konkreten Syntax unterscheiden sich Typbezeichner von Bezeichnern darin, daß sie mit *großen* Anfangsbuchstaben beginnen. Als Metavariablen für Typbezeichner wird *tid*, auch in indizierter oder gestrichelter Form, verwendet.

Für Typvariablen bleibt die Typsemantik undefiniert, d.h. für einen Typausdruck *tid* ist  $\mathcal{D}(\text{tid})$  nicht definiert. Dies verträgt sich mit dem weiteren Vorgehen, daß in den Typisierungsregeln und Semantikdefinitionen dort, wo Typbezeichner gebunden werden, mit Termersetzungen gearbeitet wird, so daß in einem typkorrekten Ausdruck beim

rekursiven Absteigen in den Term keine Typvariable bis zur Auswertung unersetzt bleibt. Alternativ könnten – mit einem über die gesamte Sprachdefinition hinweg komplizierteren Definitionsformalismus – *Typumgebungen* eingeführt werden.

Sei dazu  $E$  ein Term der Sorte  $\langle \text{Expression} \rangle$  oder  $\langle \text{Type} \rangle$ . Mit

$$E[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] \quad (\text{B.11})$$

werde der Term bezeichnet, der aus  $E$  entsteht, wenn man für alle  $i \in 0..n-1$  alle freie Vorkommen der Typvariable  $tid_i$  in  $E$  durch den Typausdruck  $T_i$  ersetzt. *Frei* sind alle Vorkommen einer Typvariablen, die nicht *bindend* oder *gebunden* sind. Bindungen von Typvariablen werden nur durch bestimmte, im weiteren noch eingeführte Sprachkonstrukte vorgenommen; auf die Bindungseigenschaften wird dann jeweils hingewiesen.

Typdefinitionen nun haben wie die Wertdefinitionen nach Abschnitt B.2.7 einen abgegrenzten Gültigkeitsbereich. Analog zu den *let*- und *letrec*-Konstrukten wird ein *lettype*-Konstrukt eingeführt, mit dem lokal für einen Typ- oder Wertausdruck ein oder mehrere Typvariablen (nichtrekursiv) definiert werden:

*Typdefinitionen*

### Syntax B.2.9.2 (Typdefinitionen)

#### Sorten

$\langle \text{TypeDefinition} \rangle$

#### Operatoren

$\text{type } \langle \text{Type} \rangle = \langle \text{Type} \rangle \rightarrow \langle \text{TypeDefinition} \rangle$

$\text{lettype } \langle \text{TypeDefinition} \rangle + \text{ in } \langle \text{Type} \rangle \rightarrow \langle \text{Type} \rangle$

$\text{lettype } \langle \text{TypeDefinition} \rangle + \text{ in } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle \quad \square$

Bei lokalen Typdefinitionen für Wertausdrücke ist die Definition eines Typbezeichners zur Sicherung der Konsistenz nur erlaubt, wenn er nicht bereits Bestandteil des Kontexts ist:

### Typisierung B.2.9.1 (Typdefinitionen)

$$\frac{\kappa \vdash x[T_0/tid_0, \dots, T_n/tid_n] : T}{\kappa \vdash \text{lettype type } tid_0 = T_0 \dots \text{type } tid_n = T_n \text{ in } x : T}$$

*falls die  $tid_0, \dots, tid_n$  paarweise verschieden sind  
und nicht frei in einem Typausdruck in  $\kappa$  vorkommen*

□

### Semantik B.2.9.1 (Typdefinitionen)

$$\llbracket \kappa \vdash \text{lettype type } tid_0 = T_0 \dots \text{type } tid_n = T_n \text{ in } x : T \rrbracket \eta = \\ \llbracket \kappa \vdash x[T_0/tid_0, \dots, T_n/tid_n] : T \rrbracket \eta$$

□

### Typäquivalenz B.2.9.1 (Typdefinitionen)

---


$$\text{lettype type } tid_0 = T_0 \dots \text{type } tid_n = T_n \text{ in } T \equiv T[T_0/tid_0, \dots, T_n/tid_n]$$

□

Die Typbezeichner  $tid_0, \dots, tid_n$  im lettype-Ausdruck sind innerhalb des Ausdrucks  $x$  bzw.  $T$  (gebundene Vorkommen) durch die Typdefinitionen (bindende Vorkommen) gebunden; sie sind also innerhalb des gesamten lettype-Ausdrucks nur dann frei, wenn sie in mindestens einem der Typausdrücke  $T_0, \dots, T_n$  frei vorkommen.

#### Typfunktionen

Analog zu Definitionen für normale Ausdrücke können auch für Typdefinitionen *Parameter* zugelassen werden; in diesem Fall sind es Typen, die als Parameter zu Typen auftreten. Im Fall normaler Ausdrücke werden Parameter auf (getypte)  $\lambda$ -Ausdrücke zurückgeführt. Die  $\lambda$ -Abstraktion ist auch hier anwendbar, jedoch in ungetypter Form (auf Metatypen wird verzichtet); dadurch entstehen *Typfunktionen*. Nachfolgend werden die  $\lambda$ -Abstraktion und die Applikation für Typen eingeführt.

### Syntax B.2.9.3 (Typfunktionen)

#### Sorten

$\langle \text{TypeDeclaration} \rangle$

**Operatoren**

$$\begin{aligned}
\text{type } \langle \text{Typeld} \rangle & \rightarrow \langle \text{TypeDeclaration} \rangle \\
( \langle \text{TypeDeclaration} \rangle ) \langle \text{Type} \rangle & \rightarrow \langle \text{Type} \rangle \\
\langle \text{Type} \rangle \langle \text{Type} \rangle & \rightarrow \langle \text{Type} \rangle
\end{aligned}
\quad \square$$

Die Syntax orientiert sich am Vorbild der Wertdeklaration und der Abstraktion und Applikation bei Wertfunktionen. Für Terme der Sorte  $\langle \text{TypeDeclaration} \rangle$  gilt im weiteren die Metavariablen *tdecl* (auch indiziert oder gestrichen).

Für die Typisierung charakteristisch ist, daß kein Wertkonstruktor ein Objekt erzeugt, das eine Typfunktion als Typ hat. Typfunktionen in Typausdrücken treten nur in angewendeter Form zur Typangabe für Werte auf. Sofern es sich nicht um vordefinierte handelt (solche enthält FSPL innerhalb der Echtzeitbeschreibungsmittel, siehe Abschnitt B.3), sind Typfunktionen nur innerhalb der Teilsprache der Typausdrücke bedeutsam und müssen bei der Typprüfung stets ausgewertet werden:

**Typäquivalenz B.2.9.2 (Typfunktionen)**

$$\frac{}{((\text{type } tid) T) T' \equiv T[T'/tid]}$$

□

Die Typisierungsregeln lassen keinen Typausdruck  $T T'$  zu, bei dem  $T$  nicht äquivalent zu einer (vordefinierten oder benutzerdefinierten) Typfunktion ist. Bei einer Typfunktion

$$(\text{type } tid) T$$

ist die Typvariable *tid* innerhalb des Typausdrucks  $T$  durch die Typdeklaration  $\text{type } tid$  gebunden, also innerhalb des gesamten Typausdrucks nicht frei.

**B.2.10 Generische Programmierung**

Die Sprachelemente für die generische Programmierung (siehe Kapitel 7.4.6) werden nachfolgend definiert:

**Syntax B.2.10.1 (Polymorphie)****Operatoren**

$$\begin{aligned}
\langle \text{TypeDeclaration} \rangle &\rightarrow \langle \text{Type} \rangle && // \forall \\
\langle \text{TypeDeclaration} \rangle &\langle \text{Expression} \rangle && // \Lambda \\
\langle \text{Expression} \rangle &\langle \text{Type} \rangle && // \text{Instantiierung}
\end{aligned}$$

□

Die Typdeklaration `type tid` in einem Typausdruck

$$\langle \text{type } tid \rangle \rightarrow T$$

bindet die Typvariable *tid* innerhalb von *T*; in einem Ausdruck

$$\langle \text{type } tid \rangle x$$

bindet sie die Typvariable innerhalb von *x*. Sie ist deshalb in den gesamten Ausdrücken nicht frei.

**Typisierung B.2.10.1 (Polymorphie)**

$$\frac{\kappa \vdash x : T}{\kappa \vdash \langle \text{type } tid \rangle x : \langle \text{type } tid \rangle \rightarrow T}$$

*falls *tid* nicht frei in einem Typausdruck in  $\kappa$  vorkommt*

$$\frac{\kappa \vdash x : \langle \text{type } tid \rangle \rightarrow T}{\kappa \vdash x \langle T' \rangle : T[T'/tid]}$$

□

Parametrische Polymorphie ist im wesentlichen eine Sache des Typsystems und ermöglicht die Typsichere Verwendung generischer Funktionen. Da die Semantikfunktion aber induktiv über Typurteile definiert ist, übertragen sich die Typparameter unmittelbar in die Semantik generischer Objekte, wobei aus der  $\Lambda$ -Abstraktion eine gewöhnliche  $\lambda$ -Abstraktion wird, indem die Typausdrücke zu gewöhnlichen Objekten werden:



**Semantik B.2.10.1 (Polymorphie)**

$$\begin{aligned}
& \llbracket \kappa \vdash < \mathbf{type} \, tid > x : < \mathbf{type} \, tid > -> T \rrbracket \eta = \\
& \quad \lambda \tau \in \mathbf{Type} . (\llbracket \kappa \vdash x[\tau/tid] : T[\tau/tid] \rrbracket \eta) \\
& \llbracket \kappa \vdash x < T' > : T[T'/tid] \rrbracket \eta = \\
& \quad \llbracket \kappa \vdash x : < \mathbf{type} \, tid > -> T \rrbracket \eta \, T'
\end{aligned}$$

□

Als Typsemantik für All-Quantifizierung kann dazu angegeben werden:

**Typsemantik B.2.10.1 (Polymorphie)**

$$\mathcal{D}(< \mathbf{type} \, tid > -> T') = \mathbf{Type} \rightarrow \bigcup_{T \in \mathbf{Type}} \mathcal{D}(T'[T/tid]) \quad \square$$

Der Name des Typparameters ist unerheblich, solange er nicht mit anderen Typvariablen kollidiert:

**Typäquivalenz B.2.10.1 (Polymorphie)**

$$\frac{T_0 \equiv T_1[tid_0/tid_1]}{< \mathbf{type} \, tid_0 > -> T_0 \equiv < \mathbf{type} \, tid_1 > -> T_1}$$

*falls  $tid_0$  nicht frei in  $T_1$*

□

**B.2.11 Module**

Die Sprachelemente für Module (siehe Kapitel 7.4.7) werden nachfolgend definiert:

**Syntax B.2.11.1 (Module)****Operatoren**

$$\begin{aligned}
& \mathbf{interface} \{ \langle \mathbf{TypeDeclaration} \rangle^* \langle \mathbf{Declaration} \rangle^* \} \rightarrow \langle \mathbf{Type} \rangle \\
& \mathbf{module} \{ \langle \mathbf{TypeDefinition} \rangle^* \langle \mathbf{Definition} \rangle^* \} \rightarrow \langle \mathbf{Expression} \rangle \\
& \mathbf{import} \langle \mathbf{Expression} \rangle \mathbf{in} \langle \mathbf{Expression} \rangle \rightarrow \langle \mathbf{Expression} \rangle \quad \square
\end{aligned}$$

Analog zu dem in früheren Syntaxdefinitionen verwendeten Zeichen  $+$  für mindestens einmalige Wiederholung stellt das  $*$  hinter einem Sortennamen eine null- oder mehrmalige Wiederholung dar. Ein Modul kann beliebig viele Typen mit beliebig vielen Operationen zusammenfassen. Falls die Typen entfallen, dient ein Modul lediglich der Wiederverwendung von Definitionen und ist äquivalent zu einem Record. Entfallen die Operationen, ist ein Modul äquivalent zum Einheitswert  $\text{unit}$  und praktisch bedeutungslos. Die Typdefinitionen sind nichtrekursiv (wie bei  $\text{lettype}$ ) die Wertdefinitionen sind rekursiv (wie bei  $\text{letrec}$ ).

### Typisierung B.2.11.1 (Module)

$$\begin{array}{c}
\kappa' \vdash x_0[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] : T'_0[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] \quad \dots \\
\kappa' \vdash x_{m-1}[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] : T'_{m-1}[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}]
\end{array}$$


---


$$\begin{array}{cc}
\text{module } \{ & \text{interface } \{ \\
\quad \text{type } tid_0 & = T_0 \quad \text{type } tid_0 \\
\quad \dots & \dots \\
\quad \text{type } tid_{n-1} & = T_{n-1} \quad \text{type } tid_{n-1} \\
\quad \text{value } id_0 & : T'_0 = x_0 \quad \text{value } id_0 : T'_0 \\
\quad \dots & \dots \\
\quad \text{value } id_{m-1} & : T'_{m-1} = x_{m-1} \quad \text{value } id_{m-1} : T'_{m-1} \\
\} & \}
\end{array}$$

*falls die  $tid_0, \dots, tid_{n-1}$  und die  $id_0, \dots, id_{n-1}$  jeweils paarweise verschieden sind und nicht frei in einem Typausdruck in  $\kappa$  vorkommen, wobei*

$$\begin{aligned}
\kappa' = \kappa[ & id_0 := T'_0[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] \dots \\
& id_{m-1} := T'_{m-1}[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] ]
\end{aligned}$$

$$\begin{array}{c}
\text{interface } \{ \\
\quad \text{type } tid_0 \\
\quad \dots \\
\quad \text{type } tid_{n-1} \\
\quad \text{value } id_0 \quad : \quad T'_0 \\
\quad \dots \\
\quad \text{value } id_{m-1} \quad : \quad T'_{m-1} \\
\} \\
\hline
\kappa[id_0 := T'_0] \dots [id_{m-1} := T'_{m-1}] \vdash x_1 : T'' \\
\kappa \vdash \text{import } x_0 \text{ in } x_1 : T''
\end{array}$$

*falls  $tid_0, \dots, tid_{n-1}$  nicht frei in  $T''$   
oder in einem Typausdruck in  $\kappa$  vorkommen*

□

Innerhalb eines Typausdrucks

$$\begin{array}{c}
\text{interface } \{ \\
\quad \text{type } tid_0 \\
\quad \dots \\
\quad \text{type } tid_{n-1} \\
\quad \text{value } id_0 \quad : \quad T'_0 \\
\quad \dots \\
\quad \text{value } id_{m-1} \quad : \quad T'_{m-1} \\
\}
\end{array}$$

sind die Typbezeichner  $tid_0, \dots, tid_{n-1}$  nicht frei. In einem Ausdruck

$$\begin{array}{c}
\text{module } \{ \\
\quad \text{type } tid_0 \quad = \quad T_0 \\
\quad \dots \\
\quad \text{type } tid_{n-1} \quad = \quad T_{n-1} \\
\quad \text{value } id_0 \quad : \quad T'_0 \quad = \quad x_0 \\
\quad \dots \\
\quad \text{value } id_{m-1} \quad : \quad T'_{m-1} \quad = \quad x_{m-1} \\
\}
\end{array}$$

kommen die Typbezeichner  $tid_0, \dots, tid_{n-1}$  nur dann frei vor, wenn sie in  $T_0, \dots, T_{n-1}$  frei vorkommen. Wenn

$$\kappa \vdash x_0 : \begin{array}{l} \text{interface } \{ \\ \quad \text{type } tid_0 \\ \quad \dots \\ \quad \text{type } tid_{n-1} \\ \quad \text{value } id_0 \quad : \quad T'_0 \\ \quad \dots \\ \quad \text{value } id_{m-1} \quad : \quad T'_{m-1} \\ \} \end{array},$$

so kommen die Typbezeichner  $tid_0, \dots, tid_{n-1}$  in einem Ausdruck

$$\text{import } x_0 \text{ in } x_1$$

im Kontext  $\kappa$  nur dann frei vor, wenn sie in  $x_0$  im Kontext  $\kappa$  frei sind. An dieser Stelle wird ersichtlich, daß die Freiheit von Typvariablen in Wertausdrücken im allgemeinen kontextabhängig ist<sup>3</sup>.

Die Typsemantik für Schnittstellentypen führt Module auf Records zurück:

#### Typsemantik B.2.11.1 (Module)

$$\mathcal{D} \left( \begin{array}{l} \text{interface } \{ \\ \quad \text{type } tid_0 \\ \quad \dots \\ \quad \text{type } tid_{n-1} \\ \quad \text{value } id_0 \quad : \quad T'_0 \\ \quad \dots \\ \quad \text{value } id_{m-1} \quad : \quad T'_{m-1} \\ \} \end{array} \right) = \bigcup_{T_0, \dots, T_{n-1} \in \text{Type}} \prod \{ (id_i : \mathcal{D}(T'_i[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}])) \mid i \in 0..m-1 \}$$

□

Entsprechend werden Module zu Records ausgewertet; sie enthalten dann keine Typinformation zu den abstrakten Typen mehr.

<sup>3</sup>In den zuvor erklärten Fällen, daß ein Typbezeichner nicht frei ist, ist dies so zu verstehen, daß er dort jeweils für alle Kontexte nicht frei ist.

**Semantik B.2.11.1 (Module)**

$$\begin{array}{c}
\left[ \left[ \begin{array}{l} \text{module } \{ \\ \quad \text{type } tid_0 = T_0 \\ \quad \dots \\ \quad \text{type } tid_{n-1} = T_{n-1} \\ \quad \text{value } id_0 : T'_0 = x_0 \\ \quad \dots \\ \quad \text{value } id_{m-1} : T'_{m-1} = x_{m-1} \\ \} \end{array} \quad : \quad \begin{array}{l} \text{interface } \{ \\ \quad \text{type } tid_0 \\ \quad \dots \\ \quad \text{type } tid_{n-1} \\ \quad \text{value } id_0 : T'_0 \\ \quad \dots \\ \quad \text{value } id_{m-1} : T'_{m-1} \\ \} \end{array} \right] \right] \eta = \\
\{ id_0 : \xi_0, \dots, id_{m-1} : \xi_{m-1} \}
\end{array}$$

wobei

$$\begin{aligned}
\xi_i &= \llbracket \kappa \vdash x_i[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] : T'_i[T_0/tid_0, \dots, T_{n-1}/tid_{n-1}] \rrbracket \eta' \\
&\text{für } i \in 0..m-1 \\
\eta' &= \eta[id_0 := \xi_0] \dots [id_{m-1} := \xi_{m-1}]
\end{aligned}$$

$$\llbracket \kappa \vdash \text{import } x_0 \text{ in } x_1 : T'' \rrbracket \eta =$$

$$\begin{aligned}
&\llbracket \kappa[id_0 := T'_0] \dots [id_{m-1} := T'_{m-1}] \vdash x_1 : T'' \rrbracket \\
&(\eta[id_0 := \mu id_0] \dots [id_{m-1} := \mu id_{m-1}])
\end{aligned}$$

$$\text{wobei } \mu = \left[ \left[ \begin{array}{l} \text{interface } \{ \\ \quad \text{type } tid_0 \\ \quad \dots \\ \quad \text{type } tid_{n-1} \\ \quad \text{value } id_0 : T'_0 \\ \quad \dots \\ \quad \text{value } id_{m-1} : T'_{m-1} \\ \} \end{array} \right] \right] \kappa \vdash x_0 : \eta$$

□

Schnittstellen, die sich nur in der Reihenfolge ihrer Deklarationen unterscheiden, sind als Typen äquivalent; desweiteren pflanzen sich Äquivalenzen der in den Deklarationen enthaltenen Typen fort:

**Typäquivalenz B.2.11.1 (Module)**

$$\begin{array}{c}
\begin{array}{c} \text{interface } \{ \\ \text{type } tid_0 \\ \dots \\ \text{type } tid_{i_{n-1}} \\ \text{value } id_0 : T_0 \\ \dots \\ \text{value } id_{m-1} : T_{m-1} \\ \} \end{array} \\
\end{array}
\equiv
\begin{array}{c}
\begin{array}{c} \text{interface } \{ \\ \text{type } tid_{i_0} \\ \dots \\ \text{type } tid_{i_{n-1}} \\ \text{value } id_{j_0} : T_{j_0} \\ \dots \\ \text{value } id_{j_{m-1}} : T_{j_{m-1}} \\ \} \end{array} \\
\end{array}$$

*falls*  $\{i_0, \dots, i_{n-1}\} = 0..n-1$  *und*  $\{j_0, \dots, j_{m-1}\} = 0..m-1$

$$\begin{array}{c}
T_0 \equiv T'_0 \quad \dots \quad T_{m-1} \equiv T'_{m-1} \\
\begin{array}{c} \text{interface } \{ \\ \text{type } tid_0 \\ \dots \\ \text{type } tid_{i_{n-1}} \\ \text{value } id_0 : T_0 \\ \dots \\ \text{value } id_{m-1} : T_{m-1} \\ \} \end{array} \\
\end{array}
\equiv
\begin{array}{c}
\begin{array}{c} \text{interface } \{ \\ \text{type } tid_0 \\ \dots \\ \text{type } tid_{i_{n-1}} \\ \text{value } id_0 : T'_0 \\ \dots \\ \text{value } id_{m-1} : T'_{m-1} \\ \} \end{array} \\
\end{array}$$

□

**B.3 Echtzeit-Beschreibungsmittel**

Um die funktionale Basissprache zu einer Programmiersprache für die Software eingebetteter Echtzeitsysteme zu vervollständigen, werden in diesem Abschnitt Beschreibungsmittel für zeitbezogenes Verhalten eingeführt. Sie übersetzen die semantischen Modelle aus Kapitel 6 in Sprachkonstrukte (siehe auch Kapitel 7.3.2 bis 7.3.5). Sie erweitern die Sprache um zusätzliche Typen und Operationen, die auf den Datenbeschreibungsmitteln aufsetzen und eine zeitliche Dimension hinzufügen. Die Mechanismen der Metasprache verwenden sie zur Systembildung.

Der Struktur der Modellbildung entsprechend, gliedern sich die Echtzeit-Beschreibungsmittel in drei Hauptteile für quasi-kontinuierliches (Abschnitt B.3.2), ereignisgesteuertes (Abschnitt B.3.3) und sequentielles Verhalten (Abschnitt B.3.4). Die gemeinsame Basis dafür, die Umsetzung des Zeit-, Prozeß- und Ereigniskonzepts in die Sprache, wird zuvor in Abschnitt B.3.1 gelegt.

## B.3.1 Zeit und Verhalten

### B.3.1.1 Zeit

Nach der in Kapitel B.3.1.1 beschriebenen Umsetzung des Zeitmodells aus Kapitel 6.3.1 erweisen sich bei der Sprachdefinition `time` und `tick` als weitere Fälle von Typumwandlungen, für die sinnrichtig der bereits existierende Operator `as` überladen werden kann. Gleiches gilt für die Rechen- und Vergleichsoperatoren. Es verbleibt die Einführung einer neuen Sorte  $\langle \text{TimeUnit} \rangle$  (mit der Metavariablen  $tu$  für Terme dieser Sorte) und der zugehörigen Konstruktoren, eines neuen primitiven Datentyps `Time` sowie des Konstruktors für Zeitkonstanten.

#### Syntax B.3.1.1 (Zeit)

##### Sorten

$\langle \text{TimeUnit} \rangle$

##### Operatoren

<code>Time</code>	$\rightarrow$	$\langle \text{Type} \rangle$
<code>ms</code>	$\rightarrow$	$\langle \text{TimeUnit} \rangle$
<code>s</code>	$\rightarrow$	$\langle \text{TimeUnit} \rangle$
<code>min</code>	$\rightarrow$	$\langle \text{TimeUnit} \rangle$
<code>h</code>	$\rightarrow$	$\langle \text{TimeUnit} \rangle$
$\langle \text{Cardinal} \rangle$	$\langle \text{TimeUnit} \rangle$	$\rightarrow$ $\langle \text{Expression} \rangle$

□

Die Typisierung folgt den Vorgaben aus Spezifikation 6.1 und der anschließenden Definitionen (mit der Ausnahme, daß `Time` zu allen Integer-Typen konvertiert werden kann):

**Typisierung B.3.1.1 (Zeit)**

$$\begin{array}{c}
\hline
\kappa \vdash c \text{ tu} : \langle \text{Time} \rangle \\
\hline
\end{array}$$

$ \frac{\kappa \vdash x : \mathbf{Time}}{\kappa \vdash x \text{ as Integer } fmt : \mathbf{Integer } fmt} $	$ \frac{\kappa \vdash x : \mathbf{Time}}{\kappa \vdash x \text{ as Real} : \mathbf{Real}} $
$ \frac{\kappa \vdash x_0 : \mathbf{Time} \quad \kappa \vdash x_1 : \mathbf{Time}}{\kappa \vdash x_0 + x_1 : \mathbf{Time}} $	$ \frac{\kappa \vdash x_0 : \mathbf{Time} \quad \kappa \vdash x_1 : \mathbf{Time}}{\kappa \vdash x_0 - x_1 : \mathbf{Time}} $
$ \frac{\kappa \vdash x_0 : \mathbf{Integer } c \text{ U} \quad \kappa \vdash x_1 : \mathbf{Time}}{\kappa \vdash x_0 * x_1 : \mathbf{Time}} $	$ \frac{\kappa \vdash x_0 : \mathbf{Time} \quad \kappa \vdash x_1 : \mathbf{Integer } c \text{ U}}{\kappa \vdash x_0 / x_1 : \mathbf{Time}} $
$ \frac{\kappa \vdash x_0 : \mathbf{Time} \quad \kappa \vdash x_1 : \mathbf{Time}}{\kappa \vdash x_0 == x_1 : \mathbf{Boolean}} $	

*analog für !=, <, <=, >= und >*

□

Die Bedeutung des Zeitdatentyps ist die abstrakte Zeitmenge aus der Modellbildung:

**Typsemantik B.3.1.1 (Datentyp Time)**

$$\mathcal{D}(\mathbf{Time}) = \text{Time}$$

□

Die Bedeutung der Zeiteinheiten liefert eine eigene Semantik-Funktion:

**Semantik B.3.1.1 (Zeiteinheiten)**

$$\begin{aligned}
\llbracket \text{ms} \rrbracket_{\text{TU}} &= 1 \\
\llbracket \text{s} \rrbracket_{\text{TU}} &= 1000 \cdot \llbracket \text{ms} \rrbracket_{\text{TU}} \\
\llbracket \text{min} \rrbracket_{\text{TU}} &= 60 \cdot \llbracket \text{s} \rrbracket_{\text{TU}} \\
\llbracket \text{h} \rrbracket_{\text{TU}} &= 60 \cdot \llbracket \text{min} \rrbracket_{\text{TU}}
\end{aligned}$$

□



Der abstrakte Datentyp **Time** und die zusätzlich definierten Operationen liefern schließlich die Semantik der Ausdrücke:

### Semantik B.3.1.2 (Zeit)

$$\begin{aligned}
\llbracket \kappa \vdash c \text{ tu} : \mathbf{Time} \rrbracket \eta &= \text{toTime} ((\llbracket c \rrbracket_{\mathbf{C}} \eta) \cdot \llbracket \text{tu} \rrbracket_{\mathbf{TU}}) \\
\llbracket \kappa \vdash x \text{ as Integer fmt} : \mathbf{Integer fmt} \rrbracket \eta &= \text{int} (\text{tick} (\llbracket \kappa \vdash x : \mathbf{Time} \rrbracket \eta)) \text{ fmt} \\
\llbracket \kappa \vdash x \text{ as Real} : \mathbf{Real} \rrbracket \eta &= \text{step}_{\mathbf{Double}} \cdot (\text{tick} (\llbracket \kappa \vdash x : \mathbf{Time} \rrbracket \eta)) \\
\\
\llbracket \kappa \vdash x_0 + x_1 : \mathbf{Time} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{+} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 - x_1 : \mathbf{Time} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{-} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 * x_1 : \mathbf{Time} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Integer} \text{ c U} \rrbracket \eta \underset{\mathbf{Time}}{*} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 / x_1 : \mathbf{Time} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{/} \llbracket \kappa \vdash x_1 : \mathbf{Integer} \text{ c U} \rrbracket \eta \\
\\
\llbracket \kappa \vdash x_0 == x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{=} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 != x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{\neq} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 < x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{<} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 <= x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{\leq} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 >= x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{\geq} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta \\
\llbracket \kappa \vdash x_0 > x_1 : \mathbf{Boolean} \rrbracket \eta &= \llbracket \kappa \vdash x_0 : \mathbf{Time} \rrbracket \eta \underset{\mathbf{Time}}{>} \llbracket \kappa \vdash x_1 : \mathbf{Time} \rrbracket \eta
\end{aligned}$$

□

### B.3.1.2 Prozeß

Für die Übertragung von Spezifikation 6.2 aus Kapitel 6.3.2 (siehe Kapitel 7.3.2.2) ist die Sprachdefinition zunächst nur um einen neuen Typkonstruktor für  $\text{Process}_X$  zu erweitern:

**Syntax B.3.1.2 (Prozesse)****Operatoren**

Process  $\rightarrow \langle \text{Type} \rangle$  □

Process ist eine vordefinierte *Typfunktion* (siehe Abschnitt B.2.9), wird also nur in angewendeter Form zur Typisierung von prozeßwertigen Ausdrücken verwendet. Die Semantik des Typkonstruktors wird durch das Prozeßmodell geliefert:

**Typsemantik B.3.1.2 (Prozeßtypen)**

$$\mathcal{D}(\text{Process } T) = \text{Process}_{\mathcal{D}(T)} \quad \square$$

*syntaktischer  
Zucker*

Für Deklarationen und Definitionen mit Prozeßtypen definiert die konkrete Syntax folgende Varianten:

$$\text{process } id : T, \quad (\text{B.12})$$

$$\text{signal } id : T \quad (\text{B.13})$$

und

$$\text{variable } id : T \quad (\text{B.14})$$

werden reduziert zu

$$\text{value } id : \text{Process } (T) \quad (\text{B.15})$$

sowie

$$\text{process } id : T = x, \quad (\text{B.16})$$

$$\text{signal } id : T = x \quad (\text{B.17})$$

und

$$\text{variable } id : T = x \quad (\text{B.18})$$

zu

$$\text{value } id : \text{Process } (T) = x. \quad (\text{B.19})$$

Die alternativen Schlüsselwörter process, signal und variable unterscheiden sich semantisch nicht, geben aber dem Programmierer eine zusätzliche Möglichkeit zur Klassifikation im Sinne der Pragmatik (vgl. Kapitel 6.3.2). Die Wahlmöglichkeit zwischen process, signal und variable dient also zu Dokumentationszwecken.

