D I P L O M A R B E I T

Rigorose objektorientierte Analyse mit LOTOS: Spezifikation eines nebenläufigen, objektorientierten Datenflußsystems für die digitale Signalverarbeitung

ausgeführt am Institut für
Computersprachen
der Technischen Universität Wien

unter der Anleitung von O.Univ.Prof. Dipl.-Ing. Dr.techn. Manfred Brockhaus und Dipl.-Ing. Dr.techn. Franz Puntigam als verantwortlich mitwirkendem Universitätsassistenten.

durch

Patrick Peck

Schmalzhofgasse 17/8a
1060 Wien

Wien, am 1. Februar 2000 ____________________

Kurzfassung

Rigorose objektorientierte Analyse (ROOA) integriert eine formale Spezifikationssprache in den Prozeß der objektorientierten Analyse, mit dem Ziel, bereits in einer frühen Phase der Softwareentwicklung über ein exaktes und ablauffähiges Modell des zu entwickelnden Systems zu verfügen. Wir zeigen ROOA anhand der formalen Spezifikationssprache LOTOS und entwickeln damit eine objektorientierte Spezifikation eines Datenflußsystems für die digitale Signalverarbeitung. Das Datenflußsystem verwendet einfache Rechenelemente, die zu komplexeren Signalverarbeitungsfunktionen zusammengeschaltet werden können. Es soll so spezifiziert werden, daß einerseits eine einfache Erweiterbarkeit gegeben ist, andererseits nebenläufige Abarbeitung unterstützt wird. Das gesteckte Ziel konnte erreicht werden und es stellte sich heraus, daß ROOA sehr gut für diese Aufgabenstellung geeignet war, wenngleich die Verwendung von LOTOS als Spezifikationssprache nicht die optimale Wahl darstellte.

Schlüsselwörter: Rigorose objektorientierte Analyse (ROOA), objektorientierte Analyse, formale Spezifikation, LOTOS, Datenflußsysteme, digitale Signalverarbeitung

Abstract

Rigorous object-oriented analysis (ROOA) integrates a formal specification language into the process of object-oriented analysis. The aim is to obtain an exact and executable model of the system in question at an early stage of development. ROOA is demonstrated, using the formal specification language LOTOS and applying the method to develop an object-oriented specification of a data flow system for digital signal processing. The data flow system uses a number of simpler functional elements to realize more complex signal processing functions. The goals of the specification are easy extendibility and concurrent execution. We come to the conclusion that ROOA is very well suited for the problem. The use of LOTOS as the specification language, however, was not an optimal choice.

Keywords: Rigorous object-oriented Anaylsis (ROOA), object-oriented analysis, formal specification, LOTOS, data flow systems, digital signal processing

Inhaltsverzeichnis

Einleitung

Objektorientierung und Parallelverarbeitung sind bedeutende Konzepte moderner Softwareentwicklung.

Objektorientierte Techniken [Weg90] unterteilen ein System in kooperierende, gekapselte Objekte, die über Nachrichten miteinander kommunizieren. Das Zusammenfassen von Objekten zu Klassen und das Ausbilden von Klassenhierarchien sind weitere Merkmale. Der Hauptvorteil objektorientierter Techniken besteht darin, daß sich damit komplexe Systeme leichter beschreiben lassen als mit früheren Paradigmen der Softwareentwicklung. Durch die Wiederverwendung von Klassen bei der Entwicklung neuer Systeme wird außerdem die komponentenbasierte Entwicklung gefördert.

Demgegenüber sind die Ziele der Parallelverarbeitung in erster Linie höhere Geschwindigkeit und bessere Resourcenauslastung. Dies wird dadurch erreicht, daß Programme aus mehreren Ausführungssträngen bestehen können, die unabhängig voneinander gleichzeitig zum Gesamtfortschritt beitragen. Durch entsprechende Kommunikations- und Synchronisationsmechanismen wird die Kooperation zwischen den einzelnen Ausführungssträngen sichergestellt.

Ziel dieser Diplomarbeit ist die Entwicklung eines Datenflußsystems für die digitale Signalverarbeitung, das sowohl objektorientierte Entwicklung als auch nebenläufige (engl.: *concurrent*) Abarbeitung vereint. Als Datenflußsystem [Sha92] bezeichnen wir eine Menge unabhängiger Rechenelemente, die über mehrere Ein- und Ausgänge verfügen können. Jedes Element realisiert eine bestimmte Funktion, die aus den Daten an den Eingängen Resultate berechnet, die an den Ausgängen zur Verfügung gestellt werden. Daten „fließen" von den Ausgängen eines Rechenelements zu den Eingängen nachgeschalteter Elemente. Schaltet man mehrere Elemente hintereinander, können auf einfache Art und Weise komplexe Signalverarbeitungsfunktionen zusammengesetzt werden.

Vor allem im Bereich technisch-naturwissenschaftlicher Anwendungen sind Datenflußsysteme sehr beliebt, weil sie den Datenfluß von den Rohdaten bis zu den Ergebnissen gut veranschaulichen. Visuelle Datenflußprogrammiersprachen wie LabView oder SimuLink erlauben es programmiertechnisch weniger versierten BenutzerInnen, auf grafischer Ebene aus einem Satz von Rechenelementen auszuwählen, diese beliebig zu verschalten und anschließend zur Ausführung zu bringen.

Datenflußsysteme eignen sich hervorragend für die Kombination von Objektorientierung und Parallelverarbeitung. Für eine objektorientierte Entwicklung spricht die Tatsache, daß die verschiedenen Rechenelemente eine ähnliche innere Struktur und ein ähnliches Verhalten besitzen, welches sich in einer gemeinsamen Basisklasse konzentrieren läßt.

Parallelverarbeitung ist bei Datenflußsystemen bereits im Konzept enthalten, da die einzelnen Rechenelemente als unabhängig voneinander betrachtet werden, gesteuert nur durch die Verfügbarkeit von Daten. Der gegenwärtige Trend hin zu Multiprozessor-Workstations ist zudem ein wichtiges Argument für die Unterstützung von Nebenläufigkeit im zu entwickelnden Datenflußsystem.

Als Entwicklungsmethode wählen wir die sogenannte rigorose objektorientierte Analyse (ROOA), eine Methode, die eine formale Spezifikationssprache (in dieser Arbeit LOTOS [BB87]) in den Prozeß der objektorientierten Analyse integriert. Die Vorteile der Methode, bezogen auf die vorliegende Arbeit, sind:

- die objektorientierte Entwicklung nebenläufiger Systeme wird unterstützt;
- sowohl OOA als auch LOTOS sind bekannte und erprobte Methoden; entsprechende Werkzeuge zur computerunterstützten Entwicklung existieren und sind leicht zugänglich;
- das Ergebnis der ROOA ist eine formale, objektorientierte Spezifikation, die ablauffähig ist und als Prototyp für eine Implementierung dienen kann.

Inhalt

Inhaltlich gliedert sich die Arbeit in zwei Teile. Der erste Teil, der sich über die Kapitel 1 bis 3 erstreckt, führt die Methoden zur formalen, objektorientierten Analyse mit LOTOS ein. Kapitel 1 behandelt die formale Spezifikationssprache LOTOS. In Kapitel 2 wird gezeigt, wie objektorientierte Konzepte in LOTOS durch das Einhalten eines bestimmten Spezifikationsstils realisiert werden können. Kapitel 3 erläutert die rigorose objektorientierte Analyse, eine Methode zur Integration von objektorientierter Analyse und formaler Spezifikation.

Im zweiten Teil, bestehend aus den Kapiteln 4 bis 8, werden die im ersten Teil erörterten Methoden angewendet, um das oben beschriebene Datenflußsystem formal zu spezifizieren. Kapitel 4 zeigt zunächst, in welchem Kontext sich dieses Datenflußsystem befindet und welche Möglichkeiten bei der Entwicklung eines solchen Systems abgewogen werden müssen. Im 5. Kapitel beginnen wir die eigentliche Spezifikation mit ROOA, indem wir die Anforderungen an das Schaltungssystem beschreiben und ein erstes Objekt-Modell erstellen. In den Kapiteln 6 und 7 wird das Objekt-Modell verfeinert und die formale Spezifikation in LOTOS erstellt. Der vollständige Quelltext der resultierenden LOTOS-Spezifikation findet sich im Anhang. Das 8. Kapitel faßt noch einmal die Erfahrungen, die mit der angewendeten Methode gemacht wurden, zusammen.

Danksagung

Zu allererst möchte ich meinen Eltern, Ing. Stefan und Mag. Anna Peck, für die langjährige Unterstützung meines Studiums den größten Dank aussprechen.

Den Herren Ing. Anton Noll und Dr. Werner Deutsch vom Institut für Schallforschung der Österreichischen Akademie der Wissenschaften danke ich für die wertvollen Diskussionen und Anregungen bei der Entwicklung des Datenflußsystems.

Last but not least möchte ich mich bei meinem Betreuer Dr. Franz Puntigam vom Institut für Computersprachen der TU-Wien für seine unkomplizierte und dennoch sorgfältige Betreuung dieser Arbeit bedanken.

Kapitel 1

LOTOS

Die formale Spezifikationssprache LOTOS dient zur Spezifikation verteilter und nebenläufiger Systeme. Sie besteht im wesentlichen aus einer Kombination von Prozeßalgebra und abstrakten Datentypen. Die Darstellung der Sprache in diesem Kapitel folgt in weiten Teilen [BB87] und [QPM+93].

1.1 Geschichte und Hintergrund

LOTOS (Language of Temporal Ordering Specification, [BB87, QPM+93, ISO89a]) ist eine 1989 von der ISO genormte Spezifikationssprache zur Beschreibung von verteilten und nebenläufigen Informationssystemen. Insbesondere wurde sie entwickelt für die Spezifikation der OSI (Open Systems Interconnection) Netzwerkarchitektur [ISO94]. Als solche mußte sie [BB87, S. 26]:

- eine eindeutige, präzise, vollständige und implementierungsunabhängige Beschreibung des Standards ermöglichen,
- lesbare Dokumente für OSI-Benutzer, -Implementierer und -Conformance-Tester liefern, und
- eine formal wohldefinierte Basis für die Verifikation und Validation des Standards bzw. für Conformance-Tests von Implementierungen des Standards darstellen.

Da die genannten Anforderungen auch für Spezifikationen in anderen Bereichen wünschenswert sind, ist LOTOS aus dem OSI-Umfeld hinausgewachsen und wird heute für die Spezifikation beliebiger verteilter und nebenläufiger Anwendungen verwendet [CV90, LMD90, Hed93, QAP93].

Die theoretische Basis von LOTOS bilden die Prozeßalgebren *Calculus of Communicating Systems* (CCS, [Mil89]) von Robin Milner und *Communicating Sequential Processes* (CSP, [Hoa85]) von C.A.R. Hoare. Abstrakte Datentypen für die Interprozeß-Kommunikation können in LOTOS in einer an ACT-ONE [EM85] angelehnten Sprache definiert werden.

Etwa zur gleichen Zeit wie LOTOS wurden noch zwei andere Spezifikationssprachen genormt, nämlich Estelle [ISO89b, Ten93] von der ISO und SDL (Specification and Description Language, [CCI88, Fær93]) von der CCITT. Das ausschlaggebende Moment für deren Entwicklung war – wie bei LOTOS – das Bedürfnis nach einer implementierungsunabhängigen Beschreibung von digitalen Kommunikationsvorgängen. Estelle trug neben LOTOS zur Beschreibung des OSI-Standards bei, SDL wurde zur Spezifikation von Telekommunikationshardware verwendet.

Die prinzipiellen Unterschiede zwischen den Spezifikationssprachen sind in Tabelle 1.1 dargestellt, für einen ausführlichen Vergleich siehe [Tur93].

	LOTOS	Estelle	SDL
Ursprung	ISO	ISO	CCITT
Standard	ISO/IEC 8807	ISO/IEC 9074	CCITT Z.100
dynamisches Modell	Prozeßalgebra	endliche Automaten	endliche Automaten
Datentypen	abstrakte Datentypen (ACT ONE)	PASCAL-Datentypen	abstrakte Datentypen (ACT ONE)
Notation	textuell	textuell	grafisch und textuell
Objektorientierung	ja – Spezifikationsstil, kein Standard	unbekannt	ja, standardisiert

Tabelle 1.1: Die Spezifikationssprachen LOTOS, Estelle und SDL

LOTOS hat sich in der Zwischenzeit weiterentwickelt. Bereits einige Jahre nach der Standardisierung wurden Erweiterungswünsche von verschiedenen AnwenderInnen artikuliert. Die wichtigsten Neuerungen werden derzeit unter dem Namen E-LOTOS (Enhancements to LOTOS) als Standard vorgeschlagen [ISO97]. E-LOTOS ist so definiert, daß LOTOS-Spezifikationen weiterverwendet werden können.

Eine grafische LOTOS-Notation namens G-LOTOS [BNT94] wurde ebenfalls diskutiert, entwickelte sich aber nie zum Standard. Die Unterstützung durch Entwicklungswerkzeuge ist zudem gering.

1.2 Grundlagen

Ein verteiltes, nebenläufiges System wird in LOTOS als *Prozeß* gesehen, der seinerseits aus mehreren *Subprozessen* bestehen kann. Ein Subprozeß ist selbst auch wieder ein Prozeß, sodaß eine LOTOS-Spezifikation eine Hierarchie von Prozessen beschreibt. Der in Abbildung 1.1 gezeigte Prozeß `Max3` besteht z. B. aus zwei Subprozessen, jeweils vom Typ `Max2`.

Ein Prozeß kann einerseits *interne, nicht beobachtbare Aktionen* durchführen und andererseits mit den Prozessen seiner Umgebung in Interaktion treten. Die Interaktion mit der Umgebung ist aus elementaren Synchronisationseinheiten, den *Ereignissen* oder *Aktionen*, aufgebaut. Alle Prozesse, die an einem bestimmten Ereignis teilhaben, werden dadurch synchronisiert, d. h., sie beobachten das Ereignis zum selben Zeitpunkt. Das Eintreten des Ereignisses selbst wird als Vorgang gesehen, der keine Zeit in Anspruch nimmt. Prozesse können zu einem bestimmten Zeitpunkt höchstens an einem Ereignis teilhaben.

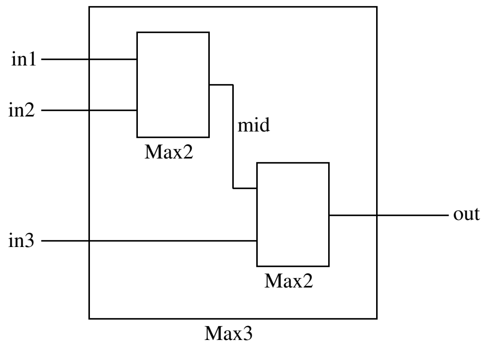

Abbildung 1.1: Max3-Prozeß

Das *Verhalten (Behaviour)* eines Prozesses wird spezifiziert, indem alle möglichen Reihenfolgen von internen und beobachtbaren Ereignissen beschrieben werden. In LOTOS werden dazu sogenannte *Behaviour Expressions* verwendet. Syntax und Semantik von Behaviour Expressions werden in Abschnitt 1.3 ausführlich behandelt.

Die Interaktionspunkte, an denen Ereignisse auftreten, werden *Gates* genannt. Prozesse können an den Gates gekoppelt werden und dadurch an gemeinsamen Ereignissen teilhaben. Bei der Koppelung von Prozessen kommt es oft vor, daß man bestimmte Gates der Umgebung nicht zur Verfügung stellen möchte. Zu diesem Zweck können in LOTOS sogenannte *versteckte Gates* definiert werden. Interaktionen an diesen Gates sind für die Umgebung nicht beobachtbar und daher interne Aktionen. Der `Max3`-Prozeß in Abbildung 1.1 kommuniziert über die Gates `in1`, `in2`, `in3` und `out` mit seiner Umgebung, während `mid` ein verstecktes Gate darstellt.

In LOTOS wird bei Ereignissen nicht zwischen „aktiver" und „passiver" Rolle eines Prozesses unterschieden. Alle an einem Ereignis teilhabenden Prozesse müssen das Ereignis gleichzeitig am entsprechenden Gate *anbieten*, d. h., zu einer Interaktion bereit sein.

1.3 Basic LOTOS

Basic LOTOS ist eine vereinfachte Form von LOTOS, die ausschließlich den prozeßalgebraischen Teil von LOTOS umfaßt. Das eigentliche LOTOS, zur besseren Unterscheidung auch *Full LOTOS* genannt, wird in Abschnitt 1.5 beschrieben.

Prozeßdefinitionen und Behaviour Expressions

Prozeßdefinitionen in Basic LOTOS haben folgende Struktur:

```
process proc_name[gate_list] :=
   behaviour expression
where
   process definitions
endproc
```

Ein Basic LOTOS Prozeß wird demnach definiert durch einen Namen (`proc_name`), eine Liste von formalen Gates (*gate_list*), eine Behaviour Expression zur Definition seines

Verhaltens (***behaviour expression***), und die Definitionen seiner Subprozesse (***process definitions***). Die nachfolgende Prozeßdefinition zeigt eine mögliche Definition des in Abbildung 1.1 gezeigten Prozesses `max3`[1]. Im Gegensatz zur Abbildung spezifiziert die Prozeßdefinition nicht nur die Hierarchie der Prozesse, sondern auch deren Verhalten:

```
1  process Max3[in1, in2, in3, out] :=
2     hide mid in
3        (Max2[in1, in2, mid]|[mid]|Max2[mid, in3, out])
4  where
5     process Max2[a, b, c] :=
6        a; b; c; stop
7        []
8        b; a; c; stop
9     endproc
10 endproc
```

Zeile 1 definiert den Prozeß `Max3` mit den formalen Gates `in1`, `in2`, `in3` und `out`. In Zeile 3 werden zwei Instanzen des Prozesses `Max2` als Subprozesse erzeugt und am Gate `mid` miteinander verbunden, welches in Zeile 2 durch den `hide`-Operator als internes Gate definiert wird. Der Prozeß `Max2` mit den formalen Gates `a`, `b` und `c` wird in den Zeilen 5–9 definiert.

Das dynamische Verhalten der Prozesse `Max2` und `Max3` wird durch ihre Behaviour Expressions spezifiziert. Der Prozeß `Max2` nimmt in beliebiger Reihenfolge an je einem Ereignis an den Gates `a` und `b` teil, anschließend an einem Ereignis am Gate `c`. Durch die Koppelung von zwei `Max2`-Prozessen erreicht der `Max3`-Prozeß, daß an den Gates `in1` bis `in3` in beliebiger Reihenfolge jeweils ein Ereignis auftreten muß, anschließend ein Ereignis am Gate `out`.

Tabelle 1.2 [BB87, S. 30] faßt die Syntax von Behaviour Expressions in Basic LOTOS zusammen. Die `g`, `g1`,…,`gn` stehen für Gates, `B`, `B1`, `B2` für Behaviour Expressions und `p` für Prozesse. Der Rest dieses Abschnitts widmet sich der Beschreibung der einzelnen Operatoren, aus denen Behaviour Expressions zusammengesetzt werden.

Inaction, Action Prefix, Choice

Inaktive Prozesse (Inaction) werden durch das Schlüsselwort `stop` spezifiziert. Inaktive Prozesse nehmen an keinen Interaktionen mit anderen Prozessen teil und führen auch keine internen Aktionen aus.

Mit Hilfe des *Action Prefix*-Operators `a; B` kann die sequentielle Ausführung der Aktion `a` und der Behaviour Expression `B` ausgedrückt werden. Interne, nicht beobachtbare (unobservable) Aktionen werden durch das Schlüsselwort `i` ausgedrückt, beobachtbare (observable) Aktionen durch den Namen des Gates, an dem sie stattfinden.

Mit dem *Choice*-Operator `[]` können alternative Verhaltensweisen spezifiziert werden. Für zwei Behaviour Expressions `B1` und `B2` beschreibt `B1 [] B2` einen Prozeß, der sich entweder wie `B1` oder `B2` verhält, abhängig von der ersten auftretenden Interaktion.

[1]Die Zeilennummern sind nicht Bestandteil von LOTOS-Spezifikationen. Sie werden in weiterer Folge immer dann angegeben, wenn im Text auf einzelne Zeilen bezug genommen wird.

Name	Syntax
inaction	`stop`
action prefix	
unobservable	`i; B`
observable	`g; B`
choice	`B1 [] B2`
parallel composition	
general case	`B1 |[g1,...,gn]| B2`
pure interleaving	`B1 ||| B2`
full synchronization	`B1 || B2`
hiding	`hide g1,...,gn in B`
process instantiation	`p[g1,...,gn]`
successful termination	`exit`
sequential composition (enabling)	`B1 >> B2`
disabling	`B1 [> B2`

Tabelle 1.2: Syntax von Behaviour Expressions in Basic LOTOS

Beispielsweise besagt die Behaviour Expression des Prozesses `Max2`, daß die Ereignisse entweder in der Reihenfolge `a`–`b`–`c` auftreten, oder in der Reihenfolge `b`–`a`–`c`, abhängig davon, ob das erste Ereignis `a` oder `b` ist. Durch den Choice-Operator kann auch indeterministisches Verhalten spezifiziert werden (siehe Unterabschnitt *Nondeterminismus und interne Aktionen*).

Parallelität

Mit dem *Parallel Composition*-Operator können Behaviour Expressions parallel zusammengesetzt werden. Solche Behaviour Expressions laufen unabhängig voneinander und synchronisieren sich für Interaktionen an bestimmten Gates. Allgemein hat der Parallel Composition-Operator die Form `B1 |[g1,...,gn]| B2` und besagt, daß `B1` und `B2` parallel ablaufen und sich an den Gates `g1,...,gn` synchronisieren, d. h., an diesen Gates gleichzeitig am selben Ereignis teilhaben müssen. An allen anderen Gates können `B1` und `B2` unabhängig voneinander an Ereignissen teilnehmen. Zwei nützliche Spezialfälle des Operators existieren:

- *Pure Interleaving*: `B1 ||| B2`
  Die Behaviour Expressions sind an keinem Gate synchronisiert, sie laufen gänzlich unabhängig voneinander.

- *Full Synchronization*: `B1 || B2`
  Die Behaviour Expressions sind an allen Gates synchronisiert, sie laufen bis auf interne Aktionen vollständig synchron.

Die Behaviour Expression des Prozesses `Max2` könnte mit dem Parallel Composition-Operator äquivalent so definiert werden:

```
a; c; stop
|[c]|
b; c; stop
```

Die beiden Behaviour Expressions laufen parallel und müssen sich am Gate `c` synchronisieren (d. h., das Ereignis `c` zur gleichen Zeit anbieten). Die Reihenfolge der Ereignisse an den Gates `a` und `b` ist wiederum beliebig.

Hiding

Hiding erlaubt, beobachtbare Ereignisse eines zusammengesetzten Prozesses in nicht beobachtbare zu verwandeln, indem bestimmte Gates zu *versteckten (hidden)* Gates erklärt werden. Durch die Behaviour Expression `hide g1,...,gn in B` werden die Gates `g1` bis `gn` der Behaviour Expression `B` zu versteckten Gates und sind damit für eine Synchronisation mit anderen Prozessen nicht verfügbar. Jedes Ereignis, das an einem solchen Gate auftritt, ist für außenstehende Prozesse eine interne Aktion. Hiding führt dadurch implizit nicht beobachtbare Aktionen ein—im Gegensatz zum `i`-Ereignis, mit dem sie explizit angegeben werden.

Prozeßinstanzierung

Prozeßdefinitionen können in Behaviour Expressions verwendet werden. `P[g1,...,gn]` instanziert den Prozeß `P` mit der aktuellen Gateliste `g1,...,gn`. Die Definition des Prozesses `P` mit einer entsprechenden n-stelligen formalen Gateliste muß an anderer Stelle in der Spezifikation gegeben sein. Der `Max3`-Prozeß instanziert z. B. zwei `Max2`-Prozesse, jeweils mit unterschiedlichen aktuellen Gatelisten.

Prozesse können sich selbst rekursiv aufrufen, was in erster Linie dazu verwendet wird, endlos laufende Prozesse zu spezifizieren. Der nachfolgende Prozeß `EndlessMax2` entspricht einem endlos laufenden `Max2`-Prozeß:

```
process EndlessMax2[a, b, c] :=
   a; b; c; EndlessMax2[a, b, c]
   []
   b; a; c; EndlessMax2[a, b, c]
endproc
```

Successful Termination und Sequential Composition

Die *erfolgreiche Beendigung (Successful Termination)* eines Prozesses wird durch das Schlüsselwort `exit` spezifiziert. Soll nach der erfolgreichen Beendigung des Prozesses ein anderer gestartet werden, kann das durch den *Sequential Composition*-Operator `>>` ausgedrückt werden:

```
(a; b; exit) >> next_process[...]
```

Der Übergang vom terminierenden zum nachfolgenden Prozeß ist ein nicht beobachtbarer Vorgang, d. h., der `>>` Operator führt implizit eine nicht beobachtbare Aktion ein.

Für den Fall parallel zusammengesetzter Prozesse ist eine erfolgreiche Beendigung aller Teilprozesse notwendig, damit der zusammengesetzte Prozeß erfolgreich terminiert.

Disabling

Bei der Ausführung von Prozessen kann es zu Ausnahmesituationen kommen, die einen Abbruch des Prozesses notwendig machen. Dieser Sachverhalt kann in LOTOS durch den *Disabling*-Operator `[>` spezifiziert werden. `B1 [> B2` bedeutet, daß `B1` ausgeführt wird, solange nicht die erste Aktion von `B2` ausgeführt werden kann. `B2` ist also die Behaviour Expression, die sowohl die Ausnahmesituation als auch deren Behandlung angibt. Falls `B1` erfolgreich terminiert, wird `B2` fallengelassen. Bei der nachfolgenden Behaviour Expression wird die Ausnahmebehandlung durch ein Ereignis am Gate `d` eingeleitet:

```
(a; b; exit)[>(d; stop)
```

Das Verhalten kann äquivalent durch den Choice-Operator ausgedrückt werden:

```
a; b; exit
[]
d; stop
[]
a; d; stop
[]
a; b; d; stop
```

Nondeterminismus und interne Aktionen

Nondeterministisches Verhalten tritt auf, wenn es an einem Punkt mehrere mögliche Aktionszweige gibt, deren Auswahl aber nicht im voraus spezifizierbar ist. Nondeterminismus kann in LOTOS mit dem Choice-Operator und dem internen Ereignis auf 3 verschiedene Arten ausgedrückt werden:

Fall 1: `(a; P) [] (a; Q)`
Das System entscheidet zum Zeitpunkt des Auftretens des Ereignisses `a`, ob es sich danach wie `P` oder `Q` verhält.

Fall 2: `(i; P) [] (i; Q)`
Das System entscheidet intern, ob es sich wie `P` oder `Q` verhält.

Fall 3: `(a; P) [] (i; Q)`
Das Ereignis `a` kann eintreten, wenn nicht vorher das interne Ereignis `i` eintritt. Bei dieser Art des Nondeterminismus hat die Umgebung die Chance, das System zu beeinflussen, wenn sie vor dem Eintreten des internen Ereignisses handelt. Da LOTOS keine absolute Zeit mitführt, ist unbestimmt, wie lange die Umgebung Zeit hat, bevor das interne Ereignis eintritt. Diese Konstruktion wird verwendet, um z. B. Timeouts, Systemfehler oder Resourcenmangel zu spezifizieren.

In den Fällen 1 und 2 hat die Umgebung keine Möglichkeit, das System in eine bestimmte Richtung zu lenken.

1.4 Datentypen

Im vorigen Abschnitt wurde Basic LOTOS beschrieben, jene Untermenge von LOTOS, mit der Prozesse und deren Synchronisation durch Ereignisse definiert werden können. Prozesse tauschen im allgemeinen auch Daten aus, welche jeweils einem bestimmten Datentyp angehören. Um Datentypen implementierungsunabhängig zu definieren, werden in LOTOS *abstrakte Datentypen (ADTs)* verwendet. Im Gegensatz zu konkreten Datentypen, die angeben, *wie* die Daten repräsentiert werden, definiert ein ADT nur die Eigenschaften der Daten unter den darauf möglichen Operationen. Eine Implementierung des abstrakten Datentyps muß diese Eigenschaften beibehalten.

Eine typische Definition eines ADT in LOTOS hat folgende Struktur:

```
type T is T1,...,Tn
   sorts ...
   opns ...
   eqns ...
endtype
```

Der Datentyp `T` verwendet in seiner Spezifikation die Typen `T1,...,Tn`. In der `sorts` Klausel werden die von dem ADT definierten *Sorten* deklariert, in der `opns` Klausel die *Operatoren*. Sorten und Operatoren zusammen werden auch *Signatur* des ADT genannt. In der `eqns` Klausel werden *Gleichungen* angegeben, mit denen die Eigenschaften der Operatoren festgelegt werden. Im folgenden Beispiel wird die Sorte `nat`, die die natürlichen Zahlen darstellen soll, durch den ADT `NaturalNumbers` definiert:

```
type NaturalNumbers is
   sorts nat
   opns
      0    : ->nat
      succ : nat->nat
      _+_  : nat, nat->nat
   eqns
      forall x, y: nat
      ofsort nat
         x+0=x;
         x+succ(y)=succ(x+y);
endtype
```

Operatoren werden definiert durch ihren Namen, den Typen ihrer Parameter und ihrem Ergebnistyp. `0` ist ein nullstelliger Operator, der eine Konstante vom Typ `nat` definiert. Der Operator `succ` bildet ein Element des Typs `nat` auf ein Element des Typs `nat` ab. Der als Infix definierte Operator `+` bildet zwei Elemente des Typs `nat` auf ein Element des Typs `nat` ab.

Das Verhältnis der Operatoren zueinander wird durch Gleichungen definiert. Für den ADT `NaturalNumbers` sind das die zwei Gleichungen `x+0=x` und `x+succ(y)=succ(x+y)`, die für alle `x` und `y` vom Typ `nat` gelten sollen (angegeben durch `forall x, y: nat`).

Der Ausdruck `ofsort nat` gibt an, daß die beiden Gleichungen Terme vom Typ `nat` miteinander in Beziehung setzen.

Die Definition von Datentypen mit vielen Operatoren ist nicht trivial. Die Gleichungen sollen den Datentyp vollständig definieren und dürfen einander nicht widersprechen. Durch folgende Mechanismen wird die Definition von Datentypen erleichtert:

- Erweiterung und Kombination von Typspezifikationen
  Bestehende Typspezifikationen können durch das Hinzufügen neuer Sorten, Operatoren und Gleichungen erweitert werden. Das erlaubt die inkrementelle Entwicklung von ADTs. Der ADT `NaturalNumbers` kann beispielsweise durch Hinzufügen des Operators `largest`, der die größere von zwei Zahlen zurückliefert, zum Typ `ExtendedNaturalNumbers` erweitert werden:

  ```
  type ExtendedNaturalNumbers is NaturalNumbers
     opns
        largest: nat, nat->nat
     eqns
        forall x, y: nat
        ofsort nat
           largest(0, x)=x;
           largest(x, y)=largest(y, x);
           largest(succ(x), succ(y))=succ(largest(x, y));
  endtype
  ```

  Durch entsprechende Gleichungen werden die Eigenschaften des hinzugefügten Operators `largest` definiert bzw. zu den Operatoren des Basistyps in Relation gesetzt.

- Parametrisierung und Aktualisierung
  Typen können unvollständig spezifiziert werden, d. h., es können formale Parameter für Sorten und Operatoren angegeben werden. Die Instanzierung des Typs mit konkreten Sorten und Operatoren wird *Aktualisierung* genannt. Ein Beispiel dafür ist ein generischer Listentyp, der mit der Sorte für den Elementtyp aktualisiert werden muß:

  ```
  type GenericList is
     formalsorts element
     sorts list
     opns
        empty  : -> list
        append : element, list->list
        first  : list->element
     eqns
        ...
  endtype
  ```

  In der `formalsorts` Klausel werden die formalen Sorten des Datentyps angegeben, im obigen Beispiel `element`. Durch Aktualisierung mit der Sorte `nat` kann aus

`GenericList` ein ADT erzeugt werden, der eine Liste von natürlichen Zahlen definiert:

```
type NaturalNumberList is GenericList
   actualizedby NaturalNumbers using
   sortnames nat for element
endtype
```

- Type Renaming
  Mit Type Renaming können neue, unabhängige Typdefinitionen aus bereits bestehenden durch die Umbenennung von Sorten und Operatoren gewonnen werden. So kann man z. B. aus dem ADT für natürliche Zahlen einen ADT für Objekt-Ids erzeugen.

```
type Ids is NaturalNumbers
   renamedby
   sortnames id for nat
   opnnames id0 for 0
endtype
```

  Nicht explizit umbenannte Sorten und Operatoren behalten ihren Namen (im angeführten Beispiel die Operatoren `succ` und `+`).

1.5 Full LOTOS

Full LOTOS integriert die in Abschnitt 1.4 beschriebenen abstrakten Datentypen in Basic LOTOS. Dadurch wird es möglich, Daten zwischen Prozessen auszutauschen und arithmetische Ausdrücke in Behaviour Expressions einzubeziehen.

Spezifikationen und Prozesse

Die nachfolgenden Programmgerüste zeigen den Aufbau von Spezifikationen und Prozessen in Full LOTOS:

```
specification spec_name[gate_list](parameter_list) : functionality
   type definitions
behaviour
   behaviour expression
where
   type definitions
   process definitions
endspec
```

Prozeß A	Prozeß B	Bedingung	Art der Interaktion	Auswirkung
g!E1	g!E2	wert(E1)=wert(E2)	Wertvergleich	Synchronisation
g!E	g?x:t	wert(E) hat Sorte t	Wertübergabe	nach der Synchronisation ist x=wert(E)
g?x:t	g?y:u	t=u	Wertgenerierung	nach der Synchronisation ist x=y=v, wobei v ein Wert der Sorte t ist

Tabelle 1.3: Arten der Interaktion

```
process proc_name[gate_list](parameter_list) : functionality :=
   behaviour expression
where
   type definitions
   process definitions
endproc
```

Die oberste Ebene einer LOTOS-Spezifikation bildet die zwischen `specification` und `endspec` angegebene Spezifikation. Spezifikationen sind ähnlich aufgebaut wie Prozesse, mit dem Unterschied, daß die Behaviour Expression einer Spezifikation mit dem Schlüsselwort `behaviour` eingeleitet wird. Die dem Schlüsselwort vorangehenden Typdefinitionen sind globale Typdefinitionen, die an die Umgebung exportiert werden. Im `where`-Teil von Spezifikationen und Prozessen werden lokale Typen in beliebiger Reihenfolge mit (lokalen) Unterprozessen definiert.

Zum Unterschied von Basic LOTOS verfügen Spezifikationen und Prozesse über *Parameterlisten*, in denen beim Aufruf Werte an die Spezifikation bzw. den Prozeß übergeben werden können (siehe Unterabschnitt *Parametrisierte Prozesse*). Die *Funktionalität* von Spezifikationen und Prozessen gibt zusätzlich ihr Terminationsverhalten an (siehe Unterabschnitt *Funktionalität von Prozessen*). Durch die Angabe einer entsprechenden Funktionalität können terminierende Prozesse Werte an die aufrufenden Prozesse bzw. Spezifikationen zurückgeben.

Strukturierte Ereignisse und Interprozeß-Kommunikation

Interprozeß-Kommunikation findet in Full LOTOS durch *strukturierte Ereignisse* der Form `g a1...an` statt, wobei `g` der Name eines Gates und `a1...an` eine Liste von *Attributen* ist. Es gibt zwei Arten von Attributen:

- *Wertdeklarationen (Value Declarations)* der Form `!expression` liefern den durch `expression` gegebenen Wert. Z. B.: `!TRUE`, `!'abcd'`, `!(3+5)`, `!largest(x, 3)`, `!(x+1)`
- *Variablendeklarationen (Variable Declarations)* der Form `?variable:sort` erwarten einen Wert der Sorte `sort` in der Variablen `variable`. Z. B.: `?x:nat`, `?y:boolean`, `?s:string`

Damit parallel verknüpfte Prozesse miteinander kommunizieren können, müssen sie gleichzeitig am selben Gate „passende“ strukturierte Ereignisse anbieten. Passend bedeutet, daß gleich viele Attribute vorhanden sein müssen und daß zwischen den Attributen

Übereinstimmung hergestellt werden kann (die Bedingungen dafür sind in Tabelle 1.3 in der Spalte *Bedingung* angegeben). Die kommunizierten Werte werden zum Zeitpunkt der Synchronisation festgelegt, alle beteiligten Prozesse arbeiten nach der Synchronisation mit den gleichen Werten weiter. Über die Arten der Interaktion bei der Interprozeß-Kommunikation und deren Auswirkungen gibt ebenfalls Tabelle 1.3 Aufschluß. Z. B. kann

```
g !succ(succ(0)) !succ(0) ?a:boolean
```

mit

```
g ?x:nat !succ(0) !TRUE
```

interagieren. Die Variable `x` hat nach der Synchronisation den Wert `succ(succ(0))`, `a` den wert `TRUE`. Interaktion ist auch mit

```
g ?x:nat !succ(0) ?b:boolean
```

möglich, wobei in diesem Fall durch Wertgenerierung für die Variablen `a` und `b` ein unbestimmter Wert der Sorte `boolean` erzeugt wird. Keine Interaktion kommt hingegen Zustande mit

```
g !succ(0) !succ(0) !TRUE    (* keine Übereinstimmung in Attribut 1 *)
```

auch nicht mit

```
g ?x:nat ?y:boolean !FALSE   (* falscher Typ von Attribut 2 *)
```

Das nachfolgende Beispiel gibt eine Spezifikation des `Max3`-Prozesses, der natürliche Zahlen an den Gates `in1` bis `in3` in beliebiger Reihenfolge entgegennimmt und das Maximum am Gate `out` zur Verfügung stellt:

```
specification Max3[in1, in2, in3, out] : noexit
   type ExtendedNaturalNumbers is ... endtype[2]
behaviour
   hide mid in
      (Max2[in1, in2, mid]|[mid]|Max2[mid, in3, out])
where
   process Max2[a, b, c] : noexit :=
      a ?x:nat; b ?y:nat; c !largest(x, y); stop
      []
      b ?x:nat; a ?y:nat; c !largest(x, y); stop
   endproc
endspec
```

[2]Für die Spezifikation des ADT `ExtendedNaturalNumbers` siehe Abschnitt 1.4.

Bedingungen

Bedingungen sind boolesche Ausdrücke, d. h. entweder Gleichungen, die Ausdrücke zueinander in Relation setzen oder Ausdrücke vom Typ `bool`. Bedingungen können in LOTOS auf zweierlei Arten verwendet werden:

- *Selection Predicates:* Damit kann man Variablendeklarationen auf einen bestimmten Wertebereich einschränken. Z. B. erwartet die Variablendeklaration `?x:nat [x<3]` für die Variable `x` eine natürliche Zahl, die kleiner als 3 ist (entsprechende Definition für den `<` Operator vorausgesetzt).
- *Guarded Expressions:* Behaviour Expressions können sogenannte *Guards* vorangestellt werden. Das Verhalten dieser Guarded Expression ist wie folgt definiert: wenn die Bedingung erfüllt ist, ist das Verhalten der Behaviour Expression möglich, ansonsten äquivalent mit `stop`. Typischerweise verwendet man Guarded Expressions in Verbindung mit dem Choice-Operator:

  ```
  [cond_1]->process_1
  []
  ...
  []
  [cond_n]->process_n
  ```

  Die Bedingungen der Guards müssen nicht exklusiv sein. Alle Zweige, deren Guards zu logisch wahr evaluieren, stehen wie beim einfachen Choice-Operator zur Auswahl.

Parametrisierte Prozesse

Full LOTOS erlaubt bei Spezifikationen und Prozessen neben der Parametrisierung durch formale Gates (bereits in Basic LOTOS enthalten) auch die Parametrisierung durch formale Variablen. Die Variablen werden zusammen mit ihren Sorten in der Parameterliste in der Titelzeile deklariert und können dann in der Behaviour Expression, die das Verhalten des Prozesses bzw. der Spezifikation definiert, auftreten. Der nachfolgende Prozeß `LowerBound` wird parametrisiert durch die Variable `bound`, die die Untergrenze für die am Gate `b` ausgegebenen Werte darstellt:

```
process LowerBound[a, b](bound: nat) : noexit :=
   a ?x:nat; b !largest(x, bound); stop
endproc
```

Bei der Instanzierung parametrisierter Prozesse werden die Variablen durch die eingesetzten Ausdrücke (die selbst Variablen enthalten können) ersetzt, wobei die Sorten der Ausdrücke den in der Parameterliste deklarierten Sorten für die Variablen entsprechen müssen. Das Ersetzen von formalen durch aktuelle Parameter ist ein ähnlicher Vorgang wie Parameterübergabe in prozeduralen Programmiersprachen.

Der Prozeß `LowerBound` wird durch die Behaviour Expression

```
LowerBound[in, out](succ(succ(0)))
```

mit einer Untergrenze von 2 instanziert.

Funktionalität

In Basic LOTOS wird der `exit` Prozeß dazu benutzt, die erfolgreiche Terminierung einer Behaviour Expression zu spezifizieren. Durch die Hinzunahme von Datentypen in Full LOTOS ist es möglich, dem `exit` Prozeß eine Liste von Ausdrücken mitzugeben, die die darauffolgende, durch den `>>` Operator angehängte Behaviour Expression, verarbeiten kann. Die Liste der Sorten der an die Terminierung angehängten Ausdrücke wird auch *Funktionalität* dieser Terminierung genannt.

In Full LOTOS muß die Funktionalität von Spezifikationen und Prozessen in der Titelzeile spezifiziert werden. Sie kann eine von 3 möglichen Formen annehmen:

- `noexit`: der Prozeß terminiert nicht
- `exit`: der Prozeß terminiert
- `exit(t1,...,tn)`: der Prozeß terminiert und liefert n Ausdrücke mit den Sorten `t1,...,tn` an den nachfolgenden Prozeß

Bei der Definition der Behaviour Expression von Prozessen muß darauf geachtet werden, daß die angegebene Funktionalität auch wirklich erreicht wird. Für das Ableiten der Funktionalität aus Behaviour Expressions gibt es eigene Regeln, deren Erklärung hier aber zu sehr ins Detail gehen würde.

Um die Werte auf der anderen Seite des `>>` Operators entgegenzunehmen, wird dessen Syntax durch das `accept`-Statement folgendermaßen erweitert, sodaß die `x1,...,xn` in der Behaviour Expression `B2` zur Verfügung stehen:

```
B1 >> accept x1:t1,...,xn:tn in B2
```

`B1` muß die Funktionalität `t1,...,tn` haben.

Kapitel 2

Objektorientiertes LOTOS

LOTOS ist keine objektorientierte Sprache. Begriffe wie „Objekt", „Klasse" oder „Methode" besitzen keine direkte Entsprechung in der Sprache selbst. Da jedoch Objekte in der objektorientierten Theorie [Weg90] ebenso wie LOTOS-Prozesse konzeptuell als unabhängige Einheiten gesehen werden, die über Nachrichten miteinander kommunizieren, ist es naheliegend, objektorientierte Systeme mit LOTOS zu spezifizieren. Durch die Arbeiten von [May89, CRS90, Rud92] hat sich ein objektorientierter LOTOS-Stil herauskristallisiert, der objektorientierte Konzepte schablonenhaft Prozeß- und ADT-Spezifikationen zuordnet. Moreira und Clark haben diesen Programmierstil in ihrer Arbeit über rigorose objektorientierte Analyse (ROOA, [MC94]) weitgehend übernommen und zum Teil durch zusätzliche Regeln für seine Anwendung erweitert. Der Programmierstil kommt ohne die Einführung neuer Schlüsselwörter aus[1].

Im Hinblick auf Kapitel 3 wird an dieser Stelle der in der ROOA gebräuchliche Stil erläutert. Da einige Konzepte der Objektorientierung auch heute noch keine eindeutige Interpretation besitzen (z. B. Vererbung, Aggregation), wird die von Moreira und Clark verwendete Interpretation übernommen.

2.1 Objekte und Klassen

Objekte sind gekapselte Einheiten, die durch folgende Eigenschaften charakterisiert sind:

- *Objektidentität:* Jedes Objekt besitzt eine eindeutige Kennzeichnung bzw. Namen. Dadurch kann es von anderen Objekten der gleichen Klasse unterschieden werden.
- *Zustand:* Jedes Objekt verfügt über einen Zustand. Der Zustand eines Objekts ist gegeben durch die Werte seiner *Attribute* zusammen mit den Bedingungen, die angeben, welche Services ein Objekt gerade anbietet.
- *Services:* Jedes Objekt bietet seiner Umgebung Services an, die die einzige Möglichkeit sind, den Zustand eines Objekts abzufragen bzw. zu verändern. In der Literatur werden häufig drei Arten von Services unterschieden: *Konstruktoren* erzeugen

[1] In [May89] wurde der Versuch unternommen, den Programmierstil durch die Einführung neuer Schlüsselwörter explizit zu machen.

Objekte und initialisieren gewöhnlich deren Zustand. *Modifikatoren* können den Zustand eines bestehenden Objekts verändern. *Selektoren* geben Auskunft über den aktuellen Zustand eines Objekts, verändern diesen aber nicht. Die Realisierung eines Services wird *Methode* genannt.

Gleichartige Objekte werden in *Klassen* zusammengefaßt. Eine *Klassenschablone* spezifiziert Zustand und Services aller *Instanzen* (erzeugter Objekte) einer Klasse. D. h., um eine Klasse von Objekten zu spezifizieren, muß die entsprechende Klassenschablone angegeben werden. Das folgende Programmgerüst zeigt, wie Klassenschablonen als LOTOS-Prozesse definiert werden:

```
process Template[g](state: State_Template) : noexit :=
   (  g !selector_1 !Get_Id(state) ?param : Param_Type ... ;
      ...
      exit(state)
      []
      g !modifier_1 !Get_Id(state) ?param : Param_Type ... ;
      ...
      exit(F1(state))
      []
      ...
   ) >> accept update_state: State_Template in
        Template[g](update_state)
```

Die Klassenschablone der Klasse `Template` wird definiert als rekursiver Prozeß mit der Funktionalität `noexit`. Die oben genannten Eigenschaften von Objekten, nämlich Objektidentität, Zustand und Services, sind darin folgendermaßen enthalten:

- Die Identität eines Objektes wird zusammen mit der Zustandsinformation in der Variablen `state` übergeben und aus dieser mit der Funktion `Get_Id(state)` ausgelesen.
- Die gesamte Zustandsinformation eines Objekts wird dem Prozeß in der Variablen `state` übergeben. Ist es nicht möglich, die Zustandsinformation in einem einzigen ADT zusammenzufassen, kann diese auch in mehreren Variablen übergeben werden.
- Services werden am Gate `g` als strukturierte Aktionen zur Verfügung gestellt. Durch den Choice-Operator können mehrere Services gleichzeitig angeboten werden. Das Service selbst wird spezifiziert durch eine LOTOS Behaviour Expression. Mit dem `exit`-Statement am Schluß des Services wird die (aktualisierte) Zustandsinformation an den nächsten Aufruf des Prozesses weitergegeben.

Sollen von einer Klasse dynamisch mehrere Objekte erzeugt werden, verwendet man *Objekt-Generatoren*. Objekt-Generatoren sind rekursive Prozesse, die die Aufgabe haben, neue Objekte mit eindeutigen Objekt-Ids zu instanzieren. Der zur Klassenschablone `Template` gehörige Objekt-Generator hat die Form:

```
process Templates[a](ids: Template_Number_Set) : noexit :=
   a !create ?id_counter: Template_Number
     ?init_val1: Value_Sort1 [id_counter notin ids];
   (  Template[a] (Make_Template(id_counter, init_val1))
      |||
      Templates[a](Insert(id_counter, ids))
   )
where
   process Template[a](state: State_Template) : noexit :=
   ...
   endproc
endproc
```

Der Objekt-Generator `Templates` verwaltet in der Variablen `ids` die Menge der bereits vergebenen Objekt-Ids. Er wird durch Synchronisation am Gate `a` der Form

```
a !create ?id !init_val
```

dazu veranlaßt, ein neues Objekt zu generieren. Die Wertgenerierung im Parameter `?id` bzw. `?id_counter` und die Bedingung `[id_counter notin ids]` sorgen dafür, daß eine eindeutige Objekt-Id für das neue Objekt erzeugt wird. Anschließend wird das Objekt mit dieser Id und einem Initialisierungswert `init_val1` instanziert, während gleichzeitig der Objekt-Generator Prozeß rekursiv aufgerufen wird, um für weitere Instanzierungen zur Verfügung zu stehen.

Um Objekte einer Klasse zerstören zu können, fügt man in der entsprechenden Klassenschablone ein Service namens `remove` ein, das den Objekt-Prozeß mit dem `stop`-Befehl terminiert.

2.2 Objektidentität und -zustand

Der Zustand eines Objekts und seine Objekt-Id werden in objektorientiertem LOTOS als ADT repräsentiert.

Für die Definition von Objekt-Ids bzw. Mengen von Objekt-Ids werden die ADTs `Id_Type` und `Id_Set_Type` herangezogen, die beide Teil der ROOA Standard-Bibliothek sind. `Id_Type` definiert die Sorte `Id` mit einem Satz eindeutiger Ids und den wichtigsten Vergleichsoperatoren. `Id_Set_Type` baut auf dem in der LOTOS Standard-Bibliothek enthaltenen ADT `Set` auf und realisiert Mengen von Ids.

Damit die Objekt-Ids klassenübergreifend eindeutig sind, wird für jede Klasse ein eigener Objekt-Id-Typ erzeugt, was durch Type Renaming des `Id_Set_Type` leicht möglich ist. Das nachfolgende Beispiel zeigt den Vorgang exemplarisch für die Klasse `Template`:

```
type Template_Id_Set_Type is Id_Set_Type
   renamed by
   sortnames
      Template_Id      for Id
      Template_Id_Set  for Set
endtype
```

Die Zustandsinformation eines Objekts besteht in der Regel aus mehreren Attributen, die selbst Instanzen abstrakter Datentypen sind. Diese werden, wenn möglich, in einem einzigen ADT zusammengefaßt und dem Objekt-Prozeß bei jedem Aufruf in der Variablen `state` als Parameter übergeben. Das nachfolgende Programmgerüst zeigt den Aufbau des ADTs für die Klasse `Template`:

```
type State_Template_Type is Template_Id_Set_Type, Attribute_Type
   sorts State_Template
   opns
      Make_Template : Template_Id, Attribute -> State_Template
      Get_Id : State_Template -> Template_Id
      Get_Attribute : State_Template -> Attribute
      Set_Attribute : State_Template, Attribute -> State_Template
      ...
   eqns
      forall s: State_Template, i: Template_Id, a: Attribute
      ofsort Template_Id
         Get_Id(Make_Template(i, a))=i;
      ofsort Attribute
         Get_Attribute(Make_Template(i, a))=a;
      ...
endtype
```

Der ADT, der die Zustandsinformation eines Objekts definiert, kann gleichermaßen als eigenes Objekt gesehen werden. Er verfügt über Konstruktoren, die den Zustand initialisieren (`Make_Template`), Selektoren, die den Wert bestimmter Attribute zurückliefern (`Get_Attribute`) und Modifikatoren, die bestimmte Attribute setzen (`Set_Attribute`).

Da in LOTOS selbst nur wenige ADTs vordefiniert sind, kann die Definition der ADTs für oben genannte Zwecke schnell überhand nehmen. In ROOA werden deshalb nur die für eine ablauffähige Spezifikation erforderlichen Gleichungen angegeben, weitere Details in der Design-Phase festgelegt. Folgende Vereinfachungen können vorgenommen werden:

- Modifikatoren werden ohne Gleichungen angegeben. Dadurch werden sie zu Konstruktoren des ADT.
- Selektoren, die keinen bestimmten Wert zurückgeben müssen, werden durch Gleichungen definiert, die einen als unbestimmt anzusehenden, konstanten Wert zurückliefern.
- Selektoren, die einen bestimmten Wert zurückgeben müssen, werden definiert durch je eine Gleichung für jeden Konstruktor.

Die ADTs für die einzelnen Attribute der Zustandsinformation können ebenfalls nach obigen Regeln vereinfacht werden.

2.3 Services

Objekte bieten Services an und nehmen selbst Services anderer Objekte in Anspruch. Der Aufruf eines angebotenen Services entspricht einem Kommunikationsvorgang zwischen Objekten. Der Anbieter eines Services wird *Server* genannt, das aufrufende Objekt *Client*. Diese Rollenverteilung ist keine statische, sondern bezogen auf die aktuell stattfindende Kommunikation zu interpretieren: ein Server kann zur Erbringung seines Services die Services anderer Objekte aufrufen, wodurch er selbst temporär zum Client wird.

Kommunizierende Objekte sind über *Message Connections* miteinander verbunden. Eine Message Connection vom Objekt A zum Objekt B bedeutet, daß das Objekt A Services des Objekts B aufruft. Der eigentliche Vorgang des Aufrufs ist je nach Programmiersprache unterschiedlich, in objektorientiertem LOTOS werden Services durch Synchronisation an gemeinsamen Gates aufgerufen.

Wie in der Klassenschablone in Abschnitt 2.1 gezeigt, werden Services als strukturiertes Ereignis der Form

```
g !op !Get_Id(state) ?param: Param_Type
```

angeboten. Damit der Aufruf zustande kommt, muß auf der Client-Seite gleichzeitig der Event

```
g !op !id !value
```

angeboten werden. (Anm.: LOTOS macht bei Kommunikationsvorgängen an Gates keinen Unterschied zwischen aktiver und passiver Rolle; beide involvierten Prozesse müssen lt. Tabelle 1.3 bereit sein, an dem Ereignis teilzunehmen). Wertvergleiche von `op` und `id` stellen sicher, daß die Operation `op` im Objekt mit der Identität `id` aufgerufen wird.

Ein Server bietet normalerweise alle seine Services an einem einzigen Gate an. Manchmal ist es auch sinnvoll, Services zu gruppieren und an verschiedenen Gates anzubieten, vor allem dann, wenn es mehrere Gruppen von Clients gibt, die jeweils einen unterschiedlichen Satz Services aufrufen. Services selbst werden durch Behaviour Expressions spezifiziert. Jedes Service (ausgenommen `remove`) wird mit dem `exit`-Statement beendet, mit dem die (aktualisierte) Zustandsinformation über den `>>` Operator an den nächsten Prozeßaufruf weitergegeben wird.

Für den Fall, daß ein Service ein Resultat zurückliefert, kann auf zweierlei Arten vorgegangen werden:

1. Wenn das Resultat direkt aus der Zustandsinformation des Objekts mittels der Funktion `Get_Result(state)` berechnet werden kann, bietet der Server

   ```
   g !op !Get_Id(state) !Get_Result(state)
   ```

   und der Client

   ```
   g !op !id ?result: Result_Type
   ```

   an. Das Resultat des Aufrufs steht nach der Synchronisation beim Client in der Variablen `result` zur Verfügung.

2. Falls das gerufene Service komplexer aufgebaut ist und zur Berechnung des Resultats seinerseits Services von anderen Objekten aufruft, wird das Resultat über einen eigenen Return-Event zur Verfügung gestellt. Der zum Service `op` gehörende Return-Event heißt gewöhnlich `rtn_op`. Die zugehörigen Aufrufpaare haben die Form

   ```
   g !op !Get_Id(state) ?param: Param_Type
   ...
   g !rtn_op !Get_Id(state) !rtn_value
   ```

   beim Server und

   ```
   g !op !id !value
   g !rtn_op !id ?result: Result_Type
   ```

   beim Client.

Die Namen von Services sind in objektorientiertem LOTOS Konstanten der Sorte `Op_Name`, die durch den ADT `Op_Names` folgendermaßen definiert wird:

```
type Op_Names is
   sorts Op_Name
   opns
      construct, select, modify, ... : -> Op_Name
endtype
```

Die ADTs für die Parameter eines Services werden nach den am Ende von Abschnitt 2.2 gezeigten Regeln definiert.

2.4 Vererbung

Vererbung und Polymorphismus zählen neben Kapselung zu den wichtigsten Elementen objektorientierter Softwareentwicklung. Durch Vererbung ist es möglich, neue Klassen aus bestehenden Klassen abzuleiten. Die Definition des Begriffes Vererbung ist auch heute noch keine eindeutige. Grundsätzlich wird unterscheiden zwischen *inkrementeller Vererbung (incremental inheritance)* und *verhaltensmäßiger Vererbung (behavioural inheritance)* [Weg90, S. 37ff]. Für erstere ist Vererbung hauptsächlich ein Mittel, Programmcode wiederzuverwenden, indem gleichartiges Verhalten abgeleiteter Klassen in gemeinsamen Basisklassen zusammengefaßt wird. Von verhaltensmäßiger Vererbung spricht man, wenn sich Objekte der abgeleiteten Klasse überall so verhalten wie Objekte der Basisklasse. Für ein und dasselbe System von Klassen führen die beiden Definitionen normalerweise zu unterschiedlichen Klassenhierarchien (z. B. [Weg90, S. 43]). In der Praxis wird meistens inkrementelle Vererbung verwendet.

Objektorientierung und Nebenläufigkeit lassen sich aufgrund einiger Komplikationen mit der Vererbungsrelation leider nicht problemlos integrieren. Matsuoka und Yonezawa [MY93] haben dafür den Ausdruck *inheritance anomaly* geprägt. Der Kern ihrer Aussage ist, daß sich Synchronisationsbedingungen (das sind jene Bedingungen, die angeben, welche Services in welchem Zustand aufgerufen werden dürfen) aktiver Objekte nur eingeschränkt vererben lassen. Je nachdem, welche Veränderungen die abgeleitete Klasse am Zustand bzw. Verhalten der Oberklasse vornimmt, können nicht-triviale Neudefinitionen vererbter Methoden in allen Oberklassen notwendig werden, was dem Kapselungsprinzip objektorientierter Softwareentwicklung widerspricht. Inwieweit objektorientierte LOTOS-Spezifikationen von diesem Problem betroffen sind, wurde in der Literatur noch nicht diskutiert. Die Arbeiten von Nierstraß [Nie93] und Puntigam [Pun95] zeigen mögliche Entwicklungen im Bereich aktiver Objekte auf.

ROOA verwendet inkrementelle Vererbung, die in bezug auf LOTOS in [Rud92] einer genauen Untersuchung unterzogen wurde. Die Basisklassen verlieren bei Vererbung die Funktionalität `noexit` und werden dadurch zu abstrakten Basisklassen, d. h., es können keine Objekte dieser Klassen instanziert werden. Basisklassen erhalten dadurch folgende Form:

```
process Template[g](state: State_Template) : exit(State_Template) :=
   g !selector_1 !Get_Id(state) ... ;
   ...
   exit(state)
   []
   g !modifier_1 !Get_Id(state) ... ;
   ...
   exit(F1(state))
   []
   ...
endproc
```

Im einfachsten Fall wird eine Klasse durch neue Services erweitert:

```
process Extended_Template[g](state: State_Template) : noexit :=
   (  (Template[g](state)
         >> accept update_state: State_Template in exit(update_state)
      )
      []
      g !modifier_2 !Get_Id(state) ... ;
      ...
      exit(F2(state))
   ) >> accept update_state: State_Template in
        Extended_Template(update_state)
endproc
```

Die Klasse `Extended_Template` erweitert die Klasse `Template`, indem sie alternativ am Gate `g` die Services der Klasse `Template` und das neu hinzugekommene Service `modifier_2` anbietet. Die abgeleitete Klasse ist wieder konkret und besitzt die Funktionalität `noexit`. Auf dieselbe Art und Weise könnte man die Klasse `Template` durch neue Gates erweitern.

Komplizierter gestaltet sich die Vererbung, wenn bestehende Services neu definiert werden müssen. Dann definiert man zuerst eine Hilfsklasse, in der alle Services definiert werden, die beibehalten werden sollen (in diesem Fall `selector_1`):

```
process Modifier_Template[g](state: State_Template) :
        exit(State_Template) :=
   g !selector_1 !Get_Id(state) ... ;
   ...
   exit(any State_Template)
endproc
```

Das LOTOS-Schlüsselwort `any` erzeugt ein unbestimmtes Element der angegebenen Sorte, im obigen Beispiel `State_Template`. Im zweiten Schritt definiert man die abgeleitete Klasse mit dem neu definierten Service:

```
process Redefined_Template[g](state: State_Template) : noexit :=
   (  (Template[g](state)|[g]|Modifier_Template[g](state))
      []
      g !modifier_1 !Get_Id(state) ... ;
      ... (* modified implementation *)
      exit(F1a(state));
   ) >> accept update_state: State_Template
        in Redefined_Template[g](update_state)
endproc
```

Das Service `modifier_1` wird ersetzt durch einen `modifier_1` mit neuer Implementierung, `selector_1` wird angeboten wie bisher.

Will man Klassen ableiten, die die Zustandsinformation der Basisklasse erweitern, stößt man in LOTOS (eigentlich ACT-ONE) auf das Problem, daß sich bestehende Datentypen nur sehr eingeschränkt durch Sorten erweitern lassen. Die zusätzliche Zustandsinformation wird deshalb in der abgeleiteten Klasse als eigenständiger ADT spezifiziert und als eigener Parameter bei jedem Prozeßaufruf übergeben:

```
process Extented_Template[g](state: State_Template,
        ext_state: State_Extended_Template) : noexit :=
   (  (Template[g](state)
         >> accept update_state: State_Template in exit(update_state)
      )
      []
      g !modifier_2 !Get_Id(state) ... ;
      ...
      exit(F2a(state), F2b(ext_state))
   ) >> accept state: State_Template, ext_state: State_Extended_Template
        in Extended_Template[g](state, ext_state)
endproc
```

Der Zustand von Objekten der Klasse `Extended_Template` ist gegeben durch `state` und `ext_state`. Das `exit`-Statement am Ende einer Methode verändert sich entsprechend (siehe neu hinzugekommenes Service `modifier_2`), ebenso das `accept`-Statement am Ende der Klassendefinition.

Die drei beschriebenen Fälle von Vererbung können auch kombiniert werden.

2.5 Assoziationen

Eine *Assoziation* ist eine Relation zwischen Klassen, die angibt, wieviele Objekte einer Klasse mit wievielen Objekten einer anderen Klasse zu welchem Zweck in Verbindung stehen[2]. Assoziationen werden meistens nach ihrer *Kardinalität* eingeteilt, man unterscheidet zwischen 1:1-, 1:n- und m:n-Assoziationen (siehe Tabelle 2.1). Zur genaueren Angabe der Kardinalität können auf beiden Seiten auch Intervalle angegeben werden, z. B. setzt eine 0,1:5,n-Assoziation 0 oder 1 Objekt der Klasse A mit 5 oder mehr Objekten der Klasse B in Verbindung.

Kardinalität	Beschreibung
1:1	ein Objekt der Klasse A mit einem Objekt der Klasse B
1:n	ein Objekt der Klasse A mit beliebig vielen Objekten der Klasse B
m:n	beliebig viele Objekte der Klasse A mit beliebig vielen Objekten der Klasse B

Tabelle 2.1: Arten von Assoziationen

Assoziationen können, wie Objekte, über eigene Attribute verfügen, die die Assoziation näher charakterisieren.

Um einfache Assoziationen (d. h. solche, die über keine eigenen Attribute verfügen) in objektorientiertem LOTOS zu spezifizieren, werden die Objekt-Ids der beteiligten Objekte dem jeweils anderen Objekt als Parameter übergeben. Tabelle 2.2 schlüsselt auf, wie dieser Parameter für die jeweilige Kardinalität definiert wird. Eine 1:n-Assoziation wird dementsprechend auf der einen Seite (enthaltendes Objekt) als Menge von Objekt-Ids definiert, auf der anderen Seite (enthaltenes Objekt) als einzelne Objekt-Id.

ein Objekt der Klasse A assoziiert mit...	Parameter-Typ
0 oder mehr Objekten der Klasse B	Menge von Objekt-Ids der Klasse B
genau einem Objekt der Klasse B	einzelne Objekt-Id der Klasse B

Tabelle 2.2: Spezifikation von Assoziationen

Im nachfolgenden Beispiel wird eine 1:n-Assoziation zwischen den Klassen `Template1` und `TemplateN` spezifiziert. Die zugehörigen Klassenschablonen sehen folgendermaßen aus:

```
process Template1[g](state: State_Template1, objs: TemplateN_Id_Set) :
        noexit :=
   (
      ...
   ) >> accept update_state: State_Template1, objs: TemplateN_Id_Set
        in exit(update_state, objs)
endproc
```

bzw.

[2]Klassen können auch zu sich selbst assoziiert sein. In diesem Fall spricht man von einer *unären* Assoziation.

```
process TemplateN[g](state: State_TemplateN, obj: Template1_Id) :
        noexit :=
   (
      ...
   ) >> accept update_state: State_TemplateN, obj: Template1_Id
        in exit(update_state, obj)
endproc
```

Oft ist es jedoch nicht notwendig, daß das enthaltene Objekt über das enthaltende bescheid weiß. In diesem Fall kann die Angabe der Objekt-Id beim enthaltenen Objekt (im obigen Beispiel `TemplateN`) wegfallen.

Assoziationen, die selbst über Attribute verfügen, werden als eigenständige Objekte realisiert, die die Objekt-Ids der beiden beteiligten Objekte enthalten.

Die Art und Weise, wie Assoziationen implementiert werden (ob z. B. als Liste oder als Hash-Tabelle), wird während der Spezifikation noch außer acht gelassen.

2.6 Aggregation

Den Aufbau komplexer Objekte aus einfachen Objekten nennt man *Aggregation*. Ein *Aggregat* ist ein komplexes Objekt, das aus mehreren Teilobjekten, den *Komponenten*, zusammengesetzt ist. Durch das Anbieten eigener Services bzw. Verwalten eigener Zustandsinformation wird aus dem Aggregat mehr als die Summe seiner Teile. Aggregation ist im Prinzip eine „besteht aus“-Assoziation und wird in LOTOS auch ähnlich wie eine Assoziation spezifiziert. Aggregation ist ein Mittel zur Strukturierung größerer Systeme. Deshalb ist es sinnvoll, sie als eigenständiges Konzept anzusehen und als solche in der Spezifikation sichtbar zu machen.

Man unterscheidet zwei Arten von Komponenten:

1. *Versteckte Komponenten (hidden components):* Sie sind den Objekten außerhalb des Aggregates nicht zugänglich. Kommunikation nach außen und von außen geschieht indirekt über das Aggregat. Die Komponenten des Aggregates können jedoch untereinander kommunizieren, ohne daß das Aggregat davon erfahren muß. Bei der Zerstörung des Aggregates werden versteckte Komponenten automatisch gelöscht.

2. *Zugängliche Komponenten (shared components):* Sie bleiben selbständige Objekte, die untereinander oder mit Objekten außerhalb des Aggregates kommunizieren können. Bei der Zerstörung des Aggregates werden zugängliche Komponenten nicht automatisch gelöscht.

Aggregationen werden als Assoziationen zwischen dem Aggregat und seinen Komponenten spezifiziert. Die Spezifikation zugänglicher Komponenten unterscheidet sich nicht von anderweitig assoziierten Objekten. Sie laufen wie normale Assoziationen unabhängig vom Aggregatprozeß. Versteckte Komponenten werden als Subprozesse des Aggregates definiert. Die Gates für die Kommunikation zwischen Aggregat und versteckten Komponenten werden durch den `hide`-Operator für außenstehende Objekte unsichtbar gemacht. Ist die Anzahl der Komponenten dynamisch, werden auch die Objekt-Generatoren als Subprozesse definiert.

Kapitel 3

Rigorose objektorientierte Analyse mit LOTOS

Im vorigen Kapitel wurde gezeigt, wie objektorientierte Systeme mit LOTOS spezifiziert werden können. Zur Spezifikation größerer Systeme benötigt man allerdings die systematische Herangehensweise der objektorientierten Analyse (OOA). Das Ziel der in diesem Kapitel vorgestellten rigorosen objektorientierten Analyse (ROOA) ist die Integration von OOA und formaler Spezifikation.

3.1 OOA und ROOA

OOA

Objektorientierte Analyse [CY91, RBP+91, JCJO92, Boo94] befaßt sich mit der Erstellung eines objektorientierten Modells des untersuchten Problembereichs:

> „OOA is the challenge of understanding the problem domain, and then the system's responsibilities in that light.“ [CY91, S. 9].

> „Analysis [...] is concerned with devising a precise, concise, understandable, and correct model of the real-world.“ [RBP+91, S. 148].

Um ein System zu beschreiben, schlagen die meisten OOA-Methoden die Erstellung von drei Modellen vor, die jeweils unterschiedliche Aspekte behandeln:

statisches Modell (Objekt-Modell): zeigt, aus welchen und wievielen Objekten ein System besteht, welche Attribute die Objekte besitzen und welche Services sie anbieten. Weiters gibt es statische Relationen zwischen den Objekten an, z. B. Vererbung, Assoziation, Aggregation, Subsystem-Bildung.

dynamisches Modell: zeigt das Verhalten des Systems zur Laufzeit. Ausgehend von Interface-Szenarien wird die Aufrufreihenfolge von Services innerhalb des Systems festgelegt.

funktionales Modell: beschreibt die Transformation der Daten bei der Erbringung von Services.

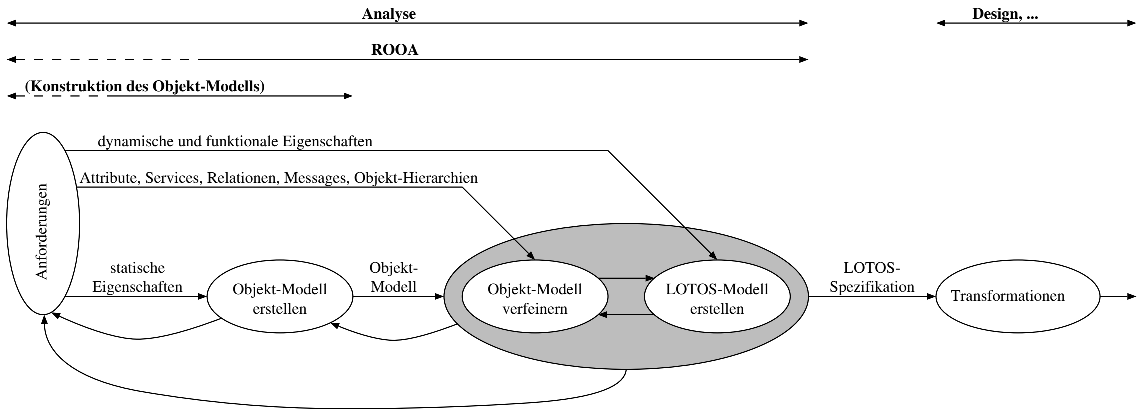

Abbildung 3.1: ROOA in der Softwareentwicklung

Das Hauptaugenmerk der „klassischen" Methoden liegt beim statischen Modell, die beiden anderen Modelle spielen eher untergeordnete Rollen. Die Dreiteilung in statisches, dynamisches und funktionales Modell hat den Nachteil, daß es schwierig ist, sie zu integrieren bzw. konsistent zu halten.

ROOA

ROOA (Rigorous Object-Oriented Analysis) erweitert die oben genannten Methoden durch die Integration einer formalen Spezifikationssprache in den Analyseprozeß. Die Methode produziert aus den statischen Eigenschaften des Objekt-Modells und den dynamischen und funktionalen Eigenschaften aus dem Anforderungsdokument eine formale objektorientierte Spezifikation. Diese ([Mor94, S. 97]):

1. formalisiert das Objekt-Modell, indem sie die Klassenschablonen und Beziehungen mathematisch definiert,
2. stattet jede Klassenschablone mit Zustandsinformation aus und beschreibt die Reihenfolge der Ereignisse,
3. zeigt die Message Connections zwischen Objekten und die zwischen Objekten kommunizierte Information,
4. definiert das Verhalten des gesamten Systems, indem sie die Klassen miteinander verbindet,
5. ist formal und exekutierbar. Rapid Prototyping kann benutzt werden, um einerseits die Spezifikation gegenüber den Anforderungen und andererseits die Anforderungen selbst auf Inkonsistenzen, Lücken und Zweideutigkeiten hin zu überprüfen.

Abbildung 3.1 [Mor94, S. 99] zeigt die ROOA im Gesamtkontext der Softwareentwicklung. Außerdem zeigt sie die schrittweise Entwicklung der Spezifikation aus dem Anforderungsdokument, in dem das System informell beschrieben ist. Die ROOA-Methode beinhaltet 3 Hauptschritte:

Schritt 1: Objekt-Modell erstellen
Das Objekt-Modell wird mit einer der bekannten OOA-Methoden (siehe Unterabschnitt *OOA*) erstellt.

Schritt 2: Objekt-Modell verfeinern
Das in Schritt 1 erstellte Objekt-Modell wird zunächst vervollständigt, um die Unterschiede der verschiedenen OOA-Methoden auszugleichen. Weiters wird mit Hilfe von Interface-Szenarien ein initiales dynamisches Modell des Systems entwickelt, mit dem zusätzliche Services und Message Connections identifiziert werden. Am Schluß von Schritt 2 wird eine erste Strukturierung in Aggregate und Subsysteme vorgenommen.

Schritt 3: LOTOS-Spezifikation erstellen
Ausgehend von Schritt 2 werden die einzelnen Klassen systematisch in LOTOS spezifiziert. Die resultierende formale Spezifikation ist exekutierbar und integriert das statische, dynamische und funktionale Modell. Die bei der Ausführung der Spezifikation gewonnenen Resultate führen meistens zu weiteren Iterationen der Schritte 2 und 3, solange bis eine zufriedenstellende Spezifikation des Gesamtsystems erreicht ist.

Die von den Anforderungen ausgehenden Pfeile in Abbildung 3.1 zeigen, welche Information aus dem Anforderungsdokument in welchem Schritt verwendet wird. Die rückwärtsgerichteten Pfeile in der Abbildung deuten an, daß es sich bei der ROOA um einen iterativen Prozeß handelt: während die Analyse durchgeführt wird, wird das Verständnis für das Problem besser. Die in einem früheren Schritt getroffenen Entscheidungen erscheinen in neuem Licht und müssen möglicherweise verändert werden. Es kann sogar sein, daß durch die formale Spezifikation Auslassungen, Inkonsistenzen oder Mehrdeutigkeiten im ursprünglichen Anforderungsdokument sichtbar werden. Dann sind dort ebenfalls die entsprechenden Berichtigungen vorzunehmen. Insbesondere gilt das für die Schritte 2 und 3, wo die endgültige Spezifikation durch mehrere Iterationen des Analyse-Spezifikation-Prototyping Zyklus hinweg immer weiter verfeinert wird.

Das Ergebnis der ROOA ist eine ablauffähige LOTOS-Spezifikation. Über mehrere Designschritte (Transformationen), in denen der Spezifikation sukzessive weitere Details hinzugefügt werden, kann eine den ursprünglichen Anforderungen entsprechende Implementierung erreicht werden. Hier kommt einer der Hauptvorteile einer formalen Spezifikation zum Tragen: bei jedem Transformationsschritt kann verifiziert werden, ob die daraus gewonnene genauere Spezifikation die vorhergehende erfüllt [Tur90].

An dieser Stelle sei darauf hingewiesen, daß ROOA nicht von LOTOS als Spezifikationssprache abhängig ist. Prinzipiell kann auch eine andere formale Spezifikationssprache in den ROOA-Prozeß integriert werden (so geschehen in [CM98]). Sie muß nach [Mor94, S. 21] folgende Eigenschaften erfüllen:

- sie ist ISO-Standard
- objektorientierte Spezifikationen können erstellt werden
- Entwicklungswerkzeuge existieren

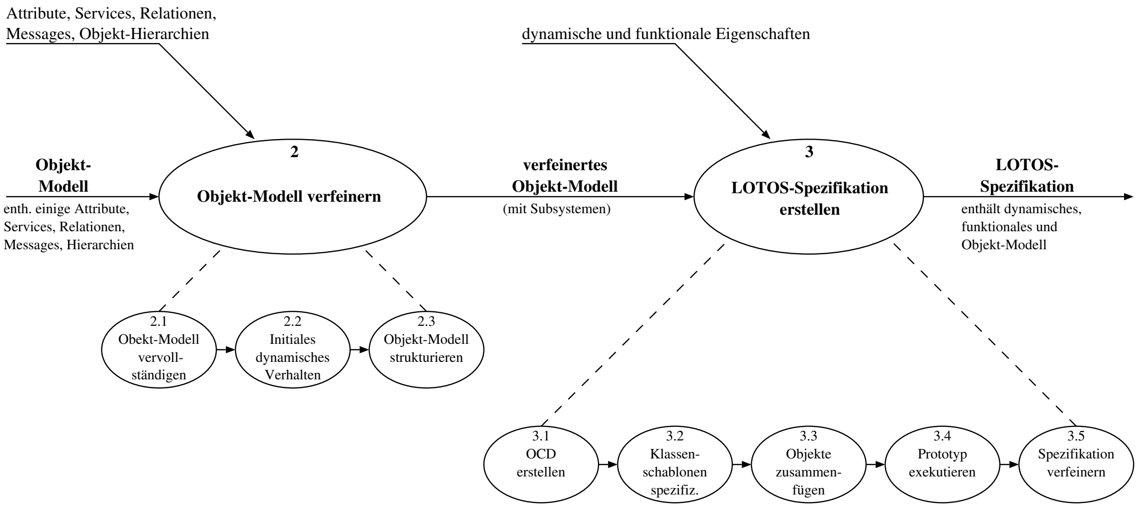

Abbildung 3.2: ROOA-Methode

- die Spezifikation ist exekutierbar und kann als Prototyp verwendet werden
- Nebenläufigkeit wird unterstützt

Abbildung 3.2 [Mor94, S. 100] zeigt den in Abbildung 3.1 grau hinterlegten Hauptteil der ROOA detaillierter, indem für die Schritte 2 und 3 die entsprechenden Unterschritte angegeben werden. Die Abschnitte 3.2 bis 3.4 dieses Kapitels erläutern ausführlich die Vorgehensweise in den einzelnen Schritten und Unterschritten.

Beispiel

Zur Demonstration der Methode verwenden wir ein praktisches Beispiel aus dem Bankwesen. Spezifiziert werden soll ein System mit folgenden Anforderungen [Mor94, S. 35][1]:

> Clients may take money from their accounts, deposit money or ask for their current balance. All these operations are accomplished using either automatic teller machines or counter tellers. Transactions on an account may be done by cheque, standing order, or using the teller machine and card. There are two kinds of accounts: savings accounts and cheque accounts. Savings accounts give interest and cannot be accessed by the automatic tellers.

3.2 Schritt 1: Objekt-Modell erstellen

Im ersten Schritt wird aus dem Anforderungsdokument mit einer der „klassischen“ OOA-Methoden (siehe Abschnitt 3.1) ein Objekt-Modell erstellt. Das Auffinden der für das System relevanten Objekte und ihrer Beziehungen untereinander stellt den schwierigsten Teil dieses Schrittes dar. Die AutorInnen der verschiedenen OOA-Methoden haben zwar ihrer

[1] Wiedergegeben in englischer Sprache, damit der Bezug zu den Klassen- und Service-Bezeichnungen in den nachfolgenden Abschnitten leichter hergestellt werden kann.

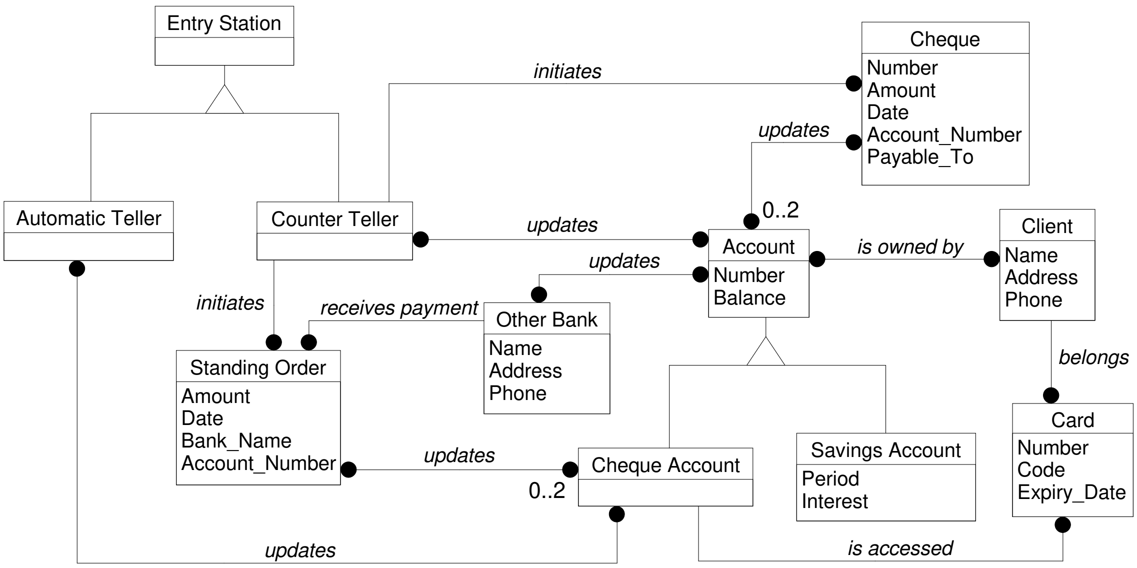

Abbildung 3.3: Einfaches Objekt-Modell mit OMT

eigenen Praxis Richtlinien darüber entnommen, dennoch hängt das Ergebnis stark von anderen Faktoren ab, wie z. B. Erfahrung mit objektorientierten Methoden oder Kenntnis des Problembereichs. Die Erstellung des initialen Objekt-Modells ist ein von der ROOA unabhängiger Schritt und kann von einem eigenen EntwicklerInnen-Team durchgeführt werden. Während der Anwendung der ROOA werden Anforderungsdokument und Objekt-Modell möglicherweise verändert. Abbildung 3.3 [Mor94, S. 102] zeigt ein einfaches, mit OMT entworfenes Objekt-Modell für das Banken-Beispiel.

3.3 Schritt 2: Objekt-Modell verfeinern

In Schritt 2 wird das in Schritt 1 erstellte Objekt-Modell zuerst vervollständigt, indem sichergestellt wird, daß es Interface-Objekte und statische Relationen enthält und außerdem alle Objekte über die wichtigsten Attribute und Services verfügen. Damit werden die Unterschiede in den Objekt-Modellen der verschiedenen OOA-Methoden ausgeglichen. Den Hauptteil des Schrittes macht die Erstellung eines dynamischen Modells der Spezifikation aus, mit dessen Hilfe zusätzliche Attribute und Services im Objekt-Modell identifiziert werden können.

Das Ergebnis von Schritt 2 ist ein verfeinertes Objekt-Modell, das folgende Elemente enthält:

- Interface-Objekte
- Statische Relationen (Klassenhierarchien, Assoziationen, Aggregationen, Subsysteme)
- Attribute und Services in jeder Klassen
- Message Connections

Das resultierende Objekt-Modell wird zusammen mit dem dynamischen Modell in Schritt 3 für die Erstellung der LOTOS-Spezifikation herangezogen.

Schritt 2.1: Objekt-Modell vervollständigen

Interface-Objekte hinzufügen. Falls nicht bereits vorhanden, werden dem Objekt-Modell Interface-Objekte hinzugefügt. Diese bilden die Schnittstelle zwischen dem System und seiner Umgebung. Die von außen kommenden Anfragen werden durch Interface-Objekte in Systemereignisse umgewandelt und umgekehrt Rückmeldungen des Systems in eine für BenutzerInnen verständliche Form gebracht.

Statische Relationen hinzufügen. Als nächstes werden statische Relationen zwischen Objekten (Klassenhierarchien, Assoziationen und Aggregationen) im Objekt-Modell identifiziert und in dieses eingetragen.

Attribute und Services hinzufügen. Zuletzt werden alle Objekte im Modell mit Attributen und Services ausgestattet. Als Faustregel gilt, daß die Menge der Attribute während der Analyse relativ stabil ist, hingegen die Services sich häufiger ändern. Man kann sich deshalb in diesem Schritt auch auf das Hinzufügen „offensichtlicher“ Services beschränken und die restlichen Services in Schritt 2.2 identifizieren, wo zum ersten Mal das dynamische Verhalten des Systems ermittelt wird.

Schritt 2.2: Initiales dynamisches Verhalten

In diesem Schritt wird begonnen, das dynamische Verhalten des Systems zu beschreiben, mit dem Ziel, weitere Services in den Objekten zu identifizieren. Dazu werden aus *Interface-Szenarien*, das sind beispielhafte Interaktionsvorgänge an den Interface-Objekten, sogenannte *Event Trace Diagramme (ETDs)* erstellt, mit denen Aufrufreihenfolgen innerhalb des Systems betrachtet werden. Die in den ETDs enthaltene Information wird anschließend in einer eigenen Tabelle, der *Object Communication Table (OCT)*, zusammengefaßt. Am Ende des Schrittes werden noch die Message Connections ins Objekt-Modell eingetragen.

Interface-Szenarien erstellen. Ein Interface-Szenario vollzieht einen typischen Interaktionsvorgang mit dem System nach. Es wird dargestellt durch eine Serie von Aufrufen an die Interface-Objekte, zusammen mit den erwarteten Ergebnissen. Jedes Interface-Szenario beschreibt einen bestimmten Teil der (nach außen sichtbaren) Funktionalität des Systems. In Schritt 3 werden Interface-Szenarien in LOTOS-Prozesse umgesetzt. Mit ihnen kann die Spezifikation des Systems getestet werden.

ETDs und initiale OCT erstellen. Event Trace Diagramme gehen von einem bestimmten Interface-Szenario aus, geben aber im Unterschied zu diesem auch die Aufrufreihenfolgen von Services innerhalb des Systems wieder. Aus diesem Grund eignen sich ETDs besonders dazu, die Services der einzelnen Objekte zu identifizieren. Die ETDs können außerdem in Schritt 3 beim Spezifizieren der Services als Vorlage dienen. Abbildung 3.4

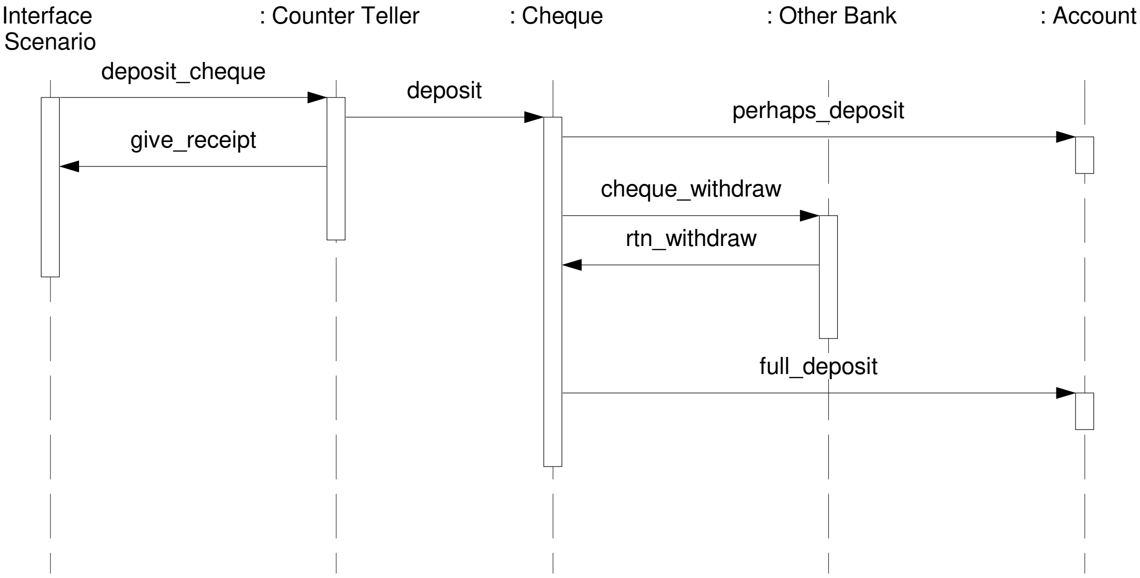

Abbildung 3.4: Event Trace Diagramm für eine Einzahlung

[Mor94, S. 105] zeigt das ETD für die Einzahlung (deposit) eines Schecks (cheque), der zu einer anderen Bank gehört.

Die Object Communication Table (OCT) stellt eine tabellarische Zusammenfassung der ETDs dar, mit dem Zweck, eine Übersicht darüber zu bekommen, welche Objekte welche Services von anderen Objekten aufrufen. Sie enthält in ihrer endgültigen Fassung fünf Spalten:

1. Spalte: eine Liste der Klassen aus dem Objekt-Modell

2. Spalte: für jede Klasse aus Spalte 1 eine Liste der angebotenen Services

3. Spalte: für jedes Service aus Spalte 2 eine Liste der verwendeten Services

4. Spalte: für jedes Service aus Spalte 2 eine Liste der Clients

5. Spalte: das Gate, an dem das Service aus Spalte 2 angeboten wird

In diesem Schritt werden nur die ersten beiden Spalten eingetragen, indem die entsprechenden Informationen dem Objekt-Modell und den ETDs abgelesen werden. Die 3. und 4. Spalte werden im nächsten Unterschritt (*Message Connections hinzufügen*) ausgefüllt, die 5. Spalte in Schritt 3.1. Tabelle 3.1 zeigt eine initiale OCT für das Bank-Beispiel (vollständig ausgeführt in [Mor94, S. 107]).

Message Connections hinzufügen. Nach dem Erstellen der ETDs werden die Message Connections als gerichtete Kanten ins Objekt-Modell eingetragen. Ruft ein Objekt der Klasse A ein Service eines Objekts der Klasse B auf, wird dies durch eine Kante von A nach B gekennzeichnet.

Simultan werden mit Hilfe der ETDs in der OCT die Spalten 3 und 4 ausgefüllt. Der Eintrag *Interface_Scenario* in der *Clients*-Spalte bedeutet, daß das angebotene Service von

Klasse	angebotene Services	verwendete Services	Clients
Entry_Station (ES)	withdraw_cash	Account.withdraw	Interface_Scenario
Counter_Teller (CT)	open_account	Account.create	Interface_Scenario
	close_account	Account.remove	Interface_Scenario
	deposit_cash	Account.deposit	Interface_Scenario
	give_balance	Account.get_balance	Interface_Scenario
	deposit_cheque	Cheque.deposit	Interface_Scenario
	ask_transfer	Account.withdraw	Interface_Scenario
		Account.deposit	
		OB.send_transfer	
	set_standing_order	SO.create	Interface_Scenario
	cancel_standing_order	SO.cancel	Interface_Scenario
Automatic_Teller (AT)	mini_statement	Cheque_Account.print_mini	Interface_Scenario
Other_Bank (OB)	receive_transfer	Account.deposit	Interface_Scenario
	send_transfer		CT, SO
	cheque_withdraw		Cheque
	remote_withdraw	Account.withdraw	
Standing_Order (SO)	create		CT
	cancel		CT
	debit	Account.withdraw	internal
		Account.deposit	
		OB.send_transfer	
...	...	...	...

Tabelle 3.1: Initiale Object Communcation Table

einem Interface-Szenario aufgerufen wird, d. h., in weiterer Folge ein Service darstellt, das BenutzerInnen direkt aufrufen können.

Schritt 2.3: Objekt-Modell strukturieren

In diesem Schritt wird das Objekt-Modell durch die Identifikation von Aggregaten und Subsystemen übersichtlicher strukturiert. Dies ist besonders wichtig bei großen und komplexen Systemen. Da die Struktur eines Systems anfangs noch nicht klar herausgebildet ist, sollten im ersten Iterationsschritt nur offensichtliche Strukturierungen vorgenommen werden.

Abbildung 3.5 [Mor94, S. 109] zeigt das verfeinerte Objekt-Modell, welches die am Anfang des Abschnitts angeführten Elemente enthält. Auf das Einzeichnen der Message Connections mußte verzichtet werden, weil das Entwurfswerkzeug dazu nicht in der Lage war.

3.4 Schritt 3: LOTOS-Spezifikation erstellen

In Schritt 3 wird aus dem in Schritt 2 gewonnenen Objekt-Modell eine ablauffähige LOTOS-Spezifikation erzeugt. Diese vereint die statischen, dynamischen und funktionalen Aspekte des Systems und kann mit den entsprechenden Werkzeugen auch ausgeführt werden.

Schritt 3 durchläuft zusammen mit Teilen von Schritt 2 mehrere Iterationen. In der ersten Iteration wird versucht, relativ rasch eine erste, vereinfachte Spezifikation zum Laufen zu bringen. Dazu wird angenommen, daß von jeder Klasse nur ein Objekt instanziert

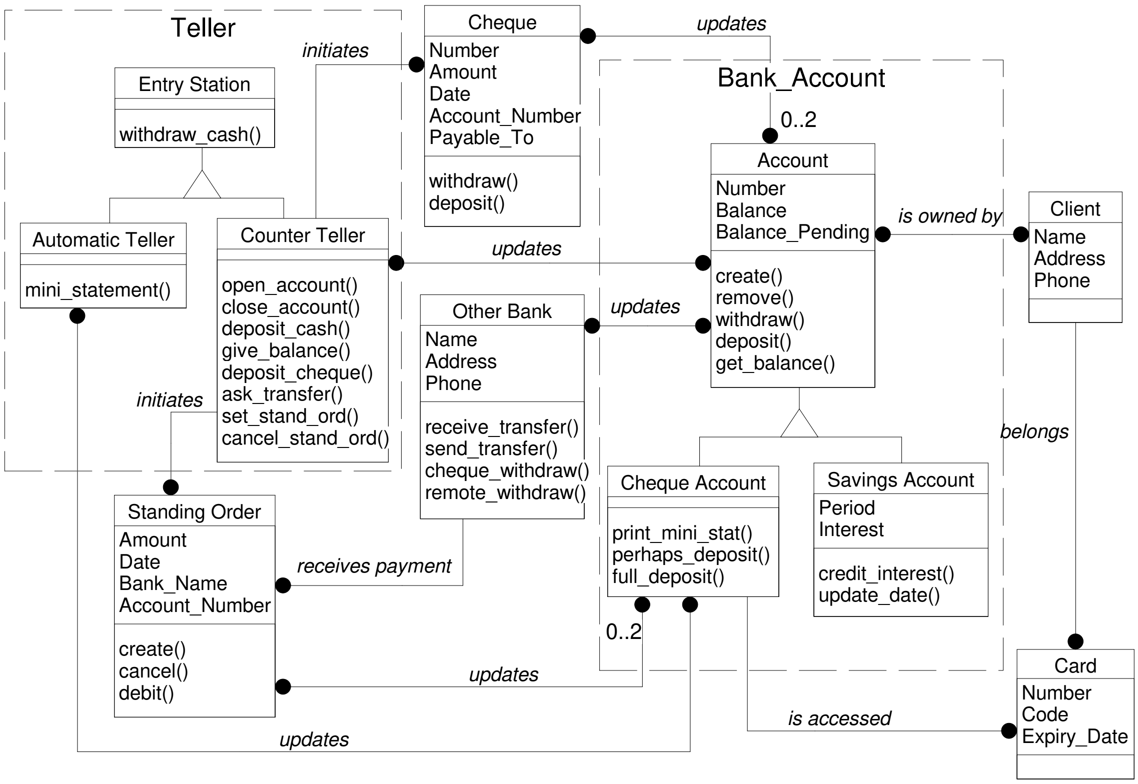

Abbildung 3.5: Verfeinertes Objekt-Modell

wird. Die weiteren Iterationen verfeinern ausgehend von Schritt 3.5 die Spezifikation, bis ein vollständiges, detailliertes Modell des Systems erreicht ist.

Schritt 3.1: OCD erstellen

Das *Object Communication Diagram (OCD)* dient als grafische Übersicht über die Spezifikation. Es zeigt einerseits die Hierarchie der Prozeßdefinitionen, andererseits die Gates, über welche die einzelnen Prozesse miteinander kommunizieren.

Das OCD ist ein Graph, in dem die Klassen die Knoten darstellen und die Message Connections die Kanten. OCD und OCT sind eng miteinander verknüpft: während der Erstellung des OCD aus dem OCT wird auch die fünfte Spalte des OCT gefüllt. Das OCD wird nach folgendem Algorithmus aus dem OCT und dem Objekt-Modell erstellt:

1. Knoten zeichnen:
   Für jede Klasse in der ersten Spalte der OCT wird ein Knoten gezeichnet. Weiters wird für jede Gruppierung von Objekten (Aggregation oder Subsystem) ein Knoten gezeichnet, der die in der Gruppierung enthaltenen Knoten umschließt.

2. Knoten durch Kanten verbinden:
   Jede Message Connection zwischen zwei Klassen wird als Kante in das OCD eingetragen. Die Message Connection zu einem Objekt einer Gruppierung wird als Kante zum übergeordneten Objekt repräsentiert. In der ersten Iteration kann es

vorkommen, daß einige Knoten ohne Kanten bleiben. Diese Knoten werden entweder in einer späteren Iterationen mit Kanten verbunden oder fallen aus dem Modell heraus.

3. Kanten beschriften:
   Dieser Schritt wird für jede Klasse in der ersten Spalte der OCT durchgeführt.

   (a) Spalte 5 in der OCT vervollständigen:
   Aus den Spalten 2 (angebotene Services) und 4 (Clients) wird der Gate-Name bestimmt:

   - den selben Gate-Namen vergeben, wenn Clients die gleiche Menge oder Untermenge von Services verwenden
   - unterschiedliche Gate-Namen vergeben, wenn Clients einen verschiedenen Satz Services verwenden

   Diese beiden Regeln ergeben sich aus der Anforderung, daß ein Objekt für die Kommunikation mit Objekten auf verschiedenem Abstraktionsniveau nicht dasselbe Gate benutzen kann.

   (b) Kanten mit Gate-Namen beschriften:
   Die Kanten im OCD werden mit den entsprechenden Gate-Namen aus Spalte 5 der OCT beschriftet.

Abbildung 3.6 [Mor94, S. 112] und Tabelle 3.2 [Mor94, S. 111] zeigen die nach Anwendung der bisherigen Schritte entstandenen OCD bzw. OCT für das Bank-Beispiel.

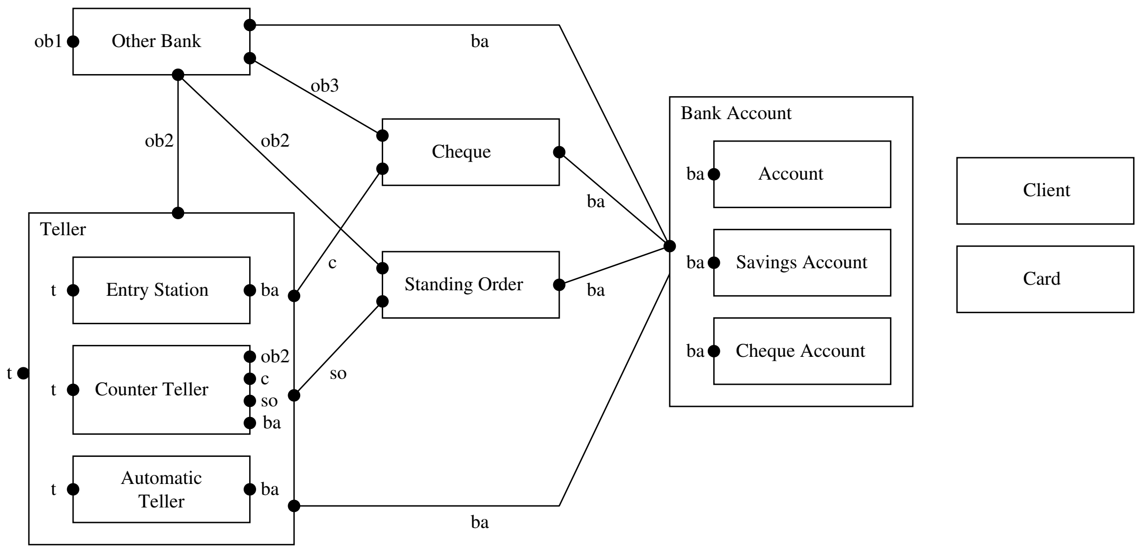

Abbildung 3.6: Initiales Object Communication Diagram

4. Oberklassen entfernen:
   Oberklassen sind in objektorientiertem LOTOS immer abstrakt. Deshalb werden sie vom OCD entfernt. Das führt dazu, daß im OCD nur noch die Blatt-Klassen des Klassenbaumes enthalten sind.

Klasse	**angebotene Services**	**verwendete Services**	**Clients**	**Gate**
Teller	open_account(CT)	BA.create	Interface_Scenario	t
[ES + CT + AT]	close_account(CT)	BA.remove	Interface_Scenario	t
	deposit_cash(CT)	BA.deposit	Interface_Scenario	t
	withdraw_cash(ES)	BA.withdraw	Interface_Scenario	t
	give_balance(CT)	BA.get_balance	Interface_Scenario	t
	deposit_cheque(CT)	Cheque.deposit	Interface_Scenario	t
	mini_statement(AT)	BA.withdraw	Interface_Scenario	t
	ask_transfer(CT)	BA.withdraw	Interface_Scenario	t
		BA.deposit		
		OB.send_transfer		
	set_standing_order(CT)	SO.create	Interface_Scenario	t
	cancel_standing_order(CT)	SO.cancel	Interface_Scenario	t
Other_Bank(OB)	receive_transfer	BA.deposit	Interface_Scenario	ob1
	send_transfer	Teller(CT), SO		ob2
	cheque_withdraw	Cheque		ob3
	remote_withdraw	BA.withdraw	Interface_Scenario	ob1
Standing_Order(SO)	create		Teller(CT)	c
	cancel		Teller(CT)	c
	debit	BA.withdraw	internal	
		BA.deposit		
		OB.send_transfer		
...	...	...	...	...

Tabelle 3.2: Vollständige Object Communication Table

5. Zusätzliche Kanten hinzufügen:
   Während der Spezifikation jedes einzelnen Objektes können neue Message Connections identifiziert werden. Wenn das der Fall ist, mit 2. und 3. fortfahren.

In den nachfolgenden Iterationen können weitere Änderungen am OCD notwendig werden:

6. Objekt-Generatoren einfügen:
   Objekt-Generatoren (siehe Abschnitt 2.1) werden dort eingefügt, wo von einer Klassenschablone mehrere Instanzen erzeugt werden sollen. Als Regel gilt, daß Objekt-Generatoren im Plural bezeichnet werden, einzelne Objekte im Singular.

7. Neue Objekt-Gruppierungen einfügen:
   Neue Objekt-Gruppierungen führen dazu, daß Kanten zusammengefaßt werden können, und zwar derart, daß alle Objekte der Gruppierung über gemeinsame Gates mit den anderen Objekten kommunizieren. Um Namenskonflikte bei den Services zu vermeiden, werden bei den Namen der angebotenen Services in der OCT (Spalte 2) die Namen der anbietenden Klasse in Klammern abgekürzt angegeben (bereits ausgeführt in Tabelle 3.2).

Schritt 3.2: Klassenschablonen spezifizieren

In diesem Schritt werden anhand des in Schritt 3.1 gewonnenen OCD und der OCT sowie des Objekt-Modells und einiger Event Trace Diagramme aus Schritt 2 die Klassenschablonen des Systems spezifiziert. Da das Ziel der ROOA eine formale Spezifikation des

Systems ist, schlagen die AutorInnen der Methode vor, sich in der ersten Iteration des Analyseprozesses nicht zu lange in den Schritten 2.2 und 3.1 aufzuhalten, sondern relativ rasch eine erste lauffähige Spezifikation zu produzieren. Ausgehend von dieser könne das Modell dann zielgerichteter verfeinert werden.

Klassenschablonen werden, wie in Kapitel 2 beschrieben, durch einen Prozeß und mehrere ADTs definiert, passive Hilfsobjekte nur durch ADTs. Die Reihenfolge bei der Definition der einzelnen Prozesse und ADTs spielt keine Rolle. In der ersten Iteration kann außerdem davon ausgegangen werden, daß von jeder Klassenschablone nur ein Objekt erzeugt wird, d. h., es müssen keine Objekt-Generatoren spezifiziert werden.

Dynamisches Verhalten spezifizieren. Zur Spezifikation des dynamischen Verhaltens einer Klasse werden die Spalten 2 (angebotene Services) und 3 (verwendete Services) der OCT herangezogen. Aus ersterer wird abgelesen, welche Services für die Klassenschablone implementiert werden müssen. Die Liste der verwendeten Services in Spalte 3 wird zusammen mit den ETDs verwendet, um die LOTOS Behaviour Expression für das Service abzuleiten. Die Namen für die Gates, an denen die für die Aufrufe der Services notwendige Kommunikation stattfindet, können aus Spalte 5 der OCT abgelesen werden.

ADTs spezifizieren. ADTs werden hauptsächlich zur Spezifikation der Zustandsinformation einer Klasse und der Operationen, die darauf möglich sind, verwendet. Des weiteren dienen sie zur Spezifikation der Datentypen, die als Parameter den Services übergeben werden sollen. Passive Hilfsobjekte werden ebenfalls als ADT spezifiziert.

Um die Spezifikation der ADTs möglichst einfach zu halten, sollten nach den Regeln in Abschnitt 2.2 nur die für eine ablauffähige Spezifikation notwendigen Gleichungen angegeben werden.

Schritt 3.3: Objekte zusammenfügen

Anhand der im OCD definierten Struktur werden die in Schritt 3.2 definierten Prozesse zu einer ablauffähigen Spezifikation zusammengefügt. Dazu werden die Objekt-Prozesse durch den Parallel-Operator verknüpft und an den im OCD angegebenen Gates verbunden. Für das Bank-Beispiel ergibt sich [Mor94, S. 116]:

```
(Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
 |[ob2, ob3]|
 ( Teller[t, ob2, c, so, ba]
  |[c, so]|
  ( Cheque[ob3, c, ba](Make_Cheque(...))
  |||
    Standing_Order[ob2, so, ba](Make_SO(...))
 ))
|[ba]|
 Bank_Account[ba]
)
where ...
```

Ebenso werden in diesem Schritt Prozeßdefinitionen für die Interface-Szenarien gegeben:

```
process Interface_Scenario[t, ob1]: noexit :=
  (* create an automatic teller and a counter teller *)
    t !create ?idc: Id_Tellers;
    t !create ?ida: Id_Tellers;
  (* open a cheque account from counter teller *)
    t !open_account !idc !cheque;
    t !rtn_open_account !idc ?nc: Account_Number
  (* deposit from counter teller *)
    t !deposit_cash !idc !nc !This_Mon;
    t !rtn_deposit_cash !idc !nc !This_Mon;
  (* balance from automatic teller *)
    t !print_mini_statement !ida !nc;
    t !rtn_print_mini_statement !ida !nc ?a: Account;
  (* withdrawal from automatic teller *)
    t !withdraw_cash !ida !nc !This_Mon;
    t !rtn_withdraw_cash !ida ?val: Bool;
    ...
  (hide success in success; stop)
endproc
```

Zum Schluß müssen die Interface-Szenarien mit dem Gesamtsystem verbunden werden, und zwar an den Gates, die die Interface-Objekte zur Verfügung stellen:

```
( (Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
   ...
   )
  |[ba]|
  Bank_Account[ba]
)
|[t, ob1]|
 Interface_Scenario[t, ob1]
```

Schritt 3.4: Prototyp exekutieren

In diesem Schritt wird die Spezifikation des Systems anhand der Interface-Szenarien getestet. Dazu kann einer der verschiedenen LOTOS-Simulatoren (z. B. SMILE[2]) verwendet werden.

In der ersten Iteration geht es hauptsächlich darum, die Spezifikationen der einzelnen Klassenschablonen auf ihre Richtigkeit und Vollständigkeit hin zu überprüfen. Falls an dieser Stelle neue Services, Attribute und Message Connections identifiziert werden oder ebensolche wegfallen, müssen die Schritte 2.2, 3.1, 3.2 und 3.3 nochmals durchlaufen werden, um das Objekt-Modell sowie OCT und OCD den Änderungen entsprechend zu modifizieren.

[2]verfügbar unter ftp://ftp.cs.utwente.nl/pub/src/lotos-tools

In späteren Iterationen liegt der Schwerpunkt bei der Überprüfung des Verhaltens des gesamten Systems, d. h., beim Testen des korrekten Zusammenspiels der verschiedenen Objekte.

Schritt 3.5: Spezifikation verfeinern

Ziel dieses Schrittes ist es, die bisher erzeugte Spezifikation anhand der Schritte 2.2, 3.1, 3.2, 3.3 und 3.4 weiter zu verfeinern, bis eine dem Anforderungsdokument entsprechende, ablauffähige LOTOS-Spezifikation erreicht ist.

Der Detailliertheitsgrad, bis zu dem die Spezifikation ausgeführt wird, hängt von den Erfordernissen des jeweiligen Projekts ab. Bevor man jedoch zur Design-Phase übergeht, sollte sichergestellt sein, daß die im Anforderungsdokument angeführten Objekte spezifiziert sind und die Spezifikation intern konsistent ist, d. h., daß für jede Message Connection die entsprechenden Aufrufe getätigt werden und auf der anderen Seite die Services zur Verfügung stehen; daß bei allen statischen Relationen die teilhabenden Objekte involviert sind, und daß für jedes Interface-Szenario ein kompletter Trace durch das Systems vorgenommen werden kann.

Folgende Tätigkeiten können in diesem Schritt anfallen:

Statische Relationen modellieren. Da an diesem Punkt während der ersten Iteration erst einzelne Objekte vorliegen, werden an dieser Stelle die Assoziationen und Aggregationen zwischen den Objekten spezifiziert (siehe Abschnitte 2.5 und 2.6). Das beeinflußt die Schritte 3.2, 3.3 und 3.4.

Objekt-Generatoren einfügen. Durch das Einfügen von Objekt-Generatoren können von einer Klasse mehrere Objekte erzeugt werden. Bei Subsystemen bzw. Klassenhierarchien kann man entweder für jede Klassenschablone einen Objekt-Generator einfügen oder einen für das ganze Subsystem. Änderungen in diesem Schritt wirken sich auf die Schritte 3.1, 3.2, 3.3 und 3.4 aus.

Neue Objekt-Gruppierungen identifizieren. Werden neue Objekt-Gruppierungen (Subsysteme oder Aggregate) identifiziert, sind OCT und OCD entsprechend zu modifizieren. Dadurch sind die Schritte 3.1, 3.2 und 3.4 betroffen.

Objekt aufwerten/abwerten. Objekte können im Laufe der Spezifikation an Bedeutung gewinnen oder verlieren. Falls ein Objekt bis jetzt nur als ADT spezifiziert ist, kann es zu einem eigenen Prozeß aufgewertet werden. Umgekehrt kann ein Objekt, das als Prozeß spezifiziert ist, zu einem ADT abgewertet werden. Dieser Schritt beeinflußt die Schritte 3.1 bis 3.4.

Prozesse und ADTs verfeinern. Mit jeder Iteration der bisherigen Schritte wird das Verständnis für das Problem besser und die Prozesse und ADTs, die die Spezifikation ausmachen, können immer weiter verfeinert werden.

Kapitel 4

S_TOOLS

Ziel dieser Diplomarbeit ist es, ein objektorientiertes, nebenläufiges Datenflußsystem mit ROOA formal zu spezifizieren. Vor der eigentlichen Spezifikation in den Kapiteln 5–7 wird in diesem Kapitel zunächst ein Überblick über S_TOOLS gegeben, das Programm, innerhalb dessen das Datenflußsystem verwendet wird. Es folgt ein Exkurs über Datenflußprogrammierung, der anschaulich machen soll, nach welchen Parametern Datenflußsysteme eingeteilt werden können und welche Design-Alternativen sich dadurch ergeben.

4.1 Übersicht

Entwicklung

S_TOOLS[1] ist ein am Institut für Schallforschung der Österreichischen Akademie der Wissenschaften entstandenes Programm zur Verarbeitung und Analyse digitaler Audiodaten. Es ist ausgelegt auf wissenschaftliche und experimentelle Anwendung. Die bisherigen Einsatzgebiete umfassen unter anderem:

- Psychoakustik: Entwicklung von psychophysikalischen Modellen der akustischen Wahrnehmung, z. B. für Lautstärke, Tonhöhe, Klangfarbe, Rhythmik
- akustische Phonetik: Sprach- und Sprechererkennung, Formantanalyse
- Musikwissenschaft: Spektrographie von Musikaufnahmen, Messung im Spektrum, automatische Grundtonextraktion
- Bioakustik: Untersuchung des Kommunikationsverhaltens von Tieren
- Qualitätsprüfung von Instrumentensaiten

S_TOOLS wurde ursprünglich in der Programmiersprache C für das Betriebssystem MS-DOS entwickelt. Über mehrere Versionen hinweg entstand ein leistungsfähiges und benutzerfreundliches Werkzeug für zahlreiche Anwendungsgebiete. Die Ablöse von MS-DOS als Standardbetriebssystem machte eine Portierung von S_TOOLS auf eine andere Plattform notwendig. Gleichzeitig sollte das System an moderne Hard- und Softwaretechnologien angepaßt werden. Dazu zählen u. a. eine objektorientierte Entwicklung, die Ausnutzung

[1] siehe http://www.kfs.oeaw.ac.at

von Multi-Threading und ein grafisches, fensterorientiertes User-Interface. Außerdem soll das neue System auf einem multimediafähigen Standard-PC lauffähig sein[2].

Mit der Portierung wurde Mitte 1996 begonnen. Aufgrund der Fülle an Veränderungen konnte nur wenig Code bzw. Design wiederverwendet werden. Was jedoch wiederverwendet werden konnte, ist die mit dem bestehenden Programm erlangte, jahrelange Anwendungserfahrung auf dem Gebiet der digitalen Signalverarbeitung. Diese macht sich vor allem bei der Strukturierung von Signalverarbeitungsanwendungen (siehe nächster Unterabschnitt) und dem Aufbau des User-Interface bezahlt.

Aus diesem Grund ist das neue System eine eigenständige Entwicklung mit einer eigenständigen inneren Struktur und teilweise völlig neuen Herausforderungen. Die vorliegende Arbeit hat sich zum Ziel gesetzt, einen Teil des neuen Systems formal zu spezifizieren. Von einer formale Spezifikation mit LOTOS erwarten wir uns eine exakte Beschreibung der Kommunikationsvorgänge im betreffenden Teil. Dadurch soll einerseits Parallelverarbeitung besser zur Geltung kommen, andererseits mögliche Synchronisationsprobleme schon früh ausgemerzt werden. Im Anschluß an die Spezifikation werden die Vor- und Nachteile der verwendeten Methode – ROOA – erörtert.

Grundstruktur von Signalverarbeitungsanwendungen

S_TOOLS fällt in die Kategorie der Signalverarbeitungsanwendungen. Diese sind in ihrer Grundstruktur sehr ähnlich (siehe Abbildung 4.1): Daten werden aus einer Quelle gelesen bzw. in ihr erzeugt, mit bestimmten – von BenutzerInnen parametrisierbaren – Algorithmen transformiert und anschließend wieder ausgegeben (Senken).

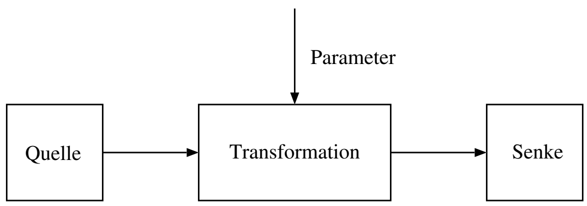

Abbildung 4.1: Grundstruktur von Signalverarbeitungsanwendungen

Im Fall von S_TOOLS, das für die Verarbeitung akustischer Signale konzipiert ist, stammen die Eingangsdaten für den Signalverarbeitungsteil meistens direkt von einem Analog-Digital-Wandler (ADC) oder von einer Datei. Eine dritte Möglichkeit besteht darin, daß Signale intern durch Berechnung erzeugt werden. Die Algorithmen zur Transformation eines Signals reichen von einfachen arithmetischen Operationen (z. B. Multiplikation mit einer Konstanten zur Verstärkung oder Dämpfung eines Signals), über schnelle Fourieranalyse (FFT) bis hin zu Filtern und speziellen phonetischen Analysen. Die Ausgabe des transformierten Signals kann auf einem Digital-Analog-Wandler (DAC) erfolgen, falls durch die Transformation wieder ein hörbares Signal entsteht. Die Möglichkeit einer grafischen Darstellung in einem Fenster oder die Umleitung in eine Datei besteht ebenfalls.

Die richtige, abgestimmte Kombination dieser drei Grundbausteine, Quelle – Transformation – Senke, bildet zusammen mit einem User-Interface und einer Steuerlogik einen

[2]Die MS-DOS Version von S_TOOLS verwendete aus Performancegründen teure Spezialhardware in Form von digitalen Signalprozessoren (DSPs).

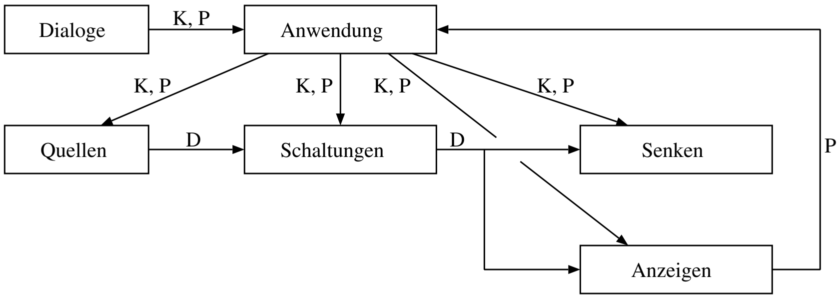

Abbildung 4.2: Dynamische Struktur des neuen Systems

charakteristischen Anwendungsfall aus dem Bereich der digitalen Signalverarbeitung.

Dynamische Struktur von S_TOOLS

Ein Ziel bei der Entwicklung des neuen S_TOOLS ist es, trotz der in Abbildung 4.1 gezeigten, relativ starren Struktur, ein hohes Maß an Interaktivität zu bieten. BenutzerInnen sollen nicht nur zusehen müssen, wie ihre Daten transformiert werden, sondern aktiv in den Programmablauf eingreifen können. Darunter verstehen wir z. B. die Veränderung von Analyseparametern zur Laufzeit oder das Messen von Werten in der grafischen Darstellung.

Abbildung 4.2 zeigt die dynamische[3] Struktur des neuen Systems und den Datenaustausch der Komponenten untereinander. Der Datenfluß von den Quellen über die Schaltungen zu den Senken realisiert die im vorigen Abschnitt beschriebene Grundstruktur. Dialoge und Anzeigen bilden das User-Interface, die Steuerungslogik der Signalverarbeitungsanwendung befindet sich in der Anwendungskomponente. Im folgenden werden die Aufgaben der einzelnen Komponenten genauer erläutert:

Anwendung: Die Anwendungskomponente bildet den Kern einer S_TOOLS-Signalverarbeitungsanwendung. Sie baut die anderen Komponenten und deren Verknüpfungen nach Bedarf auf und koordiniert ihr Zusammenspiel. Sie parametrisiert die Komponenten und nimmt Kommandos von ihnen entgegen. Anwendungen werden in S_TOOLS in einer an BASIC angelehnten Skriptsprache programmiert, was den Vorteil hat, daß die Anwendungsentwicklung relativ unabhängig von der eigentlichen Systementwicklung erfolgen kann. Die weiter unten beschriebenen Komponenten stehen in der Skriptsprache als Elemente zur Verfügung.

Dialoge: Dialoge bilden neben den Anzeigen das User-Interface einer Anwendung. Zum Unterschied von Anzeigen sind Dialoge hauptsächlich zur Eingabe von Parametern und zur Steuerung der Anwendung durch BenutzerInnen gedacht. Als Dialogelemente kommen Menüs, Knöpfe, Eingabefelder, Regler usw. zum Einsatz.

Quellen: Quellen versorgen die Anwendung mit Daten. In der Regel sind dies digitale Audiosignale, die entweder von einem Soundfile oder direkt vom ADC-Device kommen.

[3]„Dynamisch" deshalb, weil es sich um das Zusammenspiel der Komponenten während der Laufzeit handelt.

Schaltungen: Der Transformationsteil von S_TOOLS-Signalverarbeitungsanwendungen wird in sogenannten Schaltungen realisiert. Diese sind aus funktionalen Grundeinheiten, die nach dem Datenflußmodell (siehe nächster Abschnitt) beliebig zu komplexen Funktionen kombiniert werden können, aufgebaut. Funktionale Grundeinheiten sind z. B. Addierer, Multiplizierer, FFTs, Filter- und Fensterfunktionen.

Senken: Senken nehmen die in den Schaltungen berechneten Daten entgegen und schreiben sie in ein Soundfile oder übergeben sie an ein DAC-Device zur Ausgabe eines hörbaren Signals. Das Exportieren von Analysedaten zur Weiterverarbeitung in anderen Programmen kann ebenfalls als Senke realisiert werden.

Anzeigen: Anzeigen dienen zur grafischen Darstellung von Daten. Sie erlauben auch ein interaktives Ablesen der angezeigten Werte mittels zweier Cursoren, außerdem können ähnlich wie bei den Dialogen Kommandos an die Anwendung zurückgeschickt werden (z. B.: „das zwischen den Cursoren angezeigte Signal abspielen“).

Zwischen den Komponenten werden zur Laufzeit verschiedene Informationen ausgetauscht (die Pfeile zeigen jeweils vom Sender zum Empfänger):

Kommandos (K): Kommandos steuern den Ablauf einer Anwendung. Sie haben deshalb ihren Ausgangspunkt in jenen Komponenten, die direkt mit BenutzerInnen in Verbindung stehen (Dialoge und Anzeigen). Üblicherweise geben BenutzerInnen in einem fensterorientierten User-Interface Kommandos durch die Auswahl von Menüeinträgen oder durch das Drücken von Knöpfen.

Daten (D): Unter diesem Begriff werden die verschiedenen Daten verstanden, die in den Quellen erzeugt, in den Schaltungen verarbeitet und in den Senken gespeichert bzw. in den Anzeigen dargestellt werden. Die gängigsten Datentypen sind einzelne Samples, Samplevektoren (Frames), Spektren, usw.

Parameter (P): Parameter sind Daten, die nur sporadisch von den Komponenten ausgetauscht werden, z. B. wenn mit der Ausführung eines bestimmten Kommandos begonnen wird, oder wenn BenutzerInnen ein Dialogelement bedienen, um einen Analyseparameter während der Laufzeit zu verändern. Bei den Parametern, die die Anzeigen verlassen, handelt es sich um Cursorpositionen.

4.2 Datenflußprogrammierung

Die Schaltungen von S_TOOLS basieren auf dem Datenflußmodell [Sha92]. Diesem liegt die Vorstellung zugrunde, daß Daten von der Eingabe bis zur Ausgabe mehrere, teilweise unabhängig voneinander durchführbare Transformationsschritte durchlaufen. Die gängigste Repräsentation des Datenflußmodells ist in Form eines gerichteten Datenflußgraphen (siehe Abbildung 4.3). Die funktionalen Einheiten, die die Daten schrittweise transformieren, bilden die Knoten des Graphen. Die Knoten sind durch Kanten miteinander verbunden, auf denen die Daten von einem Knoten zum nächsten „wandern“. Kanten, die in einen Knoten münden, werden als Eingänge (Argumente) einer funktionalen Einheit angesehen, von Knoten wegführende Kanten als Ausgänge (Resultate).

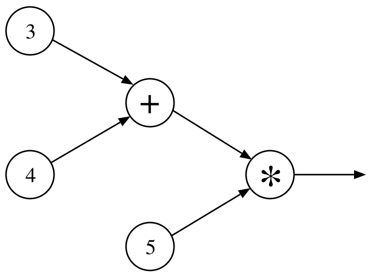

Abbildung 4.3: Datenflußgraph

Die Hauptgründe, warum der Signalverarbeitungsteil von S_TOOLS in einem Datenflußmodell realisiert werden soll, sind folgende:

- Verständlichkeit: Das Datenflußmodell bildet den Vorgang der Analyse bzw. Synthese von Audiosignalen auf „natürliche“ Art und Weise nach. Die meisten Transformationsschritte können als Knoten mit entsprechenden Ein- und Ausgängen interpretiert werden. BenutzerInnen können außerdem in ihrer Vorstellung den Datenfluß verfolgen und erhalten damit ein transparentes Bild des Verarbeitungsvorganges.

- Erweiterbarkeit: Der Satz an Analyse- und Synthesemethoden kann durch das Hinzufügen entsprechender funktionaler Grundeinheiten ständig erweitert werden. Das ist besonders dann wichtig, wenn neue Anwendungsfelder erschlossen werden sollen.

- Programmierbarkeit: Datenflußprogramme besitzen einfache grafische und textuelle Repräsentationen. Damit können Schaltungen in einer von der Implementierungssprache unabhängigen Form direkt von den AnwenderInnen angegeben werden.

Ein Nachteil des Datenflußmodells gegenüber einem „hart codierten“ Signalverarbeitungsteil ist sicher die schlechtere Performance einzelner Rechenelemente wegen des Kommunikationsaufwandes mit anderen Rechenelementen. Durch geschickte Ausnützung von Nebenläufigkeit, die im Datenflußmodell auf natürliche Art und Weise gegeben ist, kann dieser Nachteil aber – zumindest auf Multiprozessor-Computern – mehr als wettgemacht werden.

Einteilungsmöglichkeiten

Beim „reinen“ Datenflußmodell gibt es keine Konstrukte für Schleifen, sequentielle Ausführung oder Auswahlabfragen. Die Reihenfolge der Ausführung ist nicht festgelegt. Wenn alle Eingänge eines Knotens vorhanden sind, „feuert“ dieser, d. h., er konsumiert die Eingänge, führt die angegebene Funktion durch und stellt das Ergebnis an seinen Ausgängen bereit.

Das Datenflußmodell dient häufig als Grundlage für visuelle Datenflußprogrammiersprachen [Hil92], wie z. B. LabView[4] oder SimuLink[5]. Bei der Datenflußprogrammierung repräsentiert ein Graph nicht nur eine Funktion, sondern ein ganzes Programm, welches sowohl Rechen- als auch Steuereinheiten enthält.

[4]siehe http://www.ni.com
[5]siehe http://www.mathworks.com

Alternative	Möglichkeiten
Ausführungsmodus	datengesteuert/nachfragegesteuert
Iteration	Zyklen, Schleifen, Listenelemente
Prozedurale Abstraktion	ja/nein
Datenflußsteuerung	Selektoren, Distributoren, ...
Sequentielle Ausführung	ja/nein
Typüberprüfung	ja (zur Laufzeit oder beim Verbinden)/nein
Höhere Funktionen	ja/nein
Lebendigkeit	Stufen 1–4

Tabelle 4.1: Mögliche Alternativen

Das in dieser Arbeit entwickelte Schaltungssystem orientiert sich an den visuellen Datenflußprogrammiersprachen. Um das Schaltungssystem in seinem Umfeld besser bewerten zu können und einen Leitfaden für die Entwicklung zu haben, soll an dieser Stelle eine Übersicht über die wichtigsten Design-Alternativen, nach denen sich Systeme zur (visuellen) Datenflußprogrammierung einteilen lassen, gegeben werden [Hil92]. Die für diese Arbeit relevanten Alternativen werden im folgenden beschrieben (zusammengefaßt in Tabelle 4.1):

Ausführungsmodus: Generell unterscheidet man zwei Ausführungsmodi, *datengesteuert (data driven)* und *nachfragegesteuert (demand driven)*. Datengesteuert heißt, daß die Ausführungsreihenfolge der Knoten von der Verfügbarkeit der Eingänge abhängt. Beim nachfragegesteuerten Ausführungsmodus wird die Reihenfolge bestimmt nach dem Bedarf an Daten an den Ausgängen. Der datengesteuerte Ausführungsmodus hat den Nachteil, daß unter bestimmten Voraussetzungen Berechnungen unnötig durchgeführt werden. Er ist aber gegenüber dem nachfragegesteuerten Modus intuitiver und damit leichter zu verstehen.

Iteration: Es gibt mehrere Möglichkeiten, Iteration auszudrücken, z. B. durch Zyklen im Datenflußgraphen, Schleifen (while, repeat, for) sowie spezielle Knoten, die nicht nach jedem Berechnungsschritt Ausgänge liefern oder aber nach jedem Berechnungsschritt mehrmals ihre Ausgänge aktivieren.

Prozedurale Abstraktion: Prozedurale Abstraktion bedeutet, daß ein ganzer Datenflußgraph als Einheit gesehen wird und auf einer höheren Ebene als einzelner Knoten verwendet werden kann. Das ermöglicht eine hierarchische Anordnung von Datenflußgraphen und die Definition von zusammengesetzten „Unterfunktionen“ durch BenutzerInnen.

Datenflußsteuerung: Zur Steuerung des Datenflusses werden spezielle Knoten verwendet, wie z. B. die in Abbildung 4.4 gezeigten Selektoren und Distributoren. Selektoren schalten abhängig vom Kontrolleingang C einen bestimmten Eingang an den Ausgang weiter, Distributoren schalten einen Eingang an einen bestimmten Ausgang weiter. Komplexere Fallabfragen, die den `case`-Anweisungen aus prozeduralen Programmiersprachen nachgebildet sind, sind ebenfalls denkbar.

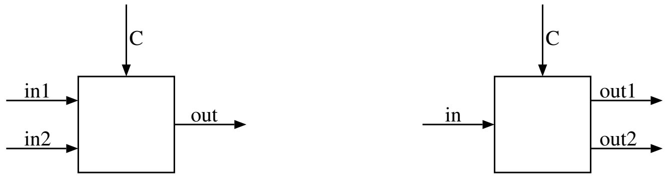

Abbildung 4.4: Selektor und Distributor

Sequentielle Ausführung: Beim reinen Datenflußmodell ist keine Reihenfolge für die Ausführung von Knoten vorgegeben—sie feuern, sobald alle Eingänge gesetzt sind. Das Modell kann erweitert werden, indem man die Spezifikation einer Reihenfolge durch BenutzerInnen zuläßt.

Typüberprüfung: Wenn die an den Kanten kommunizierten Daten unterschiedlichen Typ haben können, muß überprüft werden, ob Eingänge mit typkompatiblen Ausgängen verbunden sind. Dies kann entweder beim Verbinden der Knoten gemacht werden oder zur Laufzeit.

Höhere Funktionen: Höhere Funktionen sind Funktionen, die andere Funktionen als Parameter haben. Beispielsweise könnte ein Knoten eine als Parameter übergebene Funktion auf alle Elemente eines Vektors anwenden.

Lebendigkeit: Im Bereich der visuellen Datenflußprogrammiersprachen werden Systeme nach einer vierstufigen Lebendigkeitsskala eingeteilt. Stufe 1 entspricht einem System, bei dem Programme zwar eine grafische Interpretation besitzen, diese aber nicht direkt ausführbar ist. Bei Systemen der Stufe 2 ist die grafische Darstellung direkt exekutierbar, BenutzerInnen müssen aber selbst start- und stop-Kommandos geben. Systeme der Stufe 3 reagieren automatisch auf Änderungen des Programms oder der Eingabedaten; in der Lebendigkeitsstufe 4 ist das Programm permanent aktiv. Die meisten visuellen Datenflußprogrammiersprachen sind auf Stufe 2.

Kapitel 5

Spezifikation mit ROOA: Anforderungen und initiales Objekt-Modell

In diesem Kapitel beginnen wir mit der formalen Spezifikation des Schaltungssystems von S_TOOLS mittels rigoroser objektorientierter Analyse (ROOA). Ziel ist eine ablauffähige, objektorientierte LOTOS-Spezifikation, mit der die grundlegenden Mechanismen des Schaltungssystems getestet werden können und die als Grundlage für eine Implementierung dient.

Die Anwendung der Methode erfolgt größtenteils wie in Kapitel 3 beschrieben. Wir möchten jedoch darauf hinweisen, daß auf den folgenden Seiten nicht der gesamte Entwicklungsprozeß wiedergegeben werden kann. Auf dem Weg von der ersten Idee bis zur fertigen Spezifikation wurden zahlreiche Fehler begangen, Irrwege beschritten und Konzeptänderungen vorgenommen. Das alles zu beschreiben, würde den Rahmen dieser Arbeit sprengen und unserer Meinung nach auch den Blick auf das Wesentliche verstellen. Die Erfahrungen mit der Methode sind in Kapitel 8 zusammengefaßt.

In diesem Kapitel werden zunächst die Anforderungen an das Schaltungssystem von S_TOOLS informell dargestellt, was dem dem in der Beschreibung der ROOA häufig referenzierten Anforderungsdokument gleichkommt. Darauf aufbauend wird ein erstes Objekt-Modell des Systems erstellt. In den beiden nachfolgenden Kapiteln werden die Schritte 2 und 3 der ROOA durchgeführt.

5.1 Schaltungssystem von S_TOOLS

Das Schaltungssystem von S_TOOLS dient zur Verarbeitung digitaler Audiosignale. Als funktionale Grundbausteine, die zu Schaltungen verbunden werden, fungieren deshalb Algorithmen und Funktionen aus dem Signalverarbeitungsbereich. Die am häufigsten verwendeten Datentypen für den Datenaustausch zwischen den Elementen einer Schaltung sind Zahlen und Zahlenvektoren.

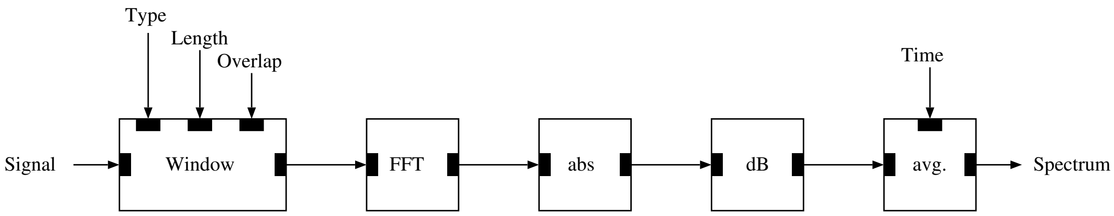

Abbildung 5.1: Schaltung zur Spektralanalyse

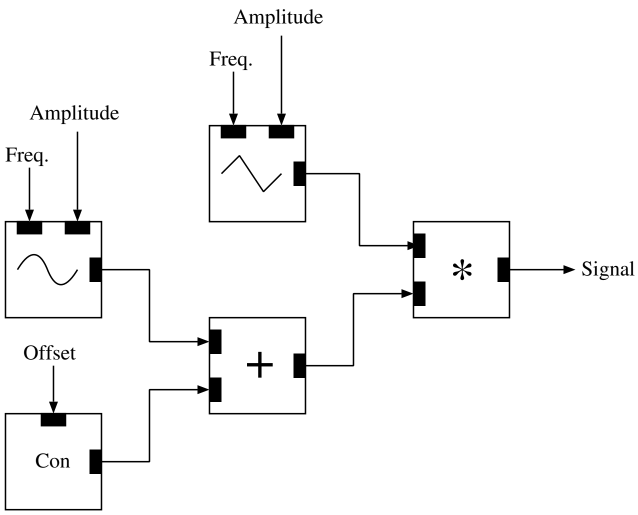

Abbildung 5.2: Schaltung zur Klangsynthese

Beispiele

Um sich ein besseres Bild von den Erfordernissen machen zu können, werden in den Abbildungen 5.1 und 5.2 Beispielschaltungen gezeigt, die typische Anwendungsfälle charakterisieren.

Abbildung 5.1 zeigt eine Schaltung zur Spektralanalyse von Audiosignalen. Die Rechtecke symbolisieren die funktionalen Grundbausteine, die ihrerseits mit Eingängen an der linken Seite und Ausgängen an der rechten Seite ausgestattet sind. An der Oberseite mancher Rechenelemente befinden sich weitere Eingänge, über die dem Element bestimmte Berechnungsparameter sporadisch gesetzt werden können.

Das an der linken Seite in Blöcken hereinkommende Signal wird über mehrere Schritte hinweg (Fensterung, FFT, Betrag, dB-isierung) in seine spektralen Anteile zerlegt, wobei am Schluß eine Mittelung über vergangene Spektren stattfindet. Als Ergebnis verlassen gemittelte logarithmische Spektren des Signals an der rechten Seite die Schaltung. Als Parameter dienen der Schaltung z. B. der Typ und die Länge des Fensters oder die Zeitkonstante bei der Mittelung. Analyseschaltungen wie diese bestehen im allgemeinen aus vorwärtsgerichteten, schrittweisen Transformationen des Eingangssignals. Schaltungen zur Filterung von Signalen sind ähnlich aufgebaut.

Eine Schaltung zur Klangsynthese zeigt Abbildung 5.2. Dort wird ein Dreieckston mit bestimmter Frequenz und Amplitude sinusförmig amplitudenmoduliert (dargestellt durch den Multiplikator an der rechten Seite der Schaltung). Die Frequenz und Amplitude des modulierenden Sinustons können ebenfalls eingestellt werden. Vor der Modulation kann über das Konstanten-Atom *Con* noch ein frei gewählter Offset zum Sinus addiert werden, um einen negativen Modulationsfaktor zu vermeiden. Bei der Synthese von Signalen kommt es oft vor, daß Schaltungen stärker verzweigt sind oder auch Rückführungen besitzen.

Funktionale Grundeinheiten

Generell sollen Schaltungen aus mehreren funktionalen Grundeinheiten, den *Atomen* bestehen. Innerhalb von S_TOOLS sollen Schaltungen aus einer textuellen Repräsentation erzeugt werden. Ein grafischer Editor für Schaltungen, der auf dieser textuellen Repräsentation arbeitet, ist eine Möglichkeit. Ebenfalls auf der textuellen Ebene soll die prozedurale Abstraktion von Schaltungen aufgelöst werden: Schaltungen, die innerhalb von Schaltungen verwendet werden, werden bei der Erzeugung aus der textuellen Repräsentation durch die Atome, aus denen sie bestehen, ersetzt. Zuletzt wird eine nur noch aus Atomen bestehende Schaltung exekutiert.

Die Atome selbst verfügen über eine beliebige Anzahl von Ein- und Ausgängen, die bei der Erzeugung des Atoms aufgebaut und initialisiert werden. Insgesamt ist in den Atomen folgende Funktionalität gekapselt:

- Verwaltung der Ein- und Ausgänge
- Verarbeitung von start-/stop-Kommandos zur Steuerung der gesamten Schaltung (siehe Unterabschnitt *Ausführung von Schaltungen*).
- Realisierung einer Funktion, die aus den an den Eingängen übergebenen Werten Resultate berechnet und den Ausgängen übergibt
- Realisierung einer Schaltbedingung, die bestimmt, wann ein Atom feuern kann

Von der Spezifikation wird erwartet, daß es eine Basisklasse für alle Atome gibt, die einen Großteil der „Routinetätigkeiten“ übernimmt und von der relativ einfach neue Atom-Klassen abgeleitet werden können. Grob betrachtet, lassen sich die Atome in zwei Kategorien einteilen:

Funktionsatome: Sie enthalten die mathematischen Funktionen einer Schaltung, wie z. B. Addierer, Multiplizierer, FFT etc. Funktionsatome verfügen über ein sehr einfaches Schaltverhalten: wenn alle Eingänge des Atoms gesetzt sind und alle Ausgänge frei sind für neue Daten, nimmt es seine Eingänge entgegen und berechnet daraus seine Ausgänge.

Datenflußatome: Datenflußatome steuern den Datenfluß innerhalb einer Schaltung. Im Gegensatz zu den Funktionsatomen führen sie keine Berechnungen auf ihren Eingangsdaten durch, definieren dafür aber ein eigenes Schaltverhalten. Die weiter oben besprochenen Selektoren und Distributoren zählen zur Klasse der Datenflußatome.

Alternative	Auswahl
Ausführungsmodus	datengesteuert
Iteration	durch spezielle Knoten
Prozedurale Abstraktion	auf syntaktischer Ebene
Datenflußsteuerung	durch Datenflußatome
Sequentielle Ausführung	nein
Typüberprüfung	ja (Laufzeit)
Höhere Funktionen	nein
Lebendigkeit	Stufe 1 (erweiterbar auf 2)

Tabelle 5.1: Ausgewählte Alternativen für das Schaltungssystem

Datentransfer

Der Datentransfer zwischen den Atomen bzw. zwischen Schaltungen und dem Rest des Systems erfolgt über Ein- und Ausgänge. Eingänge müssen mit genau einem Ausgang verbunden sein, Ausgänge können mit keinem oder mehreren Eingängen verbunden sein. Die Verbindungen werden beim Aufbau der Schaltung hergestellt.

Es soll zwischen zwei Arten von Eingängen unterschieden werden:

Synchrone Eingänge: Synchrone Eingänge folgen einem strikten Datenaustauschprotokoll mit den Ausgängen[1]. Ein gesetzter synchroner Eingang bleibt so lange für weitere Datentransfers gesperrt, bis der Besitzer des Eingangs diesen wieder freigibt. Ein Versuch, einen gesperrten synchronen Eingang zu setzen, resultiert in einem Blockieren des Senders.

Asynchrone Eingänge: Sie unterscheiden sich von den zuvor besprochenen dadurch, daß sie jederzeit, auch wenn die Schaltung gerade nicht ausgeführt wird, setz- und abfragbar sind. Asynchrone Eingänge sind vor allem dort notwendig, wo einem Atom sporadisch Laufzeitparameter mitgeteilt werden sollen, weil dort ein Blockieren des Senders nicht erwünscht ist.

Atome erzeugen ihre Ein- und Ausgänge selbst. Für den Datenaustausch von Schaltungen mit anderen Komponenten von S_TOOLS soll es die Möglichkeit geben, sogenannte „freie“ Ein- und Ausgänge zu erzeugen, d. h., Ein- und Ausgänge, die keinem Atom angehören. Eine Anzeigekomponente kann beispielsweise direkt von Schaltungen Daten übernehmen, indem sie sich freie Eingänge erzeugt und mit den entsprechenden Ausgängen der Schaltung verbindet.

Ausführung von Schaltungen

Die Ausführung von Schaltungen soll zentral gesteuert werden können. Dazu werden folgende Kommandos benötigt:

start: Die Ausführung der Schaltung für eine unbestimmte Anzahl von Schritten starten.

[1] Ausgänge sind streng genommen *synchrone* Ausgänge. Dadurch, daß aber keine asynchronen Ausgänge notwendig sind, entfällt die Unterscheidung.

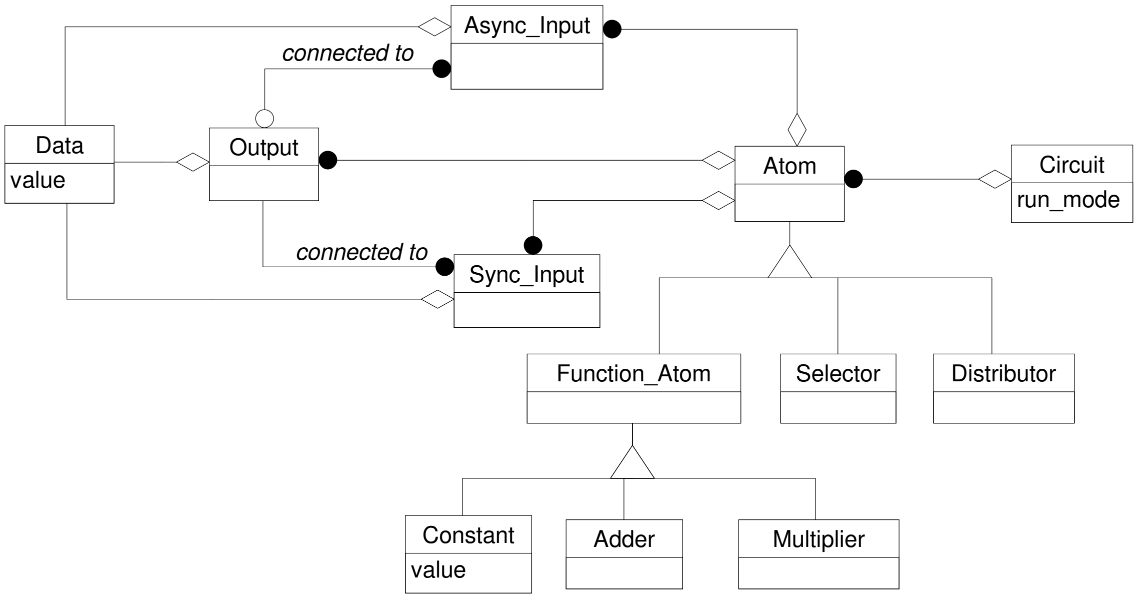

Abbildung 5.3: Erstes Objekt-Modell des Schaltungssystems

stop: Die Ausführung der Schaltung anhalten. Der Zustand der Schaltung wird eingefroren, sodaß durch ein neuerliches start-Kommando an derselben Stelle fortgesetzt werden kann.

reset: Dadurch wird die Schaltung in einen definierten Anfangszustand zurückgesetzt. Dieses Kommando kann nur einer stehenden Schaltung gegeben werden.

Tabelle 5.1 faßt zusammen, welche der Design-Alternativen für das Schaltungssystem von S_TOOLS getroffen wurden (vgl. dazu auch Tabelle 4.1).

5.2 Schritt 1: Objekt-Modell erstellen

Abbildung 5.3 zeigt das initiale Objekt-Modell des Schaltungssystems, erstellt nach der OOA-Methode OMT. Es enthält die Klassen und statischen Relationen, die aus der Beschreibung im vorherigen Abschnitt offensichtlich hervorgehen. Außerdem sind einige Attribute in den Klassen identifiziert.

Schaltungen werden durch Objekte der Klasse *Circuit* repräsentiert. Der aktuelle Ausführungszustand der Schaltung (laufend oder gestoppt) wird im Attribut *run_mode* festgehalten.

Schaltungen bestehen aus mehreren Atomen, was durch die Aggregation zwischen der Schaltungsklasse und der Basisklasse aller Atome *(Atom)* dargestellt wird. Von der abstrakten Klasse *Atom* abgeleitet sind konkrete Atom-Klassen, von denen Objekte instanziert werden können. Für die Funktionsatom-Klassen *Constant*, *Adder* und *Multiplier* wird zusätzlich die (wiederum abstrakte) Zwischenklasse *Function_Atom* eingeführt, die das gemeinsame Schaltverhalten der Funktionsatome kapseln soll. Die Datenflußatome *Selector* und *Distributor* besitzen keine offensichtlichen Gemeinsamkeiten und werden vorerst noch direkt von der Klasse *Atom* abgeleitet.

Atome ihrerseits verfügen über mehrere synchrone Eingänge *(Sync_Inputs)*, asynchrone Eingänge *(Async_Inputs)* und Ausgänge *(Outputs)*. Im Objekt-Modell sind diese Beziehungen als Aggregationen dargestellt. Ausgänge können sowohl mit synchronen als auch mit asynchronen Eingängen beliebiger Anzahl verbunden sein *(connected to)*. Synchrone Eingänge müssen, asynchrone Eingänge können mit einem Ausgang verbunden sein (dargestellt durch den leeren Kreis am *Output*-Ende der Assoziation). Für Eingänge und Ausgänge bietet sich ebenfalls eine gemeinsame Basisklasse an, z. B. unter dem Begriff *Anschluß (Connection)*. Zum jetzigen Zeitpunkt können aber noch keine genauen Angaben über das Verhalten der drei Klassen gemacht werden. Deshalb wird die Entscheidung darüber noch aufgeschoben.

Die zwischen den Ein- und Ausgängen transferierten Daten werden in Objekten der Klasse *Data* zwischengespeichert. Die Kapselung in einer eigenen Klasse hat den Vorteil, daß dadurch mehrere Datentypen für den Datenaustausch ermöglicht werden. Das Attribut *value* enthält den konkreten Wert des Datenobjekts. Die Besitz-Relationen zwischen den Anschlüssen und der *Data*-Klasse werden als 1:1-Aggregationen dargestellt.

Kapitel 6

Schritt 2: Objekt-Modell verfeinern

In diesem Schritt wird das Objekt-Modell weiter ausgebaut, sodaß es als Basis für eine formale Spezifikation in LOTOS dienen kann. Insbesondere wird mittels Interface-Szenarien und Event Trace Diagrammen das dynamische Verhalten des Systems ermittelt.

6.1 Schritt 2.1: Objekt-Modell vervollständigen

Vor der Erstellung des dynamischen Verhaltens stellen wir sicher, daß im Objekt-Modell Interface-Objekte, statische Relationen, Attribute und Services enthalten sind. Das daraus entstehende Objekt-Modell ist in Abbildung 6.1 zu sehen. Es unterscheidet sich vom initialen Objekt-Modell fast ausschließlich durch das Vorhandensein von Services in den Klassen.

Interface-Objekte hinzufügen

Beim zu entwickelnden System handelt es sich um eine objektorientierte Klassenbibliothek. Als „Benutzer" ist deshalb das aufrufende Programm anzusehen. Im Fall von S_TOOLS ist das die in Abbildung 4.2 gezeigte Anwendungskomponente. Menschliche BenutzerInnen beeinflussen den Aufbau und Ablauf von Schaltungen eigentlich nur indirekt.

Alle im bisherigen Objekt-Modell enthaltenen Objekte sind in diesem Sinne Interface-Objekte. Während des Aufbaus von Schaltungen müssen Atome erzeugt und der Schaltung bekannt gemacht werden, Eingänge müssen mit Ausgängen verbunden werden usw. Beim Ablauf einer Schaltung interagiert die Anwendungskomponente hauptsächlich mit dem Schaltungs-Objekt selbst und den freien Ein- und Ausgängen. In diesem Zusammenhang werden Datenobjekte benutzt, um den Ein- und Ausgängen Daten zu übergeben und von diesen entgegenzunehmen.

Statische Relationen hinzufügen

Das initiale Objekt-Modell enthält alle zu diesem Zeitpunkt bekannten Assoziationen und Aggregationen. Die in den Anforderungen formulierte, von der Atom-Klasse ausgehende Klassenhierarchie, ist ebenfalls im Objekt-Modell realisiert.

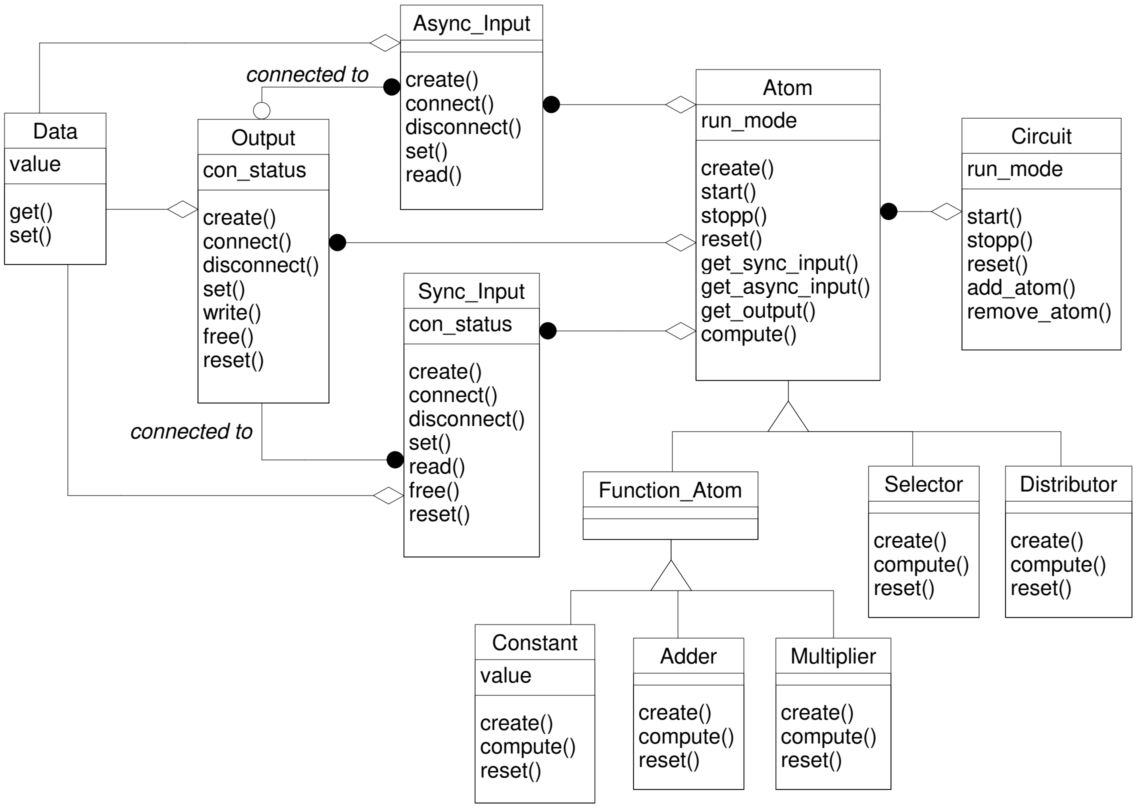

Abbildung 6.1: Objekt-Modell mit einigen Services

Attribute und Services hinzufügen

Als offensichtliche Services der Schaltungs-Klasse *Circuit* können die Kommandos zum Starten, Stoppen und Rücksetzen identifiziert werden *(start, stopp*[1]*, reset)*. Des weiteren benötigt die Klasse *Circuit* ein Service, um einer Schaltung ein Atom hinzuzufügen *(add_atom)*, bzw. wieder zu entfernen *(remove_atom)*.

Atome müssen zuallererst erzeugt werden *(create)*. Zum Verbinden der Ein- und Ausgänge eines Atoms ist es notwendig, einzelne Ein- und Ausgänge abzufragen. Dies geschieht mit den Services *get_sync_input*, *get_async_input* und *get_output*. Schaltungen müssen beim Ablaufen auf einzelne Atome zurückgreifen und diese starten, stoppen und rücksetzen können (*start*, *stopp* und *reset*). Atome merken sich im Attribut *run_mode*, in welchem Lauf-Zustand sie sich gerade befinden (gestartet oder gestoppt). Schließlich und endlich verfügen Atome über eine Schaltfunktion, die im Service *compute* zusammengefaßt ist. In den von der Klasse *Atom* abgeleiteten Atom-Klassen müssen die Services *create, compute* und *reset* überschrieben werden, um spezifische Atom-Funktionalität zu realisieren.

Synchrone und asynchrone Eingänge sowie Ausgänge werden mit *create* erzeugt. Danach werden sie meistens mit anderen Ausgängen bzw. Eingängen verbunden *(connect)*. Die Verbindungen können mit *disconnect* wieder gelöst werden. Sowohl synchrone als

[1]Mit zwei „p“ am Ende, um eine Kollision mit dem LOTOS-Schlüsselwort `stop` zu vermeiden.

auch asynchrone Eingänge können gesetzt *(set)* und gelesen werden *(read)*. Synchrone Eingänge werden mit *free* für den Empfang neuer Daten freigegeben. Sie merken sich im Attribut *con_status* ihren aktuellen Setz-Zustand (gesetzt oder frei). Ausgänge werden mit Daten beschrieben *(write)* und anschließend dazu aufgefordert, die nachgeschalteten Eingänge zu setzen *(set)*. Zum Freigeben eines Ausganges wird *free* aufgerufen. Ausgänge verfügen ebenfalls über ein *con_status*-Attribut, mit dem sie sich merken, ob sie gesetzt sind oder nicht. Synchrone Eingänge und Ausgänge können mit *reset* in den leeren Zustand zurückversetzt werden.

Das Datenelement *Data* offeriert einfache Services zum Abfragen *(get)* und Setzen *(set)* des enthaltenen Datenwertes *value*.

6.2 Schritt 2.2: Initiales dynamisches Verhalten

Das Hauptaugenmerk der Analyse des Schaltungssystems liegt bei seinem dynamischen Verhalten. Im Vordergrund stehen die Vermeidung von Deadlocks und eine hohe Effizienz bei der Abarbeitung.

Für die Ausführung von Schaltungen ist vor allem das dynamische Verhalten der Atome, ihre Interaktion mit den Ein- und Ausgängen sowie die Realisierung des Datenflusses von den Ausgängen zu den nachgeschalteten Eingängen interessant. Folgende Ziele lassen sich formulieren:

- Möglichst hoher Grad an Nebenläufigkeit: die Atome einer Schaltung sollen weitestgehend unabhängig voneinander arbeiten. Atome sollen ihre Ein- und Ausgänge nicht unnötig lange blockieren, sondern freigeben, sobald sie sie nicht mehr benötigen.

- Möglichst geringer Kommunikationsaufwand: der Austausch von Nachrichten zwischen Ein- und Ausgängen soll auf ein Minimum reduziert werden. Dies ist insbesondere dann wichtig, wenn das System verteilt implementiert ist, Atome also auf verschiedenen Rechnern liegen können. Die Kosten eines Kommunikationsvorganges sind in diesem Fall relativ hoch.

- Möglichst wenige Datentransfers: innerhalb eines Datenflußsystems müssen die in einem Element produzierten Daten zu den nachgeschalteten Elementen weitergereicht werden. Da der Transfer von Daten Zeit kostet soll nur der notwendige Datentransfer stattfinden.

Eine Besonderheit des zu entwickelnden Systems ist weiters, daß es „lebendig“ ist. Anders als bei der in Kapitel 3 skizzierten Bank-Applikation, wo Benutzerinteraktionen zu einer endlichen Aufrufreihenfolge von Services innerhalb des Systems führen, kann eine Schaltung, die einmal gestartet wurde, prinzipiell endlos weiterlaufen. Auf der anderen Seite muß eine laufende Schaltung jederzeit ohne lange Verzögerungen gestoppt werden können.

Interface-Szenarien erstellen

Als Ausgangsbasis für die Entwicklung des dynamischen Modells dient die nachfolgende informelle Beschreibung:

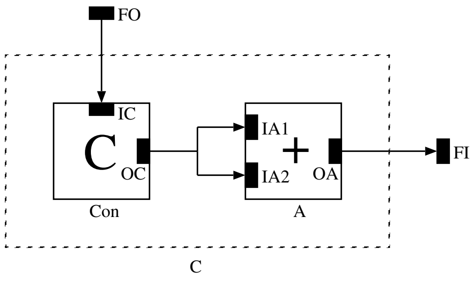

Abbildung 6.2: Beispielschaltung, anhand welcher das dynamische Verhalten erstellt wird

> Eine Schaltung *C* (siehe Abbildung 6.2) wird aufgebaut, indem ein Konstanten-Atom *Con* und ein Addierer *A* erzeugt und der Schaltung hinzugefügt werden. Ein freier Ausgang *FO* und ein freier Eingang *FI* werden für den Datenaustausch mit der Schaltung erzeugt. Anschließend werden die Eingänge *IC, IA1, IA2* und *FI* mit den entsprechenden Ausgängen *FO, OC* und *OA* verbunden und umgekehrt. Die Ausführung der Schaltung wird gestartet und am freien Eingang werden Daten entgegengenommen. Über den freien Ausgang wird der asynchrone Eingang des Konstanten-Atoms zwischendurch verändert. Nach einigen Schritten wird die Schaltung angehalten und in ihren Anfangszustand zurückgesetzt.

Auf die grafische Darstellung der Interface-Szenarien wird an dieser Stelle aus Platzgründen verzichtet; sie stellen eigentlich eine Vorstufe zu den Event Trace Diagrammen (ETDs) dar. Stattdessen werden im nächsten Unterabschnitt, ausgehend von der obigen Beschreibung sofort ETDs erstellt. Aus diesen können die in der Beschreibung enthaltenen Interface-Szenarien leicht abgelesen werden (alle Aufrufe vom Interface-Szenario an Objekte des Systems und umgekehrt bilden das Interface-Szenario).

ETDs und initiale OCT erstellen

Als erste Annäherung des dynamischen Verhaltens werden ETDs erstellt, die folgendes Verhalten abdecken:

- Atome erzeugen und der Schaltung hinzufügen
- Ein- und Ausgänge verbinden
- Schaltung starten
- Datentransfer von den Ausgängen zu den Eingängen
- Schaltvorgang innerhalb des Atoms
- Schaltung anhalten und rücksetzen

Das Event Trace Diagramm in Abbildung 6.3 zeigt das Erzeugen des Addierers *A* und das Hinzufügen zu einer existierenden Schaltung *C*. Der Addierer selbst wird vom Interface-Szenario mit *create* erzeugt und kreiert seinerseits die beiden Eingänge *IA1* und *IA2*

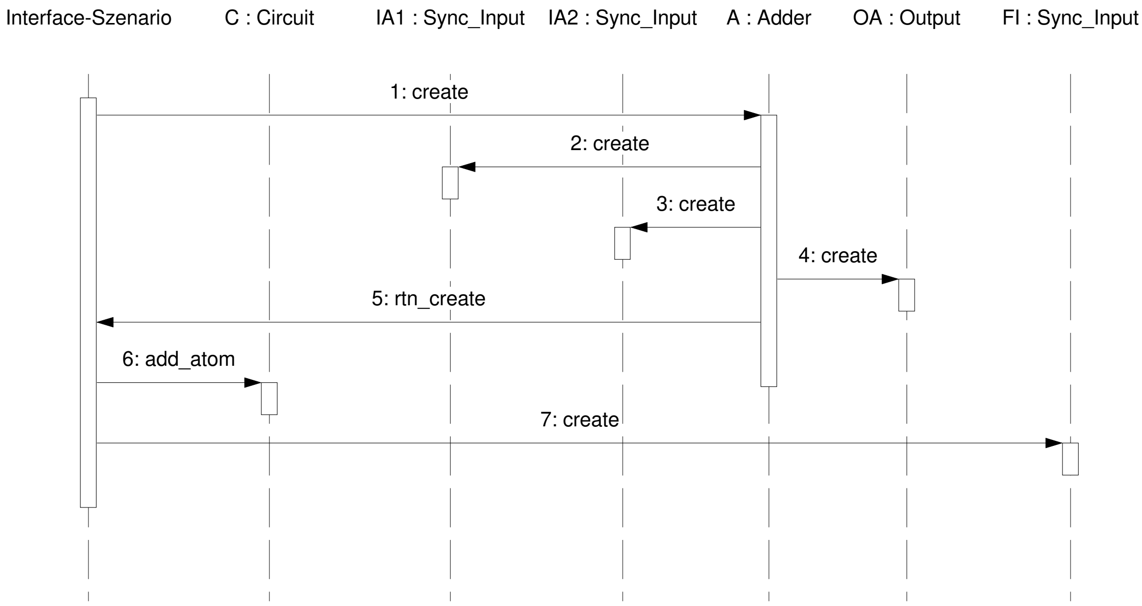

Abbildung 6.3: ETD: Addierer erzeugen und der Schaltung hinzufügen

(2, 3: create) sowie den Ausgang *OA (4: create)*. Der Abschluß des Erzeugungsvorganges wird dem Interface-Szenario mitgeteilt *(5: rtn_create)* und der Addierer wird der Schaltung hinzugefügt *(6: add_atom)*. Zuletzt wird noch demonstriert, wie ein freier Eingang *FI* direkt vom Interface-Szenario erzeugt wird *(7: create)*.

Weitere Atome bzw. freie Ein- und Ausgänge können auf die gleich Art und Weise der Schaltung hinzugefügt werden. Die Hauptaufgaben eines Atoms bei seiner Erzeugung sind das Initialisieren des Zustandes und das Erzeugen der eigenen Ein- und Ausgänge.

Abbildung 6.4 zeigt, wie der Ausgang *OC* des Konstanten-Atoms *Con* mit den beiden Eingängen *IA1* und *IA2* des Addierers *A* verbunden wird. Da dem Interface-Szenario die Ein- und Ausgänge, die von den Atomen erzeugt werden, nicht direkt bekannt sind, verfügen die Atome zum Abfragen eines bestimmten Ein- oder Ausganges über die Zugriffsfunktionen *get_sync_input* bzw. *get_output* (siehe Aufrufe 1, 3, 7 und 11). Das Verbinden selbst geschieht über das *connect* Service der Ein- und Ausgänge und besteht aus zwei Aufrufen: Ausgänge müssen mit Eingängen verbunden werden *(5, 9: connect)* und umgekehrt die Eingänge mit den Ausgängen *(6, 10: connect)*. Am unteren Ende des ETD wird der Ausgang *OA* des Addierers mit dem freien Eingang *FI* verbunden *(13, 14: connect)*.

Abbildung 6.5 zeigt vereinfacht den Start der Beispielschaltung. Das Interface-Szenario kommuniziert in diesem Fall nicht direkt mit den einzelnen Atomen, sondern fordert das übergeordnete Schaltungs-Objekt *C* zum Starten auf *(1: start)*. Das Schaltungs-Objekt startet daraufhin die beiden Atome der Schaltung, *Con* und *A (2, 6: start, 3, 7: rtn_start)*. In den Atomen beginnt sodann die Abarbeitung, was im ETD durch den Aufruf des Kontroll-Objekts *AC* angedeutet ist *(4, 7: control)*. Das Kontroll-Objekt ist von der Klasse *Atom_Controller* und deshalb notwendig, weil Prozesse in LOTOS nicht mit sich selbst kommunizieren können (auf den objektorientierten LOTOS-Stil umgelegt heißt das, daß ein Objekt keine Services von sich selbst aufrufen kann). Die Aufgabe des Kontroll-Objekts wird in den nächsten ETDs genauer beschrieben. Vorerst veranlaßt es das Konstanten-

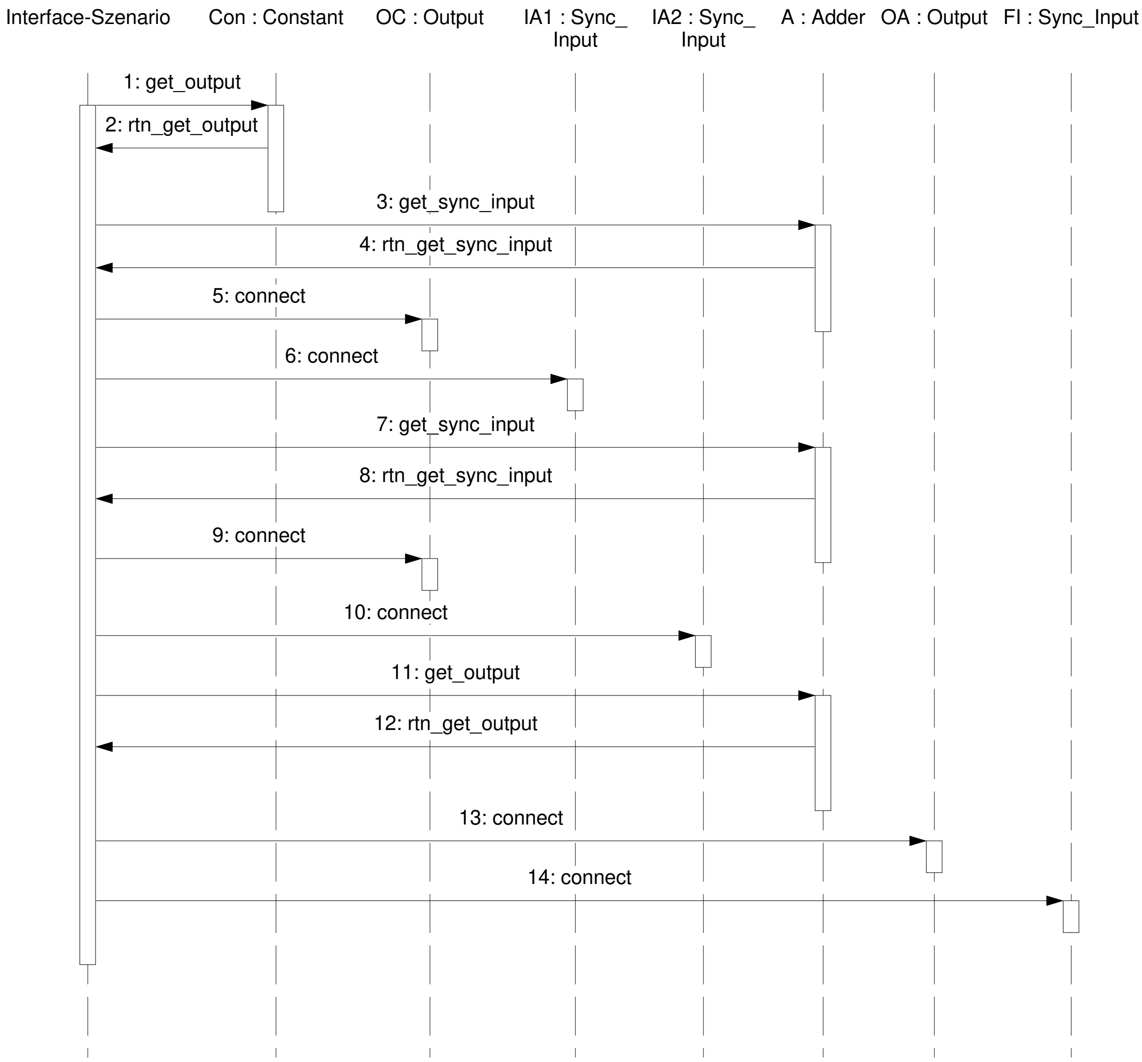

Abbildung 6.4: ETD: Ein- und Ausgänge verbinden

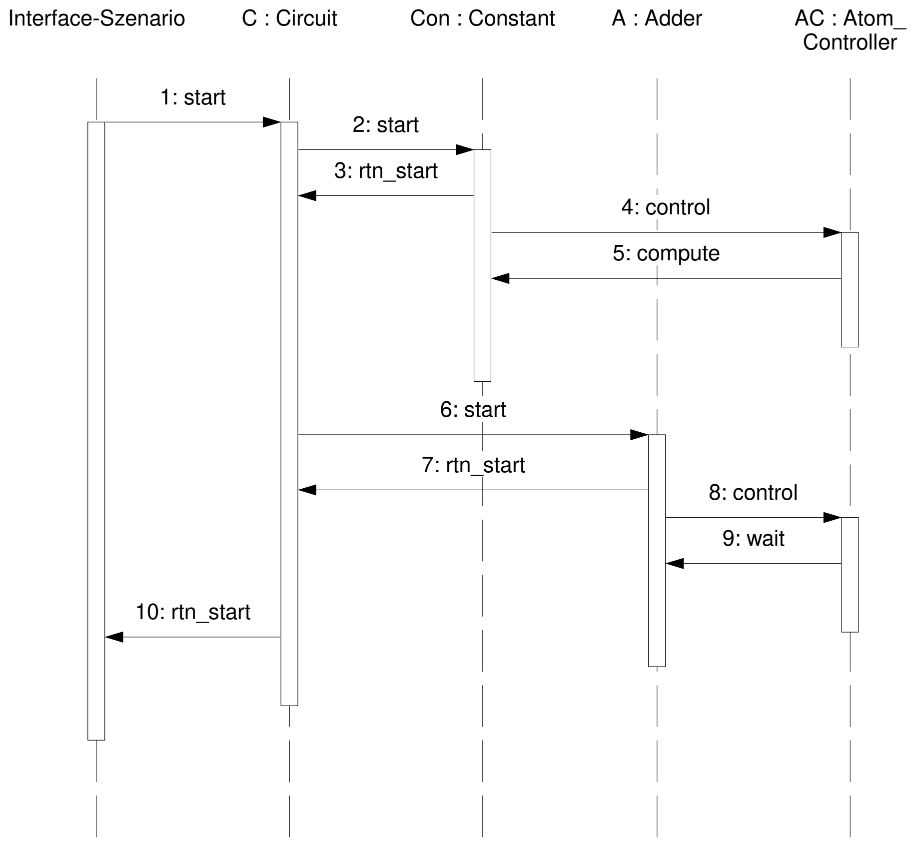

Abbildung 6.5: ETD: Start einer Schaltung

Atom, den Berechnungsschritt durchzuführen *(5: compute)*, während der Addierer noch warten muß *(9: wait)*. Abschließend erhält das Interface-Szenario vom Schaltungs-Objekt die Rückmeldung, daß die Schaltung gestartet wurde *(10: rtn_start)*.

Das ETD in Abbildung 6.6 veranschaulicht das Verhalten des Konstanten-Atoms unmittelbar nach dem Start. Insbesondere wird die Kommunikation zwischen dem Ausgang *OC* und den nachgeschalteten Eingängen *IA1* und *IA2* nachvollzogen.

Nach dem Starten des Atoms *(1: start)* durch das Schaltungs-Objekt wird zunächst der asynchrone Eingang *IC* aktualisiert *(2: update)* und der Schaltung der erfolgreiche Start zurückgemeldet *(3: rtn_start)*. Anschließend ruft das Atom das Kontroll-Objekt auf, das bestimmt, welche Aktion als nächstes durchgeführt werden soll *(4: control)*. Das Kontroll-Objekt ruft als Reaktion eines der folgenden Services des Atoms auf:

- *compute:* Die Schaltbedingung ist erfüllt und das Atom kann aus den Eingängen die Ausgänge berechnen.
- *update_connections:* Das Atom hat den Berechnungsschritt durchgeführt, die Eingänge können freigegeben und die Ausgänge gesetzt werden.
- *wait:* Die Schaltbedingung ist nicht erfüllt. Das Atom muß warten, bis weitere Eingänge gesetzt oder Ausgänge frei werden.

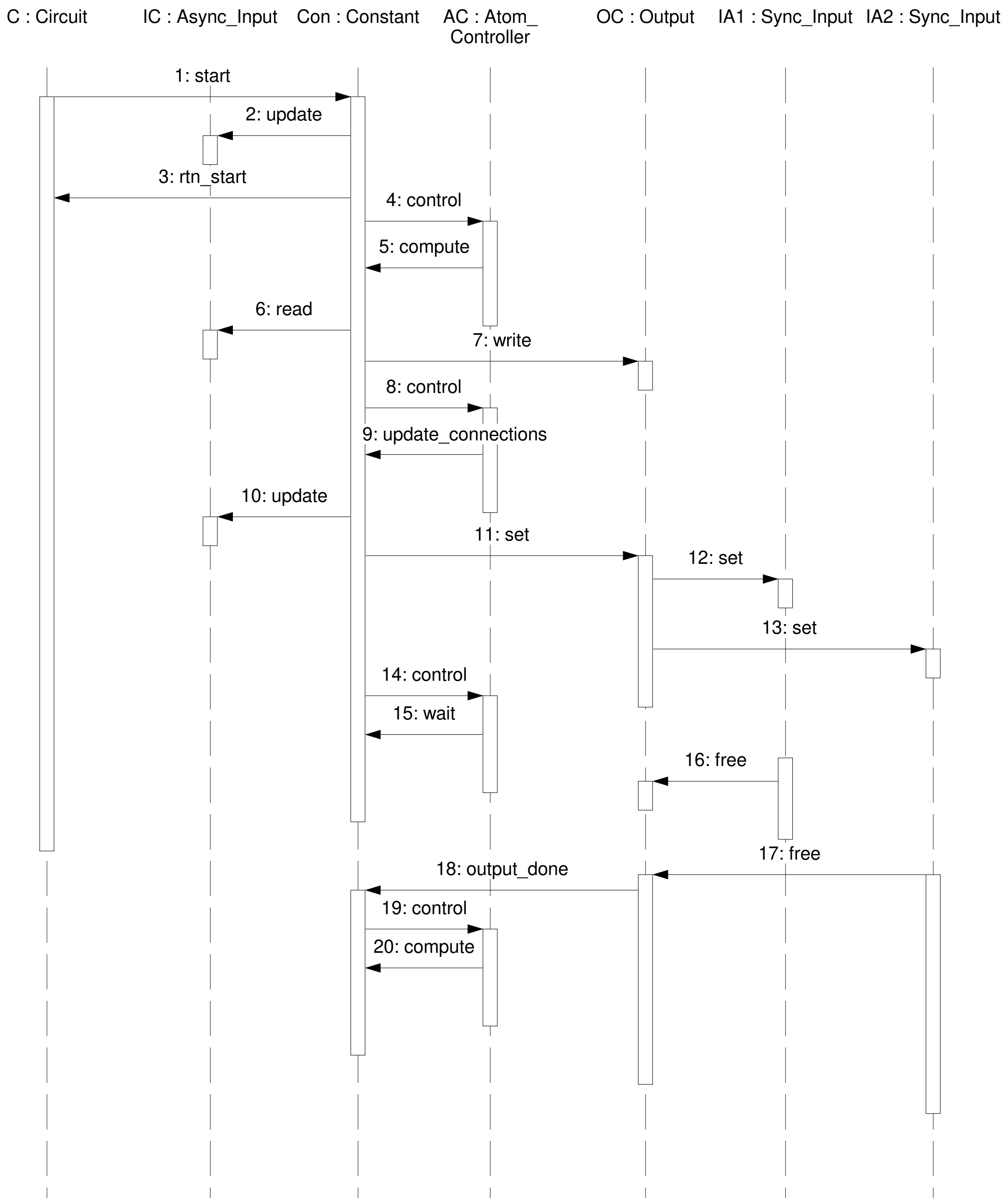

Abbildung 6.6: ETD: Schritt eines Konstanten-Atoms

Beim neu gestarteten Konstanten-Atom ist die Schaltbedingung sofort erfüllt, weil auf keine synchronen Eingänge mehr gewartet werden muß: der Berechnungsschritt kann sofort durchgeführt werden *(5: compute)*. In ihm wird der Wert des asynchronen Eingangs gelesen *(6: read)* und an den Ausgang gestellt *(7: write)*. Im nachfolgenden Aufruf des Kontroll-Objekts *(8: control)* wird festgestellt, daß der Berechnungsschritt vorbei ist und die Ein- und Ausgänge aktualisiert werden müssen *(9: update_connections)*. Im Zuge dessen wird der Wert des asynchronen Einganges aktualisiert *(10: update)* und der Ausgang gesetzt *(11: set)*. Der Ausgang versorgt seinerseits die nachgeschalteten Eingänge mit Daten *(12, 13: set)*. Es folgt wiederum ein Aufruf des Kontroll-Objekts *(14: control)*, weil aber der Ausgang nicht bereit ist, kann das Atom im Moment nicht schalten und begibt sich in Warteposition *(15: wait)*. Wir nehmen an dieser Stelle an, daß der nachgeschaltete Addierer, dem die beiden Eingänge *IA1* und *IA2* gehören, ebenfalls geschaltet hat und seine Eingänge freigibt. Die Eingänge melden das an den vorgeschalteten Ausgang *(16, 17: free)*, der wiederum dem Atom bekanntgibt, daß er frei geworden ist *(18: output_done)*. Ein freier Ausgang kann eine Veränderung in der Schaltbedingung hervorrufen und löst deshalb einen Aufruf des Kontroll-Objekts aus *(19: control)*, der mit der Aufforderung, den nächsten Berechnungsschritt durchzuführen, beantwortet wird *(20: compute)*.

Das ETD in Abbildung 6.7 vollzieht den selben Vorgang für das Addier-Atom. Besonderes Augenmerk liegt auf der Kommunikation der beiden Eingänge des Addierers mit dem Ausgang *OC* des Konstanten-Atoms. Nach dem Starten des Addierers *(1: start)* ruft dieser zunächst das Kontroll-Objekt auf *(2: control)*. Aufgrund der leeren Eingänge ist die Schaltbedingung nicht erfüllt und der Addierer muß warten *(3: wait)*. An dieser Stelle gehen wir davon aus, daß das vorgeschaltete Konstanten-Atom seinen Ausgang gesetzt hat, wodurch wiederum die Eingänge des Addierers gesetzt werden *(4, 8: set)*. Die Eingänge melden den Erhalt neuer Daten dem Atom *(5, 9: input_set)*, was zu Aufrufen des Kontroll-Objekts führt *(6, 10: control)*. Nach dem Setzen des ersten Einganges muß der Addierer noch warten *(7: wait)*, beim zweiten Eingang ist die Schaltbedingung dann erfüllt und das Atom führt seinen Rechenschritt aus *(11: compute)*. Dabei liest es die Daten von den beiden Eingängen *(12, 14: read; 13, 15: rtn_read)* und schreibt die Summe in den Ausgang *(16: write)*. Durch den nachfolgenden Aufruf des Kontroll-Objekts *(17: control)* wird der Addierer veranlaßt, die Ein- und Ausgänge zu aktualisieren *(18: update_connections)*. Im Zuge dessen werden die beiden Eingänge freigegeben *(19, 20: free)* und der Ausgang gesetzt *(21: set)*. Am Ende dieses Vorgangs wird noch einmal das Kontroll-Objekt aufgerufen *(22: control)*, welches den Addierer zum Warten veranlaßt *(23: wait)*.

Die ETDs in den Abbildungen 6.6 und 6.7 zeigen jeweils den ersten Berechnungsschritt ab dem Starten des Atoms. Das Schaltverhalten setzt sich prinzipiell endlos fort, zumindest solange am freien Eingang *FI* Daten abgeholt werden oder die Schaltung gestoppt wird.

Das Stoppen einer Schaltung gestaltet sich etwas komplizierter als das Starten, weil eine Schaltung erst dann als gestoppt gelten soll, wenn keinerlei Nachrichten mehr zwischen den beteiligten Objekten ausgetauscht werden. D. h., der gestoppte Zustand ist erst dann erreicht, wenn keine Kommunikation mehr zwischen Eingängen, Ausgängen und Atomen stattfindet, so wie sie in den vorangegangenen ETDs gezeigt wurde.

Abbildung 6.8 zeigt das Stoppen der Beispielschaltung. Nach dem Aufruf durch das Interface-Szenario *(1: stopp)* leitet das Schaltungs-Objekt *C* einen zweistufigen Vorgang ein:

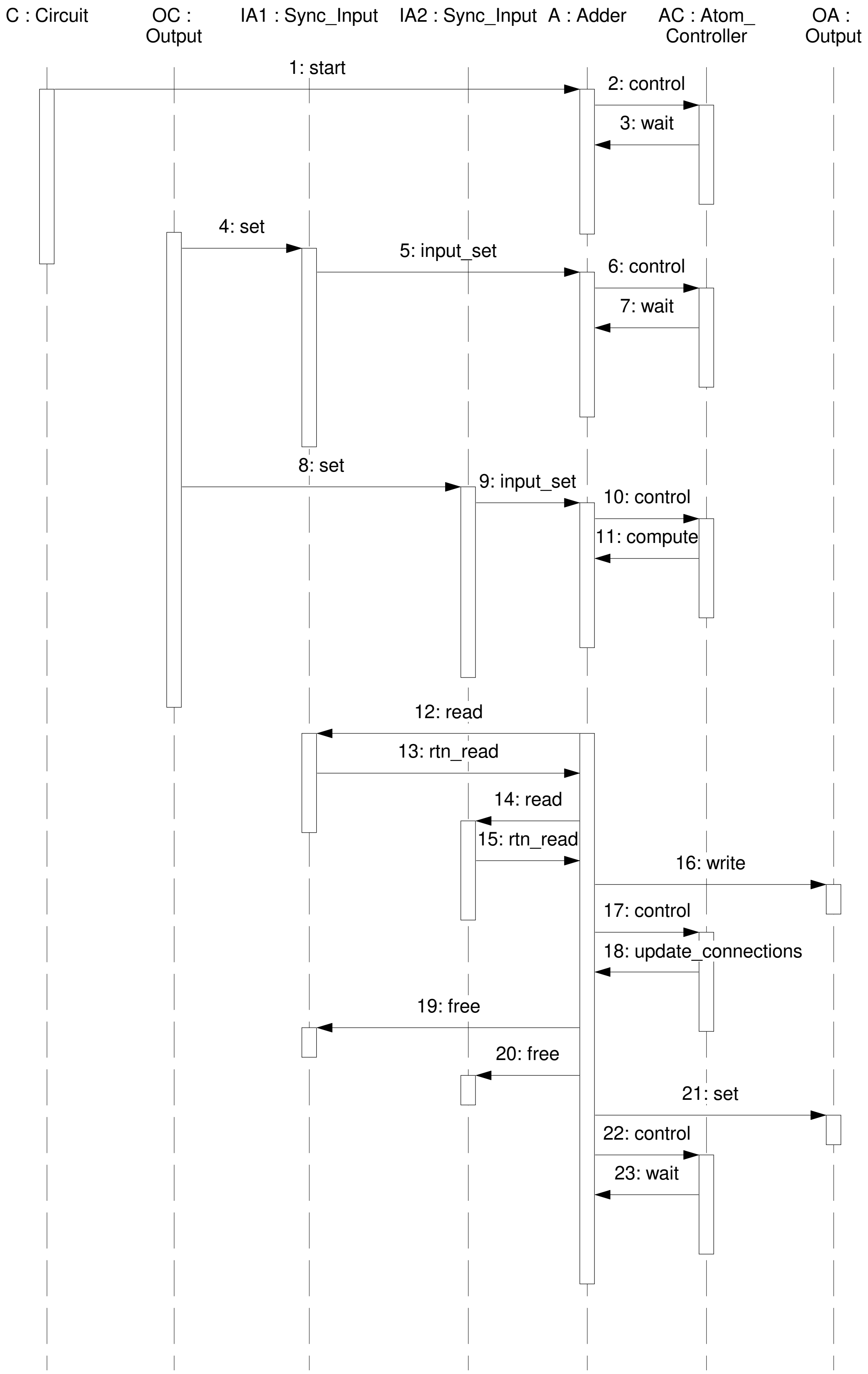

Abbildung 6.7: ETD: Schritt eines Addierers

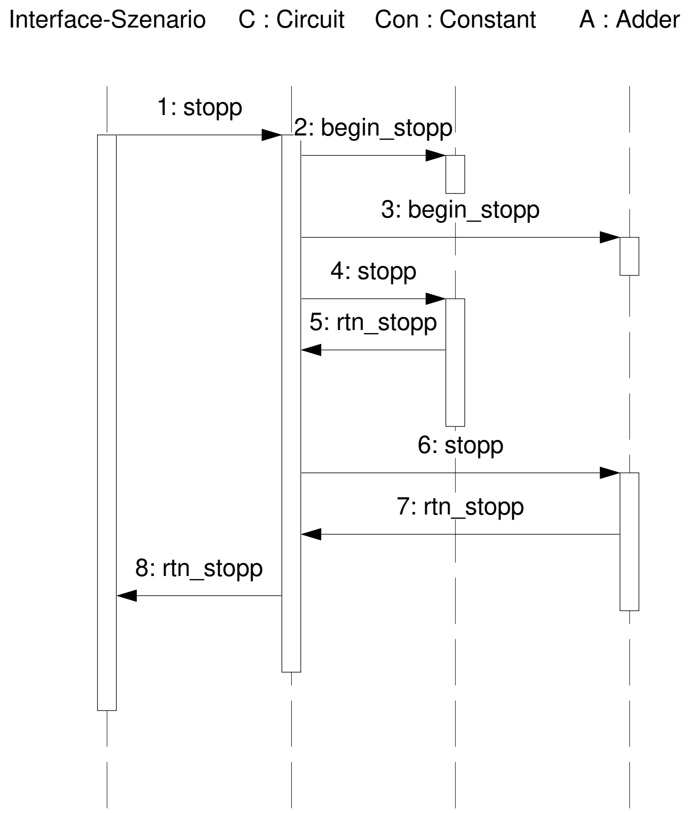

Abbildung 6.8: ETD: Stoppen einer Schaltung

1. Allen Atomen der Schaltung wird mitgeteilt, daß die Schaltung angehalten wird *(2, 3: begin_stopp)* und sie ihre Aktivitäten an den Ein- und Ausgängen einstellen sollen (keine weiteren *compute-* und *update_connections*-Messages). Dadurch wird sichergestellt, daß nach Ende der 1. Stufe kein neuer Datenfluß zwischen Atomen initiiert wird.

2. Die Atome werden endgültig gestoppt *(4, 6: stopp)*. Bei der Rückmeldung an das Schaltungs-Objekt *(5, 7: rtn_stopp)* steht fest, daß von den Ein- und Ausgängen des Atoms keine Nachrichten mehr ausgehen und sich das Atom selbst im Ruhezustand befindet.

Sind alle Atome gestoppt, meldet das Schaltungs-Objekt diesen Umstand an das Interface-Szenario *(8: rtn_stopp)*. Der Systemteil, der die Schaltung benutzt, kann nun davon ausgehen, daß keinerlei Aktivität mehr innerhalb der Schaltung vorhanden ist. Sie kann jetzt sicher abgebaut oder rückgesetzt werden.

Die 2. Stufe des Stop-Vorgangs wird anhand des Addierers in Abbildung 6.9 nachvollzogen. Wir gehen davon aus, daß das Schaltungs-Objekt *C* vom Interface-Szenario das Kommando zum Stoppen erhalten hat und bereits die 1. Stufe des Stop-Vorgangs durchgeführt hat. Zum Stoppen des Atoms kommt wieder ein Hilfsobjekt zum Einsatz, das Stop-Objekt *AS*. Es sorgt dafür, daß das Atom erst dann in den gestoppten Zustand übergeht, wenn von den Ein- und Ausgängen keine Nachrichten mehr an das Atom ge-

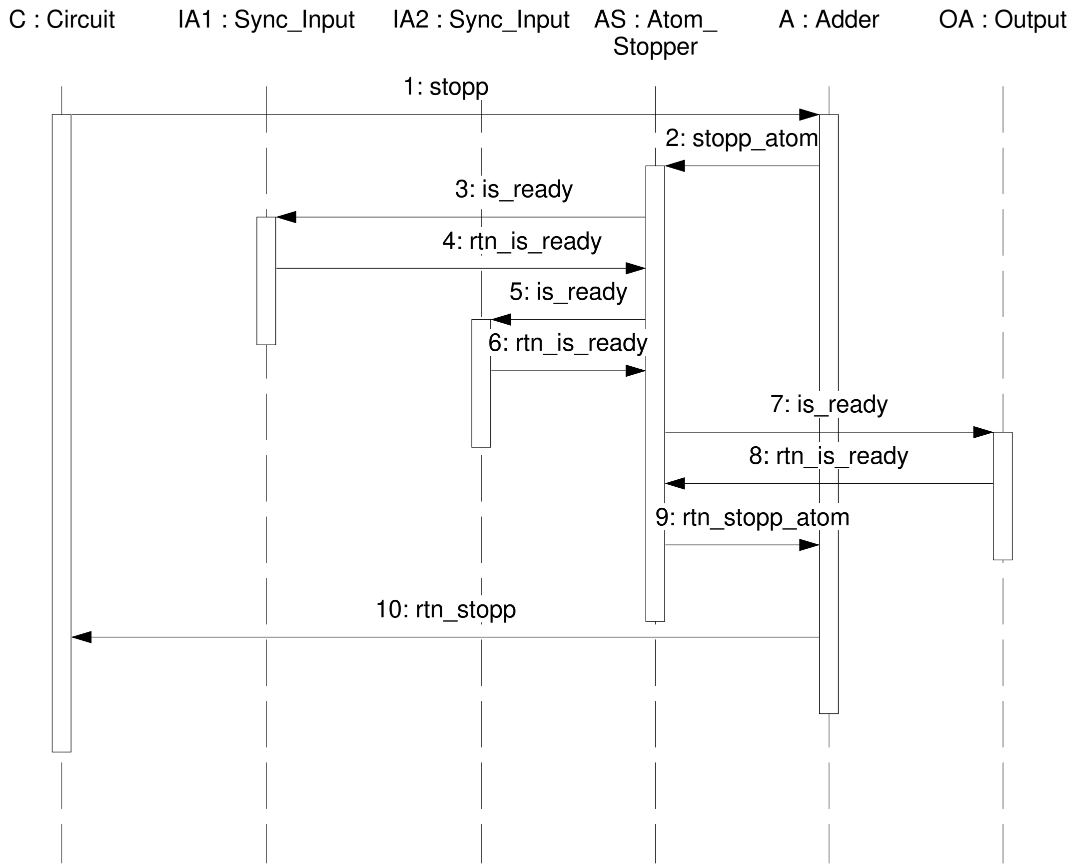

Abbildung 6.9: ETD: Stoppen eines Addierers

schickt werden können. Ein externes Objekt ist deshalb notwendig, weil es sonst zu einem Deadlock zwischen dem Atom und einem seiner Ein- bzw. Ausgängen kommen könnte.

Der Addierer ruft sofort nach dem Befehl anzuhalten *(1: stopp)* das Stop-Objekt auf *(2: stopp_atom)*. Dieses überprüft alle synchronen Eingänge und alle Ausgänge, ob sie noch beschäftigt sind *(3, 5, 7: is_ready; 4, 6, 8: rtn_is_ready)*. Ist das Stop-Objekt mit der Überprüfung fertig *(9: rtn_stopp_atom)*, kann das Atom endgültig als gestoppt angesehen werden *(10: rtn_stopp)*.

Abschließend behandeln wir noch das Rücksetzen von Schaltungen. Das ETD in Abbildung 6.10 zeigt, wie die Beispielschaltung aus Abbildung 6.2 zurückgesetzt wird. Zunächst geht vom Interface-Szenario an die Schaltung die Aufforderung aus, sich zurückzusetzen *(1: reset)*. Die Schaltung setzt daraufhin alle beteiligten Atome zurück *(2, 4: reset)* und wartet auf Bestätigung *(3, 5: rtn_reset)*, bevor es die Kontrolle an das Interface-Szenario zurückgibt *(6: rtn_reset)*. Da freie Ein- und Ausgänge nicht Teil von Schaltungen sind, müssen sie vom besitzenden Systemteil (in diesem Fall direkt vom Interface-Szenario) zurückgesetzt werden *(7, 9: reset; 8, 10: rtn_reset)*.

Beim Rücksetzen eines Atoms werden seine Ein- und Ausgänge in den leeren Zustand versetzt. Der Zustand des Atoms selbst muß ebenfalls in eine definierte Ausgangsposition gebracht werden. Den Vorgang des Rücksetzens eines Atoms zeigt Abbildung 6.11 anhand des Addierers A. Das vom Schaltungs-Objekt ausgehende Rücksetz-Kommando *(1: reset)*

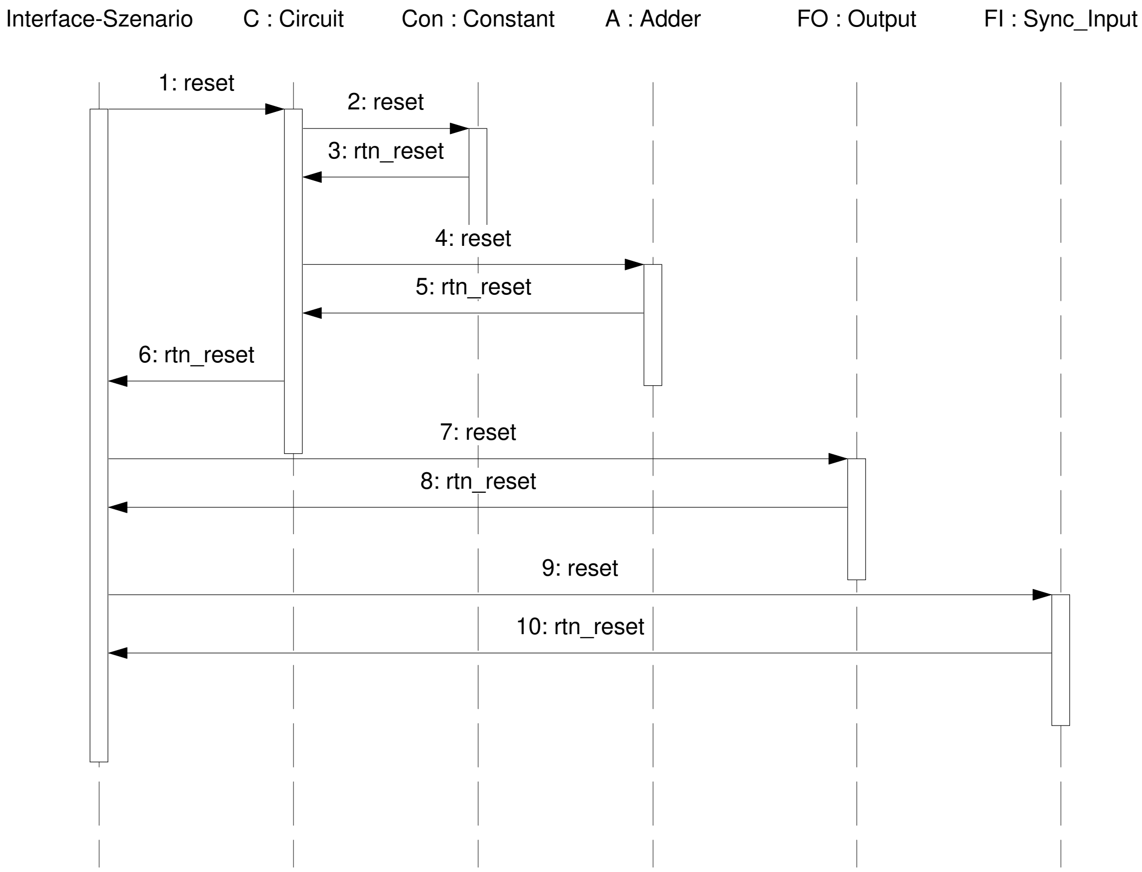

Abbildung 6.10: ETD: Rücksetzen einer Schaltung

wird vom Addierer an alle Ein- und Ausgänge weitergegeben *(2, 4, 6: reset)*. Mit dem Erhalt der Bestätigungen *(3, 5, 7: rtn_reset)* kann der Addierer als zurückgesetzt betrachtet werden *(8: rtn_reset)*.

Die Object Communication Table (OCT) wird zu Beginn von Schritt 3 gemeinsam mit dem Object Communication Diagram (OCD) erstellt.

Message Connections hinzufügen

Das in dieser Arbeit verwendete Programm zum Zeichnen der Objekt-Modelle bietet nicht die Möglichkeit, Message Connections als solche im Diagramm einzuzeichnen, weshalb wir darauf verzichten müssen.

6.3 Schritt 2.3: Objekt-Modell strukturieren

Abbildung 6.12 zeigt das verfeinerte Objekt-Modell des Schaltungssystems. Es enthält die beschriebenen Klassen mit den wichtigsten Attributen und den mit Hilfe der ETDs identifizierten Services. Gegenüber dem Objekt-Modell von Abbildung 6.1 sind die Klassen *Atom_Controller* und *Atom_Stopper* dazugekommen, die beide mit der Atom-Klasse assoziiert sind. An der Klassenhierarchie selbst hat sich nichts geändert. Für die Ein-

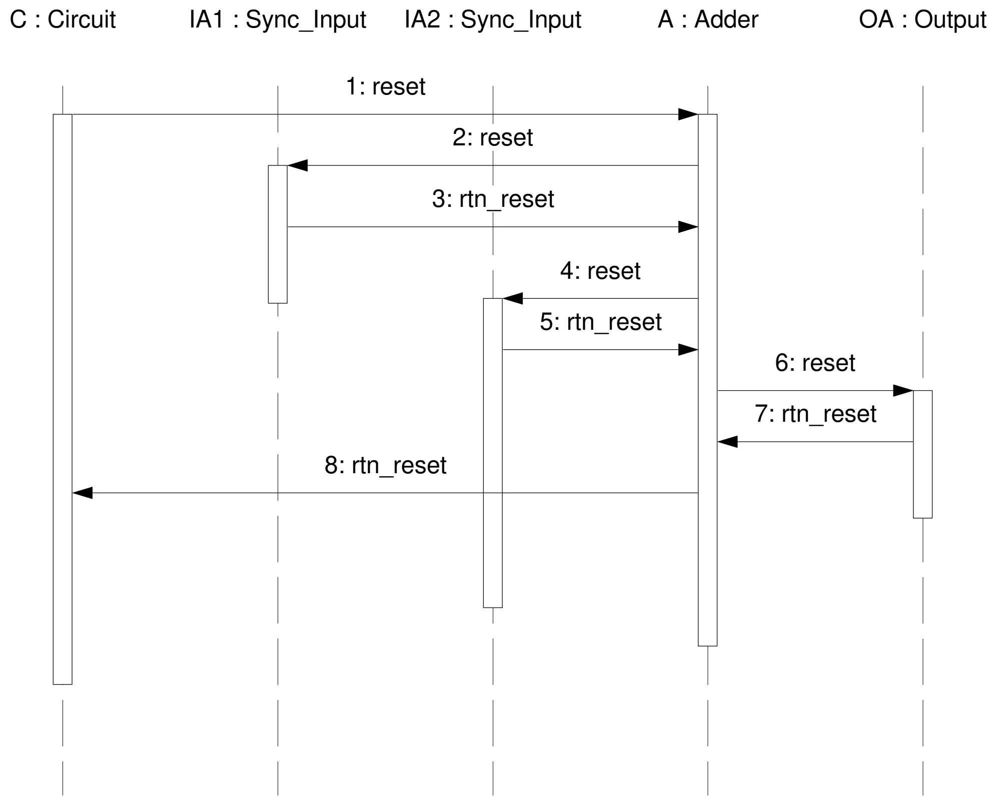

Abbildung 6.11: ETD: Rücksetzen eines Addierers

und Ausgänge wird keine Basisklasse eingeführt, weil ihr Verhalten zu unterschiedlich ist. Stattdessen werden besagte Klassen zusammen mit der Klasse *Data* im Subsystem *Connections* zusammengefaßt. Der von der Atom-Klasse ausgehende Klassenbaum ist mit den beiden Hilfsklassen *Atom_Controller* und *Atom_Stopper* im Subsystem *Atoms* untergebracht.

Die größten Änderungen hat die Klasse *Atom* erfahren. Das Attribut *step_mode* gibt an, in welchem Schritt der Abarbeitung sich das Atom gerade befindet (auf Ein- und Ausgänge wartend oder mit dem Berechnungsschritt fertig). In den Attributen *set_inputs* und *done_outputs* wird Buch geführt, welche synchronen Eingänge gesetzt bzw. welche Ausgänge frei sind. Die Attribute *required_inputs* und *required_outputs* geben die im Service *compute* verwendeten synchronen Eingänge bzw. Ausgänge an, die vom nachfolgenden *update_connections* freigegeben und gesetzt werden müssen.

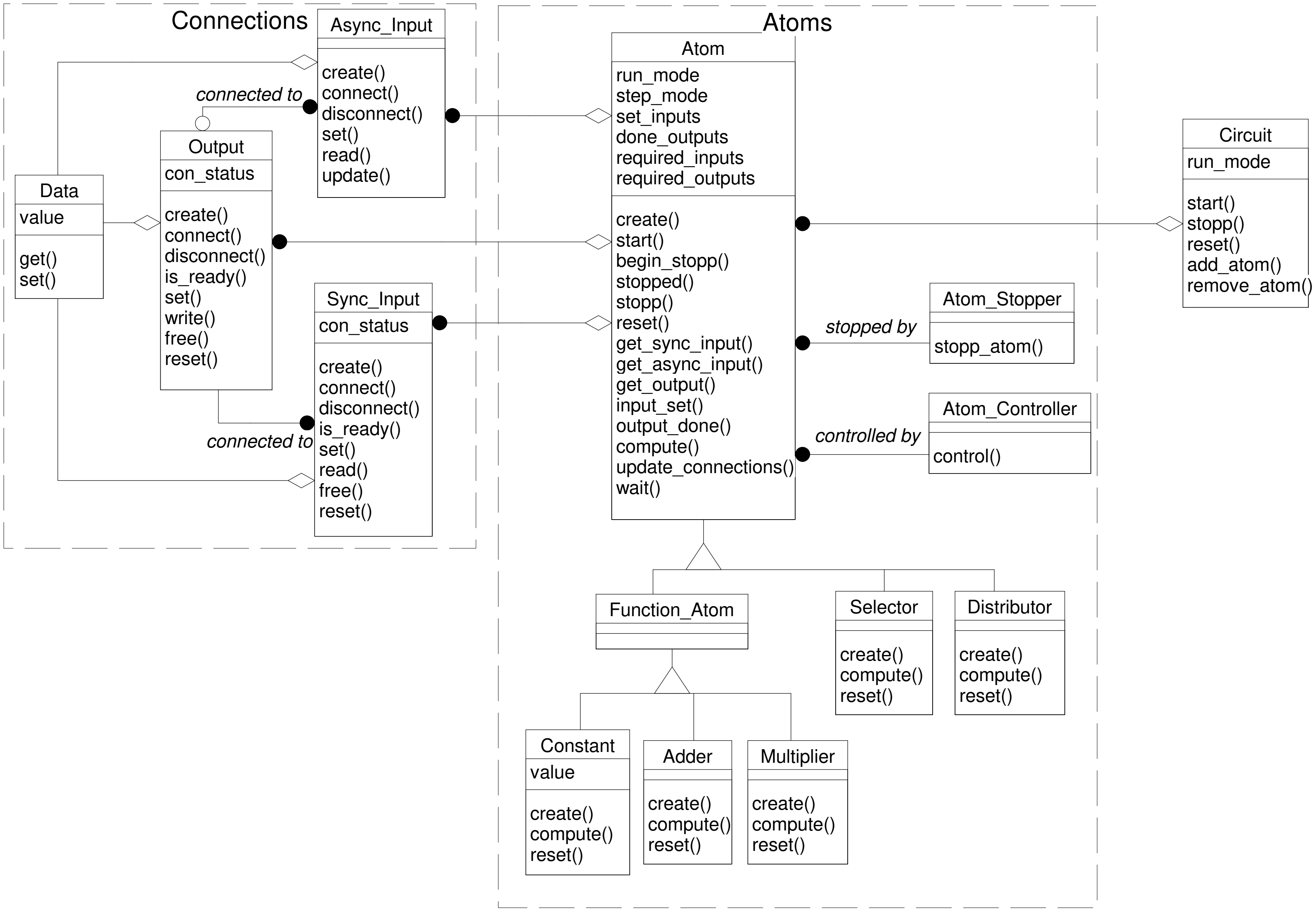

Abbildung 6.12: Verfeinertes Objekt-Modell

Kapitel 7

Schritt 3: LOTOS-Spezifikation erstellen

In diesem Kapitel wird eine ablauffähige LOTOS-Spezifikation des Schaltungssystems erstellt. Die im Objekt-Modell gegebenen Klassen werden, wie in Kapitel 2 beschrieben, in objektorientierte Prozesse umgesetzt. Um das angestrebte Verhalten des Systems zu verifizieren, soll die Spezifikation anschließend simuliert werden.

7.1 Schritt 3.1: OCT und OCD erstellen

Im Object Communication Diagram (OCD, siehe Abbildung 7.1) wird die Struktur der LOTOS-Objekt-Prozesse festgehalten. Simultan wird die Object Communication Table (OCT, siehe Tabelle 7.1) erstellt. Damit ist eine Zuordnung der Klassen zu Prozessen gegeben und eine Übersicht, welche Services an welchem Gate angeboten werden.

Das Schaltungs-Objekt *Circuit* kommuniziert über das Gate `i_c` mit der Umgebung, über das Gate `c_a` kommuniziert es mit den Atomen. Die Prozesse für die Atom-Klassen *Adders, Constants, Selectors* und *Distributors* sind bereits im Plural angegeben, weil es sich eigentlich um die Objekt-Generatoren der betreffenden Klassen handelt. Sie bilden mit den beiden Hilfsklassen *Atom_Controller* und *Atom_Stopper* ein eigenes Subsystem und werden als Subprozesse innerhalb des Prozesses *Atoms* dargestellt. Atome kommunizieren über das Gate `i_a` mit der Umgebung und über das Gate `c_a` mit dem Schaltungs-Objekt. Für die Kommunikation mit den Ein- und Ausgängen steht das Gate `a_con` zur Verfügung. Die Kommunikation mit dem Kontroll-Objekt und dem Stop-Objekt findet über die internen Gates `a_ac` bzw. `a_ac` statt (auf das Einzeichnen der Verbindungen zwischen den Atomen und den beiden letztgenannten Objekten wird aus Gründen der Übersichtlichkeit verzichtet).

Ein- und Ausgänge sind Unterprozesse des *Connections*-Prozesses. Im Diagramm scheinen wiederum die Objekt-Generatoren *Sync_Inputs*, *Async_Inputs* und *Outputs* auf. Die Umgebung kann über das Gate `i_con` Services dieser Objekte aufrufen. Wie bereits erwähnt, dient das Gate `a_con` zur Kommunikation mit den Atomen. Mittels des internen Gates `con` rufen die Ein- und Ausgänge gegenseitig Services voneinander auf.

Klasse	**angebotene Services**	**verwendete Services**	**Clients**	**Gate**
Circuit	start	Atom.start	Interface-Szenario	i_c
	stopp	Atom.begin_stopp	Interface-Szenario	i_c
		Atom.stopp	Interface-Szenario	
	reset	Atom.reset	Interface-Szenario	i_c
	add_atom		Interface-Szenario	i_c
	remove_atom		Interface-Szenario	i_c
Atoms (Constant, Adder, Selector, Distributor)	create	SI.create	Interface-Szenario	i_a
		AI.create		
		Output.create		
	start	AI.update	Circuit	c_a
		AC.control		
	begin_stopp		Circuit	c_a
	stopp	AS.stopp_atom	Circuit	c_a
	stopped		AS	a_as
	reset	SI.reset	Circuit	c_a
		Output.reset		
	get_sync_input		Interface-Szenario	i_a
	get_async_input		Interface-Szenario	i_a
	get_output		Interface-Szenario	i_a
	wait		AC	a_ac
	compute	SI.read	AC	a_ac
		AI.read		
		Output.write		
		AC.control		
	upd_conn	SI.free	AC	a_ac
		AI.update		
		Output.set		
		AC.control		
	input_set	AC.control	Sync_Input	a_con
	output_done	AC.control	Output	a_con
Atom_Controller (AC)	control	Atom.compute	Atom	a_ac
		Atom.upd_conn		
		Atom.wait		
Atom_Stopper (AS)	stopp_atom	Atom.stopped	Atom	a_as
		SI.is_ready		
		Output.is_ready		
Sync_Input (SI)	create		Atom	a_con
			Interface-Szenario	i_con
	connect		Interface-Szenario	i_con
	disconnect		Interface-Szenario	i_con
	reset		Atom	a_con
			Interface-Szenario	i_con
	set	Atom.input_set	Output	con
		Szenario.input_set		
	read		Atom	a_con
			Interface-Szenario	i_con
	free	Output.free	Atom	a_con
			Interface-Szenario	i_con
	is_ready		Atom_Stopper	a_con
			Interface-Szenario	i_con

Tabelle 7.1: Initiale Object Communication Table

Klasse	**angebotene Services**	**verwendete Services**	**Clients**	**Gate**
Async_Input (AI)	create		Atom	a_con
			Interface-Szenario	i_con
	connect		Interface-Szenario	i_con
	disconnect		Interface-Szenario	i_con
	update		Atom	a_con
			Interface-Szenario	i_con
	read		Atom	a_con
			Interface-Szenario	i_con
	set	Output.free	Output	con
Output	create		Atom	a_con
			Interface-Szenario	i_con
	connect		Interface-Szenario	i_con
	disconnect		Interface-Szenario	i_con
	reset		Atom	a_con
			Interface-Szenario	i_con
	set	SI.set	Atom	a_con
		AI.set	Interface-Szenario	i_con
	write		Atom	a_con
			Interface-Szenario	i_con
	free	Atom.output_done	Sync_Input	con
		Szenario.output_done	Async_Input	con
	is_ready		Atom_Stopper	a_con
			Interface-Szenario	i_con

Tabelle 7.1 – Fortsetzung

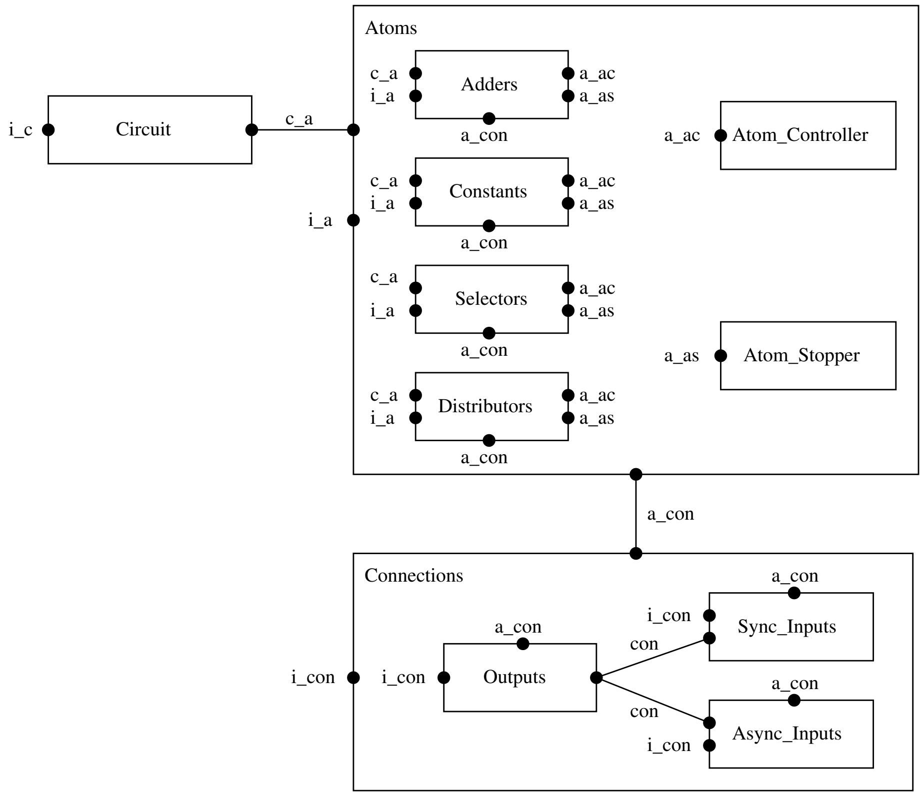

Abbildung 7.1: OCD des Schaltungssystems

7.2 Schritt 3.2: Klassenschablonen spezifizieren

In diesem Schritt wird aus dem Objekt-Modell, den Event Trace Diagrammen, der OCT und dem OCD eine ablauffähige, objektorientiert LOTOS-Spezifikation erstellt. Für jede Klasse muß dazu ein abstrakter Datentyp spezifiziert werden, der den Zustand und den eindeutigen Objekt-Kennzeichner jedes Objekts beinhaltet. Das dynamische Verhalten aktiver Objekte wird als LOTOS-Prozeß realisiert. Dort, wo mehrere Objekte von einer Klasse erzeugt werden müssen, werden anschließend noch Objekt-Generatoren eingefügt.

Spezifikation der Schaltungsklasse

Der nachfolgende ADT spezifiziert den Zustand `State_Circuit` der Schaltungsklasse:

```
220 type Circuit_Type is Circuit_Id_Type, Atom_Id_Set_Type, Run_Mode_Type
221 sorts State_Circuit
222 opns
223    Make_Circuit : Circuit_Id, Atom_Id_Set, Run_Mode -> State_Circuit
224    Get_Id : State_Circuit -> Circuit_Id
225    Get_Atoms : State_Circuit -> Atom_Id_Set
226    Add_Atom : State_Circuit, Atom_Id -> State_Circuit
227    Remove_Atom: State_Circuit, Atom_Id -> State_Circuit
228    Get_Run_Mode : State_Circuit -> Run_Mode
229    Set_Run_Mode : State_Circuit, Run_Mode -> State_Circuit
230 eqns
231    forall c: Circuit_Id, a: Atom_Id, a_ids: Atom_Id_Set,
232           r, rn: Run_Mode
233       ofsort Circuit_Id
234          Get_Id(Make_Circuit(c, a_ids, r))=c;
235       ofsort Atom_Id_Set
236          Get_Atoms(Make_Circuit(c, a_ids, r))=a_ids;
237       ofsort Run_Mode
238          Get_Run_Mode(Make_Circuit(c, a_ids, r))=r;
239       ofsort State_Circuit
240          Set_Run_Mode(Make_Circuit(c, a_ids, r), rn)=
241             Make_Circuit(c, a_ids, rn);
242          Add_Atom(Make_Circuit(c, a_ids, r), a)=
243             Make_Circuit(c, Insert(a, a_ids), r);
244          Remove_Atom(Make_Circuit(c, a_ids, r), a)=
245             Make_Circuit(c, Remove(a, a_ids), r);
246 endtype
```

Die Typen der Attribute können am Konstruktor `Make_Circuit` (Zeile 223) abgelesen werden:

- Eine Objekt-Id für jede erzeugte Schaltung, vom Typ `Circuit_Id`. Die Objekt-Id kann mit der Funktion `Get_Id` abgefragt werden.

- Eine Menge von Atomen, die zur Schaltung gehören, vom Typ `Atom_Set`. Die aktuelle Atom-Menge kann mit `Get_Atoms` abgefragt werden. Atome können mit `Add_Atom` hinzugefügt und mit `Remove_Atom` wieder entfernt werden.

- Den aktuellen Lauf-Zustand der Schaltung, festgehalten in einem Attribut vom Typ `Run_Mode`. Der Lauf-Zustand nimmt im Fall der Schaltungs-Klasse entweder den Wert `rm_stopped` oder `rm_started` an. Er wird mit `Get_Run_Mode` gelesen und mit `Set_Run_Mode` gesetzt.

Das Beispiel des ADTs für die Schaltungs-Klasse zeigt auch, wie die Gleichungen für die Operationen des ADT spezifiziert werden. Da `State_Circuit` nur über einen einzigen Konstruktor verfügt, muß für jede Operation auch nur eine Gleichung angegeben werden, die ausdrückt, wie sich die Operation zum Konstruktor `Make_Circuit` verhält. Die Selektoren `Get_Id`, `Get_Atoms` und `Get_Run_Mode` liefern jeweils das entsprechende Datenfeld aus dem Konstruktor zurück. Die Modifikatoren `Set_Run_Mode`, `Add_Atom` und `Remove_Atom` liefern auf ihrer rechten Seite wieder ein durch den Konstruktor definiertes Schaltungs-Objekt. In den nachfolgenden Darstellungen der ADTs wird auf die Angabe der Gleichungen verzichtet. Sie sind aus der vollständigen Spezifikation des Schaltungssystems im Anhang ersichtlich.

Das dynamische Verhalten der Schaltungs-Klasse geht aus der folgenden Prozeßdefinition hervor:

```
process Circuit[i_c, c_a](state: State_Circuit) : noexit :=
(
   [Get_Run_Mode(state) eq rm_stopped] ->
   (
      (
         i_c !start !Get_Id(state);
         Start_Atoms[c_a](Get_Atoms(state)) >>
         exit(Set_Run_Mode(state, rm_started))
      )
      []
      (
         i_c !reset !Get_Id(state);
         Reset_Atoms[c_a](Get_Atoms(state)) >>
         i_c !rtn_reset !Get_Id(state);
         exit(state)
      )
      []
      i_c !add_atom !Get_Id(state) ?atom: Atom_Id;
      exit(Add_Atom(state, atom))
      []
      i_c !remove_atom !Get_Id(state) ?atom:Atom_Id;
      exit(Remove_Atom(state, atom))
   )
   []
   [Get_Run_Mode(state) eq rm_started] ->
```

```
639      (
640         i_c !stopp !Get_Id(state);
641         Begin_Stopping_Atoms[c_a](Get_Atoms(state)) >>
642         Stopp_Atoms[c_a](Get_Atoms(state)) >>
643         i_c !rtn_stopp !Get_Id(state);
644         exit(Set_Run_Mode(state, rm_stopped))
645      )
646   ) >> accept update_state: State_Circuit in
647               Circuit[i_c, c_a](update_state)
```

Je nachdem, in welchem Lauf-Zustand sich die Schaltung befindet, kann ein bestimmter Satz von Services aufgerufen werden. Der Versuch, ein anderes Service aufzurufen, führt zum Blockieren des aufrufenden Objekts. Um Deadlocks zu vermeiden, dürfen deshalb nur Services aufgerufen werden, die unmittelbar oder zumindest relativ rasch zur Ausführung gelangen. Generell wird in der Spezifikation der einzelnen Klassen lediglich klar gemacht, welche Services ein Objekt in einem bestimmten Zustand zur Verfügung stellt. Um in einer Implementierung Deadlocks zu vermeiden, müßten im Falle einer Nicht-Verfügbarkeit eines Services eventuell Fehlermeldungen zurückgegeben werden.

Ist die Schaltung im gestoppten Zustand (`Get_Run_Mode(state) eq rm_stopped`, Zeile 616), kann die Schaltung gestartet und zurückgesetzt werden (`start` und `reset`). Weiters können in diesem Zustand Atome der Schaltung bekannt gemacht (`add_atom`) oder von dieser entfernt werden (`remove_atom`). Gestartete Schaltungen (`Get_Run_Mode(state) eq rm_started`, Zeile 638) können ausschließlich angehalten werden (`stopp`).

Schaltungen müssen in den Services `start`, `reset` und `stopp` bestimmte Services bei allen Atomen aufrufen. Diese Arbeit erledigen in der Spezifikation die Unterprozesse `Start_Atoms`, `Reset_Atoms`, `Begin_Stopping_Atoms` und `Stopp_Atoms`. Der nachfolgende Prozeß zeigt die Spezifikation des Unterprozesses `Start_Atoms`. Die übrigen Unterprozesse sind ähnlich aufgebaut und differieren eigentlich nur in dem aufgerufenen Atom-Service.

```
650   process Start_Atoms[x](atoms: Atom_Id_Set) : exit :=
651      [atoms ne {}] ->
652         x !start ?id: Atom_Id [id IsIn atoms];
653         Start_Atoms[x](Remove(id, atoms))
654      []
655      [atoms eq {}] ->
656         exit
657   endproc
```

Spezifikation der Klasse *Atom*

Den Großteil dieses Schrittes macht die Spezifikation der Klassen des Atom-Subsystems aus. Zunächst wird die Basisklasse der Atome *Atom* beschrieben, anschließend die beiden Hilfsklassen *Atom_Controller* und *Atom_Stopper*. Das Ableiten von Atom-Klassen wird anhand der Zwischenklasse *Function_Atom* und dem Addierer *Adder* demonstriert.

Obwohl von der Klasse *Atom* eine Klassenhierarchie ausgeht, kommt die Spezifikation mit der Definition eines einzigen ADT für alle Atom-Klassen aus:

```
type Atom_Type is Atom_Id_Type, Input_Id_List_Type, Input_Id_Type,
     Output_Id_List_Type, Output_Id_Type, Run_Mode_Type,
     Atom_Step_Type, Boolean
sorts State_Atom
opns
   Make_Atom : Atom_Id, Input_Id_List, Input_Id_List, Input_Id_List,
               Input_Id_List, Output_Id_List, Output_Id_List,
               Output_Id_List, Run_Mode, Atom_Step, Bool
               -> State_Atom
   Get_Id : State_Atom -> Atom_Id
   Get_Sync_Inputs : State_Atom -> Input_Id_List
   Get_Set_Inputs : State_Atom -> Input_Id_List
   Add_Set_Input : State_Atom, Input_Id -> State_Atom
   Remove_Set_Input : State_Atom, Input_Id -> State_Atom
   Get_Required_Inputs : State_Atom -> Input_Id_List
   Set_Required_Inputs : State_Atom, Input_Id_List -> State_Atom
   Get_Async_Inputs : State_Atom -> Input_Id_List
   Get_Outputs : State_Atom -> Output_Id_List
   Add_Done_Output : State_Atom, Output_Id -> State_Atom
   Get_Done_Outputs : State_Atom -> Output_Id_List
   Get_Required_Outputs : State_Atom -> Output_Id_List
   Set_Required_Outputs : State_Atom, Output_Id_List -> State_Atom
   Get_Run_Mode : State_Atom -> Run_Mode
   Set_Run_Mode : State_Atom, Run_Mode -> State_Atom
   Get_Atom_Step : State_Atom -> Atom_Step
   Set_Atom_Step : State_Atom, Atom_Step -> State_Atom
   Needs_Control : State_Atom -> Bool
   Set_Needs_Control : State_Atom, Bool -> State_Atom
eqns
...
endtype
```

Die Zustandsinformation eines Atoms besteht aus den folgenden Bestandteilen, die bei der Erzeugung mittels `Make_Atom` angegeben werden müssen:

- Eine eindeutige Objekt-Id vom Typ `Atom_Id`, die mit `Get_Id` abgefragt werden kann.
- Vier Listen von Eingängen vom Typ `Input_Id_List` mit folgenden Funktionen:
  1. Die Liste aller synchronen Eingänge, die das Atom besitzt. Sie kann mit `Get_Sync_Inputs` abgefragt werden.
  2. Die Liste der gesetzten synchronen Eingänge. Sie wird mit `Get_Set_Inputs` abgefragt; ein Eingang kann mit `Add_Set_Input` als gesetzt eingetragen werden.
  3. Die Liste der synchronen Eingänge, die bei der Durchführung des Rechenschrittes konsumiert wurden. Sie wird mit `Get_Required_Inputs` abgefragt und mit `Set_Required_Inputs` gesetzt.

4. Die Liste aller asynchronen Eingänge, die das Atom besitzt. Sie wird mit `Get_Async_Inputs` gelesen.

- Drei Listen von Ausgängen vom Typ `Output_Id_List`, die folgende Attribute abdecken:
  1. Die Liste aller Ausgänge des Atoms. Sie kann mit `Get_Outputs` abgefragt werden.
  2. Die Liste der freien Ausgänge. Sie kann mit `Get_Done_Outputs` gelesen werden. Ausgänge werden mit `Add_Done_Output` als frei eingetragen.
  3. Die Liste der Ausgänge, die bei der Durchführung des Rechenschrittes geschrieben wurden. Sie wird mit `Get_Required_Outputs` abgefragt und mit `Set_Required_Outputs` gesetzt.
- Den aktuellen Lauf-Zustand des Atoms, festgehalten in einem Attribut vom Typ `Run_Mode`. Im Fall der Atom-Klasse nimmt der Lauf-Zustand die Werte `rm_started`, `rm_stopping` und `rm_stopped` an. Er kann mit `Set_Run_Mode` gesetzt und mit `Get_Run_Mode` gelesen werden.
- Den aktuellen Schritt, in dem sich das Atom befindet, vom Typ `Atom_Step`. Je nachdem, ob das Atom gerade auf die Ein- und Ausgänge wartet oder mit dem Berechnungsschritt fertig ist, nimmt das Attribut den Wert `as_waiting_for_connections` bzw. `as_done_compute` an. Der aktuelle Wert kann mit `Set_Atom_Step` gesetzt und mit `Get_Atom_Step` ausgelesen werden.
- Ein Flag vom Typ `Bool`, das angibt, ob das Atom eine Nachricht vom Kontroll-Objekt erwartet. Der Wert kann mit der Funktion `Needs_Control` gelesen und mit `Set_Needs_Control` gesetzt werden.

Zur Verwaltung der Ein- und Ausgänge wird ein Listen-Typ herangezogen, weil beim Herstellen der Verbindungen von Außen auf bestimmte Ein- und Ausgänge eindeutig zugegriffen werden muß. In der vorliegenden Spezifikation verwenden wir die Position innerhalb der Liste als Schlüssel. Bei einer Implementierung würde man wahrscheinlich die Ein- und Ausgänge mit Namen versehen und so den Zugriff eindeutig regeln.

Der nächste Absatz zeigt die Spezifikation des Verhaltens der Atom-Klasse. Sie ist die Basisklasse für alle Atome und als solche nicht instanzierbar. Die Verfügbarkeit der Services ist wiederum durch Guarded Expressions eingeteilt, um sicherzustellen, daß nur die in dem Zustand möglichen Services aufgerufen werden können. Als „oberstes“ Gruppierungskriterium verwenden wir das `Needs_Control` Flag, das angibt, ob das Atom auf einen Aufruf durch das Kontroll-Objekt wartet.

```
process Atom[i_a, c_a, a_con, a_ac, a_as](state: State_Atom) :
        exit(State_Atom) :=
   [not(Needs_Control(state))] ->
   (
      [Get_Run_Mode(state) eq rm_stopped] ->
      (
         c_a !start !Get_Id(state);
```

```
            (
               Update_Async_Inputs[a_con](Get_Async_Inputs(state)) >>
               (
               let next_state: State_Atom=
                                   Set_Run_Mode(state, rm_started)
                   in
                   a_ac !control !next_state;
                   exit(Set_Needs_Control(next_state, true))
               )
            )
            []
            i_a !get_sync_input !Get_Id(state) ?n: nat;
            i_a !rtn_get_sync_input !Get_Id(state)
                !elem(Get_Sync_Inputs(state), n);
            exit(state)
            []
            i_a !get_async_input !Get_Id(state) ?n: nat;
            i_a !rtn_get_async_input !Get_Id(state)
                !elem(Get_Async_Inputs(state), n);
            exit(state)
            []
            i_a !get_output !Get_Id(state) ?n: nat;
            i_a !rtn_get_output !Get_Id(state)
                !elem(Get_Outputs(state), n);
            exit(state)
         )
         []
         [Get_Run_Mode(state) eq rm_started] ->
         (
            c_a !begin_stopp !Get_Id(state);
            c_a !rtn_begin_stopp !Get_Id(state);
            exit(Set_Run_Mode(state, rm_stopping))
         )
         []
         [(Get_Run_Mode(state) eq rm_started) or
          (Get_Run_Mode(state) eq rm_stopping)] ->
         (
            a_con !input_set !Get_Id(state) ?i_id: Input_Id;
            (
            let next_state: State_Atom=Add_Set_Input(state, i_id) in
               a_ac !control !next_state;
               exit(Set_Needs_Control(next_state, true))
            )
            []
            a_con !output_done !Get_Id(state) ?o_id: Output_id;
            (
```

```
            let next_state: State_Atom=Add_Done_Output(state, o_id) in
               a_ac !control !next_state;
               exit(Set_Needs_Control(next_state, true))
            )
         )
         []
         [Get_Run_Mode(state) eq rm_stopping] ->
         (
            c_a !stopp !Get_Id(state);
            a_as !stopp_atom !state;
            exit(state)
            []
            a_as !stopped !Get_Id(state);
            exit(Set_Run_Mode(state, rm_stopped))
         )
      )
      []
      [Needs_Control(state)] ->
      (
         (
         a_ac !update_connections !Get_Id(state);
         Update_Connections[a_con](Get_Required_Inputs(state),
              Get_Async_Inputs(state), Get_Required_Outputs(state)) >>
            (
            let next_set_inputs: Input_Id_List=
                        RemoveList(Get_Required_Inputs(state),
                                   Get_Set_Inputs(state)),
                next_done_outputs: Output_Id_List=
                        RemoveList(Get_Required_Outputs(state),
                                   Get_Done_Outputs(state))
            in let
                next_state: State_Atom=Make_Atom(Get_Id(state),
                        Get_Sync_Inputs(state), next_set_inputs, <>,
                        Get_Async_Inputs(state), Get_Outputs(state),
                        next_done_outputs, <>, Get_Run_Mode(state),
                        as_waiting_for_connections,
                        Needs_Control(state))
               in
               a_ac !control !next_state;
               exit(next_state)
            )
         )
         []
         a_ac !wait !Get_Id(state);
         exit(Set_Needs_Control(state, false))
      )
```

```
823       where
824       ...
825   endproc
```

Für den Fall, daß das Atom auf keinen Aufruf durch das Kontroll-Objekt wartet (angegeben durch `not(Needs_Control(state))`, Zeile 726) und sich im gestoppten Zustand befindet (`Get_Run_Mode(state) eq rm_stopped`, Zeile 728), können die Services `start`, `get_sync_input`, `get_async_input` und `get_output` aufgerufen werden. Letztere drei retournieren die Objekt-Id des durch `?n` gewünschten Ein- bzw. Ausganges.

Beim Starten des Atoms mit `start` (Zeile 730) werden zunächst die asynchronen Eingänge im Unterprozeß `Update_Async_Inputs` aktualisiert. Das ist deshalb notwendig, weil der Inhalt eines asynchronen Eingangs Auswirkungen auf die Schaltbedingung haben kann (wie z. B. bei den Datenflußatomen *Selector* und *Distributor*). Anschließend wird der Lauf-Zustand des Atoms auf `rm_started` gesetzt und am Gate `a_ac` das Kontroll-Objekt aufgerufen. Da das Atom von nun an auf einen Aufruf durch das Kontroll-Objekt wartet, wird noch mit `Set_Needs_Control(next_state, true)` in Zeile 738 das Zustandsflag für den nächsten Aufruf des Objekt-Prozesses gesetzt.

Im gestarteten Zustand (`Get_Run_Mode(state) eq rm_started`, Zeile 758) kann das Atom mit `begin_stopp` gestoppt werden. `begin_stopp` bringt das Atom nicht vollständig zum Stillstand, sondern leitet den zweistufigen Stop-Vorgang ein (siehe ETD in Abbildung 6.8), indem es den Lauf-Zustand auf `rm_stopping` setzt.

Während das Atom gestartet ist oder sich in der ersten Stufe des Stop-Vorgangs befindet, können synchrone Eingänge mit `input_set` (Zeile 768) melden, daß Daten vorhanden sind. Der betreffende Eingang wird daraufhin mit `Add_Set_Input` als gesetzt eingetragen. Ausgänge rufen das Service `output_done` (Zeile 775) auf, um ihre Bereitschaft zum Empfang neuer Daten zu signalisieren. Mit `Add_Done_Output` wird der Ausgang als frei markiert. In beiden Fällen (Eingang wie Ausgang) wird das Kontroll-Objekt aufgerufen und beim Verlassen des Objekt-Prozesses das `Needs_Control` Flag gesetzt.

Im Zustand `rm_stopping` können neben `input_set` und `output_done` auch die beiden Services `stopp` (Zeile 785) und `stopped` (Zeile 789) aufgerufen werden. `stopp` wird vom Schaltungs-Objekt über das Gate `c_a` aufgerufen, um den zweiten Teil des Stop-Vorganges zu initiieren. Im Zuge dessen ruft das Atom das Stop-Objekt auf (Zeile 786). Hat das Stop-Objekt das Atom erfolgreich gestoppt, ruft es am Gate `a_as` das Service `stopped` auf. Das Atom geht dadurch in den gestoppten Zustand `rm_stopped` über.

In der Basisklasse *Atom* werden zwei der vom Kontroll-Objekt aufgerufenen Services implementiert. Das Service `update_connections` ab Zeile 797 hat die Aufgabe, die von der Schaltfunktion benutzten synchronen Eingänge freizugeben, geschriebene Ausgänge zu setzen und alle asynchronen Eingänge zu aktualisieren. Dieser Vorgang wird im Unterprozeß `Update_Connections` durchgeführt. Anschließend wird der neue Zustand des Atoms in der Variablen `next_state` aufgebaut: die Menge der freigegebenen synchronen Eingänge wird von jener der gesetzten abgezogen und die Menge der gesetzten Ausgänge von jener der freien. Zudem geht das Atom in den Zustand `as_waiting_for_connections` über. Zuletzt wird wiederum das Kontroll-Objekt aufgerufen und der Objekt-Prozeß verlassen (Zeilen 815 und 816).

Das zweite Service, das vom Kontroll-Objekt aufgerufen wird, ist `wait` (Zeile 820). Seine einzige Aufgabe besteht darin, das `Needs_Control` Flag wieder auf den Wert `false` zurückzusetzen.

Spezifikation des Kontroll-Objekts

Der nächste Ausschnitt aus der Spezifikation zeigt das dynamische Verhalten des Kontroll-Objekts *Atom_Controller*. Da von dieser Klasse systemweit nur ein einziges Objekt existiert, entfällt beim Aufruf des Services `control` die Identifizierung durch eine Id. Das Objekt ist durch das Gate `a_ac`, an dem es seine Services anbietet, eindeutig identifiziert. Das Kontroll-Objekt enthält auch sonst keine Attribute, weshalb die Definition eines ADT für den Zustand unterbleiben kann.

```
process Atom_Controller[a_ac, a_con] : noexit :=
(
   a_ac !control ?state: State_Atom;
   (
      [Get_Run_Mode(state) eq rm_started] ->
      (
         [Get_Atom_Step(state) eq as_waiting_for_connections] ->
         (
            a_ac !can_compute !Get_Id(state);
            a_ac !rtn_can_compute !Get_Id(state) ?cc: Bool;
            (
               [cc eq true] ->
                  a_ac !compute !Get_Id(state);
                  exit
               []
               [cc eq false] ->
                  a_ac !wait !Get_Id(state);
                  exit
            )
         )
         []
         [Get_Atom_Step(state) eq as_done_compute] ->
            a_ac !update_connections !Get_Id(state);
            exit
      )
      []
      [Get_Run_Mode(state) ne rm_started] ->
         a_ac !wait !Get_Id(state);
         exit
   )
) >> Atom_Controller[a_ac, a_con]
endproc (* Atom_Controller *)
```

Das Kontroll-Objekt bietet ein einziges Service am Gate `a_ac` an, nämlich `control`. Dieses Service erhält als Parameter in der Variablen `state` den Zustand des zu kontrollierenden Atoms. Je nachdem, in welchem Schritt sich das Atom gerade befindet, ruft es Services des betreffenden Atoms auf.

An dieser Stelle nehmen wir eine leichte Modifikation des durch die ETDs beschriebenen Verhaltens des Kontroll-Objekts und der Atome vor. Bisher wurde davon ausgegangen, daß das Kontroll-Objekt entscheidet, wann das aufrufende Atom schalten darf. Diese Lösung hat allerdings den Nachteil, daß jede Atom-Klasse, die ein eigenes Schaltverhalten definieren möchte, auch ein eigenes Kontroll-Objekt definieren und instanzieren müßte. Das würde jedoch zur Ausbildung einer eigenen Klassenhierarchie von Kontroll-Objekten parallel zur Hierarchie der Atome selbst führen.

Die hier angegebene Lösung geht davon aus, daß Atome selbst entscheiden, wann sie schalten können. Klassen, die von *Atom* abgeleitet werden, müssen deshalb das Service *can_compute* definieren und am Gate `a_ac` zur Verfügung stellen. Mit Hilfe von *can_compute* überprüft das Kontroll-Objekt die Bereitschaft des Atoms, zu schalten.

Befindet sich das Atom im gestarteten Zustand und wartet darauf, schalten zu können (`Get_Atom_Step(state) eq as_waiting_for_connections`, Zeile 859), ruft das Kontroll-Objekt besagtes `can_compute` auf (Zeile 861). Das Atom signalisiert in der Variablen `cc` seine Bereitschaft. Kann es schalten (`cc eq true`, Zeile 864), wird als nächstes `compute` aufgerufen, ansonsten `wait`.

Gestartete Atome, die ihre Schaltfunktion abgeschlossen haben (angezeigt durch den Zustand `as_done_compute`), werden vom Kontroll-Objekt mittels `update_conntections` (Zeile 875) dazu veranlaßt, ihre Eingänge freizuschalten und Ausgänge zu setzen. Nicht gestartete Atome müssen auf jeden Fall warten (Zeilen 879–881).

Spezifikation des Stop-Objekts

Das Stop-Objekt ist dem Kontroll-Objekt in vielerlei Hinsicht ähnlich. Von ihm wird ebenfalls systemweit ein einziges Objekt erzeugt, und es bekommt beim Aufruf seines einzigen Services `stopp_atom` (Zeile 888) den Zustand des aufrufenden Atoms als Parameter übergeben.

```
886 process Atom_Stopper[a_as, a_con] : noexit :=
887 (
888    a_as !stopp_atom ?state: State_Atom;
889    (
890       All_Connections_Ready[a_con](Get_Set_Inputs(state),
891               Get_Done_Outputs(state)) >>
892       a_as !stopped !Get_Id(state);
893       exit
894    )
895 ) >> Atom_Stopper[a_as, a_con]
896
897    where
898    process All_Connections_Ready[c](i_ids: Input_Id_List,
899            o_ids: Output_Id_List) : exit :=
900       [not(isempty(i_ids))] ->
901          c !is_ready ?id: Input_Id [id iselem i_ids];
902          All_Connections_Ready[c](Remove(id, i_ids), o_ids)
903       []
```

```
904          [not(isempty(o_ids))] ->
905             c !is_ready ?id: Output_Id [id iselem o_ids];
906             All_Connections_Ready[c](i_ids, Remove(id, o_ids))
907          []
908          [isempty(i_ids) and isempty(o_ids)] ->
909             exit
910       endproc
911    endproc (* Atom_Stopper *)
```

Um ein Atom stoppen zu können, muß das Stop-Objekt prüfen, daß keine Ein- oder Ausgänge mehr aktiv sind (Unterprozeß `All_Connections_Ready`, Zeile 890). Erst wenn das der Fall ist, ruft das Stop-Objekt die Methode `stopped` (Zeile 892) des betreffenden Atoms auf.

Ab Zeile 898 beginnt die Spezifikation des Unterprozesses `All_Connections_Ready`. Ihm werden die zu überprüfenden Ein- und Ausgänge in den Listen `i_ids` und `o_ids` übergeben. Solange die beiden Listen nicht leer sind (Zeile 908), wird versucht, einen Eingang oder Ausgang mit `is_ready` zu prüfen, ihn aus der Liste zu entfernen und den Prozeß anschließend wieder rekursiv aufzurufen.

Spezifikation abgeleiteter Atom-Klassen

Als Beispiel für abgeleitete Atom-Klassen dienen an dieser Stelle die Klassen *Function_Atom* und *Adder*. Erstere stellt eine Zwischenklasse dar, die ein gemeinsames Verhalten für all jene Atome realisiert, die folgendes vereinfachtes Schaltverhalten besitzen: falls alle Eingänge gesetzt sind und alle Ausgänge frei sind, kann ein Berechnungsschritt durchgeführt werden. Danach werden alle Eingänge freigeschaltet und alle Ausgänge gesetzt.

```
913    process Function_Atom[i_a, c_a, a_con, a_ac, a_as]
914                          (state: State_Atom) : exit(State_Atom) :=
915    (
916       (Atom[i_a, c_a, a_con, a_ac, a_as](state) >>
917        accept update_state: State_Atom in exit(update_state)
918       )
919       []
920       [Needs_Control(state)] ->
921       (
922          a_ac !can_compute !Get_Id(state);
923          a_ac !rtn_can_compute !Get_Id(state)
924               !((Get_Set_Inputs(state) eq Get_Sync_Inputs(state)) and
925                 (Get_Done_Outputs(state) eq Get_Outputs(state)));
926          exit(state)
927       )
928    )
929    endproc
```

Um das beschriebene Schaltverhalten zu realisieren, wird in der Klasse *Function_Atom* das Service `can_compute` (Zeile 922) implementiert. Für die Berechnung des Rückgabewertes werden die Listen der Eingänge und der gesetzten Eingänge sowie die der Ausgänge und der freien Ausgänge auf ihre Gleichheit überprüft (Zeilen 923–925).

Alle von *Function_Atom* abgeleiteten Atom-Klassen müssen nur noch die Services `compute` und `reset` definieren, wie anhand der Addierer-Klasse *Adder* demonstriert wird:

```
989  process Adder[i_a, c_a, a_con, a_ac, a_as](state: State_Atom) :
990          noexit :=
991  (
992     (Function_Atom[i_a, c_a, a_con, a_ac, a_as](state) >>
993      accept update_state: State_Atom in exit(update_state)
994     )
995     []
996     [Needs_Control(state)] ->
997     (
998        a_ac !compute !Get_Id(state); (* add inputs to output *)
999        a_con !read !elem(Get_Sync_Inputs(state), 0) ?d1: Data;
1000       a_con !read !elem(Get_Sync_Inputs(state), succ(0)) ?d2: Data;
1001       a_con !write !elem(Get_Outputs(state), 0)
1002             !Make_Data(Get(d1) + Get(d2));
1003       (
1004          let next_state: State_Atom=
1005              Set_Atom_Step(
1006                 Set_Required_Inputs(
1007                    Set_Required_Outputs(state, Get_Outputs(state)),
1008                       Get_Sync_Inputs(state)), as_done_compute)
1009             in
1010             a_ac !control !next_state;
1011             exit(next_state)
1012       )
1013    )
1014    []
1015    [(not(Needs_Control(state))) and
1016     (Get_Run_Mode(state) eq rm_stopped)] ->
1017    (
1018       c_a !reset !Get_Id(state);
1019       Reset_Connections[a_con](Get_Sync_Inputs(state),
1020                                Get_Outputs(state))
1021       >>
1022       (
1023          let next_state: State_Atom=Make_Atom(Get_Id(state),
1024              Get_Sync_Inputs(state), <>, <>, Get_Async_Inputs(state),
1025              Get_Outputs(state), <>, <>, rm_stopped,
1026              as_waiting_for_connections, false)
1027          in
1028          exit(next_state)
1029       )
```

```
1030    )
1031 ) >> accept update_state: State_Atom in
1032                Adder[i_a, c_a, a_con, a_ac, a_as](update_state)
1033 endproc
```

Im Service `compute` werden zuerst die Eingänge gelesen (Zeilen 999–1000) und anschließend die Summe der Eingänge dem Ausgang zugewiesen (Zeilen 1001–1002). In den Zeilen 1004–1008 wird der Zustand `next_state` des Atoms für das Kontroll-Objekt einerseits und für den nächsten Prozeßaufruf andererseits erstellt. Wichtig ist dabei, daß die für die Berechnung verwendeten Ein- und Ausgänge eingetragen werden (in diesem Fall alle). Dies geschieht mit den Funktionen `Set_Required_Inputs` bzw. `Set_Required_Outputs`. Der Atom-Schritt muß außerdem auf `as_done_compute` gesetzt werden. Zum Abschluß des Berechnungsvorganges wird wieder das Kontroll-Objekt aufgerufen (Zeile 1010) und der Objekt-Prozeß verlassen.

Atome können nur im gestoppten Zustand zurückgesetzt werden. Nach dem Aufruf von `reset` (Zeile 1018) setzt der Addierer zunächst mit dem Unterprozeß `Reset_Connections` alle seine synchronen Ein- und Ausgänge zurück. Danach erstellt er in der Variablen `next_state` einen Objekt-Zustand, der dem eines neu erzeugten Addierers entspricht, und gibt ihn an den nächsten Aufruf des Prozesses weiter.

Spezifikation der Ein- und Ausgänge

Vor der eigentlichen Spezifikation der Ein- und Ausgänge wird die Klasse *Data* spezifiziert. Sie hat eine passive Rolle, deshalb genügt ein ADT für ihre Spezifikation:

```
type Data_Type is NaturalNumber
sorts Data
opns
   Make_Data: Nat -> Data
   Set: Data, Nat -> Data
   Get: Data -> Nat
eqns
...
endtype
```

Die Data-Klasse verwaltet vorerst ausschließlich natürliche Zahlen, die bei der Erzeugung mit `Make_Data` übergeben werden. Mit der Funktionen `Set` kann der Wert des Datenelements gesetzt werden, mit `Get` gelesen.

Die Spezifikation der Klasse für synchrone Eingänge *Sync_Input* besteht wiederum aus einem ADT und einem LOTOS-Prozeß:

```
type Sync_Input_Type is Input_Id_Type, Atom_Id_Type,
     Atom_Id_Set_Type, Output_Id_Type, Output_Id_Set_Type,
     Con_Status_Type, Data_Type, Boolean
sorts State_Sync_Input
opns
   Make_Sync_Input : Input_Id, Atom_Id_Set, Output_Id_Set,
```

```
                     Con_Status, Data -> State_Sync_Input
   Get_Id : State_Sync_Input -> Input_Id
   Get_Atoms : State_Sync_Input -> Atom_Id_Set
   Is_Part_Of_Atom : State_Sync_Input -> Bool
   Get_Outputs : State_Sync_Input -> Output_Id_Set
   Connect : State_Sync_Input, Output_Id -> State_Sync_Input
   Disconnect : State_Sync_Input -> State_Sync_Input
   Get_Con_Status : State_Sync_Input -> Con_Status
   Set_Con_Status : State_Sync_Input, Con_Status -> State_Sync_Input
   Get_Data : State_Sync_Input -> Data
   Set_Data : State_Sync_Input, Data -> State_Sync_Input
eqns
...
endtype
```

Folgende Attribute werden dem Konstruktor `Make_Sync_Input` angegeben:

- Eine eindeutige Objekt-Id vom Typ `Input_Id`, die mit `Get_Id` gelesen werden kann. Die Klassen *Sync_Input* und *Async_Input* teilen sich denselben Satz Bezeichner, damit Ausgänge synchrone und asynchrone Eingänge nicht getrennt verwalten müssen.
- Eine Menge von Atomen vom Typ `Atom_Id_Set`. Für den Fall, daß der Eingang zu einem Atom gehört, enthält die Menge den Bezeichner des Atoms als einziges Element. Handelt es sich um einen freien Eingang, bleibt die Menge leer. Mit der Funktion `Get_Atoms` kann das Attribut abgefragt werden und die Funktion `Is_Part_Of_Atom` stellt fest, ob der Eingang zu einem Atom gehört oder nicht.
- Eine Menge von Ausgängen vom Typ `Output_Id_Set`. Da ein Eingang nur mit einem Ausgang verbunden sein kann oder (temporär) unverbunden bleibt, enthält diese Menge entweder ein oder kein Element. Mit der Funktion `Get_Outputs` kann der Inhalt der Menge abgefragt werden. Die Funktionen `Connect` und `Disconnect` verändern die Menge insofern, als sie den in der Menge enthaltenen Ausgang setzen respektive löschen.
- Der aktuelle Setz-Zustand `Con_Status` des Einganges. Er kann die Werte `cs_free` und `cs_set` annehmen, je nachdem ob der Eingang setzbar ist oder aber Daten enthält, die erst abgeholt werden müssen. Der Zustand kann mit `Get_Con_Status` gelesen und `Set_Con_Status` gesetzt werden.
- Einen Datenwert vom Typ `Data`. Das Attribut kann mit `Get_Data` gelesen und mit `Set_Data` gesetzt werden.

Das dynamische Verhalten von Eingängen wird durch folgenden Prozeß beschrieben:

```
process Sync_Input[i_con, a_con, con](state: State_Sync_Input) :
        noexit :=
(
   i_con !connect !Get_Id(state) ?o: Output_Id;
```

```
    exit(Connect(state, o))
    []
    i_con !disconnect !Get_Id(state);
    exit(Disconnect(state))
    []
    [Is_Part_Of_Atom(state)] ->
    (
       a_con !is_ready !Get_Id(State);
       exit(state)
       []
       a_con !reset !Get_Id(state);
       exit(Set_Con_Status(state, cs_free))
       []
       [Get_Con_Status(state) eq cs_free] ->
          con !set !Get_Id(state) ?d: Data;
          a_con !input_set ?a_id: Atom_Id !Get_Id(state)
                [a_id IsIn Get_Atoms(state)];
          exit(Set_Con_Status(Set_Data(state, d), cs_set))
       []
       [Get_Con_Status(state) eq cs_set] ->
       (
          a_con !free !Get_Id(state);
          con !free ?id: Output_Id !Get_Id(state)
              [id IsIn Get_Outputs(state)];
          exit(Set_Con_Status(state, cs_free))
          []
          a_con !read !Get_Id(state) !Get_Data(state);
          exit(state)
       )
    )
    []
    [not(Is_Part_Of_Atom(state))] ->
    (
       i_con !is_ready !Get_Id(State);
       exit(state)
       []
       i_con !reset !Get_Id(state);
       exit(Set_Con_Status(state, cs_free))
       []
       [Get_Con_Status(state) eq cs_free] ->
          con !set !Get_Id(state) ?d: Data;
          i_con !input_set !Get_Id(state);
          exit(Set_Con_Status(Set_Data(state, d), cs_set))
       []
       [Get_Con_Status(state) eq cs_set] ->
       (
```

```
1170           i_con !free !Get_Id(state);
1171           con !free ?id: Output_Id !Get_Id(state)
1172               [id IsIn Get_Outputs(state)];
1173           exit(Set_Con_Status(state, cs_free))
1174           []
1175           i_con !read !Get_Id(state) !Get_Data(state);
1176           exit(state)
1177        )
1178     )
1179  ) >> accept update_state: State_Sync_Input in
1180             Sync_Input[i_con, a_con, con](update_state)
1181
1182  endproc (* Sync_Input *)
```

Die Services `connect` und `disconnect` verbinden bzw. trennen einen synchronen Eingang und einen Ausgang. Bei den anderen Services wird zunächst unterschieden, ob der Eingang Teil eines Atoms ist (`Is_Part_Of_Atom(state)`) oder nicht. Im ersten Fall werden die Services `is_ready`, `reset`, `free` und `read` am Gate `a_ac` vom besitzenden Atom aufgerufen (Zeilen 1131–1152), sonst direkt von der Umgebung am Gate `i_con` (Zeilen 1157–1177). Das Service `set` wird immer am Gate `con` vom vorgeschalteten Ausgang aufgerufen. Je nachdem, ob der Eingang Teil eines Atoms ist oder nicht, benachrichtigt er mit `input_set` entweder das Atom oder die Umgebung. Mittels zusätzlicher Guards wird sichergestellt, daß das Service `set` nur aufgerufen wird, wenn der Eingang leer ist (Zustand `cs_free`). Die Services `free` und `read` stehen ausschließlich im Zustand `cs_set` zur Verfügung.

Die im folgenden beschriebenen asynchronen Eingänge sind den synchronen Eingängen sehr ähnlich. Im Gegensatz zu diesen verfügen sie jedoch über kein Setz-Status-Attribut, da sie jederzeit gesetzt und gelesen werden können. Um die Werte der asynchronen Eingänge trotzdem zu definierten Zeitpunkten übernehmen zu können, beinhalten sie zwei Daten-Attribute (ein sog. „externes“ und ein „internes“):

```
type Async_Input_Type is Input_Id_Type, Atom_Id_Type,
     Atom_Id_Set_Type, Output_Id_Type, Output_Id_Set_Type,
     Con_Status_Type, Data_Type, Boolean
sorts State_Async_Input
opns
   Make_Async_Input : Input_Id, Atom_Id_Set, Output_Id_Set, Data,
                      Data -> State_Async_Input
   Get_Id : State_Async_Input -> Input_Id
   Get_Atoms : State_Async_Input -> Atom_Id_Set
   Is_Part_Of_Atom : State_Async_Input -> Bool
   Get_Outputs : State_Async_Input -> Output_Id_Set
   Connect : State_Async_Input, Output_Id -> State_Async_Input
   Disconnect : State_Async_Input -> State_ASync_Input
   Get_Internal_Data : State_Async_Input -> Data
   Set_Internal_Data : State_Async_Input, Data -> State_Async_Input
```

```
   Get_External_Data : State_Async_Input -> Data
   Set_External_Data : State_Async_Input, Data -> State_Async_Input
eqns
...
endtype
```

Die beiden Daten-Attribute können mit `Set_External_Data` und `Set_Internal_Data` gesetzt werden. Die entsprechenden Zugriffsfunktionen lauten `Get_External_Data` und `Get_Internal_Data`.

Die nachfolgende Spezifikation des Verhaltens ist etwas einfacher als die der synchronen Eingänge, weil keine Unterscheidung nach dem Setz-Zustand gemacht werden muß:

```
process Async_Input[i_con, a_con, con](state: State_Async_Input) :
        noexit :=
(
   i_con !connect !Get_Id(state) ?o: Output_Id;
   exit(Connect(state, o))
   []
   i_con !disconnect !Get_Id(state);
   exit(Disconnect(state))
   []
   con !set !Get_Id(state) ?d: Data;
   con !free ?id: Output_Id !Get_Id(state)
       [id IsIn Get_Outputs(state)];
   exit(Set_External_Data(state, d))
   []
   [Is_Part_Of_Atom(state)] ->
   (
      a_con !update !Get_Id(state);
      exit(Set_Internal_Data(state, Get_External_Data(state)))
      []
      a_con !read !Get_Id(state) !Get_Internal_Data(state);
      exit(state)
   )
   []
   [not(Is_Part_Of_Atom(state))] ->
   (
      i_con !update !Get_Id(state);
      exit(Set_Internal_Data(state, Get_External_Data(state)))
      []
      i_con !read !Get_Id(state) !Get_Internal_Data(state);
      exit(state)
   )
) >> accept update_state: State_Async_Input in
            Async_Input[i_con, a_con, con](update_state)
endproc
```

Die Services `connect` und `disconnect` haben die selben Funktionen wie bei synchronen Eingängen. Mit `set` kann jederzeit das externe Datenelement des asynchronen Eingangs gesetzt werden. Anders als bei den synchronen Eingängen wird bei `set` keine Nachricht an das Atom oder an die Umgebung geschickt. Bei den Services `update` und `read` muß wieder unterschieden werden, ob der Eingang zu einem Atom gehört oder frei ist. Mit `update` wird das externe Datenelement auf das interne kopiert, das mit `read` jederzeit gelesen werden kann.

Der nachfolgende ADT kapselt die Zustandsinformation der Klasse `Output`:

```
type Output_Type is Output_Id_Type, Input_Id_Type, Input_Id_Set_Type,
     Atom_Id_Type, Atom_Id_Set_Type, Con_Status_Type, Data_Type
sorts State_Output
opns
   Make_Output : Output_Id, Atom_Id_Set, Input_Id_Set, Input_Id_Set,
                 Con_Status, Data -> State_Output
   Get_Id : State_Output -> Output_Id
   Get_Atoms : State_Output -> Atom_Id_Set
   Is_Part_Of_Atom : State_Output -> Bool
   Get_Inputs : State_Output -> Input_Id_Set
   Connect : State_Output, Input_Id -> State_Output
   Disconnect : State_Output, Input_Id -> State_Output
   Get_Done_Inputs : State_Output -> Input_Id_Set
   Add_Done_Input : State_Output, Input_Id -> State_Output
   Clear_Done_Inputs : State_Output -> State_Output
   Get_Con_Status : State_Output -> Con_Status
   Set_Con_Status : State_Output, Con_Status -> State_Output
   Get_Data : State_Output -> Data
   Set_Data : State_Output, Data -> State_Output
eqns
...
endtype
```

Folgende Attribute werden dem Konstruktor `Make_Output` angegeben:

- Ein Bezeichner vom Typ `Output_Id`, der den Ausgang eindeutig identifiziert und mit `Get_Id` gelesen werden kann.
- Eine Menge von Atomen vom Typ `Atom_Id_Set`. Sie erfüllt die selbe Aufgabe wie bei den Zustandsinformationen für die beiden Eingangs-Klassen. Die Funktionen `Get_Atoms` und `Is_Part_Of_Atom` verhalten sich ebenfalls gleich.
- Eine Menge von Eingängen, mit denen der Ausgang verbunden ist (die Menge kann auch leer sein). Mit `Get_Inputs` kann die Menge abgefragt werden. Verbindungen werden mit `Connect` hergestellt und mit `Disconnect` getrennt.
- Die Menge der nachgeschalteten Eingänge, die wieder frei geworden sind. Erst, wenn alle nachgeschalteten Eingänge frei sind, kann auch der Ausgang wieder freigegeben werden. Mittels `Get_Done_Inputs` kann die Menge der frei gewordenen

Eingänge abgefragt werden. `Add_Done_Input` fügt der Menge einen Eingang hinzu und `Clear_Done_Inputs` entfernt alle Eingänge aus ihr.

- Der Setz-Status des Ausgangs, vom Typ `Con_Status`. Er kann ebenso wie beim synchronen Eingang die Werte `cs_set` und `cs_free` annehmen und wird mit der Funktion `Set_Con_Status` gesetzt und mit `Get_Con_Status` gelesen.
- Ein Datenelement vom Typ `Data`, das mit `Set_Data` gesetzt und mit `Get_Data` gelesen werden kann.

Der folgende Absatz zeigt die Spezifikation des Objekt-Prozesses für die Ausgangs-Klasse. Wie bei den beiden Eingangs-Klassen wird auch hier unterschieden, ob ein Ausgang Teil eines Atoms ist oder nicht. Im ersten Fall werden die Services `is_ready`, `reset`, `write`, `set` und `free` am Gate `a_con` angeboten, sonst am Gate `i_con`.

```
process Output[i_con, a_con, con](state: State_Output) : noexit :=
(
   i_con !connect !Get_Id(state) ?i_id: Input_Id;
   exit(Connect(state, i_id))
   []
   i_con !disconnect !Get_Id(state) ?i_id: Input_Id;
   exit(Disconnect(state, i_id))
   []
   [Is_Part_Of_Atom(state)] ->
   (
      a_con !is_ready !Get_Id(state);
      exit(state)
      []
      a_con !reset !Get_Id(state);
      exit(Clear_Done_Inputs(Set_Con_Status(state, cs_free)))
      []
      [Get_Con_Status(state) eq cs_free] ->
      (
         a_con !write !Get_Id(state) ?d: Data;
         exit(Set_Data(state, d))
         []
         (
            a_con !set !Get_Id(state);
            Set_All_Inputs[con](Get_Inputs(state), Get_Data(state))
            >>
            a_con !output_done ?a_id: Atom_Id !Get_Id(state)
                  [a_id IsIn Get_Atoms(state)];
            exit(Clear_Done_Inputs(state))
         )
      )
      []
      [Get_Con_Status(state) eq cs_set] ->
```

```
      (
         con !free !Get_Id(state) ?id: Input_Id;
         (
         let next_state: State_Output=Add_Done_Input(state, id)
             in
             [Get_Done_Inputs(state) eq Get_Inputs(state)] ->
                 a_con !output_done ?a_id: Atom_Id !Get_Id(state)
                        [a_id IsIn Get_Atoms(state)];
                 exit(Clear_Done_Inputs(
                          Set_Con_Status(next_state, cs_free)))
             []
             [Get_Done_Inputs(state) ne Get_Inputs(state)] ->
                 exit(next_state)
         )
      )
   )
   []
   [not(Is_Part_Of_Atom(state))] ->
   (
      i_con !is_ready !Get_Id(state);
      exit(state)
      []
      i_con !reset !Get_Id(state);
      exit(Clear_Done_Inputs(Set_Con_Status(state, cs_free)))
      []
      [Get_Con_Status(state) eq cs_free] ->
      (
         i_con !write !Get_Id(state) ?d: Data;
         exit(Set_Data(state, d))
         []
         (
            i_con !set !Get_Id(state);
            Set_All_Inputs[con](Get_Inputs(state), Get_Data(state))
            >>
            i_con !output_done ?a_id: Atom_Id !Get_Id(state)
                  [a_id IsIn Get_Atoms(state)];
            exit(Clear_Done_Inputs(state))
         )
      )
      []
      [Get_Con_Status(state) eq cs_set] ->
      (
         con !free !Get_Id(state) ?id: Input_Id;
         (
         let next_state: State_Output=Add_Done_Input(state, id)
             in
```

```
                [Get_Done_Inputs(state) eq Get_Inputs(state)] ->
                   i_con !output_done ?a_id: Atom_Id !Get_Id(state)
                       [a_id IsIn Get_Atoms(state)];
                   exit(Clear_Done_Inputs(
                            Set_Con_Status(next_state, cs_free)))
                []
                [Get_Done_Inputs(state) ne Get_Inputs(state)] ->
                   exit(next_state)
            )
        )
    )
) >> accept update_state: State_Output in
               Output[i_con, a_con, con](update_state)

    where
    process Set_All_Inputs[x](i_ids: Input_Id_Set, d: Data) : exit :=
        [i_ids ne {}] ->
           x !set ?id: Input_Id !d [id IsIn i_ids];
           Set_All_Inputs[x](Remove(id, i_ids), d)
        []
        [i_ids eq {}] ->
           exit
    endproc

endproc (* Output *)
```

Mit den Services `connect` und `disconnect` wird der Ausgang mit einem Eingang verbunden bzw. von diesem getrennt. `write` übernimmt den übergebenen Datenwert und trägt ihn mit `Set_Data` im Daten-Attribut ein. Das Service kann nur aufgerufen werden, wenn der Ausgang im Zustand `cs_free` ist. In diesem Zustand kann der Ausgang auch mit `set` aufgefordert werden, sein Daten-Attribut an die nachgeschalteten Eingänge weiterzuleiten. Er benutzt dazu den Unterprozeß `Set_All_Inputs`. Das Service `free` wird von den nachgeschalteten Eingängen am Gate `con` aufgerufen, um anzuzeigen, daß sie wieder Daten empfangen können. Sind alle Eingänge freigeworden, meldet das der Ausgang seinerseits am Gate `a_con` dem Atom oder am Gate `i_con` der Umgebung (je nachdem, ob der Ausgang zu einem Atom gehört oder nicht).

7.3 Schritt 3.3: Objekte zusammenfügen

Um die Spezifikation ablauffähig zu machen, müssen die in Schritt 3.2 beschriebenen Objekt-Prozesse noch an den entsprechenden Gates zusammengefügt werden. Zu diesem Zweck wird das OCD (Abbildung 7.1) herangezogen, in dem die Beziehungen zwischen den Prozessen grafisch festgehalten sind. Die Gates `i_c`, `i_a`, `i_con` und `i_t` dienen der Kommunikation mit der Umgebung und werden dementsprechend in der ersten Zeile der Spezifikation exportiert:

```
specification CircuitSystem[i_c, i_a, i_con, i_t] : noexit
```

Das Gate `i_t` dient dazu, die Aktionen des Test-Prozesses (siehe auch nächster Schritt) nach außen zugänglich zu machen.

Die Gates, die nicht direkt mit der Umgebung in Verbindung stehen sondern zur Kommunikation innerhalb des Systems dienen, werden durch den `hide`-Operator nach außen hin unsichtbar gemacht. Es sind dies die Gates `c_a` und `a_con`. Das Gesamtverhalten der Spezifikation geht aus folgender Behaviour Expression hervor:

```
591 behaviour
592    (
593       hide c_a in
594          Circuit[i_c, c_a](Make_Circuit(id1 of Circuit_Id,
595                                   {} of Atom_Id_Set, rm_stopped))
596          |[c_a]|
597          (
598             hide a_con in
599                Atoms[i_a, c_a, a_con]
600                |[a_con]|
601                Connections[i_con, a_con]
602          )
603    )
604    |[i_c, i_a, i_con]|
605    Test[i_c, i_a, i_con, i_t]
606 where
607 ...
```

Vom Prozeß `Circuit` wird eine einzige Instanz erzeugt (Zeilen 594–595), die den Bezeichner `id1` hat. Die Menge der Atome der Schaltung ist anfangs leer und die Schaltung befindet sich im gestoppten Zustand `rm_stopped`. In Zeile 599 wird der Prozeß `Atoms` instanziert, der das Atom-Subsystem beinhaltet. `Atoms` ist am Gate `a_con` mit dem Subsystem für Verbindungen (`Connections`), das in Zeile 601 erzeugt wird, verbunden. Der Prozeß `Test` (Zeile 605) beinhaltet Interface-Szenarien, mit denen das korrekte Verhalten der Spezifikation getestet werden kann.

Im folgenden Listing wird die Spezifikation des `Atoms` Subsystem gezeigt. Es erzeugt Objekt-Generatoren für die einzelnen Atom-Klassen, nämlich `Constants`, `Adders`, `Selectors` und `Distributors`. Diese werden am Gate `a_ac` mit dem Kontroll-Objekt `Atom_Controller` und am Gate `a_as` mit dem Stop-Objekt `Atom_Stopper` verbunden:

```
process Atoms[i_a, c_a, a_con] : noexit :=
   hide a_ac, a_as in
      (
         Constants[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Adders[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Selectors[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Distributors[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
```

```
      )
      |[a_ac]|
      Atom_Controller[a_ac, a_con]
      |[a_as]|
      Atom_Stopper[a_as, a_con]
   where
...
```

Der nächste Auszug zeigt die Spezifikation des Subsystems `Connections`. Es erzeugt die Objekt-Generatoren `Sync_Inputs`, `Async_Inputs` und `Outputs`:

```
process Connections[i_con, a_con] : noexit :=
   hide con in
      (
         Sync_Inputs[i_con, a_con, con]({} of Input_Id_Set)
         |||
         Async_Inputs[i_con, a_con, con]({} of Input_Id_Set)
      )
      |[con]|
      Outputs[i_con, a_con, con]({} of Output_Id_Set)
   where
...
```

Synchrone und asynchrone Eingänge laufen völlig parallel zueinander und sind beide am Gate `con` mit den Ausgängen verbunden.

Zuletzt müssen noch Interface-Szenarien spezifiziert werden, um die Spezifikation zu testen. Der nachfolgende Prozeß `Test` erzeugt die in Abbildung 6.2 gezeigte Schaltung und exekutiert sie:

```
process Test[i_c, i_a, i_con, i_t] : noexit :=
   (* construct atoms and add them to the circuit *)
   i_a !create !constant ?con: Atom_Id;
   i_c !add_atom !id1 of Circuit_Id !con;
   i_a !create !adder ?a: Atom_Id;
   i_c !add_atom !id1 of Circuit_Id !a;

   (* construct free input *)
   i_con !create !output ?fo: Output_Id;
   i_con !create !sync_input ?fi: Input_Id;

   (* connect *)
   i_a !get_async_input !con !0;
   i_a !rtn_get_async_input ?ic: Input_Id;
   i_a !get_output !con !0;
   i_a !rtn_get_output !con ?oc: Output_Id;
   i_a !get_sync_input !a !0;
   i_a !rtn_get_sync_input !a ?ia1: Input_Id;
```

```
1403      i_a !get_sync_input !a !succ(0);
1404      i_a !rtn_get_sync_input !a ?ia2: Input_Id;
1405      i_a !get_output !a !0;
1406      i_a !rtn_get_output !a ?oa: Output_Id;
1407      i_con !connect !ic !fo;
1408      i_con !connect !fo !ic;
1409      i_con !connect !oc !ia1;
1410      i_con !connect !ia1 !oc;
1411      i_con !connect !oc !ia2;
1412      i_con !connect !ia2 !oc;
1413      i_con !connect !oa !fi;
1414      i_con !connect !fi !oa;
1415
1416      (* start circuit and receive input *)
1417      i_c !start !id1 of Circuit_Id;
1418      i_con !input_set !fi;
1419      i_con !read !fi ?d1: Data;
1420      i_t !Get(d1);
1421      i_con !free !fi;
1422
1423      (* stop and reset circuit *)
1424      i_c !stopp !id1 of Circuit_Id;
1425      i_c !rtn_stopp !id1 of Circuit_Id;
1426      i_c !reset !id1 of Circuit_Id;
1427      i_c !rtn_reset !id1 of Circuit_Id;
1428      stop
1429   endproc
```

Zunächst werden das Konstanten-Atom `con` und der Addierer `a` erzeugt und der Schaltung hinzugefügt (Zeilen 1387–1390). Danach werden der für die externe Kommunikation mit der Schaltung benötigte freie Ausgang `fo` und freie Eingang `fi` erzeugt (Zeilen 1393–1394). Die Verbindungen zwischen Ein- und Ausgängen werden in den Zeilen 1397–1414 hergestellt. Die Schaltung ist damit fertig aufgebaut und kann gestartet werden (Zeile 1417). Anschließend wird darauf gewartet, daß der freie Eingang `fi` das Resultat der Berechnung erhält (Zeile 1418). Der Eingang kann dann gelesen (Zeilen 1419–1420) und wieder freigegeben werden (Zeile 1421). Zuletzt wird die Schaltung gestoppt und rückgesetzt (Zeilen 1424–1427).

7.4 Schritt 3.4: Prototyp exekutieren

In diesem Schritt wird die Spezifikation mit einem LOTOS-Simulator ausgeführt und damit getestet, ob sie den Anforderungen entspricht. Folgende Werkzeuge kamen im Rahmen dieser Arbeit zum Einsatz:

- LOLA (LOtos LAboratory)[1]:
  LOLA ist ein LOTOS-Simulator, der als Teil des TOPO[2] Environments zur Verfügung gestellt wird. Für die Simulation der vorliegenden Spezifikation ist LOLA nicht geeignet, da es keine Wertgenerierung unterstützt. Wertgenerierung ist notwendig für die dynamische Generierung von Objekt-Ids sowie die indeterministische Auswahl von Elementen aus einer Menge (siehe z. B. im Unterprozeß `Start_Atoms` der Klasse `Circuit`).

- SMILE (SyMbolic Interactive Lotos Execution)[3]:
  SMILE ist ein X-Window basierter LOTOS-Simulator, der von anderen Programmen vorübersetzte Quellen simulieren kann. Vorzugsweise ist er in das Toolkit LITE (LOTOS Integrated Tool Environment)[4], das ebenfalls an der selben Stelle distribuiert wird, integriert. SMILE ist prinzipiell für die Simulation der vorliegenden Spezifikation geeignet, weil es alle notwendigen Sprachstandards unterstützt. Mangels Stabilität des Programms konnte die Spezifikation mit diesem Simulator nur ansatzweise mit einem stark vereinfachten Interface-Szenario getestet werden.

- CADP (Caesar/Aldebaran Development Package)[5]:
  Das CADP-Toolkit ist das am weitesten entwickelte aller drei Systeme, außerdem wird das System im Gegensatz zu den beiden anderen laufend weiterentwickelt. Für objektorientierte Spezifikationen, wie sie in dieser Arbeit beschrieben werden, ist es trotzdem nur beschränkt zu empfehlen, weil ebenfalls nicht der gesamte Sprachstandard unterstützt wird. CADP kann keine generischen Datentypen, SpezifiziererInnen müßten in diesem Fall selbst alle Instanzierungen des generischen Datentyps „per Hand“ vornehmen. Da aber für die Spezifikation von Assoziationen und Aggregationen von Objekten immer wieder Mengen oder Listen von bestimmten Objekt-Ids hergestellt werden müssen, kann dies leicht in hunderten Zeilen mehr oder weniger redundanter Typdefinitionen münden.

Insgesamt stellen wir fest, daß die Unterstützung dieses Schrittes durch entsprechende Werkzeuge im Moment nicht ausreichend gewährleistet ist. Der Hauptnutzen der angeführten Werkzeuge beschränkte sich deshalb auf die Überprüfung der korrekten LOTOS-Syntax. Mit SMILE konnte die Spezifikation zumindest ansatzweise, wenn auch mit stark vereinfachten Interface-Szenarien, getestet werden.

7.5 Schritt 3.5: Spezifikation verfeinern

Wie am Anfang von Kapitel 5 angedeutet, ist die bis jetzt gezeigte Spezifikation bereits das Resultat mehrerer Iterationen. Der hier vorhandene Platz reicht nicht aus, um die Spezifikation im Detail fortzuführen. Folgende Erweiterungen können allerdings skizziert und anhand der vorliegenden Spezifikation auch relativ leicht realisiert werden:

[1] siehe ftp://ftp.dit.upm.es/pub/lotos/tools/TOPO/contrib/LOLA/
[2] siehe ftp://ftp.dit.upm.es/pub/lotos/tools/TOPO/
[3] siehe ftp://ftp.cs.utwente.nl/pub/src/lotos-tools/
[4] siehe http://www-sop.inria.fr/meije/lotostools.html
[5] siehe http://www.inrialpes.fr/vasy/pub/cadp.html

- Die Klasse *Data* kann erweitert werden, sodaß mehrere Datentypen zum Transfer zur Verfügung stehen.
- Der Satz an Atomen kann erweitert werden.
- Für die Schaltungs-Klasse kann ebenfalls ein Objekt-Generator eingeführt werden, um mehrere Schaltungen parallel ablaufen zu lassen.

Etwas mehr Aufwand würden die folgenden Änderungen bedeuten:

- Überprüfung von Schaltungen auf Konsistenz, bevor sie gestartet werden (unverbundene Eingänge, fehlerhafte Rückführungen)
- Überlegungen, wie Rückführungen in Schaltungen realisiert werden können

Kapitel 8

Erfahrungen

Die Anwendung der gewählten Methode (ROOA mit LOTOS) erfolgte nicht geradlinig. Von den ersten Vorstellungen über das Schaltungssystem bis zur fertigen Spezifikation waren mehrere Iterationen notwendig, in denen das Verständnis für die Aufgabenstellung immer besser wurde. Wir möchten in diesem Kapitel deshalb noch einmal die Erfahrungen, die wir bei der Anwendung der Methode gemacht haben, zusammenfassen, um die Methode an sich zu beurteilen. Im Anschluß daran werden die Besonderheiten bei der Entwicklung des Schaltungssystems beschrieben.

8.1 Bewertung der ROOA

Grundsätzlich halten wir ROOA für eine sinnvolle und brauchbare Methode für die Entwicklung objektorientierter Systeme. Die von den AutorInnen genannten Vorzüge, wie z. B. Exaktheit und Implementierungsunabhängigkeit haben sich bestätigt. Der Grundgedanke, eine formale, exekutierbare Spezifikationssprache in den Prozeß der objektorientierten Analyse zu integrieren, ist vielversprechend.

Es stellte sich heraus, daß die Haupthürden bei der Anwendung der Methode in der verwendeten Spezifikationssprache, LOTOS, lagen. Wir halten LOTOS für nicht geeignet, um größere objektorientierte Systeme formal zu spezifizieren. Zum einen macht sich die fehlende Unterstützung objektorientierter Mechanismen auf syntaktischer Ebene stärker bemerkbar als anfangs angenommen. Zum anderen existieren im Moment keine geeigneten Entwicklungswerkzeuge, die ein effizientes Rapid Prototyping von objektorientierten LOTOS-Spezifikationen zulassen würden.

In der folgenden Aufzählung sollen unsere Beobachtungen in loser Reihenfolge detailliert dargestellt werden:

- Objektorientierte Eigenschaften gehen aus der LOTOS-Spezifikation selbst nicht mehr deutlich hervor. Es ist ein permanentes Übersetzen und Rückübersetzen zwischen dem intendierten objektorientierten Mechanismus und der eigentlichen Spezifikation in LOTOS notwendig. Als besonders umständlich empfanden wir die Abbildung der Vererbungsrelation mit ihren verschiedenen Ausprägungen auf LOTOS. Dies stellt eine echte Einschränkung gegenüber objektorientierten Programmiersprachen, wie z. B. C++ oder Java, dar. Objektorientiertes LOTOS ist deshalb nur für relativ flache Klassenhierarchien, in denen sich zwischen abgeleiteten Klassen nur wenig ändert, interessant. Von anderen Mechanismen, beispielsweise generischen Klassen, ist überhaupt nicht klar, ob sie überhaupt in LOTOS spezifiziert werden können.

- Die Erfahrung mit LOTOS hat gezeigt, daß, um die Vorteile einer exekutierbaren Spezifikationssprache auch wirklich ausschöpfen zu können, es geeigneter Entwicklungswerkzeuge bedarf. Für LOTOS ist das im Moment nicht der Fall. Zum Testen kleiner Spezifikationen bzw. Testfälle kann SMILE empfohlen werden. Bei größeren Testfällen reagierte es mit Abstürzen und extrem langen Wartezeiten. Wir vermuten, daß dies mit der großen Anzahl der in jedem Moment zur Verfügung stehenden Services zu tun hat. Die umfangreichen ADT-Definitionen, die für die Objekt-Zustände notwendig sind, sind sicherlich auch ein Grund.

- Sehr gut finden wir, daß mit LOTOS bestimmte Aspekte des Objekt-Modells ganz genau spezifiziert werden müssen. Vor allem die Kommunikation zwischen Objekten muß mit LOTOS detailliert herausgearbeitet werden. Hier kommt der ursprüngliche Verwendungszweck von LOTOS – die formale Spezifikation des Kommunikationsverhaltens zwischen Prozessen – voll zur Geltung. Die Bindung von Services an bestimmte Gates erlaubt eine saubere Trennung der Clients eines Objekts. Nachteilig stellt sich dabei heraus, daß Services, die an verschiedenen Gates angeboten werden sollen, mehrfach spezifiziert werden müssen (siehe die Klassen für Ein- und Ausgänge). Trotzdem: Kommunikation findet an definierten Schnittstellen zu definierten Zeitpunkten statt.

- LOTOS verfügt über keinen Mechanismus, um größere Spezifikationen modular aufzubauen. Mit `library...endlib` können am Anfang der Spezifikation lediglich Typdefinitionen eingefügt werden.

- Die Trennung von Objekt-Zustand und -Verhalten hat sowohl Vorteile als auch Nachteile. Einerseits macht sie sichtbarer, worin der Zustand eines Objektes eigentlich besteht. Andererseits wird die Spezifikation dadurch örtlich „auseinandergerissen", weil in der Spezifikation zuerst alle ADTs, dann alle Prozesse angegeben sind.

- ETDs sind gut geeignet, *ein* mögliches Verhalten einer Gruppe von Objekten darzustellen. Bei aktiven Objekten muß aber daran gedacht werden, daß es unter Umständen viele verschiedene mögliche Abarbeitungsreihenfolgen geben kann. Diese gehen aus dem ETD nicht klar hervor. Andere Methoden, wie z. B. Petri-Netze oder endliche Automaten sollten deshalb als alternative Beschreibungsmethoden von Objekten in Betracht gezogen werden.

8.2 Entwicklung des Schaltungssystems

Die Entwicklung des Schaltungssystems wurde von dem Wunsch getragen, Signalverarbeitungsfunktionen modular aus einfacheren Funktionen zusammenbauen zu können. Eine mögliche nebenläufige Abarbeitung der Elemente des Schaltungssystems war ebenfalls erwünscht, weil im Bereich digitaler Signalverarbeitung immer wieder rechenintensive Aufgaben anfallen oder Zeitlimits einzuhalten sind.

Die Erstellung des ersten Objekt-Modells war einfach, weil von Anfang an klar war, daß Schaltungen, Atome und Ein- und Ausgänge eigene Klassen bilden würden. Von der Ausbildung einer Hierarchie von Atomen war bereits in den Anforderungen ausgegangen worden.

Der größte Teil des Aufwandes bei der Entwicklung des Schaltungssystems lag sicher im Entwurf seines dynamischen Modells. Erst nach mehreren Anläufen konnten die Ziele, einerseits einen möglichst effizienten Datenfluß zu realisieren, andererseits ein jederzeitiges Unterbrechen und Wiederaufnehmen des Ablaufs zu garantieren, erfüllt werden. Die Forderung, soviel „Standardaufgaben“ wie möglich in der Atom-Klasse zu konzentrieren und gleichzeitig ein adaptierbares Schaltverhalten anzubieten, war eine weitere Hürde. In dieser Phase der Entwicklung wurden zahlreiche ETDs erstellt und wieder verworfen. Zu diesem Zeitpunkt mußten wir auch feststellen, daß ETDs im Zusammenhang mit aktiven Objekten mit Vorsicht zu genießen sind: die Ausführung der Services eines aktiven Objekts kann von Aufruf zu Aufruf verschieden lang dauern. Bei mehreren beteiligten Objekten können dadurch verschiedene Aufrufreihenfolgen entstehen, von denen einige möglicherweise zu einem Deadlock führen.

Beim Übergang zur LOTOS-Spezifikation waren das Object Communication Diagram (OCD) und die Object Communication Table (OCT) von großem Nutzen, weil sie einen Überblick über die involvierten Prozesse und Gates gaben. Sie halfen damit, die Spezifikation zugänglicher und wartbarer zu halten. Im Gegensatz zu den AutorInnen der Methode, die vorschlagen, anfangs nur einzelne Objekte zu spezifizieren, gingen wir sofort daran, die richtigen Mengenverhältnisse zwischen den Objekten herzustellen. Wir beschränkten uns jedoch zu Beginn darauf, nur die wichtigsten Services zu spezifizieren.

Mit den beschriebenen Tools konnte zumindest die Syntax der Spezifikation laufend überprüft werden. Dadurch kamen bereits zahlreiche Fehler, vor allem in den Spezifikationen der ADTs, frühzeitig zum Vorschein. Mit dem Simulations-Tool SMILE waren wir eingeschränkt in der Lage, Services von Objekten aufzurufen und so den dynamischen Aspekt der Spezifikation zu testen. Die daraus gewonnenen Erkenntnisse wurden verwendet, um das Objekt-Modell und die Spezifikation weiter zu verfeinern.

Relativ spät wurden asynchrone Eingänge dem Modell hinzugefügt. Auch die Notwendigkeit, Ein- und Ausgänge außerhalb von Atomen zu erzeugen, um damit den Datenaustausch zwischen dem Schaltungssystem und seiner Umgebung zu ermöglichen, wurde erst in einem fortgeschrittenen Stadium als wichtig erkannt und realisiert.

Generell hatte die Anwendung der Methode den Vorteil, daß aus einer unscharfen Vorstellung des Schaltungssystem ein exakt definiertes Modell spezifiziert werden konnte, bevor noch eine bestimmte Architektur für die Implementierung festgelegt wurde (angestrebt ist eine nebenläufige Implementierung in C++, bei der jedes Atom in einem eigenen Thread läuft). Beim Übergang von der formalen Spezifikation zur Implementierung sind sicher wieder spezifische Probleme zu lösen. Vor allem die Frage, wie die Synchronisations- und Kommunikationsmechanismen von LOTOS auf die Zielarchitektur abgebildet werden sollen, ist zu beantworten.

Zusammenfassung

Die vorliegende Arbeit hat gezeigt, wie objektorientierte Systeme mit LOTOS formal spezifiziert werden können. Anstatt sich jedoch auf eine „freistehende“ Spezifikation zu beschränken, wurde eine Methode verwendet, die von einem objektorientierten Modell des Systems ausgeht: rigorose objektorientierte Analyse (ROOA). Ziel der ROOA ist es, den Prozeß der objektorientierten Analyse durch die Integration einer formalen Spezifikationssprache exakter und eindeutiger zu machen. Die Erstellung einer ausführbaren Spezifikation soll außerdem früh einen konzeptuellen Eindruck des fertigen Programms liefern und helfen, Problemfelder in einer frühen Phase der Entwicklung zu erkennen.

Mit der beschriebenen Methode wurde ein objektorientiertes, nebenläufiges Datenflußsystem formal spezifiziert. Mit dem Datenflußsystem können verschiedenartige Rechenelemente (*Atome*) zu *Schaltungen* zusammengesetzt werden, um damit komplexere Funktionen zu realisieren. Die Atome einer Schaltung agieren unabhängig voneinander, gesteuert durch den Datenfluß. Die Spezifikation erfolgte mit dem Ziel, einerseits die dem Schaltungssystem innewohnende Parallelverarbeitung auszunutzen, andererseits durch objektorientierte Entwicklung die Erweiterbarkeit des Systems zu vereinfachen. Besonderes Augenmerk lag bei der Entwicklung einer Basisklasse für alle Atome, die die Details des Datentransfers und der Nebenläufigkeit kapselt. EntwicklerInnen können auf einfache Weise von dieser Klasse eigene Atom-Klassen ableiten und so die Funktionalität des Systems erweitern. Auf die Vermeidung von Deadlocks, insbesondere beim Verhalten der Atome, wurde ebenfalls großer Wert gelegt.

Wir sind der Ansicht, daß die verwendeten Methoden für die beschriebene Aufgabenstellung im großen und ganzen gut geeignet waren und fassen noch einmal ihre Vor- und Nachteile zusammen. In bezug auf die vorliegende Arbeit treten folgende Vorteile hervor:

- Der Entwicklungsprozeß wird durch die Einbindung einer formalen Spezifikationssprache exakter. Mängel und Auslassungen an der ursprünglichen Aufgabenstellung können früher erkannt und korrigiert werden, wodurch der Gesamtaufwand der Entwicklung gesenkt wird.

- Die resultierende Spezifikation ist programmiersprachen- und plattformunabhängig. Auch die zugrundeliegende Art der Parallelverarbeitung wird nicht explizit vorgeschrieben, d. h., es ist eine verteilte Implementierung ebenso denkbar wie eine Implementierung auf einem Shared-Memory-System.

- Die Spezifikation ist – zumindest theoretisch – ausführbar. Trotz der Probleme bei der Ausführung der Spezifikation (siehe auch weiter unten) zeigt sie das dynamische

Verhalten des Systems deutlicher und eindeutiger als es eine verbale Beschreibung und die verschiedenen Dokumente der gewählten OOA-Methode zulassen.

Den Vorteilen stehen folgende Nachteile gegenüber:

- Durch das Fehlen echter objektorientierter Sprachelemente in LOTOS tritt das erstellte Objekt-Modell aus der Spezifikation selbst nicht mehr klar hervor. Der schablonenhafte Spezifikationsstil muß permanent „rückübersetzt" werden. Besonders Vererbung ist in ihren verschiedenen Ausformungen nur umständlich spezifizierbar und als solche in der LOTOS-Spezifikation nicht mehr einfach zu erkennen.

- Die Unterstützung durch Entwicklungswerkzeuge ist mangelhaft. Die Auswahl an Werkzeugen ist gering und aktive Weiterentwicklung findet nur bei den wenigsten statt. Für das Testen der vorliegenden Spezifikation war keines der Werkzeuge uneingeschränkt brauchbar. Einige unterstützten nicht den gesamten Sprachstandard, andere waren in der Benutzung zu instabil.

Zusammenfassend kann gesagt werden, daß die Einbindung einer formalen, exekutierbaren Spezifikationssprache in den Analyseprozeß große Vorteile mit sich bringt. Der Hauptvorteil besteht unserer Meinung nach darin, daß die Funktionalität des entwickelten Systems bereits sehr früh getestet werden kann, ohne sich jedoch an eine bestimmte Implementierung zu binden. Die Möglichkeit, relativ einfach nebenläufige Objekte zu spezifizieren, spricht ebenfalls für die Methode.

Wir sind der Meinung, daß die Verwendung von LOTOS als Spezifikationssprache in diesem Zusammenhang nicht optimal ist, sowohl aus syntaktischen Gründen als auch wegen mangelnder Unterstützung durch Werkzeuge. Die EntwicklerInnen der ROOA trafen diese Wahl wahrscheinlich aus pragmatischen Gründen—LOTOS ist leicht zu erlernen, standardisiert und relativ weit verbreitet.

Unserer Meinung nach hat diese Arbeit deutlich gezeigt, daß die Entwicklung und Verbreitung einer formalen Spezifikationssprache für aktive Objekte in diesem Zusammenhang sehr wünschenswert wäre. Da ROOA prinzipiell unabhängig von der eingesetzten Spezifikationssprache ist, würde sich an der Methode selbst und ihren Vorteilen auch wenig ändern.

Literaturverzeichnis

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BNT94] Tommaso Bolognesi, Elie Najm, and P.A.J. Tilanus. G-LOTOS: A graphical language for concurrent systems. *Computer Networks and ISDN Systems*, 26(9):1101–1127, 1994.

[Boo94] Grady Booch. *Object Oriented Analysis and Design.* Benjamin–Cummings, Redwood City, Calif., 2nd edition, 1994.

[CCI88] CCITT. *Recommendation Z.100: Specification and Description Language (SDL).* CCITT, Genf, 1988.

[CM98] Robert G. Clark and Ana M.D. Moreira. ROOA with SDL. Technical Report CSM-147, Department of Computing Science and Mathematics, University of Stirling, Stirling, UK, November 1998. verfügbar unter http://www.cs.stir.ac.uk/research/techreps/index.html.

[CRS90] Elspeth Cusack, Steve Rudkin, and Chris Smith. An object oriented interpretation of LOTOS. In S.T. Vuong, editor, *Formal description techniques II, FORTE '89 proceedings*, pages 211–226. North Holland, 1990.

[CV90] Richard C. Cam and Son T. Vuong. A formal specification, in LOTOS, of a simplified cellular mobile communications system. In S.T. Vuong, editor, *Formal description techniques II, FORTE '89 proceedings*, pages 485–499. North Holland, 1990.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press. Prentice Hall, Englewood Cliffs, 2nd edition, 1991.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specification.* EATCS monographs on theoretical computer science. Springer-Verlag, 1985.

[Fær93] O. Færgemand. Introduction to SDL. In K.J. Turner, editor, *Using Formal Description Languages: An Introduction to ESTELLE, LOTOS and SDL*, chapter 4, pages 85–124. John Wiley & Sons, Chichester, 1993.

[Hed93] Mikael Hedlund. The integration of LOTOS with an object-oriented development method. In J.C.P. Woodcock and P.G. Larsen, editors, *FME '93:*

Industrial-Strength Formal Methods. Formal Methods Europe, Springer-Verlag, 1993. Lecture Notes in Computer Science 670.

[Hil92] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, March 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[ISO89a] ISO. *LOTOS: A formal description language based on the temporal ordering of observational behviour*. ISO, Genf, 1989. ISO 8807.

[ISO89b] ISO/IEC. *Estelle: A formal description technique based on an extended state transition model*. ISO/IEC, Genf, 1989. ISO/IEC 9074.

[ISO94] ISO. *Open Systems Interconnection - Basic Reference Model: The Basic Model*. ISO/IEC, Genf, 1994. ISO/IEC 7498.

[ISO97] ISO/IEC JTC1/SC21/WG7. *Final Comite Draft on Enhancements to LOTOS*. ISO/IEC, Genf, 1997.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison Wesley/ACM Press, Reading, Mass., 1992.

[LMD90] Luigi Logrippo, Tim Melanchuk, and Robert J. DuWors. The algebraic specification language LOTOS: An industrial experience. *Proceedings of the ACM SIGSOFT*, 15(4):59–66, September 1990.

[May89] Thomas Mayr. *Spezifikation objektorientierter Systeme in LOTOS*. PhD thesis, TU-Wien, 1989.

[MC94] Ana M.D. Moreira and Robert G. Clark. Combining object-oriented analysis and formal description techniques. In M. Tokoro and R. Pareschi, editors, *Object Oriented Programming, ECOOP '94*, pages 344–364. Springer-Verlag, 1994. Lecture Notes in Computer Science 821.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.

[Mor94] Ana M.D. Moreira. *Rigorous Object-Oriented Analysis*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, Stirling, UK, 1994. Technical Report CSM-132, verfügbar unter http://www.cs.stir.ac.uk/research/techreps/index.html.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, Cambridge, MA, 1993.

[Nie93] Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proceedings of OOPSLA '93.

[Pun95] Franz Puntigam. Flexible types for a concurrent model. In *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, June 1995. vefügbar unter http://www.complang.tuwien.ac.at/franz/papers/oopmoc95.ps.gz.

[QAP93] J. Quemada, A. Azcorra, and S. Pavón. Development with LOTOS. In K.J. Turner, editor, *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*, chapter 11, pages 345–373. John Wiley & Sons, Chichester, 1993.

[QPM+93] J. Quemada, L. Ferreira Pires, J.A. Mañas, A. Azcorra, and T. Robbles. Introduction to LOTOS. In K.J. Turner, editor, *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*, chapter 3, pages 47–83. John Wiley & Sons, Chichester, 1993.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[Rud92] Steve Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal description techniques IV, FORTE '91 proceedings*, pages 409–424. North Holland, 1992.

[Sha92] John A. Sharp. A brief introduction to data flow. In John A. Sharp, editor, *Data flow computing: theory and practice*, chapter 1, pages 1–15. Ablex Publishing Co., Norwood, NJ, 1992.

[Ten93] R.L. Tenney. Introduction to ESTELLE. In K.J. Turner, editor, *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*, chapter 2, pages 17–45. John Wiley & Sons, Chichester, 1993.

[Tur90] Kenneth J. Turner. A LOTOS-based development strategy. In S.T. Vuong, editor, *Formal description techniques II, FORTE '89 proceedings*, pages 117–132. North Holland, 1990.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*. John Wiley & Sons, Chichester, 1993.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

Anhang A

Spezifikation in LOTOS

```
(*-------------------------------------------------------
SPEC : CircuitSystem
GATES: 'i_c'..gate between interface scenario and circuit
'i_a'...gate between interface scenario and atoms
'i_con'...gate between interface scenario and connections
          (inputs/outputs)
'i_t'...gate between test cases and circuit
--------------------------------------------------------*)
specification CircuitSystem[i_c, i_a, i_con, i_t] : noexit

library
        BOOLEAN, NATURALNUMBER, SET
endlib

type List_Type is Boolean, NaturalNumber
formalsorts Element
sorts List
opns
   <>     :   -> List
   _ . _  :   Element, List -> List
   _ . _  :   List, List -> List
   _ . _  :   List, Element -> List
   First  :   List -> Element
   Rest   :   List -> List
   Elem   :   List, Nat -> Element
   Elements:  List -> Nat
   Remove :   Element, List -> List
   RemoveList : List, List -> List     (* remove first list from second *)
   IsEmpty :  List -> Bool
   _ IsElem _ : Element, List -> Bool
   _ Contains _ : List, List -> Bool
   _ eq _ : List, List -> Bool
eqns
   forall e, e1, e2: Element, l, l1, l2: List, n: Nat
      ofsort List
         <> . l=l;
         <> . e=e . <>;
         (e.l1).l2=e.(l1.l2);
         (e1.l).e2=e1.(l.e2);
         Remove(e, <>)= <> ;
         Remove(e1, e2.l)=e2.Remove(e1, l);
         Remove(e, e.l)=l;
         RemoveList(<>, l)=l;
         RemoveList(e.l1, l2)=RemoveList(l1, Remove(e, l2));
         Rest(<>)= <> ;
         Rest(e.l)=l;
```

```
      ofsort Element
         First(e.l)=e;
         Elem(e.l, 0)=e;
         Elem(e.l, succ(n))=Elem(l, n);
      ofsort Nat
         Elements(<>)=0;
         Elements(e.l)=succ(Elements(l));
      ofsort Bool
         e IsElem <> =false;
         e1 IsElem (e2.l)=e1 IsElem l;
         e IsElem (e.l)=true;
         IsEmpty(<>)=true;
         IsEmpty(e.l)=false;
         l Contains <> =true;
         l1 Contains (e.l2) = (e IsElem l1) and (l1 Contains l2);
         l1 eq l2=(l1 Contains l2) and (l2 Contains l1);
endtype

type Id_Type is Boolean, NaturalNumber
sorts Id
opns id1, id2, id3, id4, id5, id6,
   id7, id8, id9, id10, id11,
   id12, id13, id14, id15, id16,
   id17, id18, id19, id20 : -> Id
   h : Id -> Nat
   _eq_, _ne_, _lt_ : Id, Id -> Bool
   First_Set : Id -> Bool
   Second_Set : Id -> Bool
   Third_Set : Id -> Bool
   Fourth_Set : Id -> Bool
eqns forall n1, n2: Id
   ofsort Nat
      h(id1) = 0;
      h(id2) = succ(h(id1));
      h(id3) = succ(h(id2));
      h(id4) = succ(h(id3));
      h(id5) = succ(h(id4));
      h(id6) = succ(h(id5));
      h(id7) = succ(h(id6));
      h(id8) = succ(h(id7));
      h(id9) = succ(h(id8));
      h(id10) = succ(h(id9));
      h(id12) = succ(h(id11));
      h(id13) = succ(h(id12));
      h(id14) = succ(h(id13));
      h(id15) = succ(h(id14));
      h(id16) = succ(h(id15));
      h(id17) = succ(h(id16));
      h(id18) = succ(h(id17));
      h(id19) = succ(h(id18));
      h(id20) = succ(h(id19));
   ofsort Bool
      n1 eq n2 = h(n1) eq h(n2);
      n1 ne n2 = h(n1) ne h(n2);
      n1 lt n2 = h(n1) lt h(n2);
      First_Set(n1) = h(n1) lt h(id5);
      Second_Set(n1) = not(h(n1) lt h(id5)) and (h(n1) lt h(id10));
      Third_Set(n1) = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
      Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
endtype


(*----------------------------------------------------
definition of various Id types as well as associated
List and Set types
```

```
------------------------------------------------------*)
type Circuit_Id_Type is Id_Type renamedby
   sortnames
      Circuit_Id for Id
endtype

type Atom_Id_Type is Id_Type renamedby
   sortnames
      Atom_Id for Id
   opnnames
      Is_Constant for First_Set
      Is_Adder for Second_Set
      Is_Selector for Third_Set
endtype

type Atom_Id_Set_Type is Set actualizedby Atom_Id_Type
   using
   sortnames
      Atom_Id_Set for Set
      Atom_Id for Element
      Bool for FBool
endtype

type Input_Id_Type is Id_Type renamedby
   sortnames
      Input_Id for Id
   opnnames
      Is_Sync_Input for First_Set
      Is_Async_Input for Second_Set
endtype

type Input_Id_List_Type is List_Type actualizedby Input_Id_Type
   using
   sortnames
      Input_Id_List for List
      Input_Id for Element
endtype

type Input_Id_Set_Type is Set actualizedby Input_Id_Type
   using
   sortnames
      Input_Id_Set for Set
      Input_Id for Element
      Bool for FBool
endtype

type Output_Id_Type is Id_Type renamedby
   sortnames
      Output_Id for Id
endtype

type Output_Id_List_Type is List_Type actualizedby Output_Id_Type
   using
   sortnames
      Output_Id_List for List
      Output_Id for Element
endtype

type Output_Id_Set_Type is Set actualizedby Output_Id_Type
   using
   sortnames
      Output_Id_Set for Set
      Output_Id for Element
      Bool for FBool
endtype
```

```
(*----------------------------------------------------
definition of various helper types for object state
-----------------------------------------------------*)

type Run_Mode_Type is Id_Type renamedby
   sortnames
      Run_Mode for Id
   opnnames
      rm_started for id1
      rm_stopping for id2
      rm_stopped for id3
endtype

type Atom_Step_Type is Id_Type renamedby
   sortnames
      Atom_Step for Id
   opnnames
      as_waiting_for_connections for id1
      as_done_compute for id2
endtype

type Con_Status_Type is Id_Type renamedby
   sortnames
      Con_Status for Id
   opnnames
      cs_free for id1
      cs_set for id3
endtype


(*----------------------------------------------------
definition of object state types
-----------------------------------------------------*)

(*
TYPE   : State_Circuit
MEMBERS: 'Id'...object id of circuit
'Atoms'...set of atoms belonging to circuit
'Run_Mode'...current run mode of circuit
*)
type Circuit_Type is Circuit_Id_Type, Atom_Id_Set_Type, Run_Mode_Type
sorts State_Circuit
opns
   Make_Circuit : Circuit_Id, Atom_Id_Set, Run_Mode -> State_Circuit
   Get_Id : State_Circuit -> Circuit_Id
   Get_Atoms : State_Circuit -> Atom_Id_Set
   Add_Atom : State_Circuit, Atom_Id -> State_Circuit
   Remove_Atom: State_Circuit, Atom_Id -> State_Circuit
   Get_Run_Mode : State_Circuit -> Run_Mode
   Set_Run_Mode : State_Circuit, Run_Mode -> State_Circuit
eqns
   forall c: Circuit_Id, a: Atom_Id, a_ids: Atom_Id_Set,
          r, rn: Run_Mode
      ofsort Circuit_Id
         Get_Id(Make_Circuit(c, a_ids, r))=c;
      ofsort Atom_Id_Set
         Get_Atoms(Make_Circuit(c, a_ids, r))=a_ids;
      ofsort Run_Mode
         Get_Run_Mode(Make_Circuit(c, a_ids, r))=r;
      ofsort State_Circuit
         Set_Run_Mode(Make_Circuit(c, a_ids, r), rn)=
            Make_Circuit(c, a_ids, rn);
```

```
         Add_Atom(Make_Circuit(c, a_ids, r), a)=
            Make_Circuit(c, Insert(a, a_ids), r);
         Remove_Atom(Make_Circuit(c, a_ids, r), a)=
            Make_Circuit(c, Remove(a, a_ids), r);
endtype

(*
TYPE   : State_Atom
MEMBERS: 'Id'...object id of atom
'Sync_Inputs'...list of synchronous atom inputs
'Set_Inputs'...list of set synchronous inputs
'Required_Inputs'...list of synchronous inputs that are required for
                    compute and will be freed during the next
                    'Update_Connections'
'Async_Inputs'...list of asynchronous inputs
'Outputs'...list of atom outputs
'Done_Outputs'...list of done Outputs
'Required_Outputs'...list of outputs that are required for compute and will
                     be set during the next 'Update_Connections'
'Run_Mode'...current run mode of circuit
'Atom_Step'...current atom step mode
'Needs_Control'...indicates whether the atom is waiting for a message from
                  the atom controller
*)
type Atom_Type is Atom_Id_Type, Input_Id_List_Type, Input_Id_Type,
     Output_Id_List_Type, Output_Id_Type, Run_Mode_Type,
     Atom_Step_Type, Boolean
sorts State_Atom
opns
   Make_Atom : Atom_Id, Input_Id_List, Input_Id_List, Input_Id_List,
               Input_Id_List, Output_Id_List, Output_Id_List,
               Output_Id_List, Run_Mode, Atom_Step, Bool
               -> State_Atom
   Get_Id : State_Atom -> Atom_Id
   Get_Sync_Inputs : State_Atom -> Input_Id_List
   Get_Set_Inputs : State_Atom -> Input_Id_List
   Add_Set_Input : State_Atom, Input_Id -> State_Atom
   Remove_Set_Input : State_Atom, Input_Id -> State_Atom
   Get_Required_Inputs : State_Atom -> Input_Id_List
   Set_Required_Inputs : State_Atom, Input_Id_List -> State_Atom
   Get_Async_Inputs : State_Atom -> Input_Id_List
   Get_Outputs : State_Atom -> Output_Id_List
   Add_Done_Output : State_Atom, Output_Id -> State_Atom
   Get_Done_Outputs : State_Atom -> Output_Id_List
   Get_Required_Outputs : State_Atom -> Output_Id_List
   Set_Required_Outputs : State_Atom, Output_Id_List -> State_Atom
   Get_Run_Mode : State_Atom -> Run_Mode
   Set_Run_Mode : State_Atom, Run_Mode -> State_Atom
   Get_Atom_Step : State_Atom -> Atom_Step
   Set_Atom_Step : State_Atom, Atom_Step -> State_Atom
   Needs_Control : State_Atom -> Bool
   Set_Needs_Control : State_Atom, Bool -> State_Atom
eqns
   forall a_id: Atom_Id,
          i_ids, si_ids, ri_ids, nri_ids, ai_ids: Input_Id_List,
          i_id: Input_Id,
          o_ids, do_ids, ro_ids, nro_ids: Output_Id_List,
          o_id: Output_Id,
          r, rn: Run_Mode,
          as, nas: Atom_Step,
          nc, nnc: Bool
      ofsort Atom_Id
         Get_Id(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids, o_ids,
                          do_ids, ro_ids, r, as, nc))=a_id;
      ofsort Input_Id_List
```

```
         Get_Sync_Inputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                   o_ids, do_ids, ro_ids, r, as, nc))=
                                   i_ids;
         Get_Set_Inputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                  o_ids, do_ids, ro_ids, r, as, nc))=
                                  si_ids;
         Get_Required_Inputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                       o_ids, do_ids, ro_ids, r, as, nc))=
                                       ri_ids;
         Get_Async_Inputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                    o_ids, do_ids, ro_ids, r, as, nc))=
                                    ai_ids;
      ofsort Output_Id_List
         Get_Outputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids, o_ids,
                               do_ids, ro_ids, r, as, nc))= o_ids;
         Get_Done_Outputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                    o_ids, do_ids, ro_ids, r, as, nc))=
                                    do_ids;
         Get_Required_Outputs(Make_Atom(a_id, i_ids, si_ids, ri_ids,
                                        ai_ids, o_ids, do_ids, ro_ids, r,
                                        as, nc))=ro_ids;
      ofsort Run_Mode
         Get_Run_Mode(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids, o_ids,
                                do_ids, ro_ids, r, as, nc))=r;
      ofsort Atom_Step
         Get_Atom_Step(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                 o_ids, do_ids, ro_ids, r, as, nc))=as;
      ofsort State_Atom
         Add_Set_Input(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                 o_ids, do_ids, ro_ids, r, as, nc), i_id)=
                 Make_Atom(a_id, i_ids, si_ids.i_id, ri_ids, ai_ids, o_ids,
                           do_ids, ro_ids, r, as, nc);
         Set_Required_Inputs(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                       o_ids, do_ids, ro_ids, r, as, nc),
                                       nri_ids)=
                             Make_Atom(a_id, i_ids, si_ids, nri_ids,
                                       ai_ids, o_ids, do_ids, ro_ids, r,
                                       as, nc);
         Add_Done_Output(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                            o_ids, do_ids, ro_ids, r, as, nc), o_id)=
                         Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                            o_ids, o_id.do_ids, ro_ids, r, as, nc);
         Set_Required_Outputs(Make_Atom(a_id, i_ids, si_ids, ri_ids,
                                 ai_ids, o_ids, do_ids, ro_ids, r, as, nc),
                                 nro_ids)=
                             Make_Atom(a_id, i_ids, si_ids, ri_ids,
                                ai_ids, o_ids, do_ids, nro_ids, r, as, nc);
         Set_Run_Mode(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids, o_ids,
                                do_ids, ro_ids, r, as, nc), rn)=
                 Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids, o_ids,
                           do_ids, ro_ids, rn, as, nc);
         Set_Atom_Step(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                 o_ids, do_ids, ro_ids, rn, as, nc), nas)=
                       Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                 o_ids, do_ids, ro_ids, rn, nas, nc);
         Set_Needs_Control(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                o_ids, do_ids, ro_ids, rn, as, nc), nnc)=
                           Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                              o_ids, do_ids, ro_ids, rn, as, nnc);
      ofsort Bool
         Needs_Control(Make_Atom(a_id, i_ids, si_ids, ri_ids, ai_ids,
                                 o_ids, do_ids, ro_ids, rn, as, nc))=nc;
endtype

(*
```

```
TYPE   : Data
MEMBERS: 'n'...the data (natural number)
*)
type Data_Type is NaturalNumber
sorts Data
opns
   Make_Data: Nat -> Data
   Set: Data, Nat -> Data
   Get: Data -> Nat
eqns
   forall n, n1: Nat
      ofsort Data
         Set(Make_Data(n), n1)=Make_Data(n1);
      ofsort Nat
         Get(Make_Data(n))=n;
endtype

(*
TYPE   : State_Sync_Input
MEMBERS: 'Id'...object id of input
'Atom'...atom to which the input belongs
'Outputs'...output to which input is connected (an empty list denotes
            an input that is not connected)
'Con_Status'...current status of input
'Data'...data that the input holds
*)
type Sync_Input_Type is Input_Id_Type, Atom_Id_Type,
     Atom_Id_Set_Type, Output_Id_Type, Output_Id_Set_Type,
     Con_Status_Type, Data_Type, Boolean
sorts State_Sync_Input
opns
   Make_Sync_Input : Input_Id, Atom_Id_Set, Output_Id_Set,
                     Con_Status, Data -> State_Sync_Input
   Get_Id : State_Sync_Input -> Input_Id
   Get_Atoms : State_Sync_Input -> Atom_Id_Set
   Is_Part_Of_Atom : State_Sync_Input -> Bool
   Get_Outputs : State_Sync_Input -> Output_Id_Set
   Connect : State_Sync_Input, Output_Id -> State_Sync_Input
   Disconnect : State_Sync_Input -> State_Sync_Input
   Get_Con_Status : State_Sync_Input -> Con_Status
   Set_Con_Status : State_Sync_Input, Con_Status -> State_Sync_Input
   Get_Data : State_Sync_Input -> Data
   Set_Data : State_Sync_Input, Data -> State_Sync_Input
eqns
   forall i_id: Input_Id, a_id: Atom_Id, a_ids: Atom_Id_Set,
          o_id: Output_Id, o_ids: Output_Id_Set, cs, csn: Con_Status,
          d, dn: Data
      ofsort Input_Id
         Get_Id(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=i_id;
      ofsort Atom_Id_Set
         Get_Atoms(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=a_ids;
      ofsort Output_Id_Set
         Get_Outputs(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=o_ids;
      ofsort Con_Status
         Get_Con_Status(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=cs;
      ofsort Data
         Get_Data(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=d;
      ofsort State_Sync_Input
         Connect(Make_Sync_Input(i_id, a_ids, o_ids, cs, d), o_id)=
                 Make_Sync_Input(i_id, a_ids, Insert(o_id, {}), cs, d);
         Disconnect(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=
                    Make_Sync_Input(i_id, a_ids, {} of Output_Id_Set,
                                    cs, d);
         Set_Con_Status(Make_Sync_Input(i_id, a_ids, o_ids, cs, d), csn)=
                        Make_Sync_Input(i_id, a_ids, o_ids, csn, d);
```

```
         Set_Data(Make_Sync_Input(i_id, a_ids, o_ids, cs, d), dn)=
                  Make_Sync_Input(i_id, a_ids, o_ids, cs, dn);
      ofsort Bool
         Is_Part_of_Atom(Make_Sync_Input(i_id, a_ids, o_ids, cs, d))=
                  a_ids ne {};
endtype

(*
TYPE   : State_Async_Input
MEMBERS: 'id'...object id of input
'Atom'...atom to which the input belongs
'outputs'...output to which input is connected (an empty list denotes
            an input that is not connected)
'internal_data'...data visible to the atom
'external_data'...data set in the latest 'set' operation
*)
type Async_Input_Type is Input_Id_Type, Atom_Id_Type,
     Atom_Id_Set_Type, Output_Id_Type, Output_Id_Set_Type,
     Con_Status_Type, Data_Type, Boolean
sorts State_Async_Input
opns
   Make_Async_Input : Input_Id, Atom_Id_Set, Output_Id_Set, Data,
                      Data -> State_Async_Input
   Get_Id : State_Async_Input -> Input_Id
   Get_Atoms : State_Async_Input -> Atom_Id_Set
   Is_Part_Of_Atom : State_Async_Input -> Bool
   Get_Outputs : State_Async_Input -> Output_Id_Set
   Connect : State_Async_Input, Output_Id -> State_Async_Input
   Disconnect : State_Async_Input -> State_ASync_Input
   Get_Internal_Data : State_Async_Input -> Data
   Set_Internal_Data : State_Async_Input, Data -> State_Async_Input
   Get_External_Data : State_Async_Input -> Data
   Set_External_Data : State_Async_Input, Data -> State_Async_Input
eqns
   forall i_id: Input_Id, a_id: Atom_Id, a_ids: Atom_Id_Set,
          o_id: Output_Id, o_ids: Output_Id_Set, id, idn, ed, edn : Data
      ofsort Input_Id
         Get_Id(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=i_id;
      ofsort Atom_Id_Set
         Get_Atoms(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=a_ids;
      ofsort Output_Id_Set
         Get_Outputs(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=o_ids;
      ofsort Data
         Get_Internal_Data(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=
                                                                     id;
         Get_External_Data(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=
                                                                     ed;
      ofsort State_Async_Input
         Connect(Make_Async_Input(i_id, a_ids, o_ids, id, ed), o_id)=
                 Make_Async_Input(i_id, a_ids, Insert(o_id, {}), id, ed);
         Disconnect(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=
                    Make_Async_Input(i_id, a_ids, {}, id, ed);
         Set_Internal_Data(Make_Async_Input(i_id, a_ids, o_ids, id, ed),
                           idn)=
                           Make_Async_Input(i_id, a_ids, o_ids, idn, ed);
         Set_External_Data(Make_Async_Input(i_id, a_ids, o_ids, id, ed),
                           edn)=
                           Make_Async_Input(i_id, a_ids, o_ids, id, edn);
      ofsort Bool
         Is_Part_Of_Atom(Make_Async_Input(i_id, a_ids, o_ids, id, ed))=
                 a_ids ne {};
endtype

(*
TYPE   : State_Output
```

```
MEMBERS: 'Id'...object id of output
'Atom'...atom to which the output belongs
'Inputs'...set of inputs which are connected to the output
'Done_Inputs'...set of inputs that are done
'Con_Status'...current status of output
'Data'...data that the output holds
*)
type Output_Type is Output_Id_Type, Input_Id_Type, Input_Id_Set_Type,
     Atom_Id_Type, Atom_Id_Set_Type, Con_Status_Type, Data_Type
sorts State_Output
opns
   Make_Output : Output_Id, Atom_Id_Set, Input_Id_Set, Input_Id_Set,
                 Con_Status, Data -> State_Output
   Get_Id : State_Output -> Output_Id
   Get_Atoms : State_Output -> Atom_Id_Set
   Is_Part_Of_Atom : State_Output -> Bool
   Get_Inputs : State_Output -> Input_Id_Set
   Connect : State_Output, Input_Id -> State_Output
   Disconnect : State_Output, Input_Id -> State_Output
   Get_Done_Inputs : State_Output -> Input_Id_Set
   Add_Done_Input : State_Output, Input_Id -> State_Output
   Clear_Done_Inputs : State_Output -> State_Output
   Get_Con_Status : State_Output -> Con_Status
   Set_Con_Status : State_Output, Con_Status -> State_Output
   Get_Data : State_Output -> Data
   Set_Data : State_Output, Data -> State_Output
eqns
   forall o_id: Output_Id, a_id: Atom_Id, a_ids: Atom_Id_Set,
          i_id: Input_Id, i_ids, di_ids: Input_Id_Set, cs, csn: Con_Status,
          d, dn: Data
      ofsort Output_Id
         Get_Id(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=o_id;
      ofsort Atom_Id_Set
         Get_Atoms(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=a_ids;
      ofsort Input_Id_Set
         Get_Inputs(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=i_ids;
         Get_Done_Inputs(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=
                                    di_ids;
      ofsort Con_Status
         Get_Con_Status(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=cs;
      ofsort Data
         Get_Data(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=d;
      ofsort State_Output
         Connect(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d), i_id)=
                 Make_Output(o_id, a_ids, Insert(i_id, i_ids), di_ids, cs,
                             d);
         Disconnect(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d), i_id)=
                 Make_Output(o_id, a_ids, Remove(i_id, i_ids), di_ids, cs,
                             d);
         Add_Done_Input(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d),
                                   i_id)=
                 Make_Output(o_id, a_ids, i_ids, Insert(i_id, di_ids), cs,
                             d);
         Clear_Done_Inputs(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=
             Make_Output(o_id, a_ids, i_ids, {} of Input_Id_Set, cs, d);
         Set_Con_Status(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d),
                        csn)=
             Make_Output(o_id, a_ids, i_ids, di_ids, csn, d);
         Set_Data(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d), dn)=
             Make_Output(o_id, a_ids, i_ids, di_ids, cs, dn);
      ofsort Bool
         Is_Part_Of_Atom(Make_Output(o_id, a_ids, i_ids, di_ids, cs, d))=
                 a_ids ne {};
endtype
```

```

(*----------------------------------------------------
definition of operation names
-----------------------------------------------------*)
type Op_Names is
sorts Op_Name
opns
   create, constant, adder, selector, sync_input, async_input, output,
   connect, disconnect,
   start, stopp, rtn_stopp, reset, rtn_reset,
   add_atom, remove_atom,
   get, rtn_get, set, rtn_set, free, read, write, input_done, update,
   get_sync_input, rtn_get_sync_input,
   get_async_input, rtn_get_async_input,
   get_output, rtn_get_output,
   input_set, output_done,
   can_compute, rtn_can_compute, compute, update_connections, wait,
   begin_stopp, rtn_begin_stopp, stopped, stopp_atom,
   is_ready, control : -> Op_Name
endtype

(*----------------------------------------------------
definition of system behaviour
-----------------------------------------------------*)
behaviour
   (
      hide c_a in
         Circuit[i_c, c_a](Make_Circuit(id1 of Circuit_Id,
                              {} of Atom_Id_Set, rm_stopped))
         |[c_a]|
         (
            hide a_con in
               Atoms[i_a, c_a, a_con]
               |[a_con]|
               Connections[i_con, a_con]
         )
   )
   |[i_c, i_a, i_con]|
   Test[i_c, i_a, i_con, i_t]
where

(*
PROCESS  : Circuit
BEHAVIOUR: process defining the 'Circuit' class template
GATES    : 'i_c'...gate between interface scenario and circuit
'c_a'...gate between circuit and atoms
*)
process Circuit[i_c, c_a](state: State_Circuit) : noexit :=
(
   [Get_Run_Mode(state) eq rm_stopped] ->
   (
      (
         i_c !start !Get_Id(state);
         Start_Atoms[c_a](Get_Atoms(state)) >>
         exit(Set_Run_Mode(state, rm_started))
      )
      []
      (
         i_c !reset !Get_Id(state);
         Reset_Atoms[c_a](Get_Atoms(state)) >>
         i_c !rtn_reset !Get_Id(state);
         exit(state)
      )
      []
      i_c !add_atom !Get_Id(state) ?atom: Atom_Id;
```

```
         exit(Add_Atom(state, atom))
         []
         i_c !remove_atom !Get_Id(state) ?atom:Atom_Id;
         exit(Remove_Atom(state, atom))
      )
      []
      [Get_Run_Mode(state) eq rm_started] ->
      (
         i_c !stopp !Get_Id(state);
         Begin_Stopping_Atoms[c_a](Get_Atoms(state)) >>
         Stopp_Atoms[c_a](Get_Atoms(state)) >>
         i_c !rtn_stopp !Get_Id(state);
         exit(Set_Run_Mode(state, rm_stopped))
      )
   ) >> accept update_state: State_Circuit in
               Circuit[i_c, c_a](update_state)

      where
      process Start_Atoms[x](atoms: Atom_Id_Set) : exit :=
         [atoms ne {}] ->
            x !start ?id: Atom_Id [id IsIn atoms];
            Start_Atoms[x](Remove(id, atoms))
         []
         [atoms eq {}] ->
            exit
      endproc

      process Reset_Atoms[x](atoms: Atom_Id_Set) : exit :=
         [atoms ne {}] ->
            x !reset ?id: Atom_Id [id IsIn atoms];
            Reset_Atoms[x](Remove(id, atoms))
         []
         [atoms eq {}] ->
            exit
      endproc

      process Begin_Stopping_Atoms[x](atoms: Atom_Id_Set) : exit :=
         [atoms ne {}] ->
            x !begin_stopp ?id: Atom_Id [id IsIn atoms];
            Begin_Stopping_Atoms[x](Remove(id, atoms))
         []
         [atoms eq {}] ->
            exit
      endproc

      process Stopp_Atoms[x](atoms: Atom_Id_Set) : exit :=
         [atoms ne {}] ->
            x !stopp ?id: Atom_Id [id IsIn atoms];
            Stopp_Atoms[x](Remove(id, atoms))
         []
         [atoms eq {}] ->
            exit
      endproc

   endproc (* Circuit *)

   (*
   PROCESS  : Atoms
   BEHAVIOUR: process encapsulating the atoms subsystem consisting of the top
   level class 'Atom' and its subclasses 'Constant', 'Adder', 'Selector' and
   'Distributor'
   GATES    : 'i_a'...gate between interface scenario and atoms
   'c_a'...gate between circuit and atoms
   'a_con'...gate between atoms and connections (inputs/outputs)
   *)
```

```
process Atoms[i_a, c_a, a_con] : noexit :=
   hide a_ac, a_as in
      (
         Constants[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Adders[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Selectors[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
         |||
         Distributors[i_a, c_a, a_con, a_ac, a_as]({} of Atom_Id_Set)
      )
      |[a_ac]|
      Atom_Controller[a_ac, a_con]
      |[a_as]|
      Atom_Stopper[a_as, a_con]
   where

   (*
   PROCESS  : Atom
   BEHAVIOUR: process defining the 'Atom' class template, which
   is the base class for all atom classes
   GATES    : 'i_a'...gate between interface scenario and atom
   'c_a'...gate between circuit and atom
   'a_con'...gate between atom and connections
   'a_ac'...gate between atom and atom controller
   'a_as'...gate between atom and atom stopper
   *)
   process Atom[i_a, c_a, a_con, a_ac, a_as](state: State_Atom) :
           exit(State_Atom) :=
      [not(Needs_Control(state))] ->
      (
         [Get_Run_Mode(state) eq rm_stopped] ->
         (
            c_a !start !Get_Id(state);
            (
               Update_Async_Inputs[a_con](Get_Async_Inputs(state)) >>
               (
               let next_state: State_Atom=
                                  Set_Run_Mode(state, rm_started)
                   in
                   a_ac !control !next_state;
                   exit(Set_Needs_Control(next_state, true))
               )
            )
            []
            i_a !get_sync_input !Get_Id(state) ?n: nat;
            i_a !rtn_get_sync_input !Get_Id(state)
                !elem(Get_Sync_Inputs(state), n);
            exit(state)
            []
            i_a !get_async_input !Get_Id(state) ?n: nat;
            i_a !rtn_get_async_input !Get_Id(state)
                !elem(Get_Async_Inputs(state), n);
            exit(state)
            []
            i_a !get_output !Get_Id(state) ?n: nat;
            i_a !rtn_get_output !Get_Id(state)
                !elem(Get_Outputs(state), n);
            exit(state)
         )
         []
         [Get_Run_Mode(state) eq rm_started] ->
         (
            c_a !begin_stopp !Get_Id(state);
            c_a !rtn_begin_stopp !Get_Id(state);
```

```
          exit(Set_Run_Mode(state, rm_stopping))
       )
       []
       [(Get_Run_Mode(state) eq rm_started) or
        (Get_Run_Mode(state) eq rm_stopping)] ->
       (
          a_con !input_set !Get_Id(state) ?i_id: Input_Id;
          (
          let next_state: State_Atom=Add_Set_Input(state, i_id) in
             a_ac !control !next_state;
             exit(Set_Needs_Control(next_state, true))
          )
          []
          a_con !output_done !Get_Id(state) ?o_id: Output_id;
          (
          let next_state: State_Atom=Add_Done_Output(state, o_id) in
             a_ac !control !next_state;
             exit(Set_Needs_Control(next_state, true))
          )
       )
       []
       [Get_Run_Mode(state) eq rm_stopping] ->
       (
          c_a !stopp !Get_Id(state);
          a_as !stopp_atom !state;
          exit(state)
          []
          a_as !stopped !Get_Id(state);
          exit(Set_Run_Mode(state, rm_stopped))
       )
    )
    []
    [Needs_Control(state)] ->
    (
       (
       a_ac !update_connections !Get_Id(state);
       Update_Connections[a_con](Get_Required_Inputs(state),
            Get_Async_Inputs(state), Get_Required_Outputs(state)) >>
          (
          let next_set_inputs: Input_Id_List=
                      RemoveList(Get_Required_Inputs(state),
                                 Get_Set_Inputs(state)),
              next_done_outputs: Output_Id_List=
                      RemoveList(Get_Required_Outputs(state),
                                 Get_Done_Outputs(state))
          in let
              next_state: State_Atom=Make_Atom(Get_Id(state),
                      Get_Sync_Inputs(state), next_set_inputs, <>,
                      Get_Async_Inputs(state), Get_Outputs(state),
                      next_done_outputs, <>, Get_Run_Mode(state),
                      as_waiting_for_connections,
                      Needs_Control(state))
             in
             a_ac !control !next_state;
             exit(next_state)
          )
       )
       []
       a_ac !wait !Get_Id(state);
       exit(Set_Needs_Control(state, false))
    )
    where
    process Update_Connections[c](i_ids: Input_Id_List,
               ai_ids: Input_Id_List, o_ids: Output_Id_List) : exit :=
       [not (isempty(i_ids))] ->
```

```
         c !free ?id: Input_Id [id iselem i_ids];
         Update_Connections[c](Remove(id, i_ids), ai_ids, o_ids)
      []
      [not (isempty(ai_ids))] ->
         c !update ?id: Input_Id [id iselem ai_ids];
         Update_Connections[c](i_ids, Remove(id, ai_ids), o_ids)
      []
      [not (isempty(o_ids))] ->
         c !set ?id: Output_Id [id iselem o_ids];
         Update_Connections[c](i_ids, ai_ids, Remove(id, o_ids))
      []
      [(isempty(i_ids)) and (isempty(ai_ids)) and (isempty(o_ids))] ->
         exit
    endproc

    process Update_Async_Inputs[c](ai_ids: Input_Id_List) : exit :=
       [not (isempty(ai_ids))] ->
          c !update ?id: Input_Id [id iselem ai_ids];
          Update_Async_Inputs[c](Remove(id, ai_ids))
       []
       [isempty(ai_ids)] ->
          exit
    endproc

endproc (* Atom *)

process Atom_Controller[a_ac, a_con] : noexit :=
(
   a_ac !control ?state: State_Atom;
   (
      [Get_Run_Mode(state) eq rm_started] ->
      (
         [Get_Atom_Step(state) eq as_waiting_for_connections] ->
         (
            a_ac !can_compute !Get_Id(state);
            a_ac !rtn_can_compute !Get_Id(state) ?cc: Bool;
            (
               [cc eq true] ->
                  a_ac !compute !Get_Id(state);
                  exit
               []
               [cc eq false] ->
                  a_ac !wait !Get_Id(state);
                  exit
            )
         )
         []
         [Get_Atom_Step(state) eq as_done_compute] ->
            a_ac !update_connections !Get_Id(state);
            exit
      )
      []
      [Get_Run_Mode(state) ne rm_started] ->
         a_ac !wait !Get_Id(state);
         exit
   )
) >> Atom_Controller[a_ac, a_con]
endproc (* Atom_Controller *)

process Atom_Stopper[a_as, a_con] : noexit :=
(
   a_as !stopp_atom ?state: State_Atom;
   (
      All_Connections_Ready[a_con](Get_Set_Inputs(state),
              Get_Done_Outputs(state)) >>
```

```
      a_as !stopped !Get_Id(state);
      exit
   )
) >> Atom_Stopper[a_as, a_con]

   where
   process All_Connections_Ready[c](i_ids: Input_Id_List,
           o_ids: Output_Id_List) : exit :=
      [not(isempty(i_ids))] ->
         c !is_ready ?id: Input_Id [id iselem i_ids];
         All_Connections_Ready[c](Remove(id, i_ids), o_ids)
      []
      [not(isempty(o_ids))] ->
         c !is_ready ?id: Output_Id [id iselem o_ids];
         All_Connections_Ready[c](i_ids, Remove(id, o_ids))
      []
      [isempty(i_ids) and isempty(o_ids)] ->
         exit
   endproc
endproc (* Atom_Stopper *)

process Function_Atom[i_a, c_a, a_con, a_ac, a_as]
                     (state: State_Atom) : exit(State_Atom) :=
(
   (Atom[i_a, c_a, a_con, a_ac, a_as](state) >>
    accept update_state: State_Atom in exit(update_state)
   )
   []
   [Needs_Control(state)] ->
   (
      a_ac !can_compute !Get_Id(state);
      a_ac !rtn_can_compute !Get_Id(state)
           !((Get_Set_Inputs(state) eq Get_Sync_Inputs(state)) and
            (Get_Done_Outputs(state) eq Get_Outputs(state)));
      exit(state)
   )
)
endproc

process Constants[i_a, c_a, a_con, a_ac, a_as](ids: Atom_Id_Set)
        : noexit :=
   i_a !create !constant ?id: Atom_Id
     [(id NotIn ids) and Is_Constant(id)];
   a_con !create !output ?o_id: Output_Id !id;
   a_con !create !async_input ?i_id: Input_Id !id;
   (
      Constant[i_a, c_a, a_con, a_ac, a_as]
              (Make_Atom(id, <>, <>, <>, <>, o_id. <>, o_id. <>, <>,
                         rm_stopped, as_waiting_for_connections, false))
      |||
      Constants[i_a, c_a, a_con, a_ac, a_as](Insert(id, ids))
   )

   where
   process Constant[i_a, c_a, a_con, a_ac, a_as](state: State_Atom) :
           noexit:=
   (
      (Function_Atom[i_a, c_a, a_con, a_ac, a_as](state) >>
       accept update_state: State_Atom in exit(update_state)
      )
      []
      [Needs_Control(state)] ->
      (
         a_ac !compute !Get_Id(state);
         a_con !read !elem(Get_Async_Inputs(state), 0) ?d: Data;
```

```
            a_con !write !elem(Get_Outputs(state), 0) !d;
            (
            let next_state: State_Atom=
                Set_Atom_Step(Set_Required_Outputs(state,
                      Get_Outputs(state)), as_done_compute)
               in
               a_ac !control !next_state;
               exit(next_state)
            )
         )
         []
         [(not(Needs_Control(state))) and
          (Get_Run_Mode(state) eq rm_stopped)] ->
         (
            c_a !reset !Get_Id(state);
            Reset_Connections[a_con](Get_Sync_Inputs(state),
                                     Get_Outputs(state))
            >>
            (
               let next_state: State_Atom=Make_Atom(Get_Id(state),
                   Get_Sync_Inputs(state), <>, <>, Get_Async_Inputs(state),
                   Get_Outputs(state), <>, <>, rm_stopped,
                   as_waiting_for_connections, false)
               in
               exit(next_state)
            )
         )
      ) >> accept update_state: State_Atom in
                     Constant[i_a, c_a, a_con, a_ac, a_as](update_state)
      endproc
   endproc (* Constants *)

   process Adders[i_a, c_a, a_con, a_ac, a_as](ids: Atom_Id_Set)
           : noexit :=
      i_a !create !adder ?id: Atom_Id [(id NotIn ids) and Is_Adder(id)];
      a_con !create !sync_input ?i_id1: Input_Id !id;
      a_con !create !sync_input ?i_id2: Input_Id !id;
      a_con !create !output ?o_id: Output_Id !id;
      (
         Adder[i_a, c_a, a_con, a_ac, a_as]
                 (Make_Atom(id, i_id1.(i_id2. <> ), <>, <>, <>, o_id. <>,
                            o_id. <>, <>, rm_stopped,
                            as_waiting_for_connections, false))
         |||
         Adders[i_a, c_a, a_con, a_ac, a_as](Insert(id, ids))
      )

      where
      process Adder[i_a, c_a, a_con, a_ac, a_as](state: State_Atom) :
              noexit :=
      (
         (Function_Atom[i_a, c_a, a_con, a_ac, a_as](state) >>
          accept update_state: State_Atom in exit(update_state)
         )
         []
         [Needs_Control(state)] ->
         (
            a_ac !compute !Get_Id(state); (* add inputs to output *)
            a_con !read !elem(Get_Sync_Inputs(state), 0) ?d1: Data;
            a_con !read !elem(Get_Sync_Inputs(state), succ(0)) ?d2: Data;
            a_con !write !elem(Get_Outputs(state), 0)
                  !Make_Data(Get(d1) + Get(d2));
            (
               let next_state: State_Atom=
                   Set_Atom_Step(
```

```
                      Set_Required_Inputs(
                         Set_Required_Outputs(state, Get_Outputs(state)),
                            Get_Sync_Inputs(state)), as_done_compute)
                  in
                  a_ac !control !next_state;
                  exit(next_state)
            )
         )
         []
         [(not(Needs_Control(state))) and
          (Get_Run_Mode(state) eq rm_stopped)] ->
         (
            c_a !reset !Get_Id(state);
            Reset_Connections[a_con](Get_Sync_Inputs(state),
                                     Get_Outputs(state))
            >>
            (
               let next_state: State_Atom=Make_Atom(Get_Id(state),
                   Get_Sync_Inputs(state), <>, <>, Get_Async_Inputs(state),
                   Get_Outputs(state), <>, <>, rm_stopped,
                   as_waiting_for_connections, false)
               in
               exit(next_state)
            )
         )
      ) >> accept update_state: State_Atom in
                     Adder[i_a, c_a, a_con, a_ac, a_as](update_state)
      endproc
   endproc (* Adders *)

   process Selectors[i_a, c_a, a_con, a_ac, a_as](ids: Atom_Id_Set)
           : noexit :=
      (* not implemented*)
      stop
   endproc

   process Distributors[i_a, c_a, a_con, a_ac, a_as](ids: Atom_Id_Set)
           : noexit :=
      (* not implemented*)
      stop
   endproc

   process Reset_Connections[c](i_ids: Input_Id_List,
                                o_ids: Output_Id_List) : exit :=
      [not(isempty(i_ids))] ->
         c !reset ?id: Input_Id [id iselem i_ids];
         Reset_Connections[c](Remove(id, i_ids), o_ids)
      []
      [not(isempty(o_ids))] ->
         c !reset ?id: Output_Id [id iselem o_ids];
         Reset_Connections[c](i_ids, Remove(id, o_ids))
      []
      [(isempty(i_ids)) and (isempty(o_ids))] ->
         exit
   endproc

endproc (* Atoms *)


(*
PROCESS  : Connections
BEHAVIOUR: process encapsulating the connections subsystem consisting of
the classes 'Input' and 'Output'
GATES    : 'i_con'...gate between interface scenario and connections
'a_con'...gate between atoms and connections
```

```
*)
process Connections[i_con, a_con] : noexit :=
   hide con in
      (
         Sync_Inputs[i_con, a_con, con]({} of Input_Id_Set)
         |||
         Async_Inputs[i_con, a_con, con]({} of Input_Id_Set)
      )
      |[con]|
      Outputs[i_con, a_con, con]({} of Output_Id_Set)
   where
   process Sync_Inputs[i_con, a_con, con](ids: Input_Id_Set) : noexit :=
      a_con !create !sync_input ?id: Input_Id ?a_id: Atom_Id
            [(id NotIn ids) and Is_Sync_Input(id)];
      (
         Sync_Input[i_con, a_con, con]
                    (Make_Sync_Input(id, Insert(a_id, {}),
                     {} of Output_Id_Set, cs_free, Make_Data(0)))
         |||
         Sync_Inputs[i_con, a_con, con](Insert(id, ids))
      )
      []
      i_con !create !sync_input ?id: Input_Id
            [(id NotIn ids) and Is_Sync_Input(id)];
      (
         Sync_Input[i_con, a_con, con]
                    (Make_Sync_Input(id, {} of Atom_Id_Set,
                     {} of Output_Id_Set, cs_free, Make_Data(0)))
         |||
         Sync_Inputs[i_con, a_con, con](Insert(id, ids))
      )

      where
      process Sync_Input[i_con, a_con, con](state: State_Sync_Input) :
              noexit :=
      (
         i_con !connect !Get_Id(state) ?o: Output_Id;
         exit(Connect(state, o))
         []
         i_con !disconnect !Get_Id(state);
         exit(Disconnect(state))
         []
         [Is_Part_Of_Atom(state)] ->
         (
            a_con !is_ready !Get_Id(State);
            exit(state)
            []
            a_con !reset !Get_Id(state);
            exit(Set_Con_Status(state, cs_free))
            []
            [Get_Con_Status(state) eq cs_free] ->
               con !set !Get_Id(state) ?d: Data;
               a_con !input_set ?a_id: Atom_Id !Get_Id(state)
                     [a_id IsIn Get_Atoms(state)];
               exit(Set_Con_Status(Set_Data(state, d), cs_set))
            []
            [Get_Con_Status(state) eq cs_set] ->
            (
               a_con !free !Get_Id(state);
               con !free ?id: Output_Id !Get_Id(state)
                   [id IsIn Get_Outputs(state)];
               exit(Set_Con_Status(state, cs_free))
               []
               a_con !read !Get_Id(state) !Get_Data(state);
               exit(state)
```

```
            )
         )
         []
         [not(Is_Part_Of_Atom(state))] ->
         (
            i_con !is_ready !Get_Id(State);
            exit(state)
            []
            i_con !reset !Get_Id(state);
            exit(Set_Con_Status(state, cs_free))
            []
            [Get_Con_Status(state) eq cs_free] ->
               con !set !Get_Id(state) ?d: Data;
               i_con !input_set !Get_Id(state);
               exit(Set_Con_Status(Set_Data(state, d), cs_set))
            []
            [Get_Con_Status(state) eq cs_set] ->
            (
               i_con !free !Get_Id(state);
               con !free ?id: Output_Id !Get_Id(state)
                   [id IsIn Get_Outputs(state)];
               exit(Set_Con_Status(state, cs_free))
               []
               i_con !read !Get_Id(state) !Get_Data(state);
               exit(state)
            )
         )
      ) >> accept update_state: State_Sync_Input in
                 Sync_Input[i_con, a_con, con](update_state)

      endproc (* Sync_Input *)

   endproc (* Sync_Inputs *)

   process Async_Inputs[i_con, a_con, con](ids: Input_Id_Set) : noexit :=
      a_con !create !async_input
            ?id: Input_Id ?a_id: Atom_Id
            [(id NotIn ids) and Is_Async_Input(id)];
      (
         Async_Input[i_con, a_con, con]
                    (Make_Async_Input(id, Insert(a_id, {}),
                     {} of Output_Id_Set, Make_Data(0), Make_Data(0)))
         |||
         Async_Inputs[i_con, a_con, con](Insert(id, ids))
      )
      []
      i_con !create !async_input
            ?id: Input_Id [(id NotIn ids) and (Is_Async_Input(id))];
      (
         Async_Input[i_con, a_con, con]
                    (Make_Async_Input(id, {} of Atom_Id_Set,
                     {} of Output_Id_Set, Make_Data(0), Make_Data(0)))
         |||
         Async_Inputs[i_con, a_con, con](Insert(id, ids))
      )

      where
      process Async_Input[i_con, a_con, con](state: State_Async_Input) :
              noexit :=
      (
         i_con !connect !Get_Id(state) ?o: Output_Id;
         exit(Connect(state, o))
         []
         i_con !disconnect !Get_Id(state);
         exit(Disconnect(state))
```

```
         []
         con !set !Get_Id(state) ?d: Data;
         con !free ?id: Output_Id !Get_Id(state)
             [id IsIn Get_Outputs(state)];
         exit(Set_External_Data(state, d))
         []
         [Is_Part_Of_Atom(state)] ->
         (
            a_con !update !Get_Id(state);
            exit(Set_Internal_Data(state, Get_External_Data(state)))
            []
            a_con !read !Get_Id(state) !Get_Internal_Data(state);
            exit(state)
         )
         []
         [not(Is_Part_Of_Atom(state))] ->
         (
            i_con !update !Get_Id(state);
            exit(Set_Internal_Data(state, Get_External_Data(state)))
            []
            i_con !read !Get_Id(state) !Get_Internal_Data(state);
            exit(state)
         )
      ) >> accept update_state: State_Async_Input in
                  Async_Input[i_con, a_con, con](update_state)
      endproc
   endproc (* Async_Inputs *)

   process Outputs[i_con, a_con, con](ids: Output_Id_Set) : noexit :=
      a_con !create !output
            ?id: Output_Id ?a_id: Atom_Id [id NotIn ids];
      (
         Output[i_con, a_con, con](Make_Output(id, Insert(a_id, {}),
                 {} of Input_Id_Set, {} of Input_Id_Set, cs_free,
                 Make_Data(0)))
         |||
         Outputs[i_con, a_con, con](Insert(id, ids))
      )
      []
      i_con !create !output
            ?id: Output_Id [id NotIn ids];
      (
         Output[i_con, a_con, con](Make_Output(id, {} of Atom_Id_Set,
                 {} of Input_Id_Set, {} of Input_Id_Set, cs_free,
                 Make_Data(0)))
         |||
         Outputs[i_con, a_con, con](Insert(id, ids))
      )

      where
      process Output[i_con, a_con, con](state: State_Output) : noexit :=
      (
         i_con !connect !Get_Id(state) ?i_id: Input_Id;
         exit(Connect(state, i_id))
         []
         i_con !disconnect !Get_Id(state) ?i_id: Input_Id;
         exit(Disconnect(state, i_id))
         []
         [Is_Part_Of_Atom(state)] ->
         (
            a_con !is_ready !Get_Id(state);
            exit(state)
            []
            a_con !reset !Get_Id(state);
            exit(Clear_Done_Inputs(Set_Con_Status(state, cs_free)))
```

```
            []
            [Get_Con_Status(state) eq cs_free] ->
            (
               a_con !write !Get_Id(state) ?d: Data;
               exit(Set_Data(state, d))
               []
               (
                  a_con !set !Get_Id(state);
                  Set_All_Inputs[con](Get_Inputs(state), Get_Data(state))
                  >>
                  a_con !output_done ?a_id: Atom_Id !Get_Id(state)
                        [a_id IsIn Get_Atoms(state)];
                  exit(Clear_Done_Inputs(state))
               )
            )
            []
            [Get_Con_Status(state) eq cs_set] ->
            (
               con !free !Get_Id(state) ?id: Input_Id;
               (
               let next_state: State_Output=Add_Done_Input(state, id)
                   in
                   [Get_Done_Inputs(state) eq Get_Inputs(state)] ->
                       a_con !output_done ?a_id: Atom_Id !Get_Id(state)
                             [a_id IsIn Get_Atoms(state)];
                       exit(Clear_Done_Inputs(
                               Set_Con_Status(next_state, cs_free)))
                   []
                   [Get_Done_Inputs(state) ne Get_Inputs(state)] ->
                       exit(next_state)
               )
            )
         )
         []
         [not(Is_Part_Of_Atom(state))] ->
         (
            i_con !is_ready !Get_Id(state);
            exit(state)
            []
            i_con !reset !Get_Id(state);
            exit(Clear_Done_Inputs(Set_Con_Status(state, cs_free)))
            []
            [Get_Con_Status(state) eq cs_free] ->
            (
               i_con !write !Get_Id(state) ?d: Data;
               exit(Set_Data(state, d))
               []
               (
                  i_con !set !Get_Id(state);
                  Set_All_Inputs[con](Get_Inputs(state), Get_Data(state))
                  >>
                  i_con !output_done ?a_id: Atom_Id !Get_Id(state)
                        [a_id IsIn Get_Atoms(state)];
                  exit(Clear_Done_Inputs(state))
               )
            )
            []
            [Get_Con_Status(state) eq cs_set] ->
            (
               con !free !Get_Id(state) ?id: Input_Id;
               (
               let next_state: State_Output=Add_Done_Input(state, id)
                   in
                   [Get_Done_Inputs(state) eq Get_Inputs(state)] ->
                      i_con !output_done ?a_id: Atom_Id !Get_Id(state)
```

```
                            [a_id IsIn Get_Atoms(state)];
                      exit(Clear_Done_Inputs(
                              Set_Con_Status(next_state, cs_free)))
                   []
                   [Get_Done_Inputs(state) ne Get_Inputs(state)] ->
                      exit(next_state)
               )
            )
         )
      ) >> accept update_state: State_Output in
                  Output[i_con, a_con, con](update_state)

         where
         process Set_All_Inputs[x](i_ids: Input_Id_Set, d: Data) : exit :=
            [i_ids ne {}] ->
               x !set ?id: Input_Id !d [id IsIn i_ids];
               Set_All_Inputs[x](Remove(id, i_ids), d)
            []
            [i_ids eq {}] ->
               exit
         endproc

      endproc (* Output *)

   endproc (* Outputs*)

endproc (* Connections *)

(*
PROCESS  : Test
BEHAVIOUR: process defining an interface scenario which can be used
to test the specification
GATES    : 'i_c'...gate between interface scenario and circuit
'i_a'...gate between interface scenario and atoms
'i_con'...gate between interface scenario and inputs/outputs
'i_t'...gate between the user and the interface scenario
(mainly for debugging)
*)
process Test[i_c, i_a, i_con, i_t] : noexit :=
   (* construct atoms and add them to the circuit *)
   i_a !create !constant ?con: Atom_Id;
   i_c !add_atom !id1 of Circuit_Id !con;
   i_a !create !adder ?a: Atom_Id;
   i_c !add_atom !id1 of Circuit_Id !a;

   (* construct free input *)
   i_con !create !output ?fo: Output_Id;
   i_con !create !sync_input ?fi: Input_Id;

   (* connect *)
   i_a !get_async_input !con !0;
   i_a !rtn_get_async_input ?ic: Input_Id;
   i_a !get_output !con !0;
   i_a !rtn_get_output !con ?oc: Output_Id;
   i_a !get_sync_input !a !0;
   i_a !rtn_get_sync_input !a ?ia1: Input_Id;
   i_a !get_sync_input !a !succ(0);
   i_a !rtn_get_sync_input !a ?ia2: Input_Id;
   i_a !get_output !a !0;
   i_a !rtn_get_output !a ?oa: Output_Id;
   i_con !connect !ic !fo;
   i_con !connect !fo !ic;
   i_con !connect !oc !ia1;
   i_con !connect !ia1 !oc;
   i_con !connect !oc !ia2;
```

```
    i_con !connect !ia2 !oc;
    i_con !connect !oa !fi;
    i_con !connect !fi !oa;

    (* start circuit and receive input *)
    i_c !start !id1 of Circuit_Id;
    i_con !input_set !fi;
    i_con !read !fi ?d1: Data;
    i_t !Get(d1);
    i_con !free !fi;

    (* stop and reset circuit *)
    i_c !stopp !id1 of Circuit_Id;
    i_c !rtn_stopp !id1 of Circuit_Id;
    i_c !reset !id1 of Circuit_Id;
    i_c !rtn_reset !id1 of Circuit_Id;
    stop
endproc

endspec
```