Formale Spezifikation und Synthese digitaler Schaltungen auf höheren Abstraktionsebenen

Christian Blumenröhr

Formale Spezifikation und Synthese digitaler Schaltungen auf höheren Abstraktionsebenen

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Christian Blumenröhr

aus Karlsruhe

Tag der mündlichen Prüfung:	21. Januar 2000
Erster Gutachter:	Prof. Dr.-Ing. D. Schmid
Zweiter Gutachter:	Prof. Dr. rer. nat. P. H. Schmitt

Für

Anne-Claire, Nicolas und Carolin

in Liebe

Vorwort

Die vorliegende Arbeit entstand in den Jahren 1995 - 1999 im Rahmen eines von der Deutschen Forschungsgemeinschaft (DFG) unterstützten Forschungsprojekts am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe.

An dieser Stelle möchte ich mich besonders bei Herrn Prof. Dr. Detlef Schmid bedanken für seine Unterstützung und konstruktive Betreuung, ohne die diese Arbeit nicht möglich gewesen wäre. Auch Herrn Prof. Dr. Peter Schmitt gilt mein herzlicher Dank für die Übernahme des Korreferates und die sorgfältige Begutachtung meiner Arbeit.

Des Weiteren möchte ich allen, die mich bei meiner Forschungstätigkeit unterstützt haben, meinen Dank aussprechen. Besonders zu erwähnen sind hierbei Herr Dr. Dirk Eisenbiegler und Herr Dr. Viktor Sabelfeld, bei denen ich mich für die vielen fruchtbaren Diskussionen und Ratschläge, aber auch für die angenehme Zusammenarbeit bedanken möchte. Auch Herrn Dr. Ramayya Kumar, Herrn Dipl.-Inform. Jörg Berdux und Herrn Dipl.-Inf. Kai Kapp gilt mein besonderer Dank für die sehr gute Zusammenarbeit. Für die orthographische Durchsicht der Arbeit möchte ich mich an dieser Stelle bei meinem Vater Dr. Friedrich Blumenröhr bedanken.

Der größte Dank gilt aber meiner Familie: meinen zwei kleinen Lieblingen Nicolas und Carolin, und vor allem meiner Frau Anne-Claire für ihre Liebe und Unterstützung, sowie für ihr Verständnis und ihre Rücksichtnahme in den heißen Phasen meiner Arbeit. Ebenso danke ich meinen Eltern für ihre (finanzielle) Unterstützung, ohne die mein Studium und meine Promotion nicht möglich gewesen wären.

Karlsruhe, im Januar 2000 *Christian Blumenröhr*

Inhaltsverzeichnis

Kapitel 1

Motivation

Im alltäglichen Sprachgebrauch bedeutet das Wort "Synthese" das Zusammenfügen einzelner Teile zu einem höheren Ganzen. Im Gegensatz dazu meint dieser Begriff im Bereich des Entwurfs digitaler Schaltungen die stufenweise Verfeinerung einer abstrakten in eine konkrete Schaltungsbeschreibung. Der Entwurf durchläuft dabei verschiedene Abstraktionsebenen mit möglichen Optimierungen innerhalb der Abstraktionsebenen. Nach der Synthese der Schaltung liegt eine konkrete Beschreibung vor, mit deren Hilfe die Schaltung auf einem Chip realisiert werden kann.

Die Synthese digitaler Systeme verschiebt sich zunehmend in Richtung höherer Abstraktionsebenen. Dies ist darin begründet, dass die Systeme immer komplexer werden. Damit wird aber auch der Prozess immer aufwendiger, sie zu entwerfen. Außerdem dringen digitale Systeme in verstärktem Maße in verschiedene Bereiche unseres Lebens ein. Von erheblicher Bedeutung ist ihr Einsatz bei sicherheitskritischen Anwendungen (Verkehrstechnik, Medizintechnik, ...). Darüber hinaus zeigt sich, dass die Entwicklung derartiger Systeme immer teurer wird und Fehler im Entwurf hohe Kosten nach sich ziehen. Aus diesen Gründen ist es umso notwendiger, dass digitale Systeme von Anfang an einwandfrei funktionieren.

Es lassen sich vier Fehlerquellen für nicht funktionierende Digitalschaltungen identifizieren: Spezifikationsfehler, Entwurfsfehler, Fertigungsfehler und Betriebsfehler. Die vorliegende Arbeit beschäftigt sich ausschließlich mit der Vermeidung von Entwurfsfehlern.

Für kleine Schaltungen kann die Synthese von Hand durchgeführt werden. Zumeist scheidet ein "Handentwurf" aber wegen der Komplexität der Schaltungen aus, sodass es heute die Regel ist, Programme einzusetzen, die eine automatische Synthese ermöglichen und die in der Lage sind, die Komplexität des Schaltungsentwurfs zu beherrschen. Der Schaltungsentwerfer interagiert mit diesen Synthesewerkzeugen; der Syntheseablauf und die produzierten Syntheseergebnisse sind von ihm aber nur noch schwer oder überhaupt nicht mehr zu übersehen.

Der Entwerfer muss darauf vertrauen können, dass die eingesetzten Synthesewerkzeuge korrekt arbeiten; eine automatisierte Synthese führt per definitionem nicht zu einem korrekten Syntheseergebnis, weil ein fehlerhaftes Syntheseprogramm in der Regel auch zu einem fehlerhaften Syntheseergebnis führt. Unter Korrekt-

heit wird dabei verstanden, dass das Syntheseergebnis (Implementierung) der Syntheseeingabe (Spezifikation) in einer bestimmten mathematischen und damit nachprüfbaren Weise genügt.

Moderne Synthesewerkzeuge sind hochgradig komplexe Softwareprogramme mit normalerweise einigen 100.000 Zeilen Programmcode, die meistens in einer imperativen Programmiersprache wie C geschrieben sind. Es werden sehr komplizierte Datenstrukturen für die Repräsentation und die Transformation der Schaltungsbeschreibungen verwendet. Gerade für die Synthese auf höheren Abstraktionsebenen kommen zudem raffinierte Algorithmen zum Einsatz, die den Entwurfsraum auf möglichst gute Lösungen untersuchen. Da eine formale Verifikation dieser Programme aus Gründen eines zu großen Aufwands scheitern muss, ist die Forderung nach fehlerfreien Syntheseprogrammen i.Allg. unerfüllbar.

Hinzu kommt, dass große Synthesewerkzeuge meist von mehreren Programmierern gemeinsam entwickelt werden. Dazu ist es erforderlich, dass Klarheit über die Schnittstellen der Programmmodule und die Datenstrukturen herrscht – eine Forderung, die nur allzu oft nicht erfüllt werden kann. Eine weitere Fehlerquelle stellen moderne Hardwarebeschreibungssprachen dar, die zur Spezifikation eingesetzt werden. Um solche Sprachen in die automatische Synthese einzubinden, müssen Programme zur Übersetzung dieser Hardwarebeschreibungssprache in das interne Format des Syntheseprogramms und umgekehrt entwickelt werden. Eine notwendige Voraussetzung dafür ist, die genaue Semantik der einzelnen Konstrukte dieser Hardwarebeschreibungssprachen zu kennen. Dies ist aber nicht so leicht, wie es den Anschein haben mag. Ein Paradebeispiel hierfür ist die Sprache VHDL [VHDL96], deren Semantik trotz massiven Einsatzes immer noch nicht vollständig festgelegt wurde bzw. oft unterschiedlich interpretiert wird, da nur eine umgangssprachliche Definition vorliegt. Aber auch wenn die Semantik der Hardwarebeschreibungssprache bekannt ist, können die Übersetzungsprogramme an sich fehlerhaft sein.