### B.3.1.3 Ereignis

Ereignisse sind wie Prozesse erstklassige Objekte. Sie bilden einen neuen Typ, für den neue Operatoren eingeführt oder existierende überladen werden. Die Namen der neu eingeführten Operatoren folgen den in der Modellbildung eingeführten Operationen mit Ausnahme von watchdog, die hier durch no within repräsentiert wird, und clock, für die clock um offset ergänzt wird:

#### Syntax B.3.1.3 (Ereignisse)

##### Operatoren

Event	→	⟨Type⟩	
never	→	⟨Expression⟩	
after ⟨Expression⟩	→	⟨Expression⟩	
clock ⟨Expression⟩ offset ⟨Expression⟩	→	⟨Expression⟩	
trigger ⟨Expression⟩	→	⟨Expression⟩	
no ⟨Expression⟩ within ⟨Expression⟩	→	⟨Expression⟩	□

Die Verknüpfungen  $\bigvee_{\text{Event}}$ ,  $\bigwedge_{\text{Event}}$ ,  $+$ ,  $-$  und  $\bigstar_{\text{Event}}$  verwenden die existierenden Operatoren  $\parallel$ ,  $\&\&$ ,  $+$ ,  $-$  bzw.  $\ast$ . guard wird durch den Operator  $\%$  und ifElseEvent wird durch den if then else-Operator repräsentiert.

Als syntaktischen Zucker erlaubt die konkrete Syntax das Weglassen des offset-Teils des clock-Operators: *syntaktischer Zucker*

$$\text{clock } dt \quad (\text{B.20})$$

wird zu

$$\text{clock } dt \text{ offset } 0 \text{ ms.} \quad (\text{B.21})$$

Weiterhin wird eine Variante der Deklarations- und Definitionssyntax eingeführt, die mit den bisherigen Varianten kombiniert werden kann:

$$\text{event } id \quad (\text{B.22})$$

wird zu

$$\text{value } id : \text{Event} \quad (\text{B.23})$$

und

$$\text{event } id = x \quad (\text{B.24})$$

zu

$$\text{value } id : \mathbf{Event} = x. \quad (\text{B.25})$$

Typisierung und Semantik folgen der Modellbildung:

### Typisierung B.3.1.2 (Ereignisse)

$$\begin{array}{c}
\frac{}{\kappa \vdash \mathbf{never} : \mathbf{Event}} \qquad \frac{\kappa \vdash dt : \mathbf{Time}}{\kappa \vdash \mathbf{after } dt : \mathbf{Event}} \\
\\
\frac{\kappa \vdash dt : \mathbf{Time} \quad \kappa \vdash t : \mathbf{Time}}{\kappa \vdash \mathbf{clock } dt \text{ offset } t : \mathbf{Event}} \qquad \frac{\kappa \vdash x : \mathbf{Process Boolean}}{\kappa \vdash \mathbf{trigger } x : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e_0 : \mathbf{Event} \quad \kappa \vdash e_1 : \mathbf{Event}}{\kappa \vdash e_0 \parallel e_1 : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e_0 : \mathbf{Event} \quad \kappa \vdash e_1 : \mathbf{Event}}{\kappa \vdash e_0 \&\& e_1 : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e_0 : \mathbf{Event} \quad \kappa \vdash e_1 : \mathbf{Event}}{\kappa \vdash e_0 + e_1 : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e_0 : \mathbf{Event} \quad \kappa \vdash e_1 : \mathbf{Event}}{\kappa \vdash e_0 - e_1 : \mathbf{Event}} \\
\\
\frac{\kappa \vdash x : \mathbf{Integer } c \mathbf{U} \quad \kappa \vdash e : \mathbf{Event}}{\kappa \vdash x * e : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e : \mathbf{Event} \quad \kappa \vdash x : \mathbf{Process Boolean}}{\kappa \vdash e \% x : \mathbf{Event}} \\
\\
\frac{\kappa \vdash x : \mathbf{Process Boolean} \quad \kappa \vdash e_0 : \mathbf{Event} \quad \kappa \vdash e_1 : \mathbf{Event}}{\kappa \vdash \mathbf{if } x \text{ then } e_0 \text{ else } e_1 : \mathbf{Event}} \\
\\
\frac{\kappa \vdash e : \mathbf{Event} \quad \kappa \vdash dt : \mathbf{Time}}{\kappa \vdash \mathbf{no } e \text{ within } dt : \mathbf{Event}}
\end{array}$$

□

**Typsemantik B.3.1.3 (Ereignistyp)**

$$\mathcal{D}(\mathbf{Event}) = \mathbf{Event}$$

□

**Semantik B.3.1.3 (Ereignisse)**

$$\llbracket \kappa \vdash \mathbf{never} : \mathbf{Event} \rrbracket \eta = \mathbf{never}$$

$$\begin{aligned} \llbracket \kappa \vdash \mathbf{after} \ dt : \mathbf{Event} \rrbracket \eta &= \mathbf{after} (\llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta) \\ &\quad \mathbf{falls} \ \llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta \in \mathbf{Time}_+ \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash \mathbf{clock} \ dt \ \mathbf{offset} \ t : \mathbf{Event} \rrbracket \eta &= \mathbf{clock} (\llbracket \kappa \vdash t : \mathbf{Time} \rrbracket \eta) \\ &\quad (\llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta) \\ &\quad \mathbf{falls} \ \llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta \in \mathbf{Time}_+ \end{aligned}$$

$$\llbracket \kappa \vdash \mathbf{trigger} \ x : \mathbf{Event} \rrbracket \eta = \mathbf{trigger} (\llbracket \kappa \vdash x : \mathbf{Process} \ \mathbf{Boolean} \rrbracket \eta)$$

$$\begin{aligned} \llbracket \kappa \vdash e_0 \ || \ e_1 : \mathbf{Event} \rrbracket \eta &= (\llbracket \kappa \vdash e_0 : \mathbf{Event} \rrbracket \eta) \underset{\mathbf{Event}}{\vee} \\ &\quad (\llbracket \kappa \vdash e_1 : \mathbf{Event} \rrbracket \eta) \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash e_0 \ \&\& \ e_1 : \mathbf{Event} \rrbracket \eta &= (\llbracket \kappa \vdash e_0 : \mathbf{Event} \rrbracket \eta) \underset{\mathbf{Event}}{\wedge} \\ &\quad (\llbracket \kappa \vdash e_1 : \mathbf{Event} \rrbracket \eta) \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash e_0 + e_1 : \mathbf{Event} \rrbracket \eta &= (\llbracket \kappa \vdash e_0 : \mathbf{Event} \rrbracket \eta) \underset{\mathbf{Event}}{+} \\ &\quad (\llbracket \kappa \vdash e_1 : \mathbf{Event} \rrbracket \eta) \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash e_0 - e_1 : \mathbf{Event} \rrbracket \eta &= (\llbracket \kappa \vdash e_0 : \mathbf{Event} \rrbracket \eta) \underset{\mathbf{Event}}{-} \\ &\quad (\llbracket \kappa \vdash e_1 : \mathbf{Event} \rrbracket \eta) \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash x * e : \mathbf{Event} \rrbracket \eta &= (\llbracket \kappa \vdash x : \mathbf{Integer} \ c \ \mathbf{U} \rrbracket \eta) \underset{\mathbf{Event}}{*} \\ &\quad (\llbracket \kappa \vdash e : \mathbf{Event} \rrbracket \eta) \end{aligned}$$

$$\mathbf{falls} \ (\llbracket \kappa \vdash x : \mathbf{Integer} \ c \ \mathbf{U} \rrbracket \eta) > 0$$

$$\llbracket \kappa \vdash e \% x : \mathbf{Event} \rrbracket \eta = \text{guard} (\llbracket \kappa \vdash x : \mathbf{Process Boolean} \rrbracket \eta) \\ (\llbracket \kappa \vdash e : \mathbf{Event} \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{if } x \text{ then } e_0 \text{ else } e_1 : \mathbf{Event} \rrbracket \eta = \text{ifElseEvent} (\llbracket \kappa \vdash x : \mathbf{Process Boolean} \rrbracket \eta) \\ (\llbracket \kappa \vdash e_0 : \mathbf{Event} \rrbracket \eta) \\ (\llbracket \kappa \vdash e_1 : \mathbf{Event} \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{no } e \text{ within } dt : \mathbf{Event} \rrbracket \eta = \text{watchdog} (\llbracket \kappa \vdash e : \mathbf{Event} \rrbracket \eta) \\ (\llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta) \\ \text{falls } (\llbracket \kappa \vdash dt : \mathbf{Time} \rrbracket \eta) \in \text{Time}_+$$

□

### B.3.2 Signalflüsse

Die spezifischen Sprachkonstrukte zur Beschreibung von Signalen und signalverarbeitenden Funktionen (siehe Kapitel 7.3.3) sind wie folgt definiert:

#### Syntax B.3.2.1 (Primitive Signale)

##### Operatoren

$$\text{const } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle \\ \text{time} \rightarrow \langle \text{Expression} \rangle$$

□

#### Typisierung B.3.2.1 (Primitive Signale)

$$\frac{\kappa \vdash x : T}{\kappa \vdash \text{const } x : \mathbf{Process } T} \\ \frac{}{\kappa \vdash \text{time} : \mathbf{Process Time}}$$

□

Die Instantiierung von  $\text{const}_X$  für eine Zustandsmenge  $X$  ergibt sich aus der Bedeutung  $\mathcal{D}(T)$  des Typs  $T$ , den der Parameter  $x$  besitzt:

**Semantik B.3.2.1 (Primitive Signale)**

$$\llbracket \kappa \vdash \text{const } x : \text{Process } T \rrbracket \eta = \text{const}_{\mathcal{D}(T)} (\llbracket \kappa \vdash x : T \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{time} : \text{Process Time} \rrbracket \eta = \text{timeProcess}$$

□

Die Operationen  $\text{apply}_{X \rightarrow Y}$ ,  $\text{zip}_{\prod(\lambda i \in I. X_i)}$  und  $\text{unzip}_{\prod(\lambda i \in I. X_i)}$  werden als überladene Operatoren eingeführt. Darüber hinaus werden alle arithmetisch-logischen Operationen und die Typumwandlung auf Signalebene angehoben:

**Syntax B.3.2.2 (Statische Systeme)****Operatoren***//Lifting*

$\text{apply } \langle \text{Expression} \rangle \text{ to } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\text{zip } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\text{unzip } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

*//Logische Operationen*

$! \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle \&\& \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle || \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\text{if } \langle \text{Expression} \rangle \text{ then } \langle \text{Expression} \rangle \text{ else } \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

*//Rechenoperationen*

$- \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle + \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle - \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle * \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle / \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle \% \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

*//Bit-Operationen*

$\sim \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle \& \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle | \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ^ \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle << \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle >> \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

//Vergleiche

$\langle \text{Expression} \rangle < \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$   
 $\langle \text{Expression} \rangle \leq \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$   
 $\langle \text{Expression} \rangle \geq \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$   
 $\langle \text{Expression} \rangle > \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$   
 $\langle \text{Expression} \rangle == \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$   
 $\langle \text{Expression} \rangle != \langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

//Typumwandlung

$\langle \text{Expression} \rangle \text{ as } \langle \text{Type} \rangle \rightarrow \langle \text{Expression} \rangle$

□

Die binären arithmetisch-logischen Operationen mit Ausnahme der Schiebeoperationen (da hier nur der linke Operand auf Signalebene angehoben werden kann) werden so überladen, daß entweder beide oder nur einer der Operanden Signale sind; im letzteren Fall wird ein Signal mit einer Konstanten verknüpft.

### Typisierung B.3.2.2 (Statische Systeme)

$$\begin{array}{c}
 \frac{\kappa \vdash f : T_0 \rightarrow T_1 \quad \kappa \vdash x : \mathbf{Process} T_0}{\kappa \vdash \text{apply } f \text{ to } x : \mathbf{Process} T_1} \\
 \\
 \frac{\kappa \vdash x : \{ l_0 : \mathbf{Process} T_0, \dots, l_{n-1} : \mathbf{Process} T_{n-1} \}}{\kappa \vdash \mathbf{zip} x : \mathbf{Process} \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}} \\
 \\
 \frac{\kappa \vdash x : (\mathbf{Process} T_0, \dots, \mathbf{Process} T_{n-1})}{\kappa \vdash \mathbf{zip} x : \mathbf{Process} (T_0, \dots, T_{n-1})} \\
 \\
 \frac{\kappa \vdash x : (\mathbf{Process} T) [c]}{\kappa \vdash \mathbf{zip} x : \mathbf{Process} (T [c])} \\
 \\
 \frac{\kappa \vdash x : \mathbf{Process} \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}}{\kappa \vdash \mathbf{unzip} x : \{ l_0 : \mathbf{Process} T_0, \dots, l_{n-1} : \mathbf{Process} T_{n-1} \}} \\
 \\
 \frac{\kappa \vdash x : \mathbf{Process} (T_0, \dots, T_{n-1})}{\kappa \vdash \mathbf{unzip} x : (\mathbf{Process} T_0, \dots, \mathbf{Process} T_{n-1})}
 \end{array}$$



$$\begin{array}{c}
\frac{\kappa \vdash x : \mathbf{Process} (T[c])}{\kappa \vdash \mathbf{unzip} x : (\mathbf{Process} T)[c]} \\
\\
\frac{\kappa \vdash x : \mathbf{Process Boolean}}{\kappa \vdash !x : \mathbf{Process Boolean}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process Boolean} \quad \kappa \vdash x_1 : \mathbf{Process Boolean}}{\kappa \vdash x_0 \&\& x_1 : \mathbf{Process Boolean}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Boolean} \quad \kappa \vdash x_1 : \mathbf{Process Boolean}}{\kappa \vdash x_0 \&\& x_1 : \mathbf{Process Boolean}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process Boolean} \quad \kappa \vdash x_1 : \mathbf{Boolean}}{\kappa \vdash x_0 \&\& x_1 : \mathbf{Process Boolean}} \\
\\
\text{analog für } || \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process Boolean} \quad \kappa \vdash x_1 : \mathbf{Process} T \quad \kappa \vdash x_2 : \mathbf{Process} T}{\kappa \vdash \mathbf{if } x_0 \mathbf{ then } x_1 \mathbf{ else } x_2 : \mathbf{Process} T} \\
\\
\frac{\kappa \vdash x : \mathbf{Process Integer } c \mathbf{ S}}{\kappa \vdash -x : \mathbf{Process Integer } c \mathbf{ S}} \quad \frac{\kappa \vdash x : \mathbf{Process Real}}{\kappa \vdash -x : \mathbf{Process Real}} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process} T \quad \kappa \vdash x_1 : \mathbf{Process} T}{\kappa \vdash x_0 + x_1 : \mathbf{Process} T} \\
\\
\text{falls } T \in \{\mathbf{Integer} \mathit{fmt} \mid \mathit{fmt} \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\frac{\kappa \vdash x_0 : T \quad \kappa \vdash x_1 : \mathbf{Process} T}{\kappa \vdash x_0 + x_1 : \mathbf{Process} T} \\
\\
\text{falls } T \in \{\mathbf{Integer} \mathit{fmt} \mid \mathit{fmt} \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process} T \quad \kappa \vdash x_1 : T}{\kappa \vdash x_0 + x_1 : \mathbf{Process} T} \\
\\
\text{falls } T \in \{\mathbf{Integer} \mathit{fmt} \mid \mathit{fmt} \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\text{analog für } -
\end{array}$$

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ T \quad \kappa \vdash x_1 : \mathbf{Process} \ T}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ T}$$

*falls*  $T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}\}$

$$\frac{\kappa \vdash x_0 : T \quad \kappa \vdash x_1 : \mathbf{Process} \ T}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ T}$$

*falls*  $T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}\}$

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ T \quad \kappa \vdash x_1 : T}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ T}$$

*falls*  $T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}\}$

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ \mathbf{Integer} \, c \, \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process} \ \mathbf{Time}}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ \mathbf{Time}}$$

$$\frac{\kappa \vdash x_0 : \mathbf{Integer} \, c \, \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process} \ \mathbf{Time}}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ \mathbf{Time}}$$

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ \mathbf{Integer} \, c \, \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Time}}{\kappa \vdash x_0 * x_1 : \mathbf{Process} \ \mathbf{Time}}$$

*analog für /*

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ \mathbf{Integer} \, fmt \quad \kappa \vdash x_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt}{\kappa \vdash x_0 \% x_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt}$$

$$\frac{\kappa \vdash x_0 : \mathbf{Integer} \, fmt \quad \kappa \vdash x_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt}{\kappa \vdash x_0 \% x_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt}$$

$$\frac{\kappa \vdash x_0 : \mathbf{Process} \ \mathbf{Integer} \, fmt \quad \kappa \vdash x_1 : \mathbf{Integer} \, fmt}{\kappa \vdash x_0 \% x_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt}$$

$$\frac{\kappa \vdash x : \mathbf{Process} \ \mathbf{Integer} \, c \, \mathbf{U}}{\kappa \vdash \sim x : \mathbf{Process} \ \mathbf{Integer} \, c \, \mathbf{U}}$$

$$\begin{array}{c}
\kappa \vdash x_0 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \\
\hline
\kappa \vdash x_0 \ \& \ x_1 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \\
\\
\kappa \vdash x_0 : \mathbf{Integer}\ c\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \\
\hline
\kappa \vdash x_0 \ \& \ x_1 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \\
\\
\kappa \vdash x_0 : \mathbf{Process\ Integer}\ c\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Integer}\ c\ \mathbf{U} \\
\hline
\kappa \vdash x_0 \ \& \ x_1 : \mathbf{Process\ Integer}\ c\ \mathbf{U}
\end{array}$$

*analog für  $|$  und  $\wedge$*

$$\begin{array}{c}
\kappa \vdash x_0 : \mathbf{Process\ Integer}\ c_0\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process\ Integer}\ c_1\ \mathbf{U} \\
\hline
\kappa \vdash x_0 << x_1 : \mathbf{Process\ Integer}\ c_0\ \mathbf{U} \\
\\
\kappa \vdash x_0 : \mathbf{Integer}\ c_0\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Process\ Integer}\ c_1\ \mathbf{U} \\
\hline
\kappa \vdash x_0 << x_1 : \mathbf{Process\ Integer}\ c_0\ \mathbf{U} \\
\\
\kappa \vdash x_0 : \mathbf{Process\ Integer}\ c_0\ \mathbf{U} \quad \kappa \vdash x_1 : \mathbf{Integer}\ c_1\ \mathbf{U} \\
\hline
\kappa \vdash x_0 << x_1 : \mathbf{Process\ Integer}\ c_0\ \mathbf{U}
\end{array}$$

*analog für  $>>$*

$$\begin{array}{c}
\kappa \vdash x_0 : \mathbf{Process}\ T \quad \kappa \vdash x_1 : \mathbf{Process}\ T \\
\hline
\kappa \vdash x_0 < x_1 : \mathbf{Process\ Boolean} \\
\\
\text{falls } T \in \{\mathbf{Integer}\ fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\kappa \vdash x_0 : T \quad \kappa \vdash x_1 : \mathbf{Process}\ T \\
\hline
\kappa \vdash x_0 < x_1 : \mathbf{Process\ Boolean} \\
\\
\text{falls } T \in \{\mathbf{Integer}\ fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\kappa \vdash x_0 : \mathbf{Process}\ T \quad \kappa \vdash x_1 : T \\
\hline
\kappa \vdash x_0 < x_1 : \mathbf{Process\ Boolean} \\
\\
\text{falls } T \in \{\mathbf{Integer}\ fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Real}, \mathbf{Time}\} \\
\\
\text{analog für } <=, >=, >
\end{array}$$

$$\begin{array}{c}
\frac{\kappa \vdash x_0 : \mathbf{Process} \ T \quad \kappa \vdash x_1 : \mathbf{Process} \ T}{\kappa \vdash x_0 == x_1 : \mathbf{Process} \ \mathbf{Boolean}} \\
\text{falls } T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Boolean}, \mathbf{Real}, \mathbf{Time}\} \\
\\
\frac{\kappa \vdash x_0 : T \quad \kappa \vdash x_1 : \mathbf{Process} \ T}{\kappa \vdash x_0 == x_1 : \mathbf{Process} \ \mathbf{Boolean}} \\
\text{falls } T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Boolean}, \mathbf{Real}, \mathbf{Time}\} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process} \ T \quad \kappa \vdash x_1 : T}{\kappa \vdash x_0 == x_1 : \mathbf{Process} \ \mathbf{Boolean}} \\
\text{falls } T \in \{\mathbf{Integer} \, fmt \mid fmt \in \mathbf{IntegerFormat}\} \cup \{\mathbf{Boolean}, \mathbf{Real}, \mathbf{Time}\} \\
\\
\text{analog für } != \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \mathbf{Integer} \, fmt_0}{\kappa \vdash x \text{ as } \mathbf{Process} \ \mathbf{Integer} \, fmt_1 : \mathbf{Process} \ \mathbf{Integer} \, fmt_1} \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \mathbf{Integer} \, fmt}{\kappa \vdash x \text{ as } \mathbf{Process} \ \mathbf{Real} : \mathbf{Process} \ \mathbf{Real}} \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \mathbf{Time}}{\kappa \vdash x \text{ as } \mathbf{Process} \ \mathbf{Integer} \, fmt : \mathbf{Process} \ \mathbf{Integer} \, fmt} \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \mathbf{Time}}{\kappa \vdash x \text{ as } \mathbf{Process} \ \mathbf{Real} : \mathbf{Process} \ \mathbf{Real}} \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}}{\kappa \vdash x : \mathbf{Process} \ \{ l_{i_0} : T_{i_0}, \dots, l_{i_{n-1}} : T_{i_{n-1}} \}} \\
\text{falls } \{i_0, \dots, i_{n-1}\} = 0..n-1 \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ \{ l_0 : T_0 \mid \dots \mid l_n : T_n \}}{\kappa \vdash x : \mathbf{Process} \ \{ l_{i_0} : T_{i_0} \mid \dots \mid l_{i_n} : T_{i_n} \}} \\
\text{falls } \{i_0, \dots, i_n\} = 0..n
\end{array}$$

□

**Semantik B.3.2.2 (Statische Systeme)**

$$\llbracket \kappa \vdash \text{apply } f \text{ to } x : \text{Process } T_1 \rrbracket \eta = \\ \text{apply}_{\mathcal{D}(T_0) \rightarrow \mathcal{D}(T_1)} (\llbracket \kappa \vdash f : T_0 \multimap T_1 \rrbracket \eta) (\llbracket \kappa \vdash x : \text{Process } T_0 \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{zip } x : \text{Process } \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta = \\ \text{zip}_{\mathcal{D}(\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \})} (\llbracket \kappa \vdash x : \{ l_0 : \text{Process } T_0, \dots, l_{n-1} : \text{Process } T_{n-1} \} \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{zip } x : \text{Process } (T_0, \dots, T_{n-1}) \rrbracket \eta = \\ \text{zip}_{\mathcal{D}((T_0, \dots, T_{n-1}))} (\llbracket \kappa \vdash x : (\text{Process } T_0, \dots, \text{Process } T_{n-1}) \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{zip } x : \text{Process } (T [c]) \rrbracket \eta = \\ \text{zip}_{\mathcal{D}(T[c])} (\llbracket \kappa \vdash x : (\text{Process } T) [c] \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{unzip } x : \{ l_0 : \text{Process } T_0, \dots, l_{n-1} : \text{Process } T_{n-1} \} \rrbracket \eta = \\ \text{unzip}_{\mathcal{D}(\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \})} (\llbracket \kappa \vdash x : \text{Process } \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{unzip } x : (\text{Process } T_0, \dots, \text{Process } T_{n-1}) \rrbracket \eta = \\ \text{unzip}_{\mathcal{D}((T_0, \dots, T_{n-1}))} (\llbracket \kappa \vdash x : \text{Process } (T_0, \dots, T_{n-1}) \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{unzip } x : (\text{Process } T) [c] \rrbracket \eta = \\ \text{unzip}_{\mathcal{D}(T[c])} (\llbracket \kappa \vdash x : \text{Process } (T [c]) \rrbracket \eta)$$

$$\llbracket \kappa \vdash ! x : \text{Process Boolean} \rrbracket \eta = \\ \text{apply}_{\mathbb{B} \rightarrow \mathbb{B}} (\lambda x \in \mathbb{B}. \neg x) (\llbracket \kappa \vdash x : \text{Process Boolean} \rrbracket \eta)$$

*analog für  $-$ ,  $\sim$*

$$\llbracket \kappa \vdash x_0 \&\& x_1 : \text{Process Boolean} \rrbracket \eta = \\ \text{apply}_{\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}} (\lambda [x_0 \in \mathbb{B}, x_1 \in \mathbb{B}]. x_0 \wedge x_1) \\ (\text{zip}_{\mathbb{B} \times \mathbb{B}} (\llbracket \kappa \vdash x_0 : \text{Process Boolean} \rrbracket \eta, \\ \llbracket \kappa \vdash x_1 : \text{Process Boolean} \rrbracket \eta))$$

$$\llbracket \kappa \vdash x_0 \&\& x_1 : \text{Process Boolean} \rrbracket \eta = \\ \text{apply}_{\mathbb{B} \rightarrow \mathbb{B}} (\lambda x_1 \in \mathbb{B}. (\llbracket \kappa \vdash x_0 : \text{Boolean} \rrbracket \eta) \wedge x_1) \\ (\llbracket \kappa \vdash x_1 : \text{Process Boolean} \rrbracket \eta)$$

$$\begin{aligned} \llbracket \kappa \vdash x_0 \ \&\& \ x_1 : \mathbf{Process \ Boolean} \rrbracket \eta = \\ &\mathbf{apply}_{\mathbb{B} \rightarrow \mathbb{B}} (\lambda x_0 \in \mathbb{B}. x_0 \wedge (\llbracket \kappa \vdash x_1 : \mathbf{Boolean} \rrbracket \eta)) \\ &(\llbracket \kappa \vdash x_0 : \mathbf{Process \ Boolean} \rrbracket \eta) \end{aligned}$$

*analog für*  $\|$ ,  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\&$ ,  $|$ ,  $\wedge$ ,  $<<$ ,  $>>$ ,  
 $<$ ,  $<=$ ,  $>=$ ,  $>$ ,  $=$ , *und*  $!$  =

$$\begin{aligned} \llbracket \kappa \vdash \mathbf{if} \ x_0 \ \mathbf{then} \ x_1 \ \mathbf{else} \ x_2 : \mathbf{Process} \ T \rrbracket \eta = \\ &\mathbf{apply}_{\mathbb{B} \times \mathcal{D}(T) \times \mathcal{D}(T) \rightarrow \mathcal{D}(T)} \\ &\left( \lambda [x_0 \in \mathbb{B}, x_1 \in T, x_2 \in T]. \begin{cases} x_1 & \text{falls } x_0 \\ x_2 & \text{sonst} \end{cases} \right) \\ &(\mathbf{zip}_{\mathbb{B} \times \mathcal{D}(T) \times \mathcal{D}(T)} [\llbracket \kappa \vdash x_0 : \mathbf{Process \ Boolean} \rrbracket \eta, \\ &\quad \llbracket \kappa \vdash x_1 : \mathbf{Process} \ T \rrbracket \eta, \\ &\quad \llbracket \kappa \vdash x_2 : \mathbf{Process} \ T \rrbracket \eta]) \end{aligned}$$

$$\begin{aligned} \llbracket \kappa \vdash x \ \mathbf{as} \ \mathbf{Process} \ \mathbf{Integer} \ \mathbf{fmt}_1 : \mathbf{Integer} \ \mathbf{fmt}_1 \rrbracket \eta = \\ &\mathbf{apply}_{\mathcal{D}(\mathbf{Integer} \ \mathbf{fmt}_0) \rightarrow \mathcal{D}(\mathbf{Integer} \ \mathbf{fmt}_1)} \\ &(\lambda x \in \mathcal{D}(\mathbf{Integer} \ \mathbf{fmt}_0). \mathbf{int} \ x \ \mathbf{fmt}_1) \\ &(\llbracket \kappa \vdash x : \mathbf{Process} \ \mathbf{Integer} \ \mathbf{fmt}_0 \rrbracket \eta) \end{aligned}$$

*analog für*  $[\mathbf{Integer} \ \mathbf{fmt}, \mathbf{Real}]$ ,  $[\mathbf{Time}, \mathbf{Integer} \ \mathbf{fmt}]$   
*und*  $[\mathbf{Time}, \mathbf{Real}]$

□

Es folgt die Einheits-Zeitverschiebung für quasi-kontinuierliche Systeme:

### Syntax B.3.2.3 (Zeitverschiebung)

#### Operatoren

$$\mathbf{previous} \ \langle \mathbf{Expression} \rangle \ \mathbf{initially} \ \langle \mathbf{Expression} \rangle \ \rightarrow \ \langle \mathbf{Expression} \rangle$$

□

### Typisierung B.3.2.3 (Zeitverschiebung)

$$\frac{\kappa \vdash x : \mathbf{Process} \ T \quad \kappa \vdash x_0 : T}{\kappa \vdash \mathbf{previous} \ x \ \mathbf{initially} \ x_0 : \mathbf{Process} \ T}$$

□

**Semantik B.3.2.3 (Zeitverschiebung)**

$$\llbracket \kappa \vdash \text{previous } x \text{ initially } x_0 : \text{Process } T \rrbracket \eta = \\ \text{delay}_{\mathcal{D}(T)} (\llbracket \kappa \vdash x_0 : T \rrbracket \eta) (\llbracket \kappa \vdash x : \text{Process } T \rrbracket \eta)$$

□

**B.3.3 Reaktive Prozesse**

Spezifisch zur Beschreibung von Phasenübergangssystemen und reaktiven Prozessen wird ein neuer Typkonstruktor für  $\text{Phase}_X$  (eine weitere vordefinierte Typfunktion) sowie Operatoren für  $\text{goto}_X$ ,  $\text{wait}_X$ ,  $\text{when}_X$ ,  $\text{phase}_X$ ,  $\text{valueInPhase}_{Y,X}$ ,  $\text{switch}_X$  und  $\text{start}_X$  definiert (siehe auch Kapitel 7.3.4):

**Syntax B.3.3.1 (Phasen)****Sorten**

$\langle \text{Continuation} \rangle, \langle \text{Transition} \rangle$

**Operatoren**

Phase	→	$\langle \text{Type} \rangle$
then $\langle \text{Expression} \rangle$	→	$\langle \text{Continuation} \rangle$
then wait	→	$\langle \text{Continuation} \rangle$
when $\langle \text{Expression} \rangle \langle \text{Continuation} \rangle$	→	$\langle \text{Transition} \rangle$
keep $\langle \text{Expression} \rangle$	→	$\langle \text{Expression} \rangle$
local $\langle \text{Declaration} \rangle := \langle \text{Expression} \rangle$ in $\langle \text{Expression} \rangle$	→	$\langle \text{Expression} \rangle$
do $\langle \text{Expression} \rangle \langle \text{Transition} \rangle^+$	→	$\langle \text{Expression} \rangle$
start $\langle \text{Expression} \rangle$	→	$\langle \text{Expression} \rangle$ □

Die beiden Varianten des then-Operators stehen für  $\text{goto}_X$  und  $\text{wait}_X$ , der when-Operator für  $\text{when}_X$ . keep steht für  $\text{phase}_X$ , do für  $\text{switch}_X$  und start für  $\text{start}_X$ . start ist einstellig; der Operand steht für die Phase, während der Startzeitpunkt auf  $t_0 = 0_{\text{Time}}$  fixiert ist. Das local-Konstrukt ähnelt syntaktisch dem let-Konstrukt für lokale Definitionen und vereinigt  $\text{valueInPhase}_{Y,X}$  mit einer  $\lambda$ -Abstraktion, um aus einem  $\text{Phase}_X$ -wertigen Ausdruck eine  $(Y \rightarrow \text{Phase}_X)$ -wertige Funktion

zu konstruieren (siehe Semantik B.3.3.1). Darüber hinaus kann zur Repräsentation von  $\text{ifElsePhase}_X$  der  $\text{if then else}$ -Operator für Phasen überladen werden.

Zum Zweck einer vereinfachten Spezifikation der Typisierung und der Semantik werden die Sorten  $\langle \text{Continuation} \rangle$  und  $\langle \text{Transition} \rangle$  als Untersorten von  $\langle \text{Expression} \rangle$  deklariert und mit entsprechenden Typkonstrukturen (als Typfunktionen) versehen. Fortsetzungen und Transitionen werden damit formal als erstklassige Objekte behandelt:

### Syntax B.3.3.2 (Fortsetzungen und Transitionen als Ausdrücke)

#### Operatoren

$\langle \text{Continuation} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
$\langle \text{Transition} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>Continuation</b>	$\rightarrow$	$\langle \text{Type} \rangle$
<b>Transition</b>	$\rightarrow$	$\langle \text{Type} \rangle$

□

In die Spezifikation der (konkreten) Syntax wird diese Einbettung jedoch nicht übernommen, wodurch sie von der Benutzung durch den Programmierer ausgenommen wird. Denn die sonst entstehenden zusätzlichen Freiheitsgrade für die Programmierung<sup>4</sup> könnten der Lesbarkeit und Wartbarkeit von Programmen sehr abträglich sein. Als Metavariablen für Fortsetzungen und Transitionen werden *cont* bzw. *tr*, auch in indizierter oder gestrichener Form, verwendet.

#### Typisierung B.3.3.1 (Fortsetzungen und Transitionen)

$$\begin{array}{c}
 \frac{\kappa \vdash ph : \text{Phase } T}{\kappa \vdash \text{then } ph : \text{Continuation } T} \\
 \\
 \frac{}{\kappa \vdash \text{then wait} : \text{Continuation } T} \\
 \\
 \frac{\kappa \vdash e : \text{Event} \quad \kappa \vdash cont : \text{Continuation } T}{\kappa \vdash \text{when } e \text{ cont} : \text{Transition } T}
 \end{array}$$

□

<sup>4</sup>Man denke z. B. an fortsetzungs- oder transitionswertige Variablen oder Funktionen.



Die Typisierung von `then wait` bildet eine Ausnahme in der Weise, daß kein eindeutiger oder bis auf Typäquivalenz eindeutiger Typ zugeordnet wird. `then wait` ist für alle Typen  $T$  überladen, wobei  $T$  nicht aus dem Ausdruck selbst abgeleitet werden kann. Die eingeschränkte Verwendung des Typkonstruktors `Continuation` (nur innerhalb der Sprachspezifikation, nicht für die Programmierung) stellt jedoch sicher, daß das in Abschnitt B.2.2 beschriebene Konzept zur Typsicherheit der Programmiersprache nicht verletzt wird.

### Typisierung B.3.3.2 (Phasen)

$$\begin{array}{c}
\frac{\kappa \vdash x : \mathbf{Process} \ T}{\kappa \vdash \mathbf{keep} \ x : \mathbf{Phase} \ T} \\
\\
\frac{\kappa \vdash x : \mathbf{Process} \ T \quad \kappa[id := T] \vdash ph : \mathbf{Phase} \ T'}{\kappa \vdash \mathbf{local \ value} \ id : T := x \ \mathbf{in} \ ph : \mathbf{Phase} \ T'} \\
\\
\frac{\kappa \vdash ph : \mathbf{Phase} \ T \quad \kappa \vdash tr_0 : \mathbf{Transition} \ T \quad \dots \quad \kappa \vdash tr_n : \mathbf{Transition} \ T}{\kappa \vdash \mathbf{do} \ ph \ tr_0 \ \dots \ tr_n : \mathbf{Phase} \ T} \\
\\
\frac{\kappa \vdash ph : \mathbf{Phase} \ T}{\kappa \vdash \mathbf{start} \ ph : \mathbf{Process} \ T} \\
\\
\frac{\kappa \vdash x_0 : \mathbf{Process} \ \mathbf{Boolean} \quad \kappa \vdash x_1 : \mathbf{Phase} \ T \quad \kappa \vdash x_2 : \mathbf{Phase} \ T}{\kappa \vdash \mathbf{if} \ x_0 \ \mathbf{then} \ x_1 \ \mathbf{else} \ x_2 : \mathbf{Phase} \ T}
\end{array}$$

□

Die Semantik der Typkonstruktoren und Operatoren ergibt sich aus der Modellbildung:

### Typsemantik B.3.3.1 (Phasentypen)

$$\begin{aligned}
\mathcal{D}(\mathbf{Phase} \ T) &= \mathbf{Phase}_{\mathcal{D}(T)} \\
\mathcal{D}(\mathbf{Continuation} \ T) &= \mathbf{Continuation}_{\mathcal{D}(T)} \\
\mathcal{D}(\mathbf{Transition} \ T) &= \mathbf{Transition}_{\mathcal{D}(T)}
\end{aligned}$$

□

**Semantik B.3.3.1 (Phasen)**

$$\begin{aligned}
\llbracket \kappa \vdash \text{then } ph : \text{Continuation } T \rrbracket \eta &= \\
&\quad \text{goto}_{\mathcal{D}(T)} (\llbracket \kappa \vdash ph : \text{Phase } T \rrbracket \eta) \\
\llbracket \kappa \vdash \text{then wait} : \text{Continuation } T \rrbracket \eta &= \\
&\quad \text{wait}_{\mathcal{D}(T)} \\
\llbracket \kappa \vdash \text{when } e \text{ cont} : \text{Transition } T \rrbracket \eta &= \\
&\quad \text{when}_{\mathcal{D}(T)} (\llbracket \kappa \vdash e : \text{Event} \rrbracket \eta) (\llbracket \kappa \vdash \text{cont} : \text{Continuation } T \rrbracket \eta) \\
\llbracket \kappa \vdash \text{keep } x : \text{Phase } T \rrbracket \eta &= \\
&\quad \text{phase}_{\mathcal{D}(T)} (\llbracket \kappa \vdash x : \text{Process } T \rrbracket \eta) \\
\llbracket \kappa \vdash \text{local value } id : T := x \text{ in } ph : \text{Phase } T' \rrbracket \eta &= \\
&\quad \text{valueInPhase}_{\mathcal{D}(T)} \\
&\quad (\llbracket \kappa \vdash x : \text{Process } T \rrbracket \eta) \\
&\quad (\lambda y \in \mathcal{D}(T). \llbracket \kappa[id := T] \vdash ph : \text{Phase } T' \rrbracket \eta[id := y]) \\
\llbracket \kappa \vdash \text{do } ph \ tr_0 \ \dots \ tr_n : \text{Phase } T \rrbracket \eta &= \\
&\quad \text{switch}_{\mathcal{D}(T)} (\llbracket \kappa \vdash ph : \text{Phase } T \rrbracket \eta) \\
&\quad [\llbracket \kappa \vdash tr_0 : \text{Transition } T \rrbracket \eta, \dots, \llbracket \kappa \vdash tr_n : \text{Transition } T \rrbracket \eta] \\
\llbracket \kappa \vdash \text{start } ph : \text{Process } T \rrbracket \eta &= \\
&\quad \text{start}_{\mathcal{D}(T)} (\llbracket \kappa \vdash ph : \text{Phase } T \rrbracket \eta) \ 0_{\text{Time}} \\
\llbracket \kappa \vdash \text{if } x_0 \text{ then } x_1 \text{ else } x_2 : \text{Phase } T \rrbracket \eta &= \\
&\quad \text{ifElsePhase}_{\mathcal{D}(T)} \\
&\quad (\llbracket \kappa \vdash x_0 : \text{Process Boolean} \rrbracket \eta) \\
&\quad (\llbracket \kappa \vdash x_1 : \text{Phase } T \rrbracket \eta) \\
&\quad (\llbracket \kappa \vdash x_2 : \text{Phase } T \rrbracket \eta)
\end{aligned}$$

□

Für die Phasenparallelisierung wird ein neuer Operator `orthogonalize` eingeführt und für alle Produkttypen und iterativen Strukturen überladen:

**Syntax B.3.3.3 (Phasenparallelisierung)****Operatoren**

orthogonalize  $\langle \text{Expression} \rangle \rightarrow \langle \text{Expression} \rangle$

□

**Typisierung B.3.3.3 (Phasenparallelisierung)**

$$\frac{\kappa \vdash ph : \{ l_0 : \mathbf{Phase} T_0, \dots, l_{n-1} : \mathbf{Phase} T_{n-1} \}}{\kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \}}$$

$$\frac{\kappa \vdash ph : (\mathbf{Phase} T_0, \dots, \mathbf{Phase} T_{n-1})}{\kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} (T_0, \dots, T_{n-1})}$$

$$\frac{\kappa \vdash ph : (\mathbf{Phase} T) [c]}{\kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} (T [c])}$$

□

**Semantik B.3.3.2 (Phasenparallelisierung)**

$$\begin{aligned} \llbracket \kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta &= \\ \text{orthogonalize}_{\mathcal{D}(\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \})} & \\ (\llbracket \kappa \vdash ph : \{ l_0 : \mathbf{Phase} T_0, \dots, l_{n-1} : \mathbf{Phase} T_{n-1} \} \rrbracket \eta) & \\ \llbracket \kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} (T_0, \dots, T_{n-1}) \rrbracket \eta &= \\ \text{orthogonalize}_{\mathcal{D}((T_0, \dots, T_{n-1}))} & \\ (\llbracket \kappa \vdash ph : (\mathbf{Phase} T_0, \dots, \mathbf{Phase} T_{n-1}) \rrbracket \eta) & \\ \llbracket \kappa \vdash \text{orthogonalize } ph : \mathbf{Phase} (T [c]) \rrbracket \eta &= \\ \text{orthogonalize}_{\mathcal{D}(T[c])} & \\ (\llbracket \kappa \vdash ph : (\mathbf{Phase} T) [c] \rrbracket \eta) & \end{aligned}$$

□

Bei  $\text{variableInPhase}_{Y,X}$  liegt konzeptionell eine starke Verwandtschaft mit  $\text{valueInPhase}_{Y,X}$  vor. Syntaktisch wird deshalb der local-Operator aus Syntax B.3.3.1 übernommen und überladen:

**Typisierung B.3.3.4 (Lokale Variable)**

$$\frac{\kappa \vdash ph : \mathbf{Phase} \ T \quad \kappa[id := \mathbf{Process} \ T] \vdash ph' : \mathbf{Phase} \ T'}{\kappa \vdash \mathbf{local\ value} \ id : \mathbf{Process} \ T := ph \ \mathbf{in} \ ph' : \mathbf{Phase} \ T'}$$

□

**Semantik B.3.3.3 (Lokale Variable)**

$$\begin{aligned} \llbracket \kappa \vdash \mathbf{local\ value} \ id : \mathbf{Process} \ T := x \ \mathbf{in} \ ph' : \mathbf{Phase} \ T' \rrbracket \eta = & \\ \text{variableInPhase}_{\mathcal{D}(T)} & \\ (\llbracket \kappa \vdash ph : \mathbf{Phase} \ T \rrbracket \eta) & \\ (\lambda y \in \mathcal{D}(\mathbf{Process} \ T) . \llbracket \kappa[id := \mathbf{Process} \ T] \vdash ph' : \mathbf{Phase} \ T' \rrbracket \eta[id := y]) & \end{aligned}$$

□

Für  $\text{valueInEvent}_X$  und  $\text{variableInEvent}_X$  im Zusammenhang mit Ereignissen wird der  $\mathbf{local}$ -Operator weitere Male überladen:

**Typisierung B.3.3.5 (Ereignislokale Auswertung und Variable)**

$$\frac{\kappa \vdash x : \mathbf{Process} \ T \quad \kappa[id := T] \vdash e : \mathbf{Event}}{\kappa \vdash \mathbf{local\ value} \ id : T := x \ \mathbf{in} \ e : \mathbf{Event}}$$

$$\frac{\kappa \vdash ph : \mathbf{Phase} \ T \quad \kappa[id := \mathbf{Process} \ T] \vdash e : \mathbf{Event}}{\kappa \vdash \mathbf{local\ value} \ id : \mathbf{Process} \ T := ph \ \mathbf{in} \ e : \mathbf{Event}}$$

□

**Semantik B.3.3.4 (Ereignislokale Auswertung und Variable)**

$$\begin{aligned} \llbracket \kappa \vdash \mathbf{local\ value} \ id : T := x \ \mathbf{in} \ e : \mathbf{Event} \rrbracket \eta = & \\ \text{valueInEvent}_{\mathcal{D}(T)} & \\ (\llbracket \kappa \vdash x : \mathbf{Process} \ T \rrbracket \eta) & \\ (\lambda y \in \mathcal{D}(T) . \llbracket \kappa[id := T] \vdash e : \mathbf{Event} \rrbracket \eta[id := y]) & \\ \llbracket \kappa \vdash \mathbf{local\ value} \ id : \mathbf{Process} \ T := ph \ \mathbf{in} \ e : \mathbf{Event} \rrbracket \eta = & \\ \text{variableInEvent}_{\mathcal{D}(T)} & \\ (\llbracket \kappa \vdash ph : \mathbf{Phase} \ T \rrbracket \eta) & \\ (\lambda y \in \mathcal{D}(\mathbf{Process} \ T) . \llbracket \kappa[id := \mathbf{Process} \ T] \vdash e : \mathbf{Event} \rrbracket \eta[id := y]) & \end{aligned}$$

□

### B.3.4 Sequentielle Prozesse

Entwurf: 7.3.5

Es folgt die Erweiterung der Sprachdefinition, beginnend mit der Syntax. Der abstrakte Datentyp  $\text{Action}_X$  wird in Form eines neuen Typkonstruktors (der analog zu **Process** und **Phase** eine vordefinierte Typfunktion darstellt) und für alle Datentypen überladener Operatoren eingeführt. Das schon mehrfach überladene **if then else**-Konstrukt wird auch hier wiederverwendet:

#### Syntax B.3.4.1 (Aktionen)

##### Operatoren

<b>Action</b>	$\rightarrow$	$\langle \text{Type} \rangle$
<b>do</b> $\langle \text{Expression} \rangle$ <b>until</b> $\langle \text{Expression} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>complete</b> $\langle \text{Expression} \rangle$ $\langle \text{Continuation} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>complete</b> $\langle \text{Expression} \rangle$ $\langle \text{Continuation} \rangle$ <b>except</b> $\langle \text{Transition} \rangle$ <b>+</b>	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>loop</b> $\langle \text{Expression} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
$\langle \text{Expression} \rangle$ ; $\langle \text{Expression} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>if</b> $\langle \text{Expression} \rangle$ <b>then</b> $\langle \text{Expression} \rangle$ <b>else</b> $\langle \text{Expression} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>repeat</b> $\langle \text{Expression} \rangle$ <b>until</b> $\langle \text{Expression} \rangle$	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>parallelize</b> $\langle \text{Expression} \rangle$ <b>terminating</b>	$\rightarrow$	$\langle \text{Expression} \rangle$
<b>parallelize</b> $\langle \text{Expression} \rangle$ <b>waiting</b>	$\rightarrow$	$\langle \text{Expression} \rangle$ $\square$

Die Typisierung und die Semantik ergeben sich aus der Modellierung:

#### Typisierung B.3.4.1 (Aktionen)

$$\begin{array}{c}
 \kappa \vdash ph : \mathbf{Phase} \ T \quad \kappa \vdash e : \mathbf{Event} \\
 \hline
 \kappa \vdash \mathbf{do} \ ph \ \mathbf{until} \ e : \mathbf{Action} \ T \\
 \\
 \kappa \vdash a : \mathbf{Action} \ T \quad \kappa \vdash cont : \mathbf{Continuation} \ T \\
 \hline
 \kappa \vdash \mathbf{complete} \ a \ cont : \mathbf{Phase} \ T
 \end{array}$$

$$\begin{array}{c}
\kappa \vdash a : \mathbf{Action} \ T \quad \kappa \vdash cont : \mathbf{Continuation} \ T \\
\kappa \vdash tr_0 : \mathbf{Transition} \ T \quad \dots \quad \kappa \vdash tr_n : \mathbf{Transition} \ T \\
\hline
\kappa \vdash \mathbf{complete} \ a \ cont \ \mathbf{except} \ tr_0 \ \dots \ tr_n : \mathbf{Phase} \ T \\
\\
\kappa \vdash a : \mathbf{Action} \ T \\
\hline
\kappa \vdash \mathbf{loop} \ a : \mathbf{Phase} \ T \\
\\
\kappa \vdash a_0 : \mathbf{Action} \ T \quad \kappa \vdash a_1 : \mathbf{Action} \ T \\
\hline
\kappa \vdash a_0 ; a_1 : \mathbf{Action} \ T \\
\\
\kappa \vdash x : \mathbf{Integer} \ c \ \mathbf{U} \quad \kappa \vdash a : \mathbf{Action} \ T \\
\hline
\kappa \vdash x * a : \mathbf{Action} \ T \\
\\
\kappa \vdash x : \mathbf{Process} \ \mathbf{Boolean} \quad \kappa \vdash a_0 : \mathbf{Action} \ T \quad \kappa \vdash a_1 : \mathbf{Action} \ T \\
\hline
\kappa \vdash \mathbf{if} \ x \ \mathbf{then} \ a_0 \ \mathbf{else} \ a_1 : \mathbf{Action} \ T \\
\\
\kappa \vdash a : \mathbf{Action} \ T \quad \kappa \vdash x : \mathbf{Process} \ \mathbf{Boolean} \\
\hline
\kappa \vdash \mathbf{repeat} \ a \ \mathbf{until} \ x : \mathbf{Action} \ T \\
\\
\kappa \vdash a_0 : \mathbf{Action} \ T_0 \quad \dots \quad \kappa \vdash a_{n-1} : \mathbf{Action} \ T_{n-1} \\
\hline
\kappa \vdash \mathbf{parallelize} \ \{ l_0 : a_0, \dots, l_{n-1} : a_{n-1} \} \ \mathbf{terminating} : \\
\mathbf{Action} \ \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \\
\\
\kappa \vdash a_0 : \mathbf{Action} \ T_0 \quad \dots \quad \kappa \vdash a_{n-1} : \mathbf{Action} \ T_{n-1} \\
\hline
\kappa \vdash \mathbf{parallelize} \ (a_0, \dots, a_{n-1}) \ \mathbf{terminating} : \\
\mathbf{Action} \ (T_0, \dots, T_{n-1}) \\
\\
\kappa \vdash a : (\mathbf{Action} \ T) [c] \\
\hline
\kappa \vdash \mathbf{parallelize} \ a \ \mathbf{terminating} : \mathbf{Action} \ (T [c]) \\
\\
\kappa \vdash a_0 : \mathbf{Action} \ T_0 \quad \dots \quad \kappa \vdash a_{n-1} : \mathbf{Action} \ T_n \\
\hline
\kappa \vdash \mathbf{parallelize} \ \{ l_0 : a_0, \dots, l_n : a_n \} \ \mathbf{waiting} : \\
\mathbf{Action} \ \{ l_0 : T_0, \dots, l_n : T_n \}
\end{array}$$

$$\begin{array}{c}
\kappa \vdash a_0 : \mathbf{Action} \, T_0 \quad \dots \quad \kappa \vdash a_n : \mathbf{Action} \, T_n \\
\hline
\kappa \vdash \mathbf{parallelize} \, (a_0, \dots, a_n) \, \mathbf{waiting} : \\
\mathbf{Action} \, (T_0, \dots, T_n) \\
\\
\kappa \vdash a : (\mathbf{Action} \, T) [c] \\
\hline
\kappa \vdash \mathbf{parallelize} \, a \, \mathbf{waiting} : \mathbf{Action} \, (T [c])
\end{array}$$

□

### Typsemantik B.3.4.1 (Aktionstypen)

$$\mathcal{D}(\mathbf{Action} \, T) = \mathbf{Action}_{\mathcal{D}(T)}$$

□

### Semantik B.3.4.1 (Aktionen)

$$\begin{aligned}
\llbracket \kappa \vdash \mathbf{do} \, ph \, \mathbf{until} \, e : \mathbf{Action} \, T \rrbracket \eta &= \\
&\quad \mathbf{until}_{\mathcal{D}(T)} (\llbracket \kappa \vdash e : \mathbf{Event} \rrbracket \eta) (\llbracket \kappa \vdash ph : \mathbf{Phase} \, T \rrbracket \eta) \\
\llbracket \kappa \vdash \mathbf{complete} \, a \, cont : \mathbf{Phase} \, T \rrbracket \eta &= \\
&\quad \mathbf{continue}_{\mathcal{D}(T)} (\llbracket \kappa \vdash a : \mathbf{Action} \, T \rrbracket \eta) (\llbracket \kappa \vdash cont : \mathbf{Continuation} \, T \rrbracket \eta) [] \\
\llbracket \kappa \vdash \mathbf{complete} \, a \, cont \, \mathbf{except} \, tr_0 \, \dots \, tr_n : \mathbf{Phase} \, T \rrbracket \eta &= \\
&\quad \mathbf{continueWithExceptions}_{\mathcal{D}(T)} \\
&\quad (\llbracket \kappa \vdash a : \mathbf{Action} \, T \rrbracket \eta) (\llbracket \kappa \vdash cont : \mathbf{Continuation} \, T \rrbracket \eta) \\
&\quad (\llbracket \kappa \vdash tr_0 : \mathbf{Transition} \, T \rrbracket \eta, \dots, \llbracket \kappa \vdash tr_n : \mathbf{Transition} \, T \rrbracket \eta) \\
\llbracket \kappa \vdash \mathbf{loop} \, a : \mathbf{Phase} \, T \rrbracket \eta &= \\
&\quad \mathbf{loop}_{\mathcal{D}(T)} (\llbracket \kappa \vdash a : \mathbf{Action} \, T \rrbracket \eta) \\
\llbracket \kappa \vdash a_0 ; a_1 : \mathbf{Action} \, T \rrbracket \eta &= \\
&\quad \mathbf{sequence}_{\mathcal{D}(T)} (\llbracket \kappa \vdash a_0 : \mathbf{Action} \, T \rrbracket \eta) (\llbracket \kappa \vdash a_1 : \mathbf{Action} \, T \rrbracket \eta) \\
\llbracket \kappa \vdash x * a : \mathbf{Action} \, T \rrbracket \eta &= \\
&\quad (\llbracket \kappa \vdash x : \mathbf{Integer} \, c \, \mathbf{U} \rrbracket \eta) \underset{\mathbf{Action}_{\mathcal{D}(T)}}{*} (\llbracket \kappa \vdash a : \mathbf{Action} \, T \rrbracket \eta) \\
\mathbf{falls} \, (\llbracket \kappa \vdash x : \mathbf{Integer} \, c \, \mathbf{U} \rrbracket \eta) > 0 &
\end{aligned}$$

$$\llbracket \kappa \vdash \text{if } x \text{ then } a_0 \text{ else } a_1 : \mathbf{Action } T \rrbracket \eta =$$

$$\begin{aligned} & \text{ifElse}_{\mathcal{D}(T)} \\ & (\llbracket \kappa \vdash x : \mathbf{Process Boolean} \rrbracket \eta) \\ & (\llbracket \kappa \vdash a_0 : \mathbf{Action } T \rrbracket \eta) \\ & (\llbracket \kappa \vdash a_1 : \mathbf{Action } T \rrbracket \eta) \end{aligned}$$

$$\llbracket \kappa \vdash \text{repeat } a \text{ until } x : \mathbf{Action } T \rrbracket \eta =$$

$$\text{repeatUntil}_{\mathcal{D}(T)} (\llbracket \kappa \vdash a : \mathbf{Action } T \rrbracket \eta) (\llbracket \kappa \vdash x : \mathbf{Process Boolean} \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{parallelize } \{ l_0 : a_0, \dots, l_{n-1} : a_{n-1} \} \text{ terminating} :$$

$$\mathbf{Action } \{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \} \rrbracket \eta \quad =$$

$$\begin{aligned} & \text{parallelizeTerminating}_{\mathcal{D}(\{ l_0 : T_0, \dots, l_{n-1} : T_{n-1} \})} \{ \\ & \quad l_0 : \llbracket \kappa \vdash a_0 : \mathbf{Action } T_0 \rrbracket \eta, \\ & \quad \dots, \\ & \quad l_{n-1} : \llbracket \kappa \vdash a_{n-1} : \mathbf{Action } T_{n-1} \rrbracket \eta \\ & \} \end{aligned}$$

$$\llbracket \kappa \vdash \text{parallelize } (a_0, \dots, a_{n-1}) \text{ terminating} :$$

$$\mathbf{Action } (T_0, \dots, T_{n-1}) \rrbracket \eta \quad =$$

$$\begin{aligned} & \text{parallelizeTerminating}_{\mathcal{D}((T_0, \dots, T_{n-1}))} [ \\ & \quad \llbracket \kappa \vdash a_0 : \mathbf{Action } T_0 \rrbracket \eta, \\ & \quad \dots, \\ & \quad \llbracket \kappa \vdash a_{n-1} : \mathbf{Action } T_{n-1} \rrbracket \eta \\ & ] \end{aligned}$$

$$\llbracket \kappa \vdash \text{parallelize } a \text{ terminating} : \mathbf{Action } (T [c]) \rrbracket \eta =$$

$$\text{parallelizeTerminating}_{\mathcal{D}(T[c])} (\llbracket \kappa \vdash a : (\mathbf{Action } T) [c] \rrbracket \eta)$$

$$\llbracket \kappa \vdash \text{parallelize } \{ l_0 : a_0, \dots, l_n : a_n \} \text{ waiting} :$$

$$\mathbf{Action } \{ l_0 : T_0, \dots, l_n : T_n \} \rrbracket \eta \quad =$$

$$\begin{aligned} & \text{parallelizeWaiting}_{\mathcal{D}(\{ l_0 : T_0, \dots, l_n : T_n \})} \{ \\ & \quad l_0 : \llbracket \kappa \vdash a_0 : \mathbf{Action } T_0 \rrbracket \eta, \\ & \quad \dots, \\ & \quad l_n : \llbracket \kappa \vdash a_n : \mathbf{Action } T_n \rrbracket \eta \\ & \} \end{aligned}$$



$$\begin{aligned}
& \llbracket \kappa \vdash \text{parallelize } (a_0, \dots, a_n) \text{ waiting :} \\
& \quad \text{Action } (T_0, \dots, T_n) \rrbracket \eta \quad = \\
& \quad \text{parallelizeWaiting}_{\mathcal{D}((T_0, \dots, T_n))} [ \\
& \quad \quad \llbracket \kappa \vdash a_0 : \text{Action } T_0 \rrbracket \eta, \\
& \quad \quad \dots, \\
& \quad \quad \llbracket \kappa \vdash a_n : \text{Action } T_n \rrbracket \eta \\
& \quad ] \\
& \llbracket \kappa \vdash \text{parallelize } a \text{ waiting : Action } (T [c]) \rrbracket \eta = \\
& \quad \text{parallelizeWaiting}_{\mathcal{D}(T[c])} (\llbracket \kappa \vdash a : (\text{Action } T) [c] \rrbracket \eta)
\end{aligned}$$

□

## B.4 Konkrete Syntax

Die textuelle Syntax von FSPL ist spezifiziert in dem Formalismus SDF (Syntax Definition Formalism) in der Version, die im ASF+SDF Meta Environment eingesetzt und in [BK02] beschrieben wird. SDF vereinigt Konzepte *kontextfreier Grammatiken* (die traditionell in Notationen wie EBNF [Wir77] beschrieben werden) zur Definition der *konkreten* Syntax und *Signatures* für die Definition der *abstrakten* Syntax zu einem durchgängigen Formalismus. Die Spezifikation umfaßt die lexikalische, grammatikalische (kontext-freie) und abstrakte Syntax.

SDF-Spezifikationen sind modular. Ein Modul besteht im allgemeinen aus folgenden Bestandteilen:

- Import anderer Module
- Deklaration von Sorten (der abstrakten Syntax)
- Regeln lexikalischer Syntax
- Regeln kontext-freier Syntax

In den Syntaxregeln spielen Sorten die Rolle, die Nichtterminalsymbole in Grammatiken spielen. An die Stelle von Produktionsregeln treten Deklarationen von Operatoren im Sinn der abstrakten Syntax. Für die Details des Formalismus sei auf [BK02] verwiesen.

Die Sprachspezifikation unterteilt sich in die Module `Layout` (Leerzeichen und Kommentare), `Identifiers` (Bezeichner und Typbezeichner), `Zahlen` (Kardinalzahlen und Literale für Ganzzahlen und Gleitkommazahlen), `Strings` (Zeichen und Zeichenketten), `Types` (Typsprache und Deklarationen) und `Expressions` (Ausdrücke).

### B.4.1 Layout

Die Sorte `LAYOUT` ist in SDF vordefiniert und repräsentiert sogenannten „White space“ (Leerräume), der vom Parser überlesen werden soll. Neben Leerzeichen, Tabulatorzeichen und Zeilenumbrüchen zur Code-Formatierung gehören auch Kommentare dazu. Die Kommentarnotation ist von C++ und Java übernommen.

```
module Layout
exports
  sorts Comment

  lexical syntax
    [\ \n\t] -> LAYOUT

    "//" ~[\n]* [\n] -> Comment
    "/*" ~[*] "*" -> Comment

  context-free syntax
    Comment -> LAYOUT

  context-free restrictions
    LAYOUT? -/- [\ \n\t]
    LAYOUT? -/- [\|/].[|/]
    LAYOUT? -/- [\|/].[|*]
```

### B.4.2 Bezeichner

Im Module `Identifiers` ist das Format von Bezeichnern und Typbezeichnern spezifiziert. Erlaubt sind alphanumerische Zeichen und Unterstriche, wobei Bezeichner stets mit einem Kleinbuchstaben und Typbezeichner mit einem Großbuchstaben beginnen müssen.

```
module Identifiers
imports Layout
exports
  sorts Id Type-Id
```

```
lexical syntax
[a-z][a-zA-Z0-9\_]* -> Id           {avoid}
[A-Z][a-zA-Z0-9\_]* -> Type-Id     {avoid}

context-free restrictions
Id      -/- [a-zA-Z0-9\_]\_
Type-Id -/- [a-zA-Z0-9\_]\_
```

### B.4.3 Zahlen

Kardinalzahlen können entweder als Dezimal- oder als Hexadezimalzahlen geschrieben werden. Die Hexadezimalschreibweise mit vorangestelltem `0x` ist von C übernommen. Ganzzahlen setzen sich aus einer Kardinalzahl (für den Wert) und einer Angabe des Bit-Formats zusammen. Gleitkommazahlen folgen der üblichen Notation aus C und vielen anderen Sprachen.

```
module Numbers
imports Layout
exports
  sorts Decimal-Literal Hex-Literal Cardinal
  sorts Integer-Format
  sorts Real-Literal

lexical syntax
[0] | ([1-9][0-9]*) -> Decimal-Literal
"0x" [0-9A-F]+      -> Hex-Literal
([0] | ([1-9][0-9]*) | "." [0-9]+ ([eE] [\+\-] ([0] | ([1-9][0-9]*)))?) -> Real-Literal

context-free restrictions
Decimal-Literal -/- [0-9]
Hex-Literal     -/- [0-9A-F]
Real-Literal    -/- [0-9]

context-free syntax
Decimal-Literal -> Cardinal
Hex-Literal     -> Cardinal

Cardinal ("u"|"U"|"s"|"S") -> Integer-Format
```

### B.4.4 Zeichenketten

Wie in C werden einzelne Zeichen mit Hochkommata, Zeichenketten mit Anführungsstrichen eingeschlossen. Die Kodierung der ein-

zernen Zeichen im Detail bleibt unterspezifiziert und wird in der prototypischen Implementierung von FSPL durch einen Interpreter (siehe Kapitel 7.5.4) von der Sprache Haskell übernommen. Die Haskell-Kodierung kann als exemplarische Vervollständigung der Spezifikation angesehen werden.

```
module Strings
imports Layout
exports
  sorts Character Character-Literal String-Literal

  lexical syntax
    ~[\'\"\\n] -> Character
    [\\][\\'] -> Character
    [\\][\\"] -> Character

    '\"' Character+ '\"' -> Character-Literal
    %% Escape combinations take more than one character

    \"'\" Character* \"'\" -> String-Literal
```

### B.4.5 Typen

Das Modul `Types` definiert die Typsprache sowie die Syntax von Deklarationen mit ihren Varianten.

```
module Types
imports Layout Identifiers Numbers
exports
  sorts Type
  sorts Pattern Labelled-Type
  sorts Declaration Type-Declaration Type-Definition

  context-free syntax
    Id -> Pattern
    Pattern "(" {Declaration ","}+ ")" -> Pattern
    Pattern "<" Type-Declaration ">" -> Pattern

  context-free syntax
    "type" Type-Id -> Type-Declaration

    Type-Declaration "(" (Type-Declaration ")" ) * "=" Type
    -> Type-Definition

    "value" Pattern ":" Type -> Declaration
    "process" Pattern ":" Type -> Declaration
```

```

"signal"    Pattern    ":" Type    -> Declaration
"variable"  Pattern    ":" Type    -> Declaration
"event"     Pattern    ":" Type    -> Declaration
"phase"     Pattern    ":" Type    -> Declaration
"action"    Pattern    ":" Type    -> Declaration

context-free syntax
"Boolean"   -> Type
"Integer"   Integer-Format -> Type
"Real"      -> Type
"Character" -> Type
"Process"   -> Type
"Event"     -> Type
"Time"      -> Type
"Phase"     -> Type
"Action"    -> Type

context-free syntax
"Boolean" -> Type-Id {reject}
"Integer" -> Type-Id {reject}
"Real"    -> Type-Id {reject}
"Process" -> Type-Id {reject}
"Event"   -> Type-Id {reject}
"Time"    -> Type-Id {reject}
"Phase"   -> Type-Id {reject}
"Action"  -> Type-Id {reject}

lexical syntax
"Integer" Integer-Format -> Type-Id {reject}

context-free syntax
Type-Id -> Type

"(" Type ")" -> Type {bracket}

%% Unlabelled product
%% (empty or of at least two types: tuple type)
"(" ")" | "(" Type "," {Type ","}+ ")" -> Type

Id ":" Type -> Labelled-Type

%% Labelled product (record type)
"{" {Labelled-Type ","}* "}" -> Type

%% Labelled sum (of at least one type: variant type)
{Labelled-Type "|" }+ -> Type

%% Array type
Type "[" Cardinal "]" -> Type

```

```

%% Function type
Type ">" Type -> Type {right}

%% Polymorphism
"<" Type-Declaration ">" ">" Type -> Type

%% Type functions
"(" Type-Declaration ")" Type -> Type
Type Type -> Type {left}

%% Modules
"interface" "{" Type-Declaration*
                Declaration*          "}" -> Type

%% Local type definitions
"lettype" Type-Definition+ "in" Type-> Type

context-free syntax
"interface" -> Id {reject}
"lettype"   -> Id {reject}
"in"        -> Id {reject}

context-free priorities
Type "[" Cardinal "]" -> Type >
Type Type -> Type >
{right:
  Type ">" Type -> Type
  "(" Type-Declaration      ")" Type -> Type
  "<" Type-Declaration      ">" ">" Type -> Type}

```

### B.4.6 Ausdrücke

Das Module `Expressions` spezifiziert den verbleibenden Teil der Syntax, der im wesentlichen aus der Syntax der Ausdrücke besteht. Ausgelagert wurden im wesentlichen die Behandlung der Literale mit nicht-trivialer lexikalischer Syntax (Module `Numbers` und `Strings`) und aller Arten von Bezeichnern (Modul `Identifiers`) sowie die Typsprache (Modul `Types`).

```

module Expressions
imports Layout Identifiers Numbers Strings Types

exports
  sorts Boolean-Literal Expression
  sorts Definition Evaluation-Definition

```

```

sorts Labelled-Expression
sorts Time-Unit Continuation Transition

%% Boolean literals

context-free syntax
  "true"  -> Boolean-Literal
  "false" -> Boolean-Literal

context-free syntax
  "true"  -> Id  {reject}
  "false" -> Id  {reject}

%% Time units

context-free syntax
  "h"    -> Time-Unit
  "min"  -> Time-Unit
  "s"    -> Time-Unit
  "ms"   -> Time-Unit

%% Definitions

context-free syntax
  "value"    Pattern ":" Type "=" Expression -> Definition
  "process"  Pattern ":" Type "=" Expression -> Definition
  "signal"   Pattern ":" Type "=" Expression -> Definition
  "variable" Pattern ":" Type "=" Expression -> Definition
  "event"    Pattern           "=" Expression -> Definition
  "phase"    Pattern ":" Type "=" Expression -> Definition
  "action"   Pattern ":" Type "=" Expression -> Definition
  Declaration           ":" "=" Expression -> Evaluation-Definition

%% Expressions

context-free syntax

  %% Constants
  Boolean-Literal      -> Expression
  Cardinal "as" Type    -> Expression
  Cardinal "#" Integer-Format -> Expression
  Real-Literal         -> Expression
  Character-Literal    -> Expression
  String-Literal       -> Expression

  %% Casts
  Expression "as" Type -> Expression

  %% Boolean operators

```

```

"!" Expression          -> Expression
Expression "&&" Expression -> Expression {left}
Expression "||" Expression -> Expression {left}

%% Comparison operators
Expression "<" Expression  -> Expression {non-assoc}
Expression "<=" Expression -> Expression {non-assoc}
Expression ">=" Expression -> Expression {non-assoc}
Expression ">" Expression  -> Expression {non-assoc}
Expression "==" Expression -> Expression {non-assoc}
Expression "!=" Expression -> Expression {non-assoc}

%% Arithmetic operators
Expression "-" Expression -> Expression
Expression "+" Expression -> Expression {left}
Expression "-" Expression -> Expression {left}
Expression "*" Expression -> Expression {left}
Expression "/" Expression -> Expression {non-assoc}
Expression "%" Expression -> Expression {non-assoc}

%% Bit manipulation
Expression "~" Expression -> Expression
Expression "&" Expression -> Expression {left}
Expression "|" Expression -> Expression {left}
Expression "^" Expression -> Expression {left}
Expression "<<" Cardinal   -> Expression
Expression ">>" Cardinal   -> Expression

%% Data structures (constructors and selectors)

Id ":" Expression -> Labelled-Expression

%% Tupel
("(", ")", " |
("(", Expression ", " {Expression ", " + "}") -> Expression
Expression "." Cardinal -> Expression

%% Record
"{ " {Labelled-Expression ", "}* "}" -> Expression
Expression "." Id -> Expression

%% Variant
Labelled-Expression "as" Type -> Expression
"case" Expression "of" Labelled-Expression+ -> Expression
"case" Expression "of" Labelled-Expression*
      "otherwise" Expression -> Expression

%% Array
 "[" {Expression ", " + "}" -> Expression

```



```

Expression "at" Expression "default" Expression -> Expression

Id -> Expression

"if" Expression "then" Expression "else" Expression -> Expression

 "(" Expression ")" -> Expression {bracket}

%% Lambda (functions): abstraction and application
 "(" {Declaration ","}+ ")" Expression -> Expression
Expression Expression -> Expression {left}

%% Polymorphism (templates): abstraction and application
 "<" Type-Declaration ">" Expression -> Expression
Expression "<" Type ">" -> Expression

%% Local definitions
"let" Definition+ "in" Expression -> Expression
"letrec" Definition+ "in" Expression -> Expression
"lettype" Type-Definition+ "in" Expression -> Expression

%% Modules
"module" "{" Type-Definition*
Definition* "}" -> Expression

%% Module imports
"import" Expression "in" Expression -> Expression

context-free syntax
%% Processes
"const" Expression -> Expression
"apply" Expression "to" Expression -> Expression
"time" -> Expression
"previous" Expression "initially" Expression -> Expression
"zip" Expression -> Expression
"unzip" Expression -> Expression

%% Time intervals
Cardinal Time-Unit -> Expression

%% Events
"never" -> Expression
"after" Expression -> Expression
"clock" Expression ("offset" Expression)? -> Expression
"trigger" Expression -> Expression
"no" Expression "within" Expression -> Expression

%% Phases
"keep" Expression -> Expression

```

```

"then" Expression                                -> Continuation
"then" "wait"                                    -> Continuation
"when" Expression Continuation                  -> Transition
"do" Expression Transition+                      -> Expression
"start" Expression                             -> Expression
"orthogonalize" Expression                     -> Expression
"local" Evaluation-Definition+ "in" Expression -> Expression

%% Actions
"do" Expression "until" Expression              -> Expression
"complete" Expression Continuation              -> Expression
  ("except" Transition+)?                      -> Expression
"loop" Expression                              -> Expression
Expression ";" Expression                      -> Expression {right}
"if" Expression "then" Expression
      "always" Expression                      -> Expression
"repeat" Expression "until" Expression          -> Expression
"while" Expression "repeat" Expression
      "finally" Expression                    -> Expression
"parallelize" Expression ("terminating" |
      "waiting")                              -> Expression

context-free syntax
"after"      -> Id {reject}
"always"     -> Id {reject}
"apply"      -> Id {reject}
"as"         -> Id {reject}
"at"         -> Id {reject}
"case"       -> Id {reject}
"clock"      -> Id {reject}
"const"      -> Id {reject}
"complete"   -> Id {reject}
"do"         -> Id {reject}
"default"    -> Id {reject}
"else"       -> Id {reject}
"except"     -> Id {reject}
"finally"    -> Id {reject}
"if"         -> Id {reject}
"import"     -> Id {reject}
"in"         -> Id {reject}
"keep"       -> Id {reject}
"let"        -> Id {reject}
"letrec"     -> Id {reject}
"lettype"    -> Id {reject}
"loop"       -> Id {reject}
"module"     -> Id {reject}
"never"      -> Id {reject}
"no"         -> Id {reject}

```

```

"of"           -> Id {reject}
"offset"       -> Id {reject}
"or"           -> Id {reject}
"parallelize"  -> Id {reject}
"repeat"       -> Id {reject}
"start"        -> Id {reject}
"then"         -> Id {reject}
"time"         -> Id {reject}
"to"           -> Id {reject}
"trigger"      -> Id {reject}
"until"        -> Id {reject}
"unzip"        -> Id {reject}
"wait"         -> Id {reject}
"waiting"      -> Id {reject}
"when"         -> Id {reject}
"while"        -> Id {reject}
"with"         -> Id {reject}
"within"       -> Id {reject}
"zip"          -> Id {reject}

```

context-free priorities

```

{Expression "." Id           -> Expression
 Expression "." Cardinal     -> Expression} >
{left:
 Expression Expression       -> Expression
 Expression "<" Type        ">" -> Expression} >
{"const" Expression         -> Expression
 "apply" Expression "to" Expression -> Expression
 "zip" Expression           -> Expression
 "unzip" Expression         -> Expression
 "after" Expression         -> Expression
 "clock" Expression ("offset" Expression)? -> Expression
 "trigger" Expression       -> Expression
 "no" Expression "within" Expression -> Expression
 "keep" Expression          -> Expression
 "loop" Expression          -> Expression
 "start" Expression         -> Expression
 "-" Expression             -> Expression
 "!" Expression             -> Expression
 "~" Expression             -> Expression} >
{left:
 Expression "*" Expression   -> Expression
 Expression "/" Expression   -> Expression
 Expression "%" Expression   -> Expression} >
{left:
 Expression "+" Expression   -> Expression
 Expression "-" Expression   -> Expression} >
{Expression "<<" Cardinal     -> Expression
 Expression ">>" Cardinal     -> Expression} >

```

```

{non-assoc:
  Expression "<" Expression -> Expression
  Expression ">" Expression -> Expression
  Expression "<=" Expression -> Expression
  Expression ">=" Expression -> Expression} >
{non-assoc:
  Expression "==" Expression -> Expression
  Expression "!=" Expression -> Expression} >
Expression "&" Expression -> Expression >
Expression "^" Expression -> Expression >
Expression "|" Expression -> Expression >
Expression "&&" Expression -> Expression >
Expression "||" Expression -> Expression >
Expression "as" Type -> Expression >
{"(" {Declaration ","}+ ")" Expression -> Expression
 "<" Type-Declaration ">" Expression -> Expression
 "let" Definition+ "in" Expression -> Expression
 "letrec" Definition+ "in" Expression -> Expression
 "lettype" Type-Definition+ "in" Expression -> Expression
 "local" Evaluation-Definition+ "in" Expression
                                     -> Expression
 "import" Expression "in" Expression -> Expression
 "previous" Expression "initially" Expression
                                     -> Expression
 "case" Expression "of" Labelled-Expression+ -> Expression
 "case" Expression "of" Labelled-Expression*
                                     "otherwise" Expression -> Expression
Expression "at" Expression
          "default" Expression -> Expression
 "if" Expression "then" Expression
          "else" Expression -> Expression
 "if" Expression "then" Expression
          "always" Expression -> Expression
 "do" Expression Transition+ -> Expression
 "do" Expression "until" Expression -> Expression
 "complete" Expression Continuation
          ("except" Transition+)? -> Expression
 "repeat" Expression "until" Expression -> Expression
 "while" Expression "repeat" Expression
          "finally" Expression -> Expression
 "parallelize" Expression ("terminating" |
                          "waiting") -> Expression} >
Expression ";" Expression -> Expression

```

## C. Programmbeispiel Kaffeemaschinensteuerung

Nachfolgend wird ein vollständiges Programmbeispiel angegeben, das sich an eine existierende Steuerungssoftware eines Kaffeeautomaten anlehnt, der in Hotels und in der Gastronomie eingesetzt wird. Des Umfangs halber wird nur ein Ausschnitt aus der Funktionalität der Kaffeemaschine betrachtet, nämlich die Reinigungsfunktion, bei der unter anderem ein Reinigungsmittel eingebracht und Spülungen der Brüheinheit und der kaffeeführenden Teile durchgeführt werden. Die auf diese Anwenderfunktion reduzierte Kaffeemaschinensteuerung umfaßt bereits wesentliche Aspekte der Betriebszustands- und Ablaufsteuerung und eine repräsentative Auswahl an beteiligter Maschinen-Hardware. So kann davon ausgegangen werden, daß die Gesamtarchitektur der Software bei Hochskalierung auf die volle Funktionalität ähnlich bleibt.

Die gleiche Anwendung diente als Demonstrator im IDESA-Projekt [IDE04] für eine Entwurfsmethodik auf Basis von ausführbaren UML-Modellen und Codegenerierung [FMG04, FSMG04]. Sie wurde als Anwendungsbeispiel freundlicherweise zur Verfügung gestellt von der Firma *TLON GmbH*, Schwäbisch-Hall, in Zusammenarbeit mit der *bremer Kaffeemaschinen GmbH*, Bad Mergentheim (jetzt: Franke Coffee Systems, Aarburg, CH). Die hier dargestellte Steuerung wandelt die Originalanwendung leicht ab; das tatsächliche Reinigungsverfahren ist dem hier beschriebenen ähnlich. In sehr knapper Form wurde die Anwendung bereits in [FSMG04] beschrieben.

## C.1 Überblick über die Anwendung

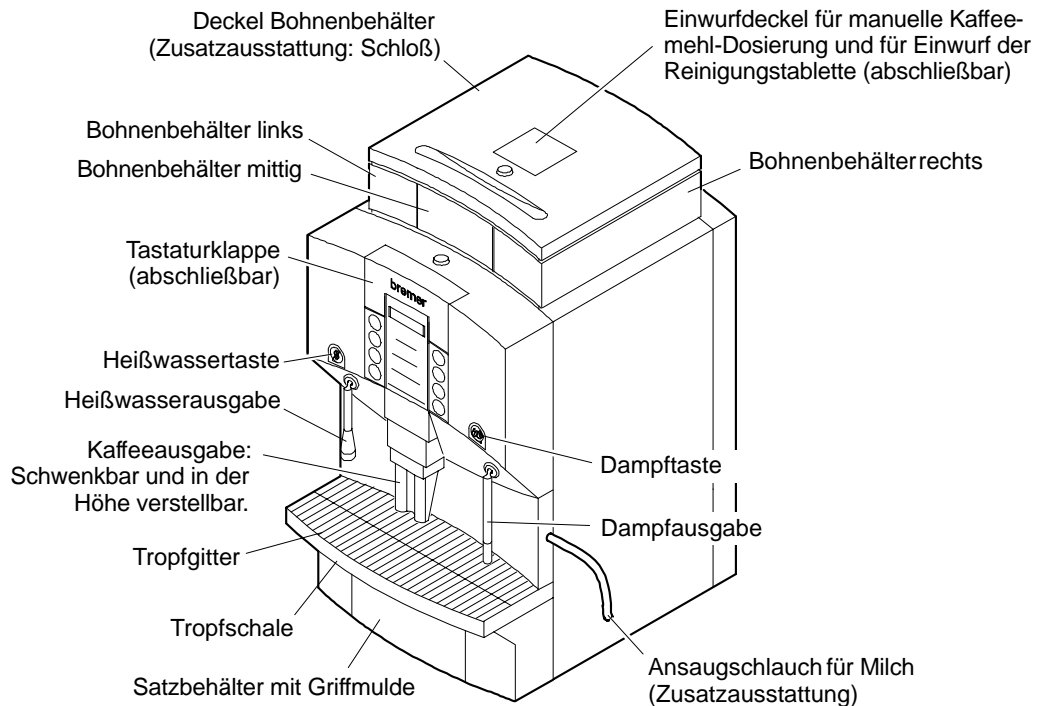


Abbildung C.1: Kaffeemaschine Viva  
(mit freundlicher Genehmigung von Franke Coffee Systems, Aarburg, CH)

Abbildung C.1 vermittelt einen groben Eindruck von der Funktionalität der Kaffeemaschine (des Typs *Viva*) aus Benutzersicht. Für die Bedienung der Reinigungsfunktion werden der Einwurfdeckel für den Einwurf der Reinigungstablette und zwei hinter der Tastaturklappe liegende Spezialtasten (in Abbildung C.2 dargestellt) verwendet. Mit der einen Taste wird die Reinigung gestartet, mit der anderen Taste bestätigt der Bediener den Einwurf der Reinigungstablette. Über ein zweizeiliges Text-Display fordert die Maschine zur Eingabe der Reinigungstablette auf und gibt Statusinformationen. Das Spülwasser läuft über die Kaffeeausgabe aus.

Die Maschine kann durch eine ebenfalls hinter der Tastaturklappe befindliche Standby-Taste ein- und ausgeschaltet werden, wobei das Ausschalten in einen Standby-Modus führt, bei der die meisten Stromverbraucher abgeschaltet sind, aber die Steuerung weiterläuft. Zum Neustart der Maschine muß sie vom Stromnetz getrennt werden. Beim Ausschalten in den Standby-Zustand wird eine laufende Reinigung abgebrochen. Abgebrochen wird eine Reinigung auch in bestimmten Feh-

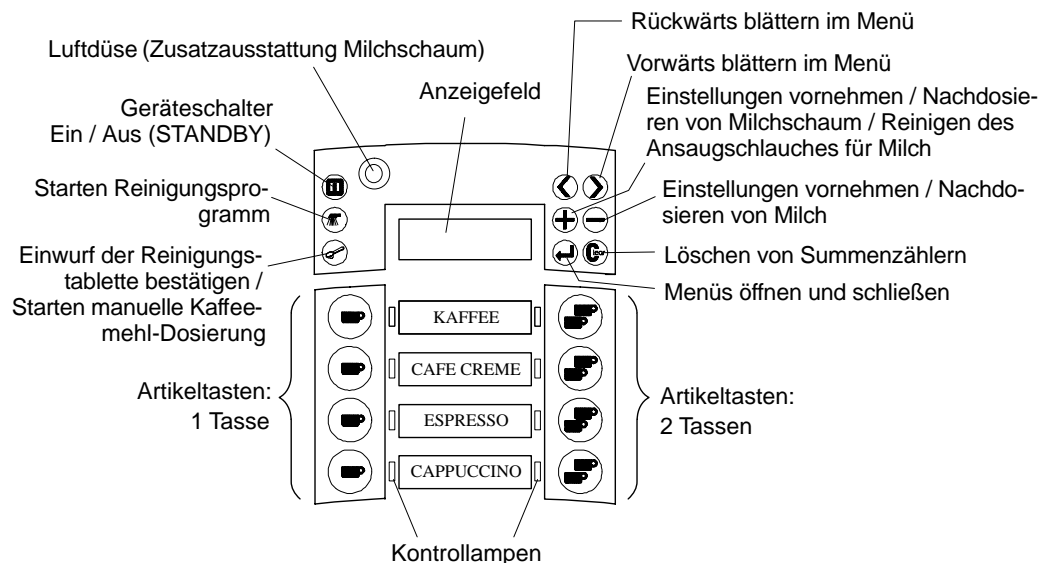


Abbildung C.2: Benutzerschnittstelle der Viva  
(mit freundlicher Genehmigung von Franke Coffee Systems, Aarburg, CH)

lersituationen, von denen exemplarisch das Herausziehen des Kaffeeausgabekopfes, das Entnehmen des Satzbehälters und eine unzureichende (z. B. unterbrochene) Frischwasserzufuhr<sup>1</sup> berücksichtigt werden. Wird die Reinigung abgebrochen, nachdem bereits die Reinigungstablette eingeworfen wurde, wird eine Wiederaufnahme der Reinigung erzwungen, bevor der Bereitschaftszustand zum Produzieren von Kaffeeprodukten erneut erreicht wird. Diese Absicherung, die auch dann wirken muß, wenn die Maschine zwischenzeitlich vom Stromnetz getrennt wird, macht eine persistente Speicherung von Zustandsinformation erforderlich; hierzu wird ein EEPROM<sup>2</sup> eingesetzt.

Die sogenannten Artikelasten, mit denen verschiedene Kaffeeprodukte angefordert werden können, sind außerhalb des Bereitschaftszustandes deaktiviert (d. h. sie werden ignoriert). Aktive Artikelasten werden durch je eine Kontrollampe (LED) angezeigt, die sich bei der jeweiligen Artikelaste befindet (siehe Abbildung C.2).

Bei der Reinigung selbst wird das Reinigungsmittel durch den Benutzer in Tablettenform über einen Eingabeschacht in die *Brüheinheit* geworfen und dort durch Wasserzufuhr aufgelöst. Die Brüheinheit besteht im Kern aus einer *Brühbuchse* und zwei Kolben, die mittels einer

<sup>1</sup>Die Maschine hat einen Wasserleitungsanschluß.

<sup>2</sup>Electronically Erasable Programmable Read-Only Memory

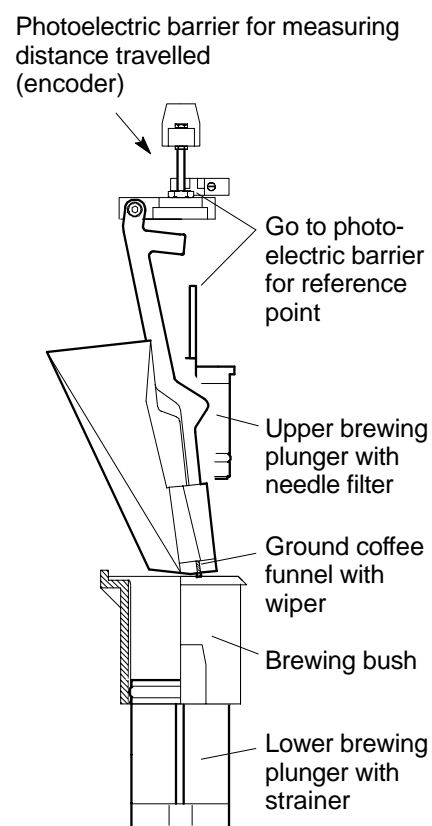


Abbildung C.3: Brühinheit

(mit freundlicher Genehmigung von Franke Coffee Systems, Aarburg, CH)



Spindel gegenüber der Buchse verschoben werden (siehe Abbildung C.3). Die Arbeitsweise dieser Konstruktion entspricht der einer Presse und ermöglicht das Verschließen der Brühbuchse zu einer *Brühkammer*, das Brühen unter Druck, das Entleeren der Brühkammer über den Ausgabekanal, das Pressen des Kaffeesatzes zu einer Tablette und das Befördern der Satztablette mit Hilfe eines Abstreifers in den Satzbehälter. Diese verschiedenen Funktionen werden allein durch das Verschieben der Kolben – mit unterschiedlicher Motorleistung (Geschwindigkeit, Anpressdruck) – betätigt. Für die Reinigung werden davon das Öffnen der Brühkammer zum Einwurf der Reinigungstablette, das Verschließen zum Spülen und das Auspressen (Entleeren) der Brühkammer vom Spülwasser benötigt. Die Position der Presse wird dabei aus der Rotation der Spindel, die bei bekannter Drehrichtung mittels einer Lochscheibe photoelektrisch detektiert wird, und einer Referenzposition, die ebenfalls mit einer Lichtschranke erkannt wird, errechnet.

Über ein Einlaßventil und eine zuschaltbare Druckpumpe kann Heißwasser mit geringem oder hohem Druck in die Brühkammer eingegeben bzw. eingespritzt werden. Die Menge wird mittels eines Durchflüssmessers bestimmt, der nach Art einer Turbine ein Schaufelrad enthält und die Umdrehungen des durch den Wasserfluß angetriebenen Schaufelrades mißt. Zum Auflösen der Reinigungstablette wird Wasser ohne die Druckpumpe dosiert; für die Spülungen wird Wasser mit Hilfe der Druckpumpe eingespritzt.

## C.2 Software-Architektur

Die Außenschnittstelle der Software zerfällt logisch in drei Teile: die Benutzerschnittstelle, die Schnittstelle zum verfahrenstechnischen Prozeß und die Datenbankschnittstelle, wobei hier an den nichtflüchtigen Speicher in Form des EEPROMs gedacht ist (siehe Abbildung C.4).<sup>3</sup>

Physikalisch besteht die Benutzerschnittstelle genauso wie der verfahrenstechnische Prozeß aus Geräten (Tastatur, Display; Druckpumpe, Ventil, Satzschublade, ...), von denen im allgemeinen Signale und Ereignisse als Eingabe in die Steuerung eingehen, und die Signale als Ausgabe aus der Steuerung erhalten. Dabei wird davon

<sup>3</sup>Im vollen Ausbau der Funktionalität kommt noch eine Netzwerkschnittstelle zur Fernbedienung oder zur Interaktion mit Abrechnungssystemen hinzu.

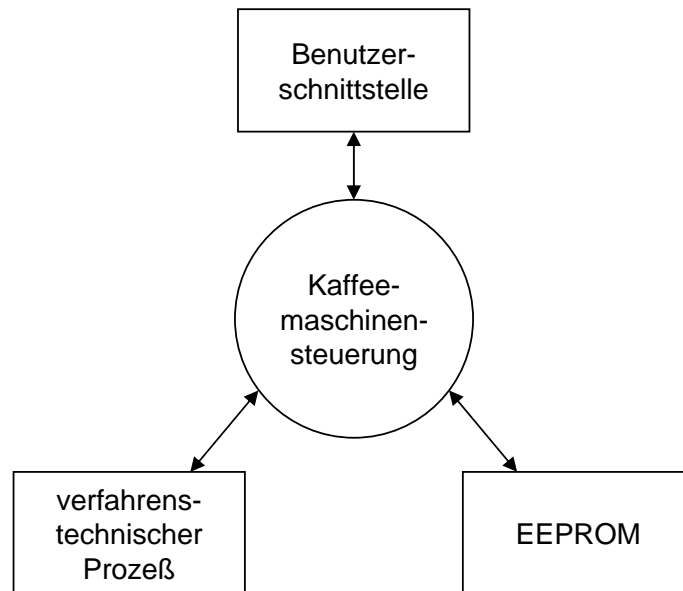


Abbildung C.4: Kontext-Diagramm

ausgegangen, daß das Laufzeitsystem von FSPL die Hardware-Ein-/Ausgabeschnittstellen geeignet als Signale und Ereignisse abstrahiert. Nichtflüchtiger Speicher wird aus Sicht eines FSPL-Programms ähnlich wie ein externes Gerät behandelt, von dem als Eingabe der Initialwert zum Startzeitpunkt eingeht (als Konstante) und das als Ausgabe den sich über der Zeit verändernden Inhalt des Speichers (als Signal) erhält.

Die Außensicht auf die Steuerung als FSPL-Programm ist dementsprechend die einer signalverarbeitenden Funktion, die Konstanten, Signale und Ereignisse als Eingabe auf Signale als Ausgabe abbildet. Genau einen solchen Typ besitzt das Hauptprogramm der Anwendung, das als Endergebnis in Abschnitt C.5.10 angegeben wird.

Die innere Architektur der Steuerung folgt jedoch nicht einer schrittweisen Verfeinerung dieser Systemfunktion im Sinne einer funktionalen Dekomposition, sondern leitet sie von der Schnittstelle einer abstrakten Maschine auf oberster Ebene einer Abstraktionshierarchie (siehe Abschnitt C.5.9) ab, die bei den konkreten Geräten des verfahrenstechnischen Prozesses und der Benutzerschnittstelle beginnt und sich bottom-up aufbaut (zur Methodik siehe Kapitel 8).

Die gesamte Anwendung zerfällt, wie in Abbildung C.5 in UML-Notation dargestellt, in eine Sammlung von Paketen (siehe Kapitel

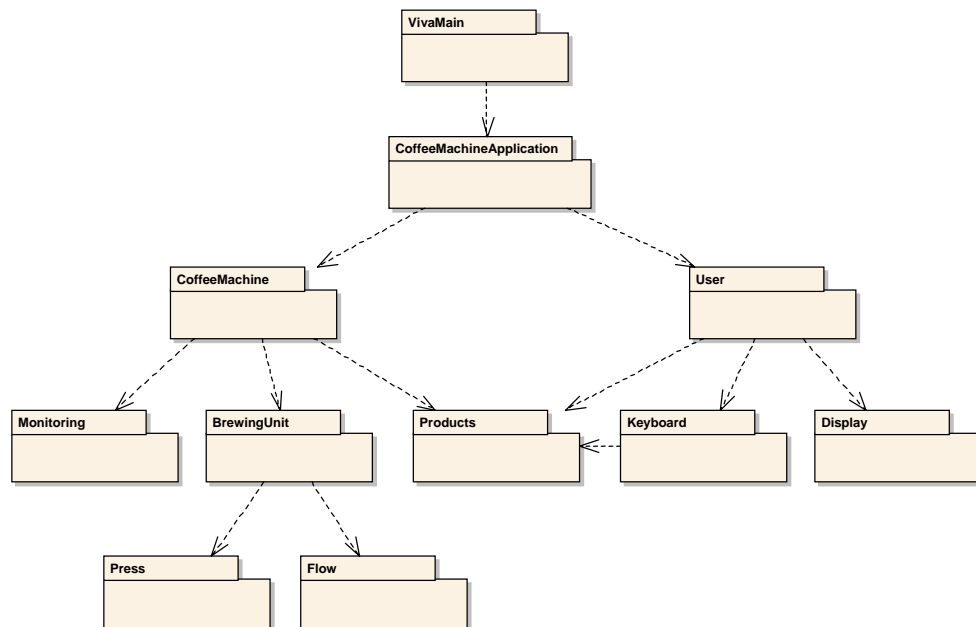


Abbildung C.5: Pakete und ihre Abhängigkeiten (in UML-Notation)

7.5.2), die sich zusätzlich einer (im Bild nicht dargestellten) Basisbibliothek bedienen. Das Paket `Monitoring` kann als Bestandteil einer domänenspezifischen Bibliothek aufgefaßt werden, da der Inhalt nicht von der speziellen Anwendung abhängt. Alle anderen Pakete aus Abbildung C.5 mit Ausnahme von `Products` stellen neben Datentyp-, Konstanten- und Funktionsdefinitionen jeweils die Schnittstelle und eine Implementierung einer abstrakten Maschine als Modul bereit. Die Abhängigkeitsstruktur der Pakete entspricht dabei der Hierarchie der abstrakten Maschinen. Im linken Ast wird der verfahrenstechnische Prozeß schrittweise zu einer abstrakten Kaffeemaschine abstrahiert, der vom rechten Ast ein abstrakter Benutzer gegenübersteht. Beide teilen das Wissen über eine Menge von Produkten (Paket `Products`), die dem Benutzer zur Auswahl stehen und von der Kaffeemaschine hergestellt werden. An der Spitze der Hierarchie steht die „Kaffeemaschinenanwendung“, die sich aus der Interaktion von Maschine und Benutzer ergibt. Aus diesem Modul wird das Hauptprogramm instanziiert.

## C.3 Basisbibliothek

Die nachfolgend beschriebenen zwei Pakete stehen exemplarisch für eine mehr oder weniger umfangreiche Basisbibliothek, die im Lieferumfang einer FSPL-Implementierung enthalten sein könnte.

### C.3.1 Standarddatentypen

FSPL bietet hinsichtlich der Bitbreite von Integer-Datentypen höchste Flexibilität. Dennoch kommt aufgrund von Hardware-Eigenschaften den Typen mit 8, 16 oder 32 Bit besondere Bedeutung zu; man wird sie häufig verwenden. Im Paket `StandardLibrary` sind daher Konstanten für die Grenzen der zugehörigen Wertebereiche zusammengefaßt.

Zur Verarbeitung von Ereignissen findet sich hier auch eine Hilfsfunktion, die an die Funktion `happened` aus Spezifikation 6.3 in Kapitel 6.3.3 angelehnt ist.

```
package StandardLibrary where

// Limits of integer value ranges

value maxInteger8U : Integer8U = 0xFF#8u      //      255
value maxInteger8S : Integer8S = 0x7F#8s      //      127
value minInteger8S : Integer8S = 0x80#8s      //     -128

value maxInteger16U : Integer16U = 0xFFFF#16u //     65535
value maxInteger16S : Integer16S = 0x7FFF#16s //     32767
value minInteger16S : Integer16S = 0x8000#16s //    -32768

value maxInteger32U : Integer32U = 0xFFFFFFFF#32u // 4294967295
value maxInteger32S : Integer32S = 0x7FFFFFFF#32s // 2147483647
value minInteger32S : Integer32S = 0x80000000#32s // -2147483648

// Functions on events

signal happened (event e) : Boolean
= start (
  do
    keep const false
  when e then
    keep const true
)
```

### C.3.2 Systemkonstanten

In diesem Paket ist die Auflösung der Systemzeit (spezifisch für eine bestimmte Zielplattform und Konfiguration) als Konstante abgelegt. Ungeachtet dessen kann sich die Anwendung auch auf einen (größeren) anwendungsspezifischen Zeittakt festlegen (vgl. Kapitel 7.3.2.1).

```
package SystemConfiguration where

// System clock resolution
value systemClockPeriod : Time = 1 ms
```

## C.4 Domänenbibliothek

Stellvertretend für eine Basisbibliothek für einen bestimmten Anwendungsbereich von Echtzeitsystemen steht ein Paket zum Thema Überwachungsfunktionen (zu weiteren möglichen Bibliotheksfunktionen siehe auch Kapitel 7.5.2).

### C.4.1 Überwachungsfunktionen

Das Paket definiert einen nicht-abstrakten, strukturierten Datentyp für Ausnahmen. Eine Ausnahme wird durch zwei Ereignisse, die Beginn und Ende der Ausnahme bedeuten, und ein boolesches Signal, das die Gültigkeit der Ausnahme anzeigt, repräsentiert. Konstruktoren für zwei Spezialfälle, eine vorübergehende Ausnahmebedingung und das Versagen als endgültige (nicht-endende) Ausnahme ergänzen die allgemeine Typdefinition.

```
package Monitoring where
include StandardLibrary

type Exception = {
  raised   : Event,
  cleared  : Event,
  holds    : Process Boolean
}

value exception (signal exceptionCondition : Boolean) : Exception = {
  raised   : trigger exceptionCondition,
  cleared  : trigger (!exceptionCondition),
  holds    : exceptionCondition
}
```

```

value failure (event failureEvent) : Exception = {
    raised  : failureEvent,
    cleared : never,
    holds   : happened failureEvent
}

```

## C.5 Implementierung der Anwendung

Der Quellcode der eigentlichen Kaffeemaschinensteuerung gliedert sich in die Pakete dieses Abschnitts, von denen mit Ausnahme von `Products` (Abschnitt C.5.4) und `VivaMain` (Abschnitt C.5.10) alle je eine abstrakte Maschine als Modul implementieren.

### C.5.1 Wasserzufluß

Das Paket `Flow` realisiert eine Wasserzuflußkontrolle auf Basis eines (Schalt-)Ventils und eines Durchflußmessers. Der Kern des Durchflußmessers ist ein Schaufelrad („Turbine“). Jede Umdrehung der Turbine zeigt an, daß eine bestimmte Wassermenge geflossen ist. Die verfahrenstechnischen Prozesse der Kaffeemaschine zählen Wassermengen in solchen „Wassereinheiten“.

Das Paket definiert einen Datentyp für Wassermengen (in den beschriebenen Wassereinheiten und daher ganzzahlig), eine Maschinenschnittstelle für eine abstrakte Maschine für den Wasserzufluß und eine Implementierung der abstrakten Maschine mit der gegebenen Sensorik und Aktorik.

Die Programmierschnittstelle der Maschine besteht aus einer Prozedur zum Abmessen einer Wassermenge, einer Nicht-Operation als Phase und einem Ereignis zur Überwachung auf mögliches Fehlverhalten der Hardware. In der Implementierung zählt die Prozedur zum Abmessen die Schaufelradumdrehungen als von außen kommende Ereignisse, während sie das Ventil geöffnet hält. Die Nicht-Operation hält das Ventil geschlossen. Das Ausbleiben eines Turbinenereignisses bei geöffnetem Ventil wird als Fehlverhalten detektiert, wobei der Timeout applizierbar ist (Parameter der Modulimplementierung).

```

package Flow where

// The core of the flow meter is a turbine. Every revolution of
// the turbine indicates that a certain amount of water has flown.

```

```

// Water is counted in such water units.
type FlowWaterAmount = Integer16U

type FlowModule(type Input)(type Output) =
interface {
  type Flow
  type FlowControl

  // Construction and I/O
  value makeFlow (process control : FlowControl) : Input -> Flow
  value flowOutput : Flow -> Output

  // Control
  action measureOff (value flow    : Flow,
                    value amount : FlowWaterAmount) : FlowControl
  phase noFlow (value flow : Flow) : FlowControl

  // Monitoring
  event flowFailure (value flow : Flow)
}

type FlowSensing    = Event // Flow meter turbine revolution
type ValveActuation = Process Boolean // Valve open

value flowMeterValveFlowModule (value flowFailureTimeout : Time)
: FlowModule(FlowSensing)(ValveActuation) =
let
  process open    : Boolean = const true
  process closed : Boolean = const false
in

module {
  type Flow = {
    waterUnit : FlowSensing,
    valve      : ValveActuation
  }
  type FlowControl = Boolean

  value makeFlow (process control : FlowControl)
                (event  flowMeterEventWaterUnit) : Flow
  = {
    waterUnit : flowMeterEventWaterUnit,
    valve      : control
  }

  signal flowOutput (value flow : Flow) : Boolean
  = flow.valve

  action measureOff(value flow    : Flow,

```

```

                                value amount : FlowWaterAmount) : FlowControl
= do
    keep open
  until
    amount * flow.waterUnit

phase noFlow(value flow : Flow) : FlowControl
= keep closed

event flowFailure(value flow : Flow)
= (no flow.waterUnit within flowFailureTimeout) %
  (previous (flow.valve == open) initially false)
}

```

### C.5.2 Presse

Der Begriff Presse ist eine Abstraktion für die Funktionsweise der Linearverschiebung der Kolben in der Brüheinheit durch einen Spindelmotor. Die Presse arbeitet vertikal. Durch Drehung der Spindel verschieben sich die Kolben. Ein optischer Sensor (Lichtschanke und Lochscheibe) detektiert die Drehung der Spindel als eine Sequenz von Ereignissen (analog zur Turbine des Durchflußmessers). Jedes Ereignis zeigt eine bestimmte Teilumdrehung und so eine bestimmte Bewegungsdistanz an. Weder die Drehrichtung, noch die absolute Position sind alleine daraus erkennbar. Die hier angegebene Implementierung nimmt vereinfachend an, das sich die Presse zum Startzeitpunkt in einer Referenzposition befindet.<sup>4</sup> Die aktuelle Bewegungsrichtung ist ein interner Zustand der Steuerung, kann aber auch aus der Ansteuerung des Motors zurückgelesen werden.

Die möglichen Positionen der Presse sind relativ zur Referenzposition als Nullpunkt in Distanzeinheiten abgezählt. Die Referenzposition befindet sich am oberen Ende und wird im Normalfall nicht überfahren. Dennoch wird als Datentyp ein vorzeichenbehafteter Integerdatentyp gewählt. Die anwendungsrelevanten Positionen befinden sich im positiven Bereich, wobei die Positionswerte in vertikaler Abwärtsrichtung numerisch aufsteigen. Diese Codierung hinsichtlich der Zählungsrichtung wird über die Funktionen `lower`, `higher`, `oneUp` und `oneDown` vor der Bewegungslogik der Presse verborgen. Der Datentyp `PressPosition` ist dennoch nicht abstrakt, da die angefahrenen Absolutpositionen als Zahlen angegeben werden.

<sup>4</sup>Im Original wird die Referenzposition durch einen zweiten optischen Sensor (Lichtschanke) detektiert und beim Hochlauf der Maschine aktiv angefahren.



Die Motorleistung (und damit die Bewegungsgeschwindigkeit bzw. der Anpress- oder Gegendruck der Presse) kann durch Pulsweitenmodulierung zwischen 0 und ihrem Maximum mit einer Auflösung von 8 Bit reguliert werden.

Der Befehlssatz der Presse unterscheidet das Anfahren einer absoluten Position (`moveTo`), die sich oberhalb oder unterhalb der aktuellen Position befinden kann, von dem Fahren in eine bestimmte Richtung auf unbestimmte Zeit (`move`). Im ersten Fall wird die Zielposition mit einer bestimmten Motorleistung (d. h. Geschwindigkeit) angefahren, und die Aktion terminiert. Falls die Presse wegen eines Widerstandes vor der Zielposition zum Halten kommt tritt das Ereignis `stalled` auf. Im zweiten Fall kann das steuernde Programm die Bewegung der Presse mittels des Signals `position` und/oder des Ereignisses `stalled` beobachten, wobei beabsichtigt sein kann, daß sich aufgrund eines Gegendrucks die Presse nicht bewegt. Vereinfachend gesagt dient `moveTo` zum Bewegen der Kolben und `move` zum Pressen. Der Timeout für `stalled` kann frei appliziert werden. Die steuernde Anwendung ist für das rechtzeitige Abschalten des Motors (Phase `noMotion`) oder ein Zurückfahren bei Blockade zuständig.

```
package Press where
include SystemConfiguration
include StandardLibrary

// The press operates vertically. An sensor reports an event every
// revolution of the axle which means that the press has moved one
// unit of distance in the direction in which it is currently
// moving. Therefore, the press positions are modelled as a
// discrete set enumerated from zero onwards, where the top
// position is the reference position (zero) and a delta of one
// corresponds to a distance unit.
type PressPosition = Integer16S
value referencePosition : PressPosition = 0 as PressPosition

value lower (value x : PressPosition)
            (value y : PressPosition) : Boolean
= x > y

value higher (value x : PressPosition)
            (value y : PressPosition) : Boolean
= x < y

value oneUp (value position : PressPosition) : PressPosition
= position - (1 as PressPosition)
```

```

value oneDown (value position : PressPosition) : PressPosition
= position + (1 as PressPosition)

type VerticalDirection = Boolean
value up    : VerticalDirection = false
value down  : VerticalDirection = true

// The power of the press motor can be scaled down from maximum
// to zero with a resolution of 8 bit, effecting a corresponding
// speed or force.
type MotorPower = Integer8U
value fullPower : MotorPower = maxInteger8U

type PressModule(type Input)(type Output) =
interface {
  type Press
  type PressControl

  // Construction and I/O
  value makePress (process control : PressControl) : Input -> Press
  value pressOutput : Press -> Output

  // Control
  action moveTo (value press          : Press,
                 value targetPosition : PressPosition,
                 value power          : MotorPower) : PressControl
  phase move (value press          : Press,
              value dir            : VerticalDirection,
              value power          : MotorPower) : PressControl
  phase noMotion (value press          : Press)      : PressControl

  // Monitoring
  signal position (value press : Press) : PressPosition
  event stalled (value press : Press)

}

// The motion sensor reporting an event every distance unit
type MotionSensing = Event

type MotorActuation = {
  power    : Process MotorPower,
  forward  : Process Boolean
}

// The press tracks position through the motion sensor reporting
// an event every distance unit, the knowledge about the current
// motion direction, and the reference position as the starting
// point at time zero.

```

```

value pressModule (value stallDetectTimeout : Time)
: PressModule(MotionSensing) (MotorActuation)
= module {
  // The motor operates in forward direction when the press is
  // moved down.
  type Press = {
    motion    : MotionSensing,
    position  : Process PressPosition,
    motor     : MotorActuation
  }
  type PressControl = {
    motorPower : MotorPower,
    direction  : VerticalDirection
  }

  value makePress (signal control : PressControl)
                  (event motionSensorEventDistanceUnit) : Press =
  let
    process direction : VerticalDirection
    = (unzip control).direction

    process motorPower : MotorPower
    = (unzip control).motorPower
  in
  {
    motion    : motionSensorEventDistanceUnit,
    position  : (
      letrec
        phase position (value p0 : PressPosition) : PressPosition
        = do
          keep const p0
          when motionSensorEventDistanceUnit then
            if (previous direction initially down) == down then
              position (oneDown p0)
            else
              position (oneUp p0)
        in
        start (position referencePosition)
      ),
    motor     : {
      power    : motorPower,
      forward  : direction == down
    }
  }

  value pressOutput (value press : Press) : MotorActuation
  = press.motor

```

```

action moveTo(value press          : Press,
              value targetPosition : PressPosition,
              value power          : MotorPower) : PressControl
= if press.position != targetPosition then
  local
    value currentPosition : PressPosition := press.position
  in let
    value direction : VerticalDirection
    = if (higher currentPosition targetPosition) then
      down
    else
      up
  in
    do keep const {
      motorPower : power,
      direction  : direction
    }
    until trigger (press.position == targetPosition)
  else
    do keep const {
      motorPower : 0 as MotorPower,
      direction  : down
    }
    until after systemClockPeriod
    // dummy (action must not be empty)

phase move (value press : Press,
           value dir    : VerticalDirection,
           value power  : MotorPower) : PressControl
= keep const {
  motorPower : power,
  direction  : dir
}

phase noMotion (value press : Press) : PressControl
= keep const {
  motorPower : 0 as MotorPower,
  direction  : down
}

signal position (value press : Press) : PressPosition
= press.position

event stalled (value press : Press)
= let
  signal motorOn : Boolean
  = previous (press.motor.power != 0 as MotorPower)
  initially false
in

```

```
(no press.motion within stallDetectTimeout) % motorOn  
}
```

### C.5.3 Brühgruppe

In die durch die Presse gebildete Brühbuchse führt von unten her ein Wassereinlaß. Die Zuleitung führt durch eine Druckpumpe, die ein- oder ausgeschaltet werden kann. So kann entweder mit dem normalen Druck der Frischwasserleitung oder mit erhöhtem Druck Wasser in die Brühkammer eingespritzt werden. Der Wassereinlaß wird durch ein Schaltventil geöffnet bzw. geschlossen wird. Die Zuleitung führt zudem durch einen Durchflußmesser, um ein hinreichend genaues Dosieren der Wassermenge zu ermöglichen.

Die Kombination aus Schaltventil und Durchflußmesser zum Dosieren von Flüssigkeit wurde zur abstrakten Maschine `Flow` abstrahiert (siehe Abschnitt C.5.1). Die abstrakte Maschine „Brühgruppe“ entsteht nun aus dem Zusammenwirken von Presse, Flußkontrolle und Druckpumpe, um abstraktere verfahrenstechnische Operationen auszuführen:

- Dosieren einer bestimmten Wassermenge in die Brühkammer mit normalem oder hohem Druck (`dose` bzw. `inject`)
- Fahren der Presse an eine bestimmte Position (mit maximaler Geschwindigkeit, `movePressTo`)
- Fahren der Presse gegen den Boden (Anpressen, `movePressToPhysicalLimit`)

Die einzelnen Operationen der Flußkontrolle und der Presse werden nicht direkt, sondern nur in ähnlicher Form weitergegeben, da es zwischen ihnen physikalische Wechselwirkungen gibt. Der von unten kommende Wasserstrahl beim Dosieren von Wasser übt bei eingeschalteter Druckpumpe eine so große Kraft auf einen Kolben der Presse aus, daß er die ganze Anordnung nach oben schieben würde. Daher wird in diesem Fall der Motor mit schwacher Leistung eingeschaltet, um die Presse mit schwacher Gegenkraft nach unten in ihrer Position zu halten.

Die verfahrenstechnische Anwendung der Bewegungsoperationen der Presse in der Brühgruppe beschränkt sich auf das Anfahren bestimmter Positionen, um die Brühkammer zum Einfüllen von Kaffeemehl

oder der Reinigungstablette zu öffnen, zum Brühen zu verschließen usw., und auf das Anpressen gegen den Boden bis zum physikalischen Stillstand, um z. B. eine Satztablette zu formen. Die Presse darf dabei jeweils mit maximaler Geschwindigkeit fahren. Im ersten Fall hat die Presse freie Fahrt, und das Ende der Aktion wird durch das Erreichen der angegebenen Zielposition erkannt. Im zweiten Fall wird die Zielposition am Stillstand der Presse erkannt; je nach Inhalt der Brühkammer kann sie an unterschiedlichen Absolutpositionen liegen. Für beide Anwendungen wird von außen her keine Motorleistung mehr eingestellt.

Ein Fehler bei der Wasserzufuhr wird von `Flow` zu `BrewingUnit` weitergereicht. Die Stillstandserkennung der Presse wird nicht weitergegeben. Eine Überwachung der Presse auf mögliche Blockaden während einer Bewegung ist nur über das Positionssignal möglich.<sup>5</sup> Ein solcher Fehlerfall wird im Beispiel nicht weiter betrachtet.

```
package BrewingUnit where
include Press
include Flow

type BrewingUnitModule (type Input) (type Output) =
interface {
  type BrewingUnit
  type BrewingUnitControl

  // Construction and I/O
  value makeBrewingUnit (process control : BrewingUnitControl)
    : Input -> BrewingUnit

  value brewingUnitOutput : BrewingUnit -> Output

  // Control
  action doseWater (value brewingUnit : BrewingUnit,
                    value amount      : FlowWaterAmount)
    : BrewingUnitControl

  action injectWater (value brewingUnit : BrewingUnit,
                     value amount      : FlowWaterAmount)
    : BrewingUnitControl
```

---

<sup>5</sup>Da die Timeouts der Stillstandserkennung in der Implementierung der Presse und in einer möglichen Überwachung der Bewegungsoperationen durch einen Anwender der Brühgruppe aufeinander abgestimmt werden müssen, liegt hier wahrscheinlich eine Design-Schwäche vor. Besser würde ein ungeplanter Stillstand der Presse als Ausnahmeereignis ausgegeben.

```

    action movePressTo (value brewingUnit : BrewingUnit,
                        value position      : PressPosition)
    : BrewingUnitControl

    action movePressToPhysicalLimit (value brewingUnit : BrewingUnit)
    : BrewingUnitControl

    phase noBrewingUnitActivity (value brewingUnit : BrewingUnit)
    : BrewingUnitControl

    // Monitoring
    event waterFailure (value brewingUnit : BrewingUnit)
    signal pressPosition (value brewingUnit : BrewingUnit) : PressPosition
}

type PumpActuation = Process Boolean

type BrewingUnitInput (type PressInput) (type FlowInput) = {
    press : PressInput,
    flow  : FlowInput
}

type BrewingUnitOutput (type PressOutput) (type FlowOutput) = {
    press : PressOutput,
    flow  : FlowOutput,
    pump  : PumpActuation
}

value brewingUnitModule
  <type PressInput><type PressOutput>
  (value pressModule : PressModule (PressInput) (PressOutput))
  <type FlowInput><type FlowOutput>
  (value flowModule : FlowModule (FlowInput) (FlowOutput))
  (value directionToPhysicalLimit : VerticalDirection,
   value directionTowardsFlow      : VerticalDirection,
   value injectionResistancePower : MotorPower)
: BrewingUnitModule (BrewingUnitInput (PressInput) (FlowInput))
                    (BrewingUnitOutput (PressOutput) (FlowOutput))
=
import pressModule in
import flowModule in

let
  process on  : Boolean = const true
  process off : Boolean = const false
in

module {
  type BrewingUnit = {

```

```

    press : Press,
    flow  : Flow,
    pump  : PumpActuation
  }
  type BrewingUnitControl = {
    press : PressControl,
    flow  : FlowControl,
    pump  : Boolean
  }

  // Construction and I/O
  value makeBrewingUnit
    (process control : BrewingUnitControl)
    (value  input    : BrewingUnitInput (PressInput)
                                           (FlowInput))

  : BrewingUnit =
  {
    press : makePress (unzip control).press input.press,
    flow  : makeFlow (unzip control).flow input.flow,
    pump  : (unzip control).pump
  }

  value brewingUnitOutput (value brewingUnit : BrewingUnit)
  : BrewingUnitOutput (PressOutput) (FlowOutput) =
  {
    press : pressOutput brewingUnit.press,
    flow  : flowOutput  brewingUnit.flow,
    pump  : brewingUnit.pump
  }

  // Control
  action doseWater (value brewingUnit : BrewingUnit,
                    value amount      : FlowWaterAmount)
  : BrewingUnitControl
  = parallelize {
    press : do noMotion (brewingUnit.press) until never,
    flow  : measureOff (brewingUnit.flow, amount),
    pump  : do keep off until never
  }
  terminating

  action injectWater (value brewingUnit : BrewingUnit,
                     value amount      : FlowWaterAmount)
  : BrewingUnitControl
  = parallelize {
    press : do
      move (brewingUnit.press,
            directionTowardsFlow,
            injectionResistancePower)

```



```

        until never,
        flow  : measureOff (brewingUnit.flow, amount),
        pump  : do keep on until never
    }
    terminating

action movePressTo (value brewingUnit : BrewingUnit,
                    value position     : PressPosition)
: BrewingUnitControl
= parallelize {
    press : moveTo (brewingUnit.press, position, fullPower),
    flow  : do noFlow (brewingUnit.flow) until never,
    pump  : do keep off until never
}
terminating

action movePressToPhysicalLimit (value brewingUnit : BrewingUnit)
: BrewingUnitControl
= do
    orthogonalize {
        press : move (brewingUnit.press,
                     directionToPhysicalLimit,
                     fullPower),
        flow  : noFlow (brewingUnit.flow),
        pump  : keep off
    }
    until
        stalled (brewingUnit.press)

phase noBrewingUnitActivity (value brewingUnit : BrewingUnit)
: BrewingUnitControl
= orthogonalize {
    press : noMotion (brewingUnit.press),
    flow  : noFlow (brewingUnit.flow),
    pump  : keep off
}

// Monitoring
event waterFailure (value brewingUnit : BrewingUnit)
= flowFailure (brewingUnit.flow)

signal pressPosition (value brewingUnit : BrewingUnit) : PressPosition
= position (brewingUnit.press)
}

```

### C.5.4 Kaffeeprodukte

Die Kaffeemaschine erlaubt die Herstellung unterschiedlicher Kaffeeprodukte, die dem Benutzer zur Auswahl angeboten werden. Um

sowohl ein einzelnes Produkt, als auch eine Teilmenge aller möglichen Produkte spezifizieren zu können, werden zwei aufeinander abgestimmte Datentypen `Product` und `Products` definiert, wobei jedem Produkt ein Bit in einer Ganzzahl zugeordnet wird. Jedes Produkt ist eine Konstante mit genau einem gesetzten Bit. Eine Menge von Produkten ist durch die gesetzten Bits einer Ganzzahl bestimmt.

```
package Products where

type Product = Integer8U

value noProduct      : Product = 0x00#8u
value cafe            : Product = 1#8u << 0#8u
value cafeCreame      : Product = 1#8u << 1#8u
value espresso        : Product = 1#8u << 2#8u
value cappuccino       : Product = 1#8u << 3#8u
value cafe2x           : Product = 1#8u << 4#8u
value cafeCreame2x     : Product = 1#8u << 5#8u
value espresso2x       : Product = 1#8u << 6#8u
value cappuccino2x     : Product = 1#8u << 7#8u

type Products = Product // Multiple bits set

value allProducts : Products = 0xFF#8u
```

### C.5.5 Kaffeemaschine

Die nächsthöhere abstrakte Maschine nach der Brüheinheit ist die gesamte Kaffeemaschine, wobei an den verfahrenstechnischen Teil gedacht ist (d.h. ohne die Benutzerschnittstelle). Die betrachtete Teilfunktionalität (Reinigung) wird im wesentlichen mit der Brüheinheit dargestellt. Im vollen Ausbau der Funktionalität kommen das Mahlwerk und andere Gerätschaften hinzu, die verfahrenstechnisch zusammenwirken.

Exemplarisch für mögliche Fehlerfälle während der Reinigung oder der Herstellung von Kaffeeprodukten wird das Herausziehen des Satzbehälters oder des Ausgabekopfes (durch den Benutzer) betrachtet. Daß sich Satzbehälter und Ausgabekopf an ihrer Position befinden, wird durch je einen mechanischen Schalter (boolesches Signal) erkannt.

Der Befehlssatz der Kaffeemaschine besteht aus einer Operation `produce` zum Herstellen eines Kaffeeprodukts, das durch einen Parameter der Aktion spezifiziert wird, und Operationen zur Durchführung

der Reinigung. Die Reinigung zerfällt in drei Teile: das Dosieren des Reinigungsmittels, den Spülvorgang und eine Neutralisierung. Aus Sicherheitsgründen ist es relevant zu unterscheiden, ob sich Reinigungsmittel in der Brühkammer befindet oder nicht. Dies ist ein Zustand, der sich während einer vollständigen Reinigung zweimal ändert (während der Dosierung und mit Abschluß der Spülungen). Wird eine Reinigung vorzeitig abgebrochen und ist die Dosierung bereits erfolgt, muß für eine vollständige Spülung gesorgt werden, ohne daß erneut Reinigungsmittel dosiert wird. Diese beiden Teile der Reinigung treten daher als eigenständige Operationen (`doseDetergent` und `rinse`) auf. Im Anschluß an die Spülung wird noch eine „Neutralierung“ durchgeführt, indem ein Kaffee gebrüht wird und in den Ausguß läuft. Dieser dritte Teil der Reinigung ist eine Anwendung von `produce`.

Die Aktion `doseDetergent` ist interaktiv: Zunächst muß die Presse eine bestimmte Position anfahren, um die Brühbuchse für das Einbringen der Reinigungstablette zu öffnen. Anschließend wird der Benutzer zum Einwerfen der Tablette aufgefordert (die abstrakte Kaffeemaschine gibt ein Ereignis `detergentRequest` aus), was wiederum der Benutzer durch Drücken einer Taste bestätigt (die abstrakte Kaffeemaschine erhält das Ereignis `detergentInsertion` als Parameter zur Aktion `doseDetergent`). Anschließend wird die Reinigungstablette durch Zugabe von Wasser aufgelöst.

Die drei betrachteten möglichen Fehlerfälle (Herausziehen der Satzschublade, Abziehen des Ausgabekopfes und nicht ausreichende Wasserzufuhr) werden als Ausnahmesituationen im Sinn des im Paket `Monitoring` definierten Datentyps `Exception` (siehe Abschnitt C.4.1) gemeldet.

Die Implementierung der abstrakten Maschine kapselt das zentrale verfahrenstechnische Know-How. Drei auszumessende Positionen der Presse sind verfahrenstechnisch relevant und als Parameter des Moduls applizierbar. Applizierbar sind weiterhin diverse Parameter der Reinigungsprozedur wie Wassermengen, Anzahlen von Durchläufen und Wartezeiten. Die Kaffeeproduktion ist hier nur als Dummy implementiert. Die Dosierung erfolgt interaktiv wie oben beschrieben. Der Spülvorgang besteht aus zwei Zyklen von Spülungen. Im ersten Zyklus werden die Spülungen durch Wartezeiten unterbrochen, abschließend erfolgt ein „Auspressen“. Im zweiten Zyklus wechseln sich Spülung und Auspressen ab.

Die Realisierung des abstrakten Typs zur Ansteuerung der Maschine zerfällt in die Ansteuerung der Brühgruppe und die „Ansteuerung“ der Aufforderung zur Reinigungsmittelzugabe. Letztere Aufforderung ist ein Ereignis, das neben dem Verhalten der Brüheinheit und den betrachteten Ausnahmesituationen das Verhalten der Kaffeemaschine ausmacht (siehe die Implementierung des Typs `CoffeeMachine`). Die „Ansteuerung“ dieses Ereignisses ist ein boolesches Signal, auf dessen positive Flanken das Ereignis triggert (siehe die Implementierung des Typs `CoffeeMaschineControl` und des Konstruktors `makeCoffeeMachine`).

```
package CoffeeMachine where
include BrewingUnit
include Products
include Monitoring
include SystemConfiguration

type CoffeeMachineModule (type Input) (type Output) =
interface {
  type CoffeeMachine
  type CoffeeMachineControl

  value makeCoffeeMachine (process control : CoffeeMachineControl)
    : Input -> CoffeeMachine

  value coffeeMachineOutput : CoffeeMachine -> Output

  action produce (value machine : CoffeeMachine,
                  value product : Product) : CoffeeMachineControl

  /* Detergent dosing protocol:
     An execution of doseDetergent raises an event detergentRequest
     and awaits a following event detergentInsertion to proceed.
     Rationale: Detergent dosing is a user supported procedure
     where on request the user must insert a cleaning tablet and
     confirm this insertion.
  */
  action doseDetergent (value machine : CoffeeMachine,
                        event detergentInsertion)
    : CoffeeMachineControl

  event detergentRequest (value machine : CoffeeMachine)

  action rinse (value machine : CoffeeMachine) : CoffeeMachineControl

  phase noCoffeeMachineActivity (value machine : CoffeeMachine)
    : CoffeeMachineControl
```

```

    value missingGroundsContainer (value machine : CoffeeMachine) : Exception
    value missingDispenserBlock   (value machine : CoffeeMachine) : Exception
    value waterSupplyFailure       (value machine : CoffeeMachine) : Exception
}

type GroundsContainerInput = Process Boolean // grounds container in place
type DispenserBlockInput   = Process Boolean // dispenser block in place

type CoffeeMachineInput (type BrewingUnitInput) = {
    brewingUnit      : BrewingUnitInput,
    groundsContainer : GroundsContainerInput,
    dispenserBlock   : DispenserBlockInput
}
type CoffeeMachineOutput (type BrewingUnitOutput) = {
    brewingUnit : BrewingUnitOutput
}

value coffeeMachineModule
  <type BrewingUnitInput><type BrewingUnitOutput>
  (value brewingUnitModule : BrewingUnitModule(BrewingUnitInput)
                                     (BrewingUnitOutput))

  (value fillingPosition      : PressPosition,
   value closingPosition      : PressPosition,
   value bottomPosition       : PressPosition)
  (value detergentDissolutionWater : FlowWaterAmount,
   value rinsingWater           : FlowWaterAmount)
  (value rinsingCyclesWithWaiting : Integer16U,
   value rinsingCyclesWithPressOut : Integer16U)
  (value detergentResidenceTime : Time,
   value rinsingWaitingTime     : Time)
: CoffeeMachineModule (CoffeeMachineInput (BrewingUnitInput))
                      (CoffeeMachineOutput (BrewingUnitOutput))
=

import
  brewingUnitModule
in

module {
  // In a more complete CoffeeMachine, a grinding unit and other
  // devices would join the brewing unit
  type CoffeeMachine = {
    brewingUnit      : BrewingUnit,
    detergentRequest : Event,
    missingGroundsContainer : Exception,
    missingDispenserBlock   : Exception,
    waterSupplyFailure       : Exception
  }
}

```

```

type CoffeeMachineControl = {
  brewingUnit          : BrewingUnitControl,
  detergentRequest     : Boolean
}

value makeCoffeeMachine (process control : CoffeeMachineControl)
                        (value input    : CoffeeMachineInput
                          (BrewingUnitInput))

: CoffeeMachine =
let
  value brewingUnitInput : BrewingUnitInput
  = input.brewingUnit

  process groundsContainerInPlace : Boolean
  = input.groundsContainer

  process dispenserBlockInPlace : Boolean
  = input.dispenserBlock
in let
  value brewingUnit : BrewingUnit
  = makeBrewingUnit (unzip control).brewingUnit brewingUnitInput

  event detergentRequest
  = trigger (unzip control).detergentRequest
in
{
  brewingUnit          : brewingUnit,
  detergentRequest     : detergentRequest,
  missingGroundsContainer : exception (!groundsContainerInPlace),
  missingDispenserBlock  : exception (!dispenserBlockInPlace),
  waterSupplyFailure     : failure (waterFailure brewingUnit)
}

value coffeeMachineOutput (value machine : CoffeeMachine)
: CoffeeMachineOutput (BrewingUnitOutput)
= {
  brewingUnit : brewingUnitOutput (machine.brewingUnit)
}

action produce (value machine : CoffeeMachine,
                value product : Product)
: CoffeeMachineControl =
(
  // Dummy implementation
  do
    noCoffeeMachineActivity (machine)
  until (after systemClockPeriod)
)

```

```

action doseDetergent (value machine : CoffeeMachine,
                      event detergentInsertion)
: CoffeeMachineControl =
(
  parallelize {
    brewingUnit      : movePressTo (machine.brewingUnit,
                                    fillingPosition),
    detergentRequest : do keep const false until never
  }
  terminating;

  do orthogonalize {
    brewingUnit      : noBrewingUnitActivity (machine.brewingUnit),
    detergentRequest : keep const true
  }
  until detergentInsertion;

  parallelize {
    brewingUnit      : (
      movePressTo (machine.brewingUnit, closingPosition);
      doseWater    (machine.brewingUnit, detergentDissolutionWater);
      do
        noBrewingUnitActivity (machine.brewingUnit)
      until (after detergentResidenceTime)
    ),
    detergentRequest : do keep const false until never
  }
  terminating
)

event detergentRequest (value machine : CoffeeMachine)
= machine.detergentRequest

action rinse (value machine : CoffeeMachine) : CoffeeMachineControl
= (
  parallelize {
    brewingUnit : (
      let
        action pressOut : BrewingUnitControl = (
          movePressToPhysicalLimit (machine.brewingUnit);
          movePressTo (machine.brewingUnit, closingPosition)
        )
      in (
        // First rinsing cycle
        rinsingCyclesWithWaiting * (
          injectWater (machine.brewingUnit, rinsingWater);
          do
            noBrewingUnitActivity (machine.brewingUnit)
          until (after rinsingWaitingTime)
        )
      )
    )
  }
)

```

```

        );
        pressOut;

        // Second rinsing cycle
        rinsingCyclesWithPressOut * (
            injectWater (machine.brewingUnit, rinsingWater);
            pressOut
        )
    ),
    detergentRequest :
        do keep const false until never
    }
    terminating
)

phase noCoffeeMachineActivity (value machine : CoffeeMachine)
: CoffeeMachineControl =
(
    orthogonalize {
        brewingUnit      : noBrewingUnitActivity (machine.brewingUnit),
        detergentRequest : keep const false
    }
)

value missingGroundsContainer (value machine : CoffeeMachine)
: Exception
= machine.missingGroundsContainer

value missingDispenserBlock (value machine : CoffeeMachine)
: Exception
= machine.missingDispenserBlock

value waterSupplyFailure (value machine : CoffeeMachine)
: Exception
= machine.waterSupplyFailure
}

```

### C.5.6 Tastatur

Zur (anwendungsspezifischen) Tastatur gehören die in Abbildung C.2 dargestellten Tasten und LEDs. Die LEDs befinden sich an den sogenannten Artikeltasten (Produktasten), die den zur Auswahl stehenden Kaffeeprodukten zugeordnet sind. Die Tasten sind wie die Produkte bitcodiert, wobei für die Produktasten der Produktcode um 16 Bit verschoben eingeblendet wird. Der vom Datentyp `Key` abgeleitete Da-



`tentyp Keys` dient analog zu `Products` zur Darstellung einer Menge von Tasten.

Die Programmierschnittstelle der abstrakten Maschine `Keyboard` kennt nur eine, allerdings parametrisierte Phase zur Ansteuerung der Tastatur. Der Parameter gibt eine Auswahl von Produkten an, deren LEDs aktiviert werden sollen. Als Rückmeldung über die Tastatureingaben steht das Signal `keysDown`, das zu jedem Zeitpunkt die Menge aller gedrückten Tasten angibt, und die parametrisierten Ereignisse `keyPressed` und `keyReleased` zur Verfügung, die das Drücken bzw. Loslassen einer beliebigen aus einer durch den Parameter gegebenen Auswahl von Tasten anzeigen.

```
package Keyboard where
include Products

type Key = Integer32U

value noKey      : Key = 0x00000000#32u
value standbyKey : Key = 0x00000001#32u
value cleaningKey : Key = 0x00000002#32u
value handKey    : Key = 0x00000004#32u

value backwardKey : Key = 0x00000100#32u
value forwardKey  : Key = 0x00000200#32u
value plusKey     : Key = 0x00000400#32u
value minusKey    : Key = 0x00000800#32u
value menuKey     : Key = 0x00001000#32u
value clearKey    : Key = 0x00002000#32u

value productKey (value product : Product) : Key
= (product as Integer32U) << 16#8u

value productOfKey (value key : Key) : Product
= (key >> 16#8u) as Product

type Keys = Key

value allKeys : Keys
= 0xFFFFFFFF#32u

value productKeys (value products : Products) : Keys
= productKey (products)

value allProductKeys : Keys
= productKeys (allProducts)
```

```

value minKey (value keys : Keys) : Key
= if keys == noKey then noKey
  else if (keys & (1 as Key)) != noKey then (1 as Key)
  else (minKey (keys >> 1#8u)) << 1#8u

type KeyboardModule (type Input) (type Output) =
interface {
  type Keyboard
  type KeyboardControl

  // Construction and I/O
  value makeKeyboard (process keyboardControl : KeyboardControl)
    : Input -> Keyboard

  value keyboardOutput : Keyboard -> Output

  // Control
  phase enableProductLEDs (value keyboard : Keyboard,
                           value products : Products)
    : KeyboardControl

  signal keysDown      (value keyboard : Keyboard) : Keys
  event  keyPressed    (value keyboard : Keyboard, value keyMask : Keys)
  event  keyReleased   (value keyboard : Keyboard, value keyMask : Keys)
}

type KeyboardInput  = Process Keys
type KeyboardOutput = Process Products

value keyboardModule : KeyboardModule (KeyboardInput)
                                   (KeyboardOutput) =
module {
  type Keyboard = {
    keysDown          : Process Keys,
    keysPreviouslyDown : Process Keys,
    productLEDs       : Process Products
  }

  type KeyboardControl = Integer8U

  // Construction and I/O
  value makeKeyboard (process keyboardControl : KeyboardControl)
    (process keys : Keys)
    : Keyboard
  = {
    keysDown          : keys,
    keysPreviouslyDown : previous keys initially noKey,
  }

```

```

    productLEDs      : keyboardControl
  }

  process keyboardOutput (value keyboard : Keyboard) : Products
  = keyboard.productLEDs

  // Control

  phase enableProductLEDs (value keyboard : Keyboard,
                           value products : Products) : KeyboardControl
  = keep const products

  signal keysDown (value keyboard : Keyboard) : Keys
  = keyboard.keysDown

  event keyPressed (value keyboard : Keyboard, value keyMask : Keys)
  = trigger ((keyboard.keysDown & ~keyboard.keysPreviouslyDown & keyMask)
            != noKey)

  event keyReleased (value keyboard : Keyboard, value keyMask : Keys)
  = trigger ((~keyboard.keysDown & keyboard.keysPreviouslyDown & keyMask)
            != noKey)
}

```

### C.5.7 Display

Die abstrakte Maschine `Display` realisiert generisch eine Anzeige. Das Modul ist mit einem Typparameter für den Inhalt der Anzeige universell polymorph. Als Programmierschnittstelle steht eine einzige, mit dem angezeigten Inhalt parametrisierte Phase `show` zur Verfügung.

```

package Display where

type DisplayModule (type Input) (type Output)
                  (type DisplayMessage) =
interface {
  type Display
  type DisplayControl

  // Construction and I/O
  value makeDisplay (process displayControl
                    : DisplayControl) : Input -> Display
  value displayOutput : Display -> Output

  // Control
  phase show (value display : Display,
              value message : DisplayMessage) : DisplayControl
}

```

```

}

type DisplayInput = ()
type DisplayOutput (type DisplayMessage) = Process DisplayMessage

value displayModule <type DisplayMessage>
: DisplayModule (DisplayInput)
                (DisplayOutput (DisplayMessage))
                (DisplayMessage) =
module {
  type Display = Process DisplayMessage
  type DisplayControl = DisplayMessage

  value makeDisplay (process displayControl : DisplayControl)
  : () -> Display
  = (value dummy : ()) displayControl

  value displayOutput (value display : Display)
  : Process DisplayMessage
  = display

  phase show (value display : Display,
              value message : DisplayMessage) : DisplayControl
  = keep const message
}

```

### C.5.8 Benutzerinteraktion

Die abstrakte Maschine `User` realisiert den anwendungsspezifischen Dialog mit dem Benutzer der Kaffeemaschine. Zur Programmierschnittstelle gehören eine Reihe von Phasen, die die dem Benutzer angezeigten Zustände der Maschine repräsentieren, und eine Reihe von Ereignissen und ein Signal, die die Eingaben des Benutzers zurückliefern.

Die abstrakte Maschine ist auf Basis der abstrakten Tastatur und des abstrakten Display implementiert, wobei der Datentyp der Anzeige weiterhin generisch bleibt, aber ein bestimmter Satz von Nachrichten (des gewählten Typs) als Parameter erwartet wird. Je ein solcher Nachrichtensatz wird als Modul zu der vorgegebenen Schnittstelle definiert. Für einen bestimmten Anzeigetyp (zwei Zeilen Text à 16 Zeichen) und eine bestimmte Anwendersprache (Deutsch) wird ein Nachrichtensatz angegeben.

```
package User where
include Products
include Display
include Keyboard

type UserModule(type Input) (type Output) =
interface {
  type User
  type UserControl

  // Construction
  value makeUser (process control : UserControl) : Input -> User
  value userOutput (value user : User) : Output

  // Standby interaction

  phase informOnStandby      (value user : User) : UserControl
  phase informOnGoingStandby (value user : User) : UserControl

  event powerOn (value user : User)
  // awaited only while in informOnStandby

  event powerOff (value user : User)
  // awaited only while not in informOnStandby

  // Product or cleaning selection

  phase requestSelection (value user          : User,
                          value allowedProducts : Products,
                          value cleaningRecommended : Boolean)
    : UserControl

  event productSelected (value user : User)
  // awaited only while in requestSelection

  signal selectedProduct (value user : User) : Product
  // != noProduct when productSelected

  event cleaningSelected (value user : User)
  // awaited only while in requestSelection

  // Production information

  phase informOnProduction (value user      : User,
                            value product : Product) : UserControl

  // Cleaning interaction

  phase informOnCleaning (value user : User) : UserControl
```

```

    phase informOnNeutralization (value user : User) : UserControl

    phase requestDetergent (value user : User) : UserControl

    event detergentConfirmed (value user : User)
    // waited only while in requestDetergent

    // Alarms

    phase alarmMissingGroundsContainer (value user : User) : UserControl
    phase alarmMissingDispenserBlock   (value user : User) : UserControl
    phase alarmWaterSupplyFailed        (value user : User) : UserControl
}

type DisplayMessages (type DisplayText) = interface {
    value standbyMsg           : DisplayText
    value readyMsg             : DisplayText
    value enteringStandbyMsg   : DisplayText
    value cleaningRequestMsg   : DisplayText
    value productionRunningMsg (value product : Product) : DisplayText
    value cleaningRunningMsg   : DisplayText
    value neutralizationRunningMsg : DisplayText
    value detergentRequestMsg  : DisplayText
    value groundsContainerPulledMsg : DisplayText
    value dispenserBlockPulledMsg : DisplayText
    value waterSupplyFailedMsg  : DisplayText
}

type UserInput (type DisplayInput) (type KeyboardInput) = {
    display : DisplayInput,
    keyboard : KeyboardInput
}

type UserOutput (type DisplayOutput) (type KeyboardOutput) = {
    display : DisplayOutput,
    keyboard : KeyboardOutput
}

value userModule
    <type DisplayInput>
    <type DisplayOutput>
    <type DisplayText>
    (value displayModule : DisplayModule (DisplayInput)
                                     (DisplayOutput)
                                     (DisplayText))

    <type KeyboardInput>
    <type KeyboardOutput>
    (value keyboardModule : KeyboardModule (KeyboardInput)
                                     (KeyboardOutput))

    (value messages : DisplayMessages (DisplayText))

```

```

: UserModule (UserInput (DisplayInput) (KeyboardInput))
              (UserOutput (DisplayOutput) (KeyboardOutput)) =

import displayModule in
import keyboardModule in
import messages      in

let
  phase inform (value display  : Display,
                 value keyboard : Keyboard,
                 value message : DisplayText) :
  {
    display : DisplayControl,
    keyboard : KeyboardControl
  } =
  orthogonalize {
    display : show (display, message),
    keyboard : enableProductLEDs (keyboard, noProduct)
  }
in

module {
  type User = {
    display : Display,
    keyboard : Keyboard
  }
  type UserControl = {
    display : DisplayControl,
    keyboard : KeyboardControl
  }

  // Construction
  value makeUser (process control : UserControl)
                 (value input   : UserInput (DisplayInput)
                 (KeyboardInput))

  : User = {
    display : makeDisplay ((unzip control).display)
                  (input.display),
    keyboard : makeKeyboard ((unzip control).keyboard)
                  (input.keyboard)
  }

  value userOutput (value user : User)
  : UserOutput (DisplayOutput) (KeyboardOutput) = {
    display : displayOutput (user.display),
    keyboard : keyboardOutput (user.keyboard)
  }

  // Interaction

```

```

phase informOnStandby (value user : User) : UserControl
= inform (user.display, user.keyboard, standbyMsg)

phase informOnGoingStandby (value user : User) : UserControl
= inform (user.display, user.keyboard, enteringStandbyMsg)

event powerOn (value user : User)
= keyPressed (user.keyboard, standbyKey) +
  (keyReleased (user.keyboard, standbyKey) - after 1s)

event powerOff (value user : User)
= keyPressed (user.keyboard, standbyKey) +
  (after 1s - keyReleased (user.keyboard, standbyKey))

phase requestSelection (value user          : User,
                        value allowedProducts : Products,
                        value cleaningRecommended : Boolean)
: UserControl
= orthogonalize {
  display : show (user.display,
                  if cleaningRecommended then
                    cleaningRequestMsg
                  else
                    readyMsg),
  keyboard : enableProductLEDs (user.keyboard, allowedProducts)
}

event productSelected (value user : User)
= keyPressed (user.keyboard, allProductKeys)

signal selectedProduct (value user : User) : Product
= apply productOfKey to (apply minKey to (keysDown (user.keyboard)))

phase informOnProduction (value user      : User,
                           value product : Product) : UserControl
= inform (user.display, user.keyboard, productionRunningMsg product)

event cleaningSelected (value user : User)
= keyPressed (user.keyboard, cleaningKey)

phase informOnCleaning (value user : User) : UserControl
= inform (user.display, user.keyboard, cleaningRunningMsg)

phase informOnNeutralization (value user : User) : UserControl
= inform (user.display, user.keyboard, neutralizationRunningMsg)

phase requestDetergent (value user : User) : UserControl
= inform (user.display, user.keyboard, detergentRequestMsg)

```



[illegible]

```

        else
            ("Unbekanntes      ",
             "Produkt          ")

        value cleaningRunningMsg
        : DisplayText2Lines =
            ("Reinigung läuft ",
             "                  ")

        value neutralizationRunningMsg
        : DisplayText2Lines =
            ("Neutralisierung ",
             "läuft           ")

        value detergentRequestMsg
        : DisplayText2Lines =
            ("Reinigungstabl. ",
             "einwerfen        ")

        value groundsContainerPulledMsg
        : DisplayText2Lines =
            ("Satzschublade  ",
             "offen             ")

        value dispenserBlockPulledMsg
        : DisplayText2Lines =
            ("Ausgabekopf    ",
             "fehlt             ")

        value waterSupplyFailedMsg
        : DisplayText2Lines =
            ("Frischwasser    ",
             "fehlt             ")
    }

```

### C.5.9 Kaffeemaschinenanwendung

Die abstrakte Maschine der obersten Ebene kapselt die globale Zustandsmaschine des Kaffeeautomaten und implementiert die gesamte Anwendung auf Basis der abstrakten (verfahrenstechnischen) Kaffeemaschine und des abstrakten Benutzers. Zum globalen Zustand gehören auch persistent (d. h. im EEPROM) zu speichernde Informationen, in diesem Fall der Zustand der Reinigung. Da das EEPROM in FSPL wie ein externes Gerät behandelt wird, gehört der initial gespeicherte Zustand mit zur Eingabe und der sich im Laufe des Betriebs ändernde Zustand zur Ausgabe des Systems (siehe die Definitionen von `CoffeeMachineApplicationInput` und `CoffeeMachineApplicationOutput`).

Auf oberster Ebene besitzt die Maschine zwei Betriebsmodi, zwischen denen durch Tastendruck umgeschaltet werden kann: Standby und laufender Betrieb (Phasen `standBy` und `run`). Im laufenden Betrieb (`run/operate`) befindet sich die Maschine entweder in einer Ausnahmebehandlung (Aktion `handleException`) oder arbeitet normal (Phase `serve`). Die normale Arbeit besteht aus Warten in Bereitschaft (Phase `waitReady`), dem Herstellen eines Produkts (`makeProduct`), einer Reinigung (`clean`) oder dem Fortsetzen einer

abgebrochenen Reinigung, abhängig vom aktuellen Reinigungszustand (`resumeCleaning`).

Hinter diesen Phasen stehen die Aktionen `produce`, `doseDetergent`, `rinse` und `neutralize`, die für die Ausführung auf entsprechende Aktionen der darunterliegenden abstrakten Kaffeemaschine und für Benutzerinteraktion und Statusinformation auf den darunterliegenden abstrakten Benutzer abgestützt sind. Neben die Ansteuerung dieser beiden Maschinen tritt die „Ansteuerung“ des persistenten Zustands der Reinigung. Drei unterschiedliche Reinigungszustände (vor und nach der Dosierung von Reinigungsmittel, vor und nach der Neutralisierung) werden auf dieser Ebene unterschieden.

Die Anwendung sieht als Option vor, daß bei einer Reinigung die Maschine nach Abschluß der Spülungen selbsttätig in den Standby-Modus verfällt und beim Wiedereinschalten automatisch zunächst die Neutralisierung durchführt.<sup>6</sup> In dem bisher beschriebenen hierarchischen Phasenübergangssystem liegt hier eine hierarchieebenenübergreifende Transition vor. Sie wird unter Wahrung der Kompositionalität ausgeprägt durch einen zusätzlichen Prozeß `powerDownRequest` booleschen Zustands, der zum Teil der Ansteuerung und des Maschinenmodells wird (siehe die Definitionen von `CoffeeMachineApplication` und `CoffeeMachineApplicationControl`) und der auf der obersten Ebene des Phasenübergangssystems, wo der Wechsel zwischen Standby und laufendem Betrieb erfolgt, ausgewertet wird. Zur Auslösung des Wechsels in den Standby aus der Anwendung heraus gibt es die zusätzliche Aktion `enterStandby`, die die Flanke auf `powerDownRequest` erzeugt und durch die Transition auf oberster Ebene sofort abgebrochen wird.

Die Programmierschnittstelle der abstrakten Maschine besteht aus genau einem Prozeß (`runCoffeeMachine`).

```
package CoffeeMachineApplication where
include CoffeeMachine
include User
```

```
type CoffeeMachineApplicationModule (type Input) (type Output) =
interface {
  type CoffeeMachineApplication
```

<sup>6</sup>So kann z. B. die Reinigung abends als letzte Benutzung der Maschine angestoßen werden; die Kaffeebrühung zur Neutralisierung erfolgt dann morgens bei der Wiederinbetriebnahme.

```

type CoffeeMachineApplicationControl

value makeCoffeeMachineApplication
    (process control : CoffeeMachineApplicationControl)
: Input -> CoffeeMachineApplication

value coffeeMachineApplicationOutput
: CoffeeMachineApplication -> Output

process runCoffeeMachine
    (value application : CoffeeMachineApplication)
: CoffeeMachineApplicationControl
}

// Aspect of machine state stored persistently
type CleaningState = Integer2U
value detergentInside : CleaningState = 3#2u
value rinsed          : CleaningState = 1#2u
value neutral          : CleaningState = 0#2u

type CoffeeMachineApplicationInput (type CoffeeMachineInput)
                                   (type UserInput)
= {
    machine : CoffeeMachineInput,
    user    : UserInput,
    cleaning : CleaningState // initial cleaning state
}
type CoffeeMachineApplicationOutput (type CoffeeMachineOutput)
                                   (type UserOutput)
= {
    machine : CoffeeMachineOutput,
    user    : UserOutput,
    cleaning : Process CleaningState
}

value coffeeMachineApplicationModule
    <type CoffeeMachineInput>
    <type CoffeeMachineOutput>
    (value coffeeMachineModule :
        CoffeeMachineModule (CoffeeMachineInput)
                             (CoffeeMachineOutput))
    <type UserInput>
    <type UserOutput>
    (value userModule : UserModule (UserInput) (UserOutput))
    (value standbyAfterCleaning : Boolean)

: CoffeeMachineApplicationModule
    (CoffeeMachineApplicationInput (CoffeeMachineInput)
                                     (UserInput))

```

```

        (CoffeeMachineApplicationOutput (CoffeeMachineOutput)
                                         (UserOutput))
=

import coffeeMachineModule in
import userModule in

module {
  type CoffeeMachineApplication = {
    machine          : CoffeeMachine,
    user             : User,
    cleaningState    : Process CleaningState,
    previousCleaningState : Process CleaningState,
    powerDownRequest : Process Boolean
  }

  type CoffeeMachineApplicationControl = {
    machine          : CoffeeMachineControl,
    user             : UserControl,
    cleaningState    : CleaningState,
    powerDownRequest : Boolean
  }

  value makeCoffeeMachineApplication
    (process control : CoffeeMachineApplicationControl)
    (value input    : CoffeeMachineApplicationInput
      (CoffeeMachineInput)
      (UserInput))
  : CoffeeMachineApplication =
  let
    process coffeeMachineControl : CoffeeMachineControl
    = (unzip control).machine

    process userControl : UserControl
    = (unzip control).user

    process cleaningState : CleaningState
    = (unzip control).cleaningState

    process powerDownRequest : Boolean
    = (unzip control).powerDownRequest

    value coffeeMachineInput : CoffeeMachineInput
    = input.machine

    value userInput : UserInput
    = input.user

    value initialCleaningState : CleaningState

```

```

    = input.cleaning
in
{
  machine          : makeCoffeeMachine coffeeMachineControl
                    coffeeMachineInput,
  user             : makeUser userControl userInput,
  cleaningState    : cleaningState,
  previousCleaningState : previous cleaningState
                    initially initialCleaningState,
  powerDownRequest : powerDownRequest
}

value coffeeMachineApplicationOutput
  (value application : CoffeeMachineApplication)
: CoffeeMachineApplicationOutput (CoffeeMachineOutput)
  (UserOutput) =
{
  machine : coffeeMachineOutput (application.machine),
  user    : userOutput (application.user),
  cleaning : application.cleaningState
}

process runCoffeeMachine
  (value application : CoffeeMachineApplication)
: CoffeeMachineApplicationControl =

let
  value machine : CoffeeMachine
  = application.machine

  value user : User
  = application.user

  process cleaningState : CleaningState
  = application.cleaningState

  process previousCleaningState : CleaningState
  = application.previousCleaningState

  process powerDownRequest : Boolean
  = application.powerDownRequest
in

let
  action produce (value product : Product)
: CoffeeMachineApplicationControl = (
  parallelize {
    machine          : produce (machine, product),
    user             : do informOnProduction (user, product)
  }
)

```

```

        until never,
        cleaningState      : do keep const neutral until never,
        powerDownRequest   : do keep const false until never
    }
    terminating
)

action doseDetergent : CoffeeMachineApplicationControl = (
    parallelize {
        machine :
            doseDetergent (machine, detergentConfirmed (user)),

        user :
            do (
                do
                    informOnCleaning (user)
                when detergentRequest (machine) then

                do
                    requestDetergent (user)
                when detergentConfirmed (user) then

                informOnCleaning (user)
            )
        until never,

        cleaningState :
            do (
                do
                    keep const neutral
                when detergentConfirmed (user) then

                keep const detergentInside
            )
        until never,

        powerDownRequest :
            do keep const false until never
    }
    terminating
)

action rinse : CoffeeMachineApplicationControl = (
    parallelize {
        machine      : rinse (machine),
        user          : do informOnCleaning (user)
                        until never,
        cleaningState : do keep const detergentInside
                        until never,
    }
)

```

```

        powerDownRequest : do keep const false
                           until never
    }
    terminating
)

action neutralize : CoffeeMachineApplicationControl = (
    // Technically, neutralization is just a normal coffee
    // production, only the coffee flows away.
    parallelize {
        machine          : produce (machine, cafe),
        user              : do informOnNeutralization (user)
                           until never,
        cleaningState     : do keep const rinsed
                           until never,
        powerDownRequest : do keep const false
                           until never
    }
    terminating
)

action enterStandby (value cleaningState : CleaningState)
: CoffeeMachineApplicationControl =
(
    do orthogonalize {
        machine          : noCoffeeMachineActivity (machine),
        user              : informOnGoingStandby (user),
        cleaningState     : keep const cleaningState,
        powerDownRequest : keep const true
    }
    until never
    // powerDownRequest to be caught by outer control,
    // which will terminate this action
)
in

let
    phase serve : CoffeeMachineApplicationControl = (
        letrec
            phase waitReady : CoffeeMachineApplicationControl = (
                do
                    orthogonalize {
                        machine          : noCoffeeMachineActivity (machine),
                        user              : requestSelection (user,
                                                             allProducts,
                                                             false),
                        cleaningState     : keep const neutral,
                        powerDownRequest : keep const false
                    }

```



```
when productSelected (user) then (
  local
    value product : Product := selectedProduct (user)
  in
    makeProduct product
)
when cleaningSelected (user) then
  clean
)

phase makeProduct (value product : Product)
: CoffeeMachineApplicationControl =
(
  complete (
    produce product
  )
  then waitReady
)

phase clean : CoffeeMachineApplicationControl = (
  complete (
    doseDetergent;
    rinse;
    if standbyAfterCleaning then
      enterStandby (rinsed)
    else
      neutralize
  )
  then waitReady
)

phase resumeCleaning : CoffeeMachineApplicationControl = (
  complete (
    if (previousCleaningState == detergentInside) then (
      rinse;
      neutralize
    )
    else
      neutralize
  )
  then waitReady
)
in (
  if previousCleaningState != neutral then
    resumeCleaning
  else
    waitReady
)
)
```

```

in

let
  phase operate : CoffeeMachineApplicationControl = (
    let
      action handleException (value exception : Exception,
                             phase alarmUser : UserControl)
      : CoffeeMachineApplicationControl = (
        do
          orthogonalize {
            machine      : noCoffeeMachineActivity (machine),
            user         : alarmUser,
            cleaningState : keep previousCleaningState,
            powerDownRequest : keep const false
          }
        until exception.cleared
      )
    in (
      loop (
        if ((waterSupplyFailure (machine)).holds) then
          handleException (waterSupplyFailure (machine),
                          alarmWaterSupplyFailed (user))
        else if ((missingDispenserBlock (machine)).holds) then
          handleException (missingDispenserBlock (machine),
                          alarmMissingDispenserBlock (user))
        else if ((missingGroundsContainer (machine)).holds) then
          handleException (missingGroundsContainer (machine),
                          alarmMissingGroundsContainer (user))
        else (
          do
            serve
          until (
            (waterSupplyFailure (machine)).raised ||
            (missingDispenserBlock (machine)).raised ||
            (missingGroundsContainer (machine)).raised
          )
        )
      )
    )
  )
in

letrec
  phase standBy : CoffeeMachineApplicationControl =
  do
    orthogonalize {
      machine      : noCoffeeMachineActivity (machine),
      user         : informOnStandby (user),
      cleaningState : keep previousCleaningState,

```

```

        powerDownRequest : keep const false
    }
    when powerOn (user) then run

    phase run : CoffeeMachineApplicationControl = (
        do
            operate
            when powerOff (user)
            then standBy
            when (trigger (previous powerDownRequest initially false))
            then standBy
        )
    in
        start standBy
}

```

### C.5.10 Hauptprogramm

Das Hauptprogramm-Paket stellt für eine bestimmte Konfiguration alle applizierbaren Parameter ein und baut die abstrakten Maschinen bottom-up zusammen (Abschnitt „Modules“). Das Hauptprogramm selbst (Abschnitt „System (Main Program)“) instantiiert die abstrakte Maschine der obersten Ebene mit ihrem eindeutigen Prozess `runCoffeeMachine`. Es ist im Ergebnis eine Funktion (in dieser Konfiguration heißt sie `viva_German`), die die gesamten Eingaben des Systems auf seine Ausgaben abbildet (Systemfunktion).

```

package VivaMain where
include CoffeeMachineApplication

// ***** //
// *** Modules ***** //
// ***** //

type VivaPressInput  = MotionSensing
type VivaPressOutput = MotorActuation

value vivaStallDetectTimeout : Time = 5 ms

value vivaPressModule : PressModule (VivaPressInput)
                                (VivaPressOutput)
= pressModule (vivaStallDetectTimeout)

type VivaFlowInput  = FlowSensing

```

```

type VivaFlowOutput = ValveActuation

value vivaFlowFailureTimeout : Time = 500 ms

value vivaFlowModule : FlowModule (VivaFlowInput)
                           (VivaFlowOutput)
= flowMeterValveFlowModule (vivaFlowFailureTimeout)

type VivaBrewingUnitInput  = BrewingUnitInput  (VivaPressInput)
                                   (VivaFlowInput)

type VivaBrewingUnitOutput = BrewingUnitOutput (VivaPressOutput)
                                   (VivaFlowOutput)

value vivaDirectionToPhysicalLimit : VerticalDirection = down
value vivaDirectionTowardsFlow     : VerticalDirection = down
value vivaInjectionResistancePower : MotorPower = 16 as MotorPower

value vivaBrewingUnitModule : BrewingUnitModule
                           (VivaBrewingUnitInput)
                           (VivaBrewingUnitOutput)
= brewingUnitModule
  <VivaPressInput> <VivaPressOutput> vivaPressModule
  <VivaFlowInput>  <VivaFlowOutput>  vivaFlowModule
  (vivaDirectionToPhysicalLimit,
   vivaDirectionTowardsFlow,
   vivaInjectionResistancePower)

type VivaCoffeeMachineInput
= CoffeeMachineInput (VivaBrewingUnitInput)

type VivaCoffeeMachineOutput
= CoffeeMachineOutput (VivaBrewingUnitOutput)

value vivaFillingPosition : PressPosition = 800 as PressPosition
value vivaClosingPosition : PressPosition = 700 as PressPosition
value vivaBottomPosition  : PressPosition = 950 as PressPosition

value vivaDetergentDissolutionWater : FlowWaterAmount
= 50 as FlowWaterAmount

value vivaRinsingWater : FlowWaterAmount
= 200 as FlowWaterAmount

value vivaRinsingCyclesWithWaiting : Integer16U = 4 as Integer16U
value vivaRinsingCyclesWithPressOut : Integer16U = 2 as Integer16U

```

```

value vivaDetergentResidenceTime : Time = 60 s
value vivaRinsingWaitingTime      : Time = 5 s

value vivaCoffeeMachineModule
: CoffeeMachineModule (VivaCoffeeMachineInput)
                      (VivaCoffeeMachineOutput)
= coffeeMachineModule
  <BrewingUnitInput (MotionSensing) (FlowSensing)>
  <BrewingUnitOutput (MotorActuation) (ValveActuation)>
  vivaBrewingUnitModule
    (vivaFillingPosition,
     vivaClosingPosition,
     vivaBottomPosition)
    (vivaDetergentDissolutionWater,
     vivaRinsingWater)
    (vivaRinsingCyclesWithWaiting,
     vivaRinsingCyclesWithPressOut)
    (vivaDetergentResidenceTime,
     vivaRinsingWaitingTime)

type VivaDisplayInput  = DisplayInput
type VivaDisplayOutput = DisplayOutput (DisplayText2Lines)
type VivaDisplayMessage = DisplayText2Lines

value vivaDisplayModule : DisplayModule (VivaDisplayInput)
                                (VivaDisplayOutput)
                                (VivaDisplayMessage)
= displayModule <VivaDisplayMessage>

type VivaKeyboardInput  = KeyboardInput
type VivaKeyboardOutput = KeyboardOutput

value vivaKeyboardModule : KeyboardModule (VivaKeyboardInput)
                                (VivaKeyboardOutput)
= keyboardModule

type VivaUserInput  = UserInput  (VivaDisplayInput)
                                (VivaKeyboardInput)
type VivaUserOutput = UserOutput (VivaDisplayOutput)
                                (VivaKeyboardOutput)

value vivaUserModule_German : UserModule (VivaUserInput)
                                (VivaUserOutput)
= userModule
  <VivaDisplayInput><VivaDisplayOutput><VivaDisplayMessage>
  vivaDisplayModule

```

```

    <VivaKeyboardInput><VivaKeyboardOutput>
    vivaKeyboardModule
    displayMessages_German

type VivaCoffeeMachineApplicationInput
= CoffeeMachineApplicationInput (VivaCoffeeMachineInput)
                                (VivaUserInput)
type VivaCoffeeMachineApplicationOutput
= CoffeeMachineApplicationOutput (VivaCoffeeMachineOutput)
                                (VivaUserOutput)

value vivaStandbyAfterCleaning : Boolean = true

value vivaCoffeeMachineApplicationModule_German
: CoffeeMachineApplicationModule (VivaCoffeeMachineApplicationInput)
                                (VivaCoffeeMachineApplicationOutput)
= coffeeMachineApplicationModule
    <VivaCoffeeMachineInput>
    <VivaCoffeeMachineOutput>
    vivaCoffeeMachineModule
    <VivaUserInput>
    <VivaUserOutput>
    vivaUserModule_German
    vivaStandbyAfterCleaning

// ***** //
// *** System (Main Program) ***** //
// ***** //

value viva_German (value input : VivaCoffeeMachineApplicationInput)
: VivaCoffeeMachineApplicationOutput
= import
    vivaCoffeeMachineApplicationModule_German
in letrec
    value vivaCoffeeMachineApplication : CoffeeMachineApplication
    = makeCoffeeMachineApplication
        (runCoffeeMachine (vivaCoffeeMachineApplication))
        (input)
in
    coffeeMachineApplicationOutput (vivaCoffeeMachineApplication)

```

## C.6 Testmodule

Zum Testen der Module/Pakete der Anwendung (siehe Abschnitt C.5) wurden zahlreiche Testtreiber und -stubs implementiert.

Nachfolgend ist exemplarisch eine Dummy-Implementierung der Schnittstelle `BrewingUnitModule` zum Testen des Moduls `coffeeMachineModule` angegeben.

```

value testBrewingUnitModule
: BrewingUnitModule() (Process TestBrewingUnitOutput) =
module {
  type BrewingUnit = {
    position          : Process PressPosition,
    previousPosition : Process PressPosition,
    water             : Process Boolean,
    pump              : Process Boolean
  }
  type BrewingUnitControl = {
    position : PressPosition,
    water    : Boolean,
    pump     : Boolean
  }

  value makeBrewingUnit (process control : BrewingUnitControl)
                        (value dummy : ()) : BrewingUnit
= let
  process position : PressPosition = (unzip control).position
  process water    : Boolean        = (unzip control).water
  process pump      : Boolean        = (unzip control).pump
  in
  {
    position          : position,
    previousPosition : previous position
                        initially referencePosition,
    water             : water,
    pump              : pump
  }

  process brewingUnitOutput (value brewingUnit : BrewingUnit)
: TestBrewingUnitOutput
= zip {
  position : brewingUnit.position,
  water    : brewingUnit.water,
  pump     : brewingUnit.pump
}

  action doseWater (value brewingUnit : BrewingUnit,
                    value amount      : FlowWaterAmount)
: BrewingUnitControl
= do orthogonalize {
  position : keep brewingUnit.previousPosition,
  water    : keep const true,

```

```

        pump      : keep const false
    }
    until (after 2 ms)

    action injectWater (value brewingUnit : BrewingUnit,
                        value amount      : FlowWaterAmount)
    : BrewingUnitControl
    = do orthogonalize {
        position : keep brewingUnit.previousPosition,
        water    : keep const true,
        pump     : keep const true
    }
    until (after 2 ms)

    action movePressTo (value brewingUnit : BrewingUnit,
                        value position     : PressPosition)
    : BrewingUnitControl
    = do orthogonalize {
        position : keep const position,
        water    : keep const false,
        pump     : keep const false
    }
    until (after 1 ms)

    action movePressToPhysicalLimit (value brewingUnit : BrewingUnit)
    : BrewingUnitControl
    = do orthogonalize {
        position : keep const 1000 as PressPosition,
        water    : keep const false,
        pump     : keep const false
    }
    until (after 1 ms)

    phase noBrewingUnitActivity (value brewingUnit : BrewingUnit)
    : BrewingUnitControl
    = orthogonalize {
        position : keep brewingUnit.previousPosition,
        water    : keep const false,
        pump     : keep const false
    }
    }

    event waterFailure (value brewingUnit : BrewingUnit)
    = never

    signal pressPosition (value brewingUnit : BrewingUnit)
    : PressPosition
    = brewingUnit.position
}

```



# Literatur

- [Abr96] J.-R. Abrial: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [Ada95] Ada Reference Manual. ISO/IEC/ANSI 8652:19 Intermetrics, Cambridge, MA, 1995.
- [AGH05] K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language*. Addison Wesley Professional, 4. Aufl., 2005.
- [AMS05] K. Altisen, F. Maraninchi, D. Stauch: Aspect-Oriented Programming for Reactive Systems: a Proposal in the Synchronous Framework. Research Report TR-2005-18, Verimag, Nov. 2005.
- [And96] C. André: Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In: *Proc. CESA'96, Lille, France*, S. 19–29. Juli 1996.
- [Arm97] J. Armstrong: The development of Erlang. In: *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, S. 196–203. New York, NY, USA: ACM Press, 1997.
- [Aug98] L. Augustsson: Cayenne – a language with dependent types. In: *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, S. 239–250. New York, NY, USA: ACM Press, 1998.
- [Bac78] J. Backus: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, Bd. 21, Nr. 8, Aug. 1978.

- [Bar84] H. P. Barendregt: *The lambda calculus : its syntax and semantics*. Amsterdam: North-Holland, 1984.
- [BB91] A. Beneviste, G. Berry: The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, Bd. 79, Nr. 9: S. 1270–1282, Sept. 1991.
- [BBD<sup>+</sup>00] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull: *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [BCE<sup>+</sup>03] A. Beneviste, P. Caspi, S. A. Edwards, N. Halbwachs, P. le Guernic, R. de Simone: The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, Bd. 91, Nr. 1, Jan. 2003.
- [BCJ<sup>+</sup>02] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, J. Nordlander: Techn. Ber. CSE 02-002, Department of Computer Science and Engineering, OGI School of Science & Engineering, Oregon Health & Science University, 20000 NW Walker Road, Beaverton, OR 97006-8921, USA, Apr. 2002.
- [Ber00] G. Berry: *The Esterel v5 Language Primer*. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, France, 2000.
- [Ber02] A. S. Berger: *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. Lawrence, Kansas: CMP Books, 2002.
- [BK02] M. G. J. van den Brand, P. Klint: *ASF+SDF Meta-Environment User Manual*. Kruislaan 413, 1089 SJ Amsterdam, The Netherlands, Juli 2002. Revision 1.125.
- [Bou91] F. Boussinot: Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, Bd. 21, Nr. 4: S. 401–428, Apr. 1991.
- [BP01] R. Budde, A. Poigné: Complex Reactive Control with Simple Synchronous Models. In: *LCTES 2000* (herausgegeben von J. Davidson, S. Min), Bd. 1985 von *Lecture Notes in Computer Science*, S. 19–32. Berlin Heidelberg: Springer-Verlag, 2001.

- [BPS04] R. Budde, A. Poigné, K.-H. Sylla: synERJY – an Object-oriented Synchronous Language. In: *Proc. SLAP2004 (Synchronous Languages, Applications, and Programming)*. 2004.
- [BPS05] R. Budde, A. Poigné, K.-H. Sylla: synERJY 3.1 – An Introduction to the Language. Techn. Ber., Fraunhofer Institut Autonome intelligente Systeme (Fraunhofer AiS), Juli 2005.
- [Bre03] P. Breedveld: Port-based modeling of mechatronic systems. In: *Proceedings of the 4th MATHMOD Vienna – International IMACS Symposium on Mathematical Modelling* (herausgegeben von I. Troch, F. Breiteneker). Febr. 2003.
- [Bro97] M. Broy: Compositional refinement of interactive systems. *J. ACM*, Bd. 44, Nr. 6: S. 850–891, 1997.
- [Bro98] M. Broy: *Informatik – Eine grundlegende Einführung*, Bd. 1. Springer-Verlag, 2. Aufl., 1998.
- [BRS05] A. Bauer, J. Romberg, B. Schätz: Integrierte Entwicklung von Automotive-Software mit AutoFOCUS. *Informatik – Forschung und Entwicklung*, Bd. 19, Nr. 4: S. 194–205, Apr. 2005.
- [BS91] F. Boussinot, R. de Simone: The ESTEREL Language. *Proceedings of the IEEE*, Bd. 79, Nr. 9: S. 1293–1303, Sept. 1991.
- [BS96] F. Boussinot, R. de Simone: The SL Synchronous Language. *IEEE Trans. Softw. Eng.*, Bd. 22, Nr. 4: S. 256–266, 1996.
- [BS01] M. Broy, K. Stølen: *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.
- [BS03] E. Börger, R. Stärk: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

- [BS05] B. Bouyssounouse, J. Sifakis (Hg.): *Embedded Systems Design – The ARTIST Roadmap for Research and Development*, Bd. 3436 von *Lecture Notes in Computer Science*. Berlin; Heidelberg; New York: Springer, 2005.
- [BSMM01] I. N. Bronstein, K. A. Semendjajew, G. Musiol, H. Mühlig: *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 2001.
- [Bur99] A. Burns: The Ravenscar Profile. *Ada Letters*, Bd. XIX, Nr. 4: S. 49–52, 1999.
- [CGHP04] J.-L. Colaço, A. Girault, G. Hamon, M. Pouzet: Towards a higher-order synchronous data-flow language. In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, S. 230–239. New York, NY, USA: ACM Press, 2004.
- [Che76] P. P.-S. Chen: The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, Bd. 1, Nr. 1: S. 9–36, 1976.
- [Chu41] A. Church: *The Calculi of Lambda-Conversion*. Nr. 6 in *Annals of Mathematics Studies*. Princeton, NJ: Princeton University Press, 1941.
- [CL04] J. Carlson, B. Lisper: An event detection algebra for reactive systems. In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, S. 147–154. New York, NY, USA: ACM Press, 2004.
- [COM02] COMTESSA: Project Web Site, 2002. URL [http://www.fzi.de/esm/projects/Comtessa/Comtessa\\_uk.html](http://www.fzi.de/esm/projects/Comtessa/Comtessa_uk.html).
- [CP96] P. Caspi, M. Pouzet: Synchronous Kahn networks. In: *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, S. 226–238. New York, NY, USA: ACM Press, 1996.
- [CP00] P. Caspi, M. Pouzet: Lucid Synchrone, a Functional Extension of Lustre, 2000. Draft submitted to publication.

- [CPP05] J.-L. Colaço, B. Pagano, M. Pouzet: A conservative extension of synchronous data-flow with state machines. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, S. 173–182. New York, NY, USA: ACM Press, 2005.
- [CW85] L. Cardelli, P. Wegner: On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, Bd. 17, Nr. 4, Dez. 1985.
- [Dav92] A. J. T. Davie: *An introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.
- [Dav95] R. David: Grafcet: a powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, Bd. 3, Nr. 3: S. 253–268, Sept. 1995.
- [Dij68] E. W. Dijkstra: The structure of the “THE”-multiprogramming system. *Comm. ACM*, Bd. 11, Nr. 5: S. 341–346, 1968.
- [Dij69] E. W. Dijkstra: Complexity controlled by hierarchical ordering of function and variability. In: *Software Engineering*. NATO Science Committee, 1969.
- [Dij72] E. W. Dijkstra: Notes on Structured Programming. In: *Structured Programming*. London u. New York: Academic Press, 1972.
- [dSP06] dSPACE GmbH: Homepage, 2006. URL <http://www.dspace.de>.
- [E<sup>+</sup>88] H. Engesser, et al. (Hg.): *Duden Informatik*. Mannheim, Wien, Zürich: Duden-Verlag, 1988.
- [EH97] C. Elliott, P. Hudak: Functional reactive animation. In: *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, S. 263–273. New York, NY, USA: ACM Press, 1997.
- [Emb99] Embedded C++ Technical Committee: The Embedded C++ specification, Version WP-AM-003, 1999. URL <http://www.caravan.net/ec2plus/spec.html>.

- [ETA04] ETAS GmbH, Stuttgart: *ASCET V5.0 – Referenzhandbuch*, 2004. Dokument EC010005 R5.0.1 DE.
- [ETA06] ETAS GmbH: ETAS – Engineering Products and Services, 2006. URL <http://de.etasgroup.com>.
- [ETF03] Y. Etsion, D. Tsafir, D. G. Feitelson: Effects of clock resolution on the scheduling of interactive and soft real-time processes. *SIGMETRICS Perform. Eval. Rev.*, Bd. 31, Nr. 1: S. 172–183, 2003. ISSN 0163-5999.
- [FAZ04] Die echte Zeit. *FAZ*, 22.6.2004.
- [Föl92] O. Föllinger: *Regelungstechnik*. Heidelberg: Hüthig, 3. Aufl., 1992.
- [FMG01] G. Frick, K. D. Müller-Glaser: Information Management Concepts for Multi-Tool Modeling. In: *Proceedings of the IASTED International Conference MODELLING, IDENTIFICATION, AND CONTROL, February 19–22, 2001, Innsbruck, Austria*, S. 803–806. 2001.
- [FMG02] G. Frick, K. D. Müller-Glaser: Generative Development of Embedded Real-Time Systems. In: *ECOOP 2002 Workshop on Generative Programming, June 10, 2002, Malaga, Spain*. 2002.
- [FMG03] G. Frick, K. D. Müller-Glaser: Semantic Integration of Modelling Languages Based on a Reference Language. In: *Proceedings of the 4th MATHMOD Vienna – International IMACS Symposium on Mathematical Modelling* (herausgegeben von I. Troch, F. Breitenecker), S. 1564–1573. Febr. 2003.
- [FMG04] G. Frick, K. D. Müller-Glaser: A Design Methodology for Distributed Embedded Systems in Industrial Automation. In: *DESIGN&ELEKTRONIK, embedded world 2004 Conference, 18. Februar 2004, Nürnberg*, S. 575–579. 2004. Online verfügbar unter [IDE04].
- [FMO94] M. Fränzle, M. Müller-Olm: Towards Provably Correct Code Generation for a Hard Real-Time Programming Language. In: *CC '94: Proceedings of the 5th International*

- Conference on Compiler Construction*, S. 294–308. London, UK: Springer-Verlag, 1994.
- [FSMG01] G. Frick, E. Sax, K. D. Müller-Glaser: An Object-Based Model Representation System Lending OO Features to Non-OO Modeling Languages. In: *OMER-2 Workshop on Object-oriented Modeling of Embedded Real-time Systems, May 10–12, 2001, Herrsching, Germany*. 2001.
- [FSMG04] G. Frick, B. Scherrer, K. D. Müller-Glaser: Designing the Software Architecture of an Embedded System with UML 2.0. In: *Proceedings of the UML 2004 Workshop on Software Architecture Description & UML, October 11–15, 2004, Lisbon, Portugal*. 2004. Online verfügbar unter [IDE04].
- [GGBM91] P. le Guernic, T. Gautier, M. le Borgne, C. le Maire: Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, Bd. 79, Nr. 9: S. 1321–1335, Sept. 1991.
- [GH83] G. Goos, J. Hartmanis (Hg.): *The Programming Language Ada Reference Manual*, Bd. 155 von *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1983. American National Standards Institute, ANSI/MIL-STD-1815A-1983.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [GJSB95] V. Gupta, R. Jagadeesan, V. A. Saraswat, D. G. Bobrow: Programming in Hybrid Constraint Languages. In: *Hybrid Systems II*, S. 226–251. London, UK: Springer-Verlag, 1995.
- [Gün97] M. Günther: *Kontinuierliche und zeitdiskrete Regelungen*. Stuttgart: Teubner, 1997.
- [GNU] GNU Emacs. URL <http://www.gnu.org/software/emacs/>.
- [GR99] M. Gunzert, T. Ringler: ViPER – A Component-Based Approach for Designing Real-Time Systems. In: *Proc. INTERKAMA-ISA TECH 1999 Conference*. Düsseldorf, 1999.

- [GS90] C. A. Gunter, D. S. Scott: Semantic Domains. In: *Handbook of Theoretical Computer Science* (herausgegeben von J. van Leeuwen), Bd. B, S. 633–674. Elsevier Science Publishers B.V., 1990.
- [GS93] D. Gries, F. B. Schneider: *A logical approach to discrete math.* Springer-Verlag, 1993.
- [Gun02] M. Gunzert: *Komponentenbasierte Softwareentwicklung für sicherheitskritische eingebettete Systeme.* Dissertation, Universität Stuttgart, 2002. Shaker Verlag, Aachen, 2003.
- [Gut77] J. V. Guttag: Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, Bd. 20, Nr. 6: S. 396–404, 1977.
- [GZD<sup>+</sup>00] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao: *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.
- [Har87] D. Harel: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, Bd. 8: S. 231–274, 1987.
- [Har01] R. Harper: *Programming in Standard ML.* Carnegie Mellon University, 2001. Spring Semester, Working draft of June 28, 2001.
- [Has] The Haskell Homepage. URL <http://www.haskell.org/>.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud: The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, Bd. 79, Nr. 9: S. 1305–1319, Sept. 1991.
- [Hen96] T. A. Henzinger: The theory of hybrid automata. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, S. 278–292. 1996.
- [HHK03] T. A. Henzinger, B. Horowitz, C. M. Kirsch: Giotto: A Time-triggered Language for Embedded Programming. *Proceedings of the IEEE*, Bd. 91: S. 84–99, 2003.



- [HM03] K. Hammond, G. Michaelson: Hume: A Domain-Specific Language for Real-Time Embedded Systems. In: *GPCE 2003* (herausgegeben von F. Pfenning, Y. Smaragdakis), Bd. 2830 von *Lecture Notes in Computer Science*, S. 37–56. Berlin Heidelberg: Springer-Verlag, 2003.
- [HN96] D. Harel, A. Naamad: The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, Bd. 5, Nr. 4: S. 293–333, 1996. ISSN 1049-331X.
- [Hoa85] C. A. R. Hoare: *Communicating Sequential Processes*. London: Prentice Hall, 1985.
- [Hoa88] C. A. R. Hoare (Hg.): *Occam 2 Reference Manual: Inmos Limited*. Prentice Hall, 1988.
- [HP98] L. Holenderski, A. Poigné: The Multi-Paradigm Synchronous Programming Language LEA. In: *Proc. of the Intl. Workshop on Formal Techniques for Hardware and Hardware-like Systems*. 1998.
- [HP99] D. Harel, M. Politi: *Modeling Reactive Systems with Statecharts: The Statemate Approach*. I-Logix Inc., 3 Riverside Drive, Andover, MA 01810 U.S.A., 1999.
- [HPS02] F. Huber, J. Philipps, O. Slotosch: Model-based development of embedded systems. In: *Embedded Intelligence*. WEKA Fachzeitschriften-Verlag, 2002.
- [HR02] P. Hruschka, C. Rupp: *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. München, Wien: Carl Hanser Verlag, 2002.
- [HS01] F. Huber, B. Schätz: Integrated Development of Embedded Systems with AutoFocus. Techn. Ber. TUM-I0107, TU München, Institut für Informatik, Dez. 2001.
- [HSS96] F. Huber, B. Schätz, A. Schmidt, K. Spies: AutoFocus: A Tool for Distributed Systems Specification. In: *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, S. 467–470. London, UK: Springer-Verlag, 1996. Siehe auch Website, URL <http://autofocus.in.tum.de>.

- [Hud00] P. Hudak: *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- [HW01] D. M. Hoffman, D. M. Weiss (Hg.): *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley, 2001.
- [IBM] IBM Software: Rational Rose Technical Developer – Product Overview. URL <http://www.ibm.com/software/awdtools/developer/technical/>.
- [IDE04] IDESA: Project Web Site, 2004. URL <http://www.fzi.de/esm/eng/idesa.html>.
- [IEE85] IEEE standard for binary floating-point arithmetic. Bd. 22, Nr. 2: S. 9–25, 1985.
- [ILo] I-Logix. URL <http://www.ilogix.com>.
- [ILo01] I-Logix, 3 Riverside Drive, Andover, MA 01810, U.S.A.: *Rhapsody in MicroC, Quick Reference*, 2001. Part No. 2265.
- [JHM04] W. M. Johnston, J. R. P. Hanna, R. J. Millar: Advances in dataflow programming languages. *ACM Comput. Surv.*, Bd. 36, Nr. 1: S. 1–34, 2004.
- [KJ02] U. Kiencke, H. Jäkel: *Signale und Systeme*. München, Wien: Oldenbourg, 2002.
- [Kop97] H. Kopetz: *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
- [KR88] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*. Prentice Hall, 2. Aufl., 1988.
- [Kre91] H.-J. Kreowski: *Logische Grundlagen der Informatik*. Oldenbourg, 1991.
- [KRSW98] T. Kropf, J. Ruf, K. Schneider, M. Wild: A synchronous language for modeling and verifying real time and embedded

- systems. In: *GI/ITG/GME Workshop: Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen*. HNI-Verlagsschriften, 1998. ISBN 3-931466-35-3.
- [Lam78] L. Lamport: Time, Clock, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Bd. 21, Nr. 7, Juli 1978.
- [Lee00] E. A. Lee: What's Ahead for Embedded Software? *Computer*, Sept. 2000.
- [May88] O. Mayer: *Programmieren in COMMON LISP*. Nr. 638 in BI-Hochschultaschenbücher. Mannheim; Wien; Zürich: BI-Wiss.-Verl., 1988.
- [McC62] J. McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, S. 184–195, Mai 1962.
- [Mel02] S. J. Mellor: Coping with changing requirements. In: *Embedded Systems Conference*. München, 2002.
- [MG96] K. D. Müller-Glaser: CAD of Microsystems: A Challenge for System Engineering. In: *EURO-DAC'96 with EURO-VHDL'96, Geneva, Switzerland*. Sept. 1996.
- [MGFSK04] K. D. Müller-Glaser, G. Frick, E. Sax, M. Kühl: Multi-paradigm Modeling in Embedded Systems Design. *IEEE Transactions on Control Systems Technology*, Bd. 12, Nr. 2: S. 279–292, März 2004.
- [Mit90] J. C. Mitchell: Type Systems for Programming Languages. In: *Handbook of Theoretical Computer Science* (herausgegeben von J. van Leeuwen), Bd. B, S. 365–458. Elsevier Science Publishers B.V., 1990.
- [MMP92] O. Maler, Z. Manna, A. Pnueli: From Timed to Hybrid Systems. In: *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, S. 447–484. London, UK: Springer-Verlag, 1992.

- [Mos90] P. D. Mosses: Denotational Semantics. In: *Handbook of Theoretical Computer Science* (herausgegeben von J. van Leeuwen), Bd. B, S. 575–631. Elsevier Science Publishers B.V., 1990.
- [MR03] F. Maraninchi, Y. Rémond: Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, Bd. 46, Nr. 3: S. 219–254, 2003.
- [MSR] MSR-Konsortium. URL <http://www.msr-wg.de>.
- [MSR01] MSR-Arbeitsgruppe MEGMA: Standardization of library blocks for graphical model exchange, Version 1.12, 2001. URL <http://www.msr-wg.de/megma/download.html>.
- [MTHM97] R. Milner, M. Tofte, R. Harper, D. MacQueen: *The Definition of Standard ML – Revised*. The MIT Press, 1997.
- [NS62] J. Naas, H. L. Schmid: *Mathematisches Wörterbuch*. Berlin, Stuttgart, 1962.
- [OMG02] OMG: Meta Object Facility (MOF) Specification, Version 1.4. OMG Specification formal/02-04-03, Object Management Group, Apr. 2002.
- [OSE] OSEK VDX Portal. URL <http://www.osek-vdx.org/>.
- [P<sup>+</sup>] S. Pestov, et al.: jEdit Programmer’s Text Editor. URL <http://www.jedit.org/>.
- [Par72] D. L. Parnas: On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, Bd. 15, Nr. 12: S. 1053–1058, Dez. 1972.
- [Par76] D. L. Parnas: On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, Bd. SE-2, Nr. 1: S. 1–9, März 1976.
- [Par78] D. L. Parnas: Some Software Engineering Principles. Infotech state of the art report on structured analysis and design, Infotech International, 1978. Abgedruckt in [HW01].

- [Par79] D. L. Parnas: Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, S. 128–138, März 1979.
- [Pie02] B. C. Pierce: *Types and Programming Languages*. Cambridge, London: The MIT Press, 2002.
- [Rey98] J. C. Reynolds: *Theories of Programming Languages*. Cambridge University Press, 1998.
- [Rup93] W. Rupperecht: *Signale und Übertragungssysteme*. Springer-Verlag, 1993.
- [Sch01] U. Schöning: *Theoretische Informatik – kurzgefasst*. Heidelberg, Berlin: Spektrum Akademischer Verlag, 4. Aufl., 2001.
- [Sch05] P. Scholz: *Softwareentwicklung eingebetteter Systeme*. Berlin; Heidelberg; New York: Springer, 2005.
- [SDL00] Specification and description language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, 2000.
- [Seb04] R. W. Sebesta: *Concepts of Programming Languages*. Pearson Education, Inc., 6. Aufl., 2004.
- [SGW94] B. Selic, G. Gullekson, P. T. Ward: *Real-Time Object-Oriented Modeling*. New York: Wiley, 1994.
- [SJG99] V. Saraswat, R. Jagadeesan, V. Gupta: Timed Default Concurrent Constraint Programming. *J. Symbolic Computation*, , Nr. 11, 1999.
- [SM05] V. C. Sreedhar, M.-C. Marinescu: From Statecharts to ESP\*: Programming with events, states and predicates for embedded systems. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded Software*, S. 48–51. New York, NY, USA: ACM Press, 2005.
- [Som01] I. Sommerville: *Software Engineering*. Addison-Wesley, 6. Aufl., 2001.

- [SR98] B. Selic, J. Rumbaugh: Using UML for Modeling Complex Real-Time Systems, 1998. White Paper, URL [http://www.ibm.com/developerworks/rational/library/content/03July/1000/1155/1155\\_umlmodeling.pdf](http://www.ibm.com/developerworks/rational/library/content/03July/1000/1155/1155_umlmodeling.pdf).
- [Str97] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley Professional, 3. Aufl., 1997.
- [Sun] Java Technology. URL <http://java.sun.com/>.
- [Tay95] B. N. Taylor: Guide for the Use of the International System of Units (SI). NIST Special Publication 811, 1995 Edition, National Institute of Standards and Technology, Apr. 1995.
- [The06a] The MathWorks, Inc.: MATLAB – The Language of Technical Computing, 2006. URL <http://www.mathworks.com/products/matlab/>.
- [The06b] The MathWorks, Inc.: Real-Time Workshop Embedded Coder, 2006. URL <http://www.mathworks.com/products/rtwembedded/>.
- [The06c] The MathWorks, Inc.: *Stateflow and Stateflow Coder User's Guide, Version 6*, 2006. Revised for Stateflow 6.4 (Release R2006a), URL [http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/stateflow/sf\\_ug.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf).
- [The06d] The MathWorks, Inc.: *Using Simulink, Version 6*, 2006. Revised for Simulink 6.4 (Release 2006a), URL [http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/simulink/sl\\_using.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf).
- [UBH04] Virtuelle Fachbibliothek Mathematik, 2004. URL <http://www.ub.uni-heidelberg.de/helios/fachinfo/www/math/math.htm>.
- [UML03] OMG Unified Modeling Language Specification, Version 1.5. OMG Specification formal/03-03-01, Object Management Group, März 2003.

- [UML05a] Unified Modeling Language: Superstructure, version 2.0. OMG Specification formal/05-07-04, Object Management Group, 2005.
- [UML05b] UML Profile for Schedulability, Performance, and Time Specification, Version 1.1. OMG Specification formal/05-01-02, Object Management Group, 2005.
- [Unb97] R. Unbehauen: *Systemtheorie*, Bd. 1. München, Wien: Oldenbourg, 1997.
- [WA85] W. W. Wadge, E. A. Ashcroft: *Lucid, the Dataflow Programming Language*. Nr. 22 in APIC Studies in Data Processing. Academic Press, 1985.
- [Wat04] D. A. Watt: *Programming Language Design Concepts*. Wiley, 2004. With contributions by William Findlay.
- [WB05] H. Wörn, U. Brinkschulte: *Echtzeitsysteme*. Berlin; Heidelberg; New York: Springer, 2005.
- [WH00] Z. Wan, P. Hudak: Functional reactive programming from first principles. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, S. 242–252. New York, NY, USA: ACM Press, 2000.
- [WI02] B. C. Williams, M. D. Ingham: Model-Based Programming: Controlling Embedded Systems by Reasoning About Hidden State. In: *CP 2002* (herausgegeben von P. V. Hentenryck), Bd. 2470 von *Lecture Notes in Computer Science*, S. 508–524. Berlin; Heidelberg: Springer-Verlag, 2002.
- [Wik05a] Wikipedia: Aussagenlogik, 24.5.2005. URL <http://de.wikipedia.org/wiki/Aussagenlogik>.
- [Wik05b] Wikipedia: Prädikatenlogik, 24.5.2005. URL <http://de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik>.
- [Wik05c] Wikipedia: Quantifizierung (Logik), 24.5.2005. URL [http://de.wikipedia.org/wiki/Quantifizierung\\_%28Logik%29](http://de.wikipedia.org/wiki/Quantifizierung_%28Logik%29).

- [Wik05d] Wikipedia: Type safety, 4.1.2005. URL [http://en.wikipedia.org/wiki/Type\\_safety](http://en.wikipedia.org/wiki/Type_safety).
- [Wir71] N. Wirth: The programming language Pascal. *Acta Informatica*, Bd. 1, Nr. 1, Mai 1971.
- [Wir77] N. Wirth: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, Bd. 20, Nr. 11: S. 822–823, 1977.
- [Wir90] M. Wirsing: Algebraic Specification. In: *Handbook of Theoretical Computer Science* (herausgegeben von J. van Leeuwen), Bd. B, S. 675–788. Elsevier Science Publishers B.V., 1990.
- [WL99] D. M. Weiss, C. T. R. Lai: *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.
- [WMB01] A. Wohnhaas, R. Moser, P. Brangs: Standardblockbibliothek für Steuergerätesoftwareentwicklung. In: *Simulation technischer Systeme – Berichte aus der Fachgruppe* (herausgegeben von I. Bausch-Gall), SCS Frontiers in Simulation – ASIM Fortschritte in der Simulationstechnik. 2001. ISBN 1-56555-186-9.
- [WR95a] M. Wallace, C. Runciman: Extending a functional programming system for embedded applications. *Software Practice and Experience*, Bd. 25, Nr. 1: S. 73–96, Jan. 1995.
- [WR95b] M. Wallace, C. Runciman: Lambdas in the liftshaft – functional programming and an embedded architecture. In: *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, S. 249–258. New York, NY, USA: ACM Press, 1995.
- [WTH01] Z. Wan, W. Taha, P. Hudak: Real-time FRP. In: *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, S. 146–156. New York, NY, USA: ACM Press, 2001.



- [WTH02] Z. Wan, W. Taha, P. Hudak: Event-Driven FRP. In: *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, S. 155–172. London, UK: Springer-Verlag, 2002.



# Lebenslauf

**Name:** Gerd Frick

**Geboren:** am 27.2.1974 in Sobernheim

**Schulausbildung:**

1980 – 1984: Grundschule in Norheim  
1984 – 1992: Gymnasium an der Stadtmauer, Bad Kreuznach  
(Erwerb der allgemeinen Hochschulreife)

**Studium:**

10/1992 – 1/1998: Studium der Informatik  
mit Nebenfach Elektrotechnik  
an der Universität Kaiserslautern

**Beruflicher Werdegang:**

10/1994 – 12/1997: Wissenschaftliche Hilfskraft an der  
Universität Kaiserslautern,  
Fachbereich Informatik,  
Arbeitsgruppe VLSI-Entwurf und Architektur

1/1998 - 12/2004: Wissenschaftlicher Mitarbeiter am  
Forschungszentrum Informatik  
an der Universität Karlsruhe (FZI),  
Forschungsbereich Elektronische Systeme  
und Mikrosysteme (ESM)

1/2005 - 6/2005: Stipendiat am FZI, Forschungsbereich ESM

Seit 7/2005: Entwicklungsingenieur bei der  
Robert Bosch GmbH, Abstatt,  
Geschäftsbereich Chassis Systems Control