Als Konsequenz all dieser Punkte ergeben sich notwendigerweise Syntheseprogramme, die erst durch massiven praktischen Einsatz getestet und korrigiert werden können. Auch gut getestete Werkzeuge sind aber keine Garantie für Fehlerfreiheit. Dem Entwerfer bleibt also bisher nichts anderes übrig, als die Zuverlässigkeit des Syntheseprogramms in der Praxis zu überprüfen und sodann auf dessen Korrektheit zu vertrauen.

Dass Vertrauen zwar gut, Kontrolle aber besser ist, ist den meisten Schaltungsentwerfern bewusst. Aus diesem Grunde ist es heute üblich, zur Überprüfung eines Syntheseergebnisses eine vergleichende Simulation mit der Syntheseeingabe durchzuführen. Da aber die Größe der Schaltungen immer mehr zunimmt, ist eine vollständige Simulation in angemessener Zeit nur noch in seltenen Fällen möglich. Meistens kann der Entwerfer nur einige Bereiche, die er als kritisch einstuft, überprüfen. Das macht es lediglich möglich, Fehler aufzuspüren, aber nicht, einen Nachweis für die Korrektheit zu erbringen. Als Alternative zur Simulation drängen zurzeit Methoden der formalen Verifikation [Keut96] auf den Markt. Insbesondere Werkzeuge zur Modellprüfung [ClGL96] werden dabei eingesetzt. Diese Verfahren haben aber mehrere Nachteile.

Die Aufgabenstellung der Modellprüfung ist i.Allg. NP-vollständig. Zwei Schaltungsbeschreibungen werden dabei miteinander verglichen, und es muss der Beweis gefunden werden, dass diese in einer bestimmten mathematischen Relation stehen. Die Information, *wie* die Synthese durchgeführt wurde, steht in der Regel nicht mehr zur Verfügung, und der Beweis für die Korrektheit des Synthesevorgangs ist daher nur schwierig zu führen. Das hat zur Folge, dass gewöhnlich nur kleine Schaltungen mit einer kleinen Zustandsmenge automatisch verifiziert werden können. Obwohl die Forschung in diesem Bereich weit fortgeschritten ist, gibt es zahlreiche Klassen von Schaltungen, für die es keine effizienten Modellprüfungsverfahren gibt. Ein weiterer Nachteil der Verwendung von Modellprüfungsprogrammen ist, dass deren Implementierungen ebenfalls nicht erwiesenermaßen korrekt sind. Sie bestehen meistens aus vielen tausenden Zeilen Programmcode, der in der Regel nicht verifiziert ist bzw. nicht verifiziert werden kann.

Des Weiteren werden bei der Modellprüfung Temporallogiken eingesetzt, die zur Beschreibung von Schaltungen auf höheren Abstraktionsebenen nicht ausreichen. Dort müssen unentscheidbare Logiken eingesetzt werden, die eine automatische Verifikation nicht zulassen. Ein anderes Verfahren der formalen Verifikation, das Theorembeweisen, lässt sich zwar auch auf höheren Abstraktionsebenen anwenden, kann aber nicht automatisch durchgeführt werden. Der zeitliche Aufwand für einen interaktiven Beweis nach jedem Synthesedurchlauf ist schwer abzuschätzen.

1.1 Ziel der Arbeit

In Anbetracht dieser Situation verfolgt diese Arbeit das Ziel, eine andere Methode zu entwickeln, mit deren Hilfe man zu erwiesenermaßen korrekten Syntheseergebnissen gelangt. Statt die Synthese in konventioneller Weise durchzuführen und eine Simulation oder Verifikation nachzuschalten, soll das Syntheseergebnis innerhalb des Logikkalküls eines Theorembeweisers durch Anwendung elementarer mathematischer Regeln abgeleitet werden. Dieser Synthesevorgang wird mit dem Begriff "Formale Schaltungssynthese" bezeichnet.

Grundlage für diesen Ansatz ist der Theorembeweiser HOL (*higher order logic*) [GoMe93]. Der Kalkül von HOL umfasst fünf Axiome und acht Inferenz-Regeln. Auf diesem Kalkül baut die logische Argumentation dieser Arbeit auf. Alle in dieser Arbeit vorgestellten Schaltungstransformationen sind nichts anderes als mathematische Umformungen, die sich ausschließlich aus diesen Elementarschritten zusammensetzen. Die Software-Implementierung dieses Kalküls erstreckt sich auf wenige hundert Zeilen Code in der funktionalen Programmiersprache SML. Obwohl diese Software nicht formal verifiziert wurde, stellt sie doch eine ungleich vertrauenswürdigere Basis für die Synthese dar als ein herkömmliches Syntheseprogramm. Da alle Schaltungstransformationen aus dem Kalkül von HOL abgeleitet wurden, kann die Formale Synthese als sehr sicher angesehen werden. Insbesondere ändert sich durch dieses Vorgehen mit der Anzahl der Transformationen und daher mit der Größe des Syntheseprogramms nichts an dessen Korrektheit – sehr zum Unterschied von kon-

ventionellen Syntheseprogrammen.

Im Bereich der Formalen Schaltungssynthese existieren bislang lediglich Arbeiten für eine Synthese auf unteren Abstraktionsebenen (RT- und Gatterebene) sowie für reine Datenpfade auf algorithmischer Ebene. In dieser Arbeit soll ein neues Verfahren für die Formale Synthese allgemeiner μ-rekursiver Funktionen auf algorithmischer Ebene und für eine Synthese auf Systemebene entwickelt werden. Auf der algorithmischen Ebene werden Einprozessbeschreibungen behandelt, während auf der Systemebene Strukturen nebenläufiger Prozesse betrachtet werden. Die Schaltungssynthese basiert auf einer neuentwickelten funktionalen Hardwarebeschreibungssprache namens Gropius. Im Gegensatz zu herkömmlichen Hardwarebeschreibungssprachen wie etwa VHDL hat Gropius eine mathematisch exakte Semantik, da alle Konstrukte in der Prädikatenlogik höherer Ordnung definiert wurden.

1.2 Gliederung der Arbeit

Im anschließenden Kapitel folgt zunächst eine Einführung in die Logik höherer Ordnung und den in dieser Arbeit verwendeten Theorembeweiser HOL. Daran schließt sich eine Einführung in die Grundlagen der digitalen Schaltungssynthese an. Es wird eine Übersicht der Methoden zur Gewinnung korrekter Syntheseergebnisse vorgestellt und der Stand der Technik beleuchtet. In Kapitel 4 werden die Konzepte für die dieser Arbeit zugrunde liegende Hardwarebeschreibungssprache vorgestellt, bevor die Definition der Sprache für die algorithmische Ebene und die Systemebene in den Kapiteln 5 und 6 folgt. Die zwei nachfolgenden Kapitel behandeln die eigentliche Synthese. In Kapitel 7 wird die Synthese auf der algorithmischen Ebene und in Kapitel 8 die Synthese auf der Systemebene beschrieben. In Kapitel 9 folgen experimentelle Ergebnisse zur Laufzeiteffizienz des in dieser Arbeit vorgestellten Ansatzes und Kapitel 10 bringt eine Zusammenfassung der Arbeit.

Kapitel 2

Einführung in die Logik höherer Ordnung und den Theorembeweiser HOL

Die in dieser Arbeit verwendete Hardwarebeschreibungssprache Gropius wurde in einem Theorembeweiser namens HOL [GoMe93] definiert. HOL ist ein interaktiver Theorembeweiser für Prädikatenlogik höherer Ordnung, dessen Grundlagen in diesem Kapitel vorgestellt werden sollen.

Die Version der in HOL verwendeten Logik höherer Ordnung wurde von Mike Gordon an der University of Cambridge entwickelt [Gord85]. Sie vereinigt Konzepte des λ-Kalküls, der von Church entwickelt wurde, und der Typtheorie, die auf Whitehead und Russell zurückgeht. Der λ-Kalkül ist die einfachste algorithmische Sprache, um alle berechenbaren Probleme zu formulieren. Ein anderes Modell ist die Turing-Maschine. Während imperative Programmiersprachen wie Pascal oder C auf dem Konzept der Turing-Maschine basieren, bildet der λ-Kalkül die Grundlage für funktionale Programmiersprachen wie ML [Paul91] oder Miranda. So ist im HOL-System die Logik in die Sprache ML eingebettet. HOL ist also eine Version von ML, die um eine Sammlung von ML-Funktionen zum Beweisen von Theoremen der Logik höherer Ordnung ergänzt worden ist. In Abschnitt 2.7 wird dies näher erläutert. In dieser Arbeit wird die Version HOL90 des Theorembeweisers benutzt. Sie basiert auf Standard ML New Jersey von AT&T (SML/NJ).

Im Folgenden soll ein Überblick über die Logik höherer Ordnung gegeben werden, wie sie im HOL-System verwendet wird. Ausführlichere Beschreibungen finden sich beispielsweise in [Gord85, Melh93, Bare92, Goos97a]. Anschließend soll kurz die Funktionsweise von HOL erläutert werden.

2.1 Terme

Die erste und einfachste Stufe der mathematischen Logik [Kond83, Schö89] ist die *Aussagenlogik*. In ihr wird lediglich der Wahrheitswert einer Aussage betrachtet.

Eine Aussage wird als Term bezeichnet und kann entweder "wahr" oder "falsch" sein. Die einfachen Aussagen werden als unzerlegbar angesehen. Zusammengesetzte Aussagen ergeben sich durch Anwendung der klassischen Operatoren wie der Konjunktion, Disjunktion und Negation. Der Wahrheitswert einer zusammengesetzten Aussage hängt nur von den Wahrheitswerten der zugrunde liegenden Aussagen ab, nicht aber von deren Inhalt. Die *Prädikatenlogik* ist eine Erweiterung der Aussagenlogik, die auch die innere Struktur von Aussagen in die Betrachtung einbezieht. Was hinzukommt sind Funktions- und Prädikatenoperatoren. Quantoren sind dabei spezielle Funktionsoperatoren. In der *Prädikatenlogik der 1. Ordnung* werden als Erweiterung der Aussagenlogik Quantifizierungen über Variablen betrachtet. Die *Prädikatenlogik höherer Ordnung* erlaubt schließlich beliebige Funktions- und Prädikatenoperatoren. Insbesondere kann über beliebige Funktionen quantifiziert werden. In der Prädikatenlogik höherer Ordnung können Variablen einen beliebigen Typ, insbesondere den einer Funktion, annehmen. Man spricht dabei von *Variablen höherer Ordnung*. Darüber hinaus können sogar Funktionen andere Funktionen als Argumente annehmen und als Ergebnis eine Funktion liefern. Man spricht von *Funktionen höherer Ordnung* oder von *Funktionalen*.

2.1.1 Syntax

Die Syntax eines Terms in der Logik höherer Ordnung ergibt sich durch die folgende Backus-Naur-Form (BNF):

$$
\begin{array}{rcl}
\mathit{Konstante} & ::= & \text{spezieller Bezeichner} \\
\mathit{Variable} & ::= & \text{beliebiger Bezeichner} \\
\mathit{Term} & ::= & \mathit{Konstante} \;\mid \\
 & & \mathit{Variable} \;\mid \\
 & & \text{“(”}\;\; \mathit{Term}\;\; \mathit{Term}\;\; \text{“)”} \;\mid \\
 & & \text{“(}\;\lambda\text{”}\;\; \mathit{Variable}\;\; \text{“.”}\;\; \mathit{Term}\;\; \text{“)”}
\end{array}
\tag{2.1}
$$

Es gibt vier Arten, einen Term zu erzeugen: Ein Term kann entweder eine Konstante c oder eine Variable v sein. Eine *Funktionsanwendung* zweier Terme $(M\ N)$ bedeutet, dass die Funktion M (Operator) auf den Wert N (Operand) angewandt wird. Eine *Abstraktion* $(\lambda v.M)$ kennzeichnet schließlich eine Funktion mit Parameter v und Funktionsrumpf M. Sie beschreibt eine Abbildung $v \mapsto M[v]$. Terme, die nach der Syntaxregel (2.1) gebildet werden, heißen auch λ-*Ausdrücke*.

Kommt eine Variable v innerhalb des Funktionsrumpfes M einer λ-Abstraktion mit dem gleichen Parameter v $(\lambda v.M)$ vor, bezeichnet man sie als *gebunden* durch die Bindung λv. Im Gegensatz dazu bezeichnet man eine Variable v, die in einem Term M außerhalb eines Teilterms $(\lambda v.N)$ vorkommt, als *frei* in M. Im Folgenden Term

$$(\lambda x.f\ x)\ ((\lambda y.x + y)\ z)$$

ist die Variable x folglich im Funktionsrumpf des Operators $(\lambda x.f\ x)$ gebunden, während sie im Operanden $((\lambda y.x+y)\ z)$ frei vorkommt. Einen Term, dessen Varia-

blen alle gebunden sind, nennt man *geschlossen*. Sind $v_1, \ldots, v_n$ die in einem Term M frei vorkommenden Variablen, so schreibt man oft $M[v_1, \ldots, v_n]$.

Um die zahlreichen Klammern in λ-Ausdrücken zu vermeiden, gibt es folgende Konventionen:

- Funktionsanwendungen: Der Term $(M_1\ M_2\ \cdots\ M_n)$ bezeichnet den Term $((\cdots (M_1\ M_2)\ M_3)\ \cdots)\ M_n)$.
- Verschachtelte λ-Abstraktionen: Der Term $(\lambda v_1.(\lambda v_2.(\ldots(\lambda v_n.M)\ldots)))$ darf abgekürzt werden mit $(\lambda v_1 \ldots v_n.M)$.

Diese Schreibweisen werden vom "Parser" und vom "Pretty Printer" unterstützt. Der Parser übersetzt eingegebene Terme in die interne Darstellung des HOL-Systems, und der Pretty-Printer sorgt für den umgekehrten Weg, indem er für eine bessere Lesbarkeit der ausgegebenen Terme sorgt. Parser und Printer sind spezielle ML-Programme, die für die Interaktion zwischen dem Benutzer und HOL sorgen (siehe Abschnitt 2.7).

2.1.2 Umformungen im λ-Kalkül

Die Umformung eines λ-Ausdrucks wird *Konversion* oder *Reduktion* des Ausdrucks genannt. Seien $N_1, \ldots, N_n$ Terme und $v_1, \ldots, v_n$ verschiedene Variablen, dann bezeichnet die Schreibweise $M[N_1, \ldots, N_n/v_1, \ldots, v_n]$ das Ergebnis, wenn gleichzeitig jede Variable v_i an jeder Stelle, an der sie im Term M frei vorkommt, durch den entsprechenden Term N_i *substituiert* wird. Eine Substitution ist aber nur dann zulässig, wenn keine freie Variable in einem der Terme N_i durch das Ergebnis der Substitution gebunden wird. So ist es zum Beispiel nicht zulässig, in dem Term $(\lambda x.(f\ x)\ +\ y)$ die Variable y durch den Term $(x + z)$ zu substituieren. In einem solchen Fall kann nur nach Umbenennung der gebundenen Variablen x im Term $(\lambda x.(f\ x)\ +\ y)$ die Substitution durchgeführt werden. Die Umbenennung gebundener Variablen ist die erste von drei Grundregeln zur Konversion von λ-Ausdrücken. Die beiden anderen definieren die Funktionsanwendung:

α-Konversion: Jede λ-Abstraktion $(\lambda v.M)$ darf ersetzt werden durch $(\lambda v'.M[v'/v])$, falls die Substitution von v' für v zulässig ist.

β-Konversion: Eine Funktionsanwendung $((\lambda v.M)\ N)$, dessen Operator eine λ-Abstraktion ist, nennt man *β-Redex*. Er darf ersetzt werden durch den Ausdruck $M[N/v]$, falls die Substitution von N für v in M zulässig ist.

η-Konversion: Eine λ-Abstraktion $(\lambda v.M\ v)$ kann durch M ersetzt werden, falls v in M nicht frei vorkommt.

Wird ein Ausdruck M solange umgeformt, bis ein Term N entsteht, auf den keine β- oder η-Konversion mehr angewandt werden kann, bezeichnet man N als *Normalform* von M. Alle weiteren Terme, die sich durch α-Konversion aus N erzeugen lassen,

werden auch als Normalformen von M bezeichnet. Unabhängig von der Reihenfolge, in der die Konversionen angewandt werden, erhält man bis auf α-Konversion immer das gleiche Ergebnis. Man sagt, der λ-Kalkül ist *konfluent*.

2.2 Typen

Die Logik höherer Ordnung ist streng typisiert. Jeder logische Term hat einen fest zugeordneten Typ. Typen sind unbedingt erforderlich, um Inkonsistenzen in der Logik zu vermeiden. Man betrachte zum Beispiel das Paradoxon von Russell. Gegeben sei folgender Ausdruck:

$$\Omega = \lambda P.\neg(P\ P) \tag{2.2}$$

Würde man den Term Ω auf sich selbst anwenden und anschließend eine β-Konversion durchführen, so ergäbe sich:

$$\Omega\ \Omega = (\lambda P.\neg(P\ P))\ \Omega = \neg(\Omega\ \Omega)$$

Dieser Widerspruch ergibt sich nur dadurch, dass untypisierte Terme verwendet wurden. Bei einer Typisierung ließe sich die Funktion Ω rein syntaktisch gar nicht definieren, wie im Folgenden erläutert wird.

2.2.1 Syntax

Die Syntax von Typen in der Logik höherer Ordnung ist gegeben durch die folgende BNF:

$$\begin{array}{lcl} \textit{Typ} & ::= & \textit{Typkonstante} \mid \\ & & \textit{Typvariable} \mid \\ & & \text{“(”}\ \ \textit{Typ}\ \ \left\{\ \text{“,”}\ \ \textit{Typ}\ \right\}\ \ \text{“)”}\ \ \textit{Typoperator} \end{array} \tag{2.3}$$

Einen Typ, der aus einer Typkonstanten c oder einer Typvariablen v besteht, nennt man *atomaren* Typ. Wird ein Typ aus mehreren Teiltypen $(\sigma_1, \ldots, \sigma_n)$ und einem Typoperator *op* gebildet, spricht man von einem *zusammengesetzten* Typ.

Typkonstanten bezeichnen eine festgesetzte Wertemenge. So bezeichnet z.B. der Typ bool die Menge der Wahrheitswerte $\{\mathsf{T}, \mathsf{F}\}$. Ein variabler Typ dagegen ist ein Platzhalter für einen beliebigen Typ und kann durch jeden beliebigen Typ instantiiert werden. Variable Typen werden durch kleine griechische Buchstaben bezeichnet. Ein Typ, der einen variablen Typ enthält, wird als *polymorph* bezeichnet. Man kann von einer gewissen Allquantifizierung über Typen in der Logik höherer Ordnung sprechen, da ein Theorem, das einen polymorphen Typ σ enthält, auch für alle Instanzen von σ gültig ist. Zusammengesetzte Typen bezeichnen eine Menge, die aus den beteiligten Teiltypen konstruiert ist. Ein Beispiel für einen Typoperator ist der zweistellige Operator fun. Er ordnet zwei Typen σ_1 und σ_2 einen neuen Typ (σ_1, σ_2)fun zu, dessen Elemente die totalen Funktionen von σ_1 nach σ_2 sind. Um

Typausdrücke leichter lesbar zu machen, unterstützt der HOL-Parser die Abkürzung $\sigma_1 \to \sigma_2$ für $(\sigma_1, \sigma_2)\mathsf{fun}$.

Es gibt in HOL zwei grundlegende Typkonstanten, nämlich bool und ind. Der Typ ind steht für eine nicht näher spezifizierte Individuenmenge, die nichts anderes ist als eine unendliche Menge verschiedener Elemente. Mehr ist über ind nicht bekannt – ind ist nur partiell spezifiziert. Es gibt auch nur einen grundlegenden Typoperator: den zweistelligen Operator fun. Prinzipiell können mit bool, ind, fun und Typvariablen alle Typen beschrieben werden. Aus pragmatischen Gesichtspunkten ist es aber erstrebenswert, weitere Typkonstanten und -operatoren einzuführen. In Abschnitt 2.5 wird der Mechanismus vorgestellt, wie solche neuen Typen definiert werden.

2.2.2 Wohltypisierte Terme

Es sollen hier nur "wohltypisierte" Terme T mit Typ σ betrachtet werden. Sie werden bezeichnet als $(T : \sigma)$. Wohltypisiertheit ist gemäß dem syntaktischen Aufbau von Termen (2.1) wie folgt definiert:

- Jede Konstante hat einen festen *generischen* Typ. Ist der generische Typ σ einer Konstante polymorph, dann ist $c : \sigma'$ wohltypisiert für jede Instanz σ' von σ. Konstanten werden in Abschnitt 2.4 näher betrachtet.
- Variablen können jeden beliebigen Typ haben.
- Wenn zwei Terme $M_1 : \alpha \to \beta$ und $M_2 : \alpha$ wohltypisiert sind, dann ist auch die Funktionsanwendung $(M_1\ M_2) : \beta$ wohltypisiert.
- Wenn $v : \alpha$ eine Variable ist und $M : \beta$ ein wohltypisierter Term, dann stellt auch die Abstraktion $(\lambda v.M) : \alpha \to \beta$ einen wohltypisierten Term dar.

Damit wird auch verständlich, warum das Paradoxon von Russell, das auf Seite 8 beschrieben wurde, in HOL nicht auftreten kann. Da bei einer Funktionsanwendung Operator und Operand unterschiedliche Typen haben müssen, kann kein Term auf sich selbst angewandt werden. Der Ausdruck (2.2) kann also niemals wohltypisiert sein.

Wenn Terme aufgestellt werden, müssten prinzipiell für alle Teilterme die Typen spezifiziert werden. Das HOL-System verfügt aber über einen automatischen Typ-Inferenz-Mechanismus. Damit müssen solche Typen nicht explizit vom Anwender angegeben werden, die aufgrund des Kontextes bereits festgelegt sind. Somit ist es zum Beispiel nicht nötig, im Term $(f : \alpha \to \beta)\ x$ den Typ für die Variable x anzugeben, da dieser bereits mit α festgelegt ist.

2.3 Formeln, Sequenzen, Axiome, Theoreme

In der Logik höherer Ordnung bezeichnet man Terme vom Typ bool als *Formeln*. Formeln werden verwendet, um Sequenzen und Theoreme zu beschreiben.

Eine *Sequenz* wird geschrieben als $(, , P)$. Dabei ist , eine Menge von Formeln, genannt Prämissen, und P ist ebenfalls eine Formel, genannt Konklusion.

Axiome sind Sequenzen ohne Beweis, d.h. bei ihnen wird vorausgesetzt, dass sie wahr seien. Sie werden geschrieben als , $\vdash P$, oder, falls , leer ist, einfach als $\vdash P$.

Ein *Theorem* ist eine Sequenz, die entweder ein Axiom ist oder durch eine Inferenz-Regel von anderen Theoremen abgeleitet wird. Theoreme werden ebenfalls geschrieben als , $\vdash P$. Die Axiome und Inferenz-Regeln in HOL werden in Abschnitt 2.6 beschrieben.

2.4 Konstanten

2.4.1 Elementare Konstanten

In der Logik höherer Ordnung gibt es die folgenden drei elementaren Konstanten:

$$=: \alpha \to \alpha \to \mathsf{bool} \qquad \Rightarrow: \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \qquad \varepsilon : (\alpha \to \mathsf{bool}) \to \alpha$$

Die beiden Konstanten = und $\Rightarrow$ stehen für die beiden zweistelligen Relationen der Gleichheit und der Implikation. Diese Relationen sind in der Logik höherer Ordnung durch Funktionen höherer Ordnung beschrieben. Wenn sie auf einen Wert angewandt werden, ist das Resultat eine boolesche Funktion. So bezeichnet z.B. die Funktionsanwendung $(\Rightarrow P)$ eine Funktion vom Typ $\mathsf{bool} \to \mathsf{bool}$. Wird diese Funktion auf einen weiteren Term Q angewandt, ergibt sich die Aussage $(\Rightarrow\ P\ Q)$, P impliziere Q.

Die dritte Konstante repräsentiert den Hilbert'schen Auswahl-Operator. Die Semantik von ε kann informell wie folgt beschrieben werden: Wenn $(P : \sigma \to \mathsf{bool})$ ein Prädikat über die Werte vom Typ σ ist, dann bezeichnet die Funktionsanwendung $(\varepsilon\ P)$ einen beliebigen Wert des Typs σ, für den P wahr ist. Falls mehrere Werte das Prädikat P erfüllen, legt die Semantik nicht fest, welcher davon durch den Term $(\varepsilon\ P)$ spezifiziert wird. Falls kein Wert existiert, der P erfüllt, bezeichnet der Term $(\varepsilon\ P)$ einen unbekannten, aber festen Wert des Typs σ.

2.4.2 Konstantendefinition

Die drei soeben vorgestellten Konstanten stellen die Basis dar, um weitere Konstanten durch *Konstantendefinitionen* einzuführen. Damit eine Konstante verwendet werden kann, muss einfach ein neuer Konstantenname und ein Typ angegeben werden. Es gibt aber dann noch keine Theoreme, die dieser Konstanten eine Eigenschaft zuordnen. Bei einer Konstantendefinition wird deshalb eine Formel eingeführt, die eine Gleichung zwischen der neuen Konstante c und einem Term M aufstellt. Für den Term $M : \sigma$ gilt die Einschränkung, dass er geschlossen sein muß, dass er die Konstante c nicht enthalten darf und dass in M keine Typvariablen vorkommen dürfen, die nicht in σ vorkommen. Außerdem darf keine andere Konstante bereits

eingeführt sein, die denselben Namen wie c aufweist. Das resultierende Theorem

$$\vdash\ c\ =\ M$$

erweitert die Logik, indem die neue Konstante c als atomare Abkürzung des Terms M eingeführt wird. Bei dem Vorgang der Konstantendefinition handelt es sich um eine *konservative Erweiterung* der Logik. Dies bedeutet, dass für eine Formel P, die die neue Konstante c nicht enthält, das Theorem $\vdash P$ nur dann ein Theorem der erweiterten Logik ist, wenn es bereits ein Theorem der ursprünglichen Logik war. Insbesondere ist also das falsche Theorem $\vdash \mathsf{F}$ nur dann ein Theorem, wenn es ursprünglich schon enthalten war. Daher wird durch das Hinzufügen von Axiomen, die neue Konstanten definieren, keine Inkonsistenz in die Logik gebracht.

Eine erweiterte Form der Konstantendefinition sieht vor, dass auf der linken Seite der Formel neben einer Konstanten auch Variablen als Parameter vorkommen dürfen, in denen die freien Variablen des Terms M der rechten Seite enthalten sind:

$$\vdash\ c\ v_1\ \ldots v_n\ =\ M \quad \text{bzw.} \quad \vdash\ c\ (v_1, \ldots, v_n)\ =\ M$$

Diese erweiterte Form kann auf die ursprüngliche Form der Konstantendefinition zurückgeführt werden.

Einige grundlegende Konstanten, die in HOL, aufbauend auf den Konstanten $=$, $\Rightarrow$ und ε, durch Konstantendefinitionen eingeführt worden sind, sind in Tabelle 2.1 aufgeführt. Diese Konstanten sind Bestandteile der konventionellen mathematischen Logik. In der Logik der höheren Ordnung müssen sie aber nicht als elementare Konstanten eingeführt werden, sondern können formal so definiert werden, dass sie ihre gewohnten logischen Eigenschaften erhalten.

Auch hier sind wieder einige Besonderheiten bezüglich der Schreibweise zu beachten. Um logische Terme leichter lesbar zu machen, gilt auch hier die Regel, dass zweistellige Operatoren in Infix-Notation geschrieben werden. Aus $(\Rightarrow\ P\ Q)$ beispielsweise wird also $(P \Rightarrow Q)$. Konstanten wie $\forall, \exists$ und ε werden als *Binder* bezeichnet. Wenn sie auf eine λ-Abstraktion angewandt werden, darf der Bezeichner " λ " weggelassen werden, und bei ineinander geschachtelten Bindern dürfen die inneren Binder weggelassen werden. So wird etwa anstatt $\forall(\lambda x.\ \ldots)$ die Schreibweise $\forall x.\ \ldots$ verwendet, und $\forall x_1 x_2 x_3.\ \ldots$ ersetzt $\forall x_1.(\forall x_2.(\forall x_3.\ \ldots))$. Der Parser übersetzt die auf diese Weise abgekürzten Terme in ihre eigentliche Form. Zu jeder Konstanten, die in HOL durch eine Konstantendefinition eingeführt wird, wird festgelegt, ob sie Infix- oder Binder-Status haben soll oder nicht.

Theorem	**Typ**	**Beschreibung**
$\vdash \mathsf{T} = ((\lambda x.x) = (\lambda x.x))$	bool	*wahr*
$\vdash \forall = (\lambda P.P = (\lambda x.\mathsf{T}))$	$(\alpha \to \mathsf{bool}) \to \mathsf{bool}$	Allquantor
$\vdash \exists = \lambda P.P(\varepsilon P)$	$(\alpha \to \mathsf{bool}) \to \mathsf{bool}$	Existenzquantor
$\vdash \mathsf{F} = \forall b.b$	bool	*falsch*
$\vdash \neg = \forall b.(b \Rightarrow \mathsf{F})$	$\mathsf{bool} \to \mathsf{bool}$	Negation
$\vdash \wedge = \lambda b_1 b_2.\forall b.(b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$	$\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$	Konjunktion
$\vdash \vee = \lambda b_1 b_2.\forall b.$ $(b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)$	$\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$	Disjunktion
$\vdash \exists^1 = (\lambda P.\ \exists P) \wedge$ $(\forall xy.(Px) \wedge (Py) \Rightarrow (x = y))$	$(\alpha \to \mathsf{bool}) \to \mathsf{bool}$	eindeutiger Existenzquantor

Tabelle 2.1: Einige Konstantendefinitionen

2.4.3 Konstantenspezifikation

Die *Konstantenspezifikation* ist ein allgemeinerer Mechanismus als die Konstantendefinition. Die Formel muss die Konstante nicht in eindeutiger Weise beschreiben – es muss lediglich bewiesen werden, dass ein derartiger Wert existiert. Es wird ein Theorem

$$\vdash P\ [c] \tag{2.4}$$

eingeführt, das die Konstante c beschreibt. Das Prädikat P kann jedoch nicht in beliebiger Weise gewählt werden. Ein "ungünstig" gewähltes P könnte dazu führen, dass durch die Einführung des Theorems (2.4) die Konsistenz des Theorembeweisers verletzt wird. Das Prädikat P könnte so gewählt sein, dass $P\ [c]$ widersprüchlich ist, oder dass das Theorem (2.4) zu einer Inkonsistenz mit den bisher abgeleiteten Theoremen führt. Bei einer Konstantenspezifikation wird deshalb vorausgesetzt, dass das Theorem

$$\vdash \exists x.\ P\ [x] \tag{2.5}$$

bewiesen wurde. Aufgrund des Theorems (2.5) kann anschließend die Konstante c durch das Theorem

$$\vdash c\ =\ (\varepsilon x.\ P\ [x]) \tag{2.6}$$

per Konstantendefinition eingeführt werden. Zusammen mit dem Axiom über ε (siehe SELECT_AX in Tabelle 2.2 in Abschnitt 2.6) kann das gewünschte Theorem (2.4) abgeleitet werden.

2.4.4 Rekursive Definition

Bei einer Konstantendefinition $\vdash c = M$ darf die zu definierende Konstante c im Term M nicht vorkommen. Damit werden inkonsistente rekursive Definitionen wie $\vdash P = \neg P$ verhindert. Konstanten, die rekursiven Gleichungen genügen, können also nicht direkt über den im Abschnitt 2.4.2 beschriebenen Weg eingeführt werden, vielmehr wird ein Weg ähnlich dem der Konstantenspezifikation eingeschlagen.

Zuerst muss bewiesen werden, dass die angestrebte rekursive Gleichung überhaupt erfüllbar ist. Anschließend kann die Konstante mit Hilfe des Hilbert-Operators ε definiert werden. Soll beispielsweise eine Konstante mit

$$\vdash c = M[c] \tag{2.7}$$

eingeführt werden, so muss zunächst bewiesen werden, dass die Gleichung konsistent, also zumindest durch einen Wert erfüllbar ist. Dies kann erreicht werden, indem das Theorem

$$\vdash \exists x.x = M[x]$$

bewiesen wird.

2.5 Einführung neuer Typen

Es ist aus pragmatischen Gesichtspunkten leicht einsichtig, dass für einen praktischen Einsatz der Logik eine reichere Typensyntax notwendig ist, als sie in Abschnitt 2.2 vorgestellt wurde. In Abschnitt 2.5.1 soll daher ein Mechanismus vorgestellt werden, um die Logik durch weitere Typkonstanten und Typoperatoren zu erweitern. Typen werden dabei durch Typausdrücke beschrieben, wie sie bereits in der Logik vorhanden sind. In Abschnitt 2.5.2 wird anschließend ein Beispiel gezeigt.

2.5.1 Typendefinition

Bei einer Typdefinition wird ein neuer Typ σ_n eingeführt. Gleichzeitig wird ein Axiom eingeführt, das die Eigenschaft des neuen Typs beschreibt. Um diesen Typ zu beschreiben, muss zuerst in geeigneter Weise ein bereits vorhandener Typ σ ausgewählt werden. Anschließend wird ein Prädikat $P : \sigma \to \mathsf{bool}$ festgelegt, das eine nichtleere Teilmenge $\{x|P\ x\}$ von σ beschreibt. Das ist erforderlich, da alle Typen in HOL nichtleere Mengen bezeichnen müssen. Der Beweis, dass diese Menge über mindestens ein Element verfügt, muss explizit erbracht werden. Es muss also zunächst das Theorem

$$\vdash\ \exists x.\, P\ x \tag{2.8}$$

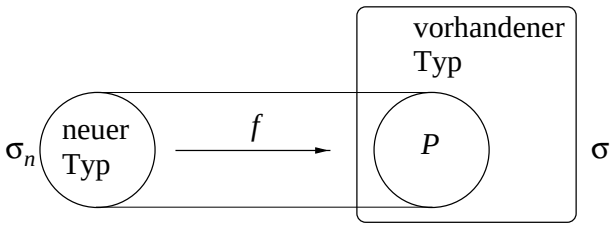


Abbildung 2.1: Typendefinition

bewiesen werden. Es wird nun eine Isomorphiebeziehung zwischen der Wertemenge des neueinzuführenden Typs σ_n und $\{x|P\ x\}$ aufgebaut. Diese Beziehung wird durch ein Theorem beschrieben, das die Aussage macht, dass es eine injektive Funktion $f : \sigma_n \to \sigma$ gibt, deren Wertemenge gerade $\{x|P\ x\}$ ist. Durch diese Funktion f liegt eine bijektive Abbildung von der Wertemenge von σ_n nach $\{x|P\ x\}$ vor. Man beachte aber, dass f in der Regel keine bijektive Funktion von σ_n nach σ ist, da über die Werte des Typs σ, für die P nicht erfüllt ist, keine Aussage gemacht wird. In Abbildung 2.1 ist dieser Zusammenhang veranschaulicht.

Aufgrund der Isomorphie, die durch f hergestellt wird, kann gezeigt werden, dass die Menge, die durch den Typ σ_n beschrieben wird, dieselben Eigenschaften hat wie die Untermenge von σ, die durch das Prädikat P definiert ist. Der neue Typ σ_n ist also durch den alten Typ σ definiert, und seine Eigenschaften werden durch die Wahl von P festgelegt. Insbesondere "erbt" der neue Typ auch die Konstanten und Funktionen des alten Typs.

Neue Typkonstanten werden auf diese Weise ebenso eingeführt wie neue Typoperatoren. Der Unterschied besteht darin, dass für die Definition einer Typkonstanten der Typ σ keine Typvariablen enthalten darf. Im anderen Fall wird ein neuer n-stelliger Typoperator $(\alpha_1, \ldots, \alpha_n)op$ erzeugt. Die Stelligkeit des Operators ergibt sich dabei aus der Anzahl aller Typvariablen in σ.

Zur Unterstützung der Typendefinition gibt es im HOL-System folgende Konstantendefinition:

$$\begin{array}{l} \vdash\ \mathsf{TYPE\_DEFINITION}\ = \\ \quad \lambda P\ f.\ (\forall y_1 y_2.\ (f\ y_1\ =\ f\ y_2) \Rightarrow (y_1\ =\ y_2)) \\ \qquad (\forall x.\ P\ x\ =\ (\exists y.\ x\ =\ f\ y)) \end{array} \tag{2.9}$$

($\mathsf{TYPE\_DEFINITION}\ P\ f$) drückt aus, dass die Funktion f injektiv ist und dass die Menge $\{x|P\ x\}$ die Bildmenge von f ist. Bei der Typendefinition wird das folgende Axiom in die Logik eingeführt, das die in Theorem (2.9) vorkommende Bezeichnung verwendet:

$$\vdash\ \exists f.\ \mathsf{TYPE\_DEFINITION}\ P\ f \tag{2.10}$$

Voraussetzung für die Definition ist, dass das Prädikat P geschlossen ist, dass das Existenz-Theorem (2.8) bewiesen ist und σ_n nicht der Name einer bereits definierten

Typkonstante ist. Darüber hinaus muss bei der Definition einer Typkonstanten gelten, dass sowohl σ als auch P über keine Typvariablen verfügen. Die Typendefinition ist damit abgeschlossen. Die Konsistenz wird durch die gleichzeitige Einführung des Typs σ_n und des Axioms (2.10) nicht verletzt.

Neben dem hier vorgestellten Verfahren der Typendefinition gibt es noch das allgemeinere Verfahren der Typenspezifikation. Ähnlich wie bei der Konstantenspezifikation, muss auch bei der Typenspezifikation ein Beweis erbracht werden, durch den abgesichert wird, dass die Typenspezifikation tatsächlich konsistenzerhaltend ist. Hier soll aber auf dieses Verfahren nicht näher eingegangen werden. Es ist in [HOL93] ausführlich beschrieben.

Der Mechanismus der Typendefinition ist zwar aus logischer Sicht ausreichend, für eine praktische Anwendbarkeit aber nicht befriedigend. Man benötigt vielmehr Theoreme, die den neuen Typ in abstrakter Weise beschreiben. Es müssen also mehrere Theoreme bewiesen werden, die die wesentlichen Eigenschaften des Typs beschreiben, ohne auf dessen Definition Bezug zu nehmen. Im nächsten Abschnitt soll daher an einem Beispiel gezeigt werden, wie ein neuer Typ definiert wird und wie nützliche Theoreme abgeleitet werden können, um ihn in abstrakter Weise zu beschreiben.

2.5.2 Beispiel einer Typendefinition

Der Typoperator prod wird für die Beschreibung des kartesischen Produkts zweier Terme benötigt. Dabei bezeichnen Werte des Typs $(\sigma_1, \sigma_2)\mathsf{prod}$ geordnete Paare, deren erste Komponente den Typ σ_1 und deren zweite Komponente den Typ σ_2 hat. Die Schreibweise $(\sigma_1 \times \sigma_2)$ wird dabei vom Parser in $(\sigma_1, \sigma_2)\mathsf{prod}$ umgewandelt.

Es soll nun kurz erläutert werden, wie dieser Typ in HOL definiert ist. Als bereits vorhandener Typ, mit dessen Hilfe der neue Typ beschrieben werden soll, wird zunächst $\sigma_1 \to \sigma_2 \to \mathsf{bool}$ ausgewählt. Anschließend werden zwei Konstantendefinitionen gemacht:

$$\vdash \forall x_1 x_2.\, \mathsf{MK\_PAIR}\, x_1\, x_2 = (\lambda v_1 v_2.\, (v_1 = x_1) \wedge (v_2 = x_2)) \tag{2.11}$$

$$\vdash \forall x.\, \mathsf{IS\_PAIR}\, x = (\exists x_1 x_2.\, x = \mathsf{MK\_PAIR}\, x_1\, x_2) \tag{2.12}$$

Dabei erhält MK_PAIR den generischen Typ $\sigma_1 \to \sigma_2 \to (\sigma_1 \to \sigma_2 \to \mathsf{bool})$ und IS_PAIR ist eine Konstante des Typs $(\sigma_1 \to \sigma_2 \to \mathsf{bool}) \to \mathsf{bool}$. Das Prädikat IS_PAIR wird nun verwendet, um eine nichtleere Teilmenge zu beschreiben. Dieser Zusammenhang wird in Abbildung 2.2 verdeutlicht. Ausgehend von diesen Definitionen kann das Existenz-Theorem

$$\vdash \exists x.\, \mathsf{IS\_PAIR}\, x \tag{2.13}$$

leicht bewiesen werden, da das Theorem $\vdash \mathsf{IS\_PAIR}\ (\mathsf{MK\_PAIR}\ x_1\ x_2)$ aus der Definition (2.12) folgt. Nun kann der Typoperator prod definiert werden. Es ergibt sich dabei das folgende Axiom:

$$\vdash \exists f.\, \mathsf{TYPE\_DEFINITION\ IS\_PAIR}\ f \tag{2.14}$$

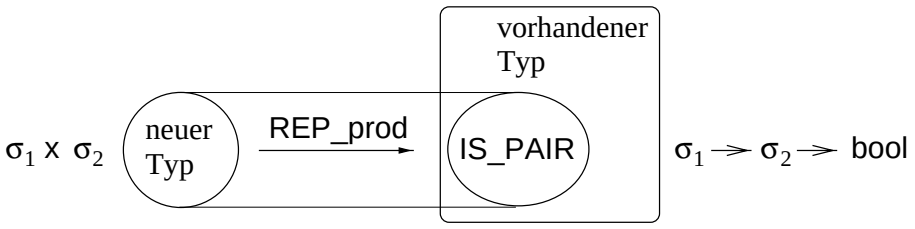


Abbildung 2.2: Typdefinition für kartesisches Produkt

Anschließend wird eine Konstante REP_prod definiert, die gerade die bijektive Abbildung von der Wertemenge des neuen Typs zu der Teilmenge $\{x \mid \mathsf{IS\_PAIR}\ x\}$ des vorhandenen Typs realisiert (siehe Abbildung 2.2).

$$\begin{aligned}\vdash \mathsf{REP\_prod} \;=\; \varepsilon f.\ (\forall y_1 y_2.\ (f\ y_1 \;=\; f\ y_2) \;\Rightarrow\; (y_1 \;=\; y_2)) \wedge \\ (\forall x.\ \mathsf{IS\_PAIR}\ x \;=\; (\exists y.\ x \;=\; f\ y))\end{aligned}$$

Anschließend werden die folgenden Konstanten definiert:

$$\vdash\ \forall x_1 x_2.\ \mathsf{COMMA}\ x_1\ x_2 \;=\; (\varepsilon y.\ \mathsf{REP\_prod}\ y \;=\; \mathsf{MK\_PAIR}\ x_1\ x_2) \tag{2.15}$$

$$\vdash\ \forall y.\ \mathsf{FST}\ y \;=\; (\varepsilon x_1.\ \exists x_2.\ \mathsf{MK\_PAIR}\ x_1\ x_2 \;=\; \mathsf{REP\_prod}\ y) \tag{2.16}$$

$$\vdash\ \forall y.\ \mathsf{SND}\ y \;=\; (\varepsilon x_2.\ \exists x_1.\ \mathsf{MK\_PAIR}\ x_1\ x_2 \;=\; \mathsf{REP\_prod}\ y) \tag{2.17}$$

Zur besseren Lesbarkeit wird dabei die Konstante COMMA durch die Konstante ',' ersetzt, die in Infix-Notation verwendet wird. Mit Hilfe der Definitionen (2.15), (2.16), (2.17) und des eingeführten Axioms (2.14) können nun leicht die folgenden Theoreme abgeleitet werden.

$$\vdash\ \forall x_1\ x_2.\ \mathsf{FST}\ (x_1, x_2) \;=\; x_1 \tag{2.18}$$

$$\vdash\ \forall x_1\ x_2.\ \mathsf{SND}\ (x_1, x_2) \;=\; x_2 \tag{2.19}$$

$$\vdash\ \forall x.\ (\mathsf{FST}\ x, \mathsf{SND}\ x) \;=\; x \tag{2.20}$$

$$\vdash\ \forall x_1\ x_2\ a\ b.\ ((x_1, x_2) \;=\; (a, b)) \;=\; ((x_1 \;=\; a) \wedge (x_2 \;=\; b)) \tag{2.21}$$

Diese vier Theoreme beschreiben die wesentlichen Eigenschaften des neuen Typs. Für die weitere Argumentation über Paare in der Logik sind nur noch diese vier Theoreme wesentlich, und es spielt keine Rolle mehr, wie der Operator prod definiert wurde.

2.6 Der Kalkül von HOL

Der Kalkül von HOL basiert auf fünf Axiomen und acht Inferenz-Regeln. Axiome (siehe Abschnitt 2.3) sind per definitionem allgemeingültig. Die fünf Axiome sind in Tabelle 2.2 aufgeführt. Das Axiom ETA_AX beschreibt die bereits in Abschnitt 2.1 erwähnte η-Konversion. SELECT_AX ist das Axiom über den Hilbert-Operator

BOOL_CASES_AX	$\vdash \forall b.\, (b = \mathsf{T}) \vee (b = \mathsf{F})$
IMP_ANTISYM_AX	$\vdash \forall b_1\, b_2.\, (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$
ETA_AX	$\vdash \forall f.\, (\lambda v.f\, v) = f$
SELECT_AX	$\vdash \forall P.\, P\, x \;\Rightarrow\; P(\varepsilon\, P)$
INFINITY_AX	$\vdash \exists f.\, (\forall x\, y.(f\, x = f\, y) \Rightarrow (x = y)) \wedge (\neg\, \forall x.\exists y.x = f\, y)$

Tabelle 2.2: Axiome in HOL

ε und INFINITY_AX dient zur Charakterisierung des elementaren Grundtyps ind. Es behauptet, dass es eine Funktion $f : \mathsf{ind} \to \mathsf{ind}$ gibt, die zwar injektiv, aber nicht surjektiv ist. Daraus folgt, dass es unendlich viele Werte des Typs ind gibt. Weitere Axiome ließen sich prinzipiell in beliebiger Weise einführen. Es bestünde dann aber die Gefahr, dass die Konsistenz des Kalküls verlorenginge.

Mit Hilfe von Inferenz-Regeln (kurz: Regeln) können aus bereits abgeleiteten oder axiomatisch eingeführten Theoremen neue Theoreme erzeugt werden. Die Schreibweise für Regeln lautet:

$$\frac{,_1 \vdash P_1, \ldots, ,_n \vdash P_n}{, \vdash P}$$

Regeln können andere Theoreme voraussetzen. Solche vorausgesetzten Theoreme werden bei der obigen Schreibweise gleichsam als "Zähler" über der horizontalen Linie aufgeführt. Das abgeleitete Theorem befindet sich unter der Linie (im "Nenner"). Die acht Regeln des HOL-Systems sind in Tabelle 2.3 aufgelistet.

Die Regel BETA_CONV beschreibt die β-Konversion des λ-Kalküls, die bereits in Abschnitt 2.1 vorgestellt wurde. Die β-Konversion ist dabei so definiert, dass sie nur durchführbar ist, wenn durch die Substitution keine freie Variable gebunden wird. Ansonsten ist eine β-Konversion nicht möglich. Um ein solches Scheitern der Konversion zu umgehen, wird im HOL-System bei Bedarf durch eine Variablenumbenennung (α-Konversion) die β-Konversion möglich gemacht. Beispielsweise wird für den Term $(\lambda x.(\lambda y.x + y))\, y$ die folgende β-Konversion durchgeführt:

$$(\lambda x.(\lambda y.x + y))\, y \quad \xrightarrow{\alpha} \quad (\lambda x.(\lambda y'.x + y'))\, y \quad \xrightarrow{\beta} \quad \lambda y'.y + y'$$

Wendet man die Regel BETA_CONV auf den Term an, erhält man also das Theorem $\vdash\ (\lambda x.(\lambda y.x + y))\, y\ =\ \lambda y'.y + y'$. Dasselbe Prinzip greift bei Anwendung der Regel SUBST. Gebundene Variablen im Prädikat P werden zuerst umbenannt, damit keine freien Variablen in den Termen N_i' fälschlicherweise in $P[N_1', \ldots, N_n']$ gebunden werden. Bei der Regel ABS ist zu beachten, dass die Variable v in der Prämisse , nicht frei vorkommen darf. Für INST_TYPE gilt, dass die Typvariablen α_i in der Prämisse , nicht vorkommen dürfen. Eine weitere Gefahr bestünde, wenn

ASSUME	$\dfrac{}{P \vdash P}$
REFL	$\dfrac{}{\vdash M = M}$
BETA_CONV	$\dfrac{}{\vdash (\lambda v.\, N)\ M\ =\ N\ [M/v]}$
SUBST	$\dfrac{,_1 \vdash N_1 = N_1' \ \cdots\ ,_n \vdash N_n = N_n' \qquad , \vdash P[N_1, \ldots, N_n]}{,_1 \cup \cdots \cup ,_n \cup , \ \vdash P[N_1', \ldots, N_n']}$
ABS	$\dfrac{, \ \vdash M = N}{, \ \vdash (\lambda v.\, M) = (\lambda v.\, N)}$
INST_TYPE	$\dfrac{, \ \vdash P}{, \ \vdash P[\sigma_1, \ldots, \sigma_n \,/\, \alpha_1, \ldots, \alpha_n]}$
DISCH	$\dfrac{, \ \vdash P}{, \ -\ \{Q\} \vdash Q \Rightarrow P}$
MP	$\dfrac{,_1 \vdash P \Rightarrow Q \qquad ,_2 \vdash P}{,_1 \cup ,_2 \vdash Q}$

Tabelle 2.3: Inferenz-Regeln in HOL

zwei verschiedene Variablen in P, die sich nicht im Namen, sondern nur im Typ unterscheiden, durch die Typensubstitution gleichgesetzt würden. Das wird aber in HOL automatisch verhindert, indem solche Variablen umbenannt werden.

2.6.1 Einige nützliche Konstanten in HOL

Für eine bessere Lesbarkeit der Terme wurden im HOL-System die beiden Konstanten UNCURRY und LET eingeführt, die vom Parser und Pretty-Printer unterstützt werden. Die Konstante UNCURRY hat den generischen Typ $(\alpha \to \beta \to \gamma) \to ((\alpha \times \beta) \to \gamma)$ und dient der Beschreibung gepaarter λ-Abstraktionen. Dabei wird etwa der Term

$$\lambda(v_1, v_2).M$$

durch den Parser in die interne Darstellung UNCURRY $(\lambda v_1.\ (\lambda v_2.M))$ umgewandelt. v_1 und v_2 sind dabei Variablen oder Tupel von Variablen (sog. Variablenstrukturen).

Die Konstante LET verfügt über den generischen Typ $(\alpha \to \beta) \to \alpha \to \beta$ und wird verwendet, um let-Terme zu repräsentieren. let-Terme werden genutzt, um β-Redizes $(\lambda v.M)N$ leichter lesbar zu machen. Insbesondere tief geschachtelte β-

Redizes, wie sie in dieser Arbeit vorkommen, sind kaum noch zu überblicken, da die Variable v und der ihr zugeordnete Term N räumlich weit auseinander stehen. let-Terme sind daher lediglich eine andere Schreibweise für β-Redizes, bei der der Term N direkt hinter der ihm zugehörigen Variablen v steht. Der folgende let-Term ist definitionsgemäß äquivalent dem β-Redex $(\lambda v.M)N$:

$$\mathsf{let}\ v\ =\ N\ \mathsf{in}\ M$$

Dieser let-Term wird durch den Parser in den Term (LET $(\lambda v.M)$ N) umgewandelt. Die Schreibweise mit let-Termen ist nicht beschränkt auf β-Redizes, deren Operator eine gewöhnliche λ-Abstraktion ist. Auch gepaarte λ-Abstraktionen, die mit Hilfe von UNCURRY erzeugt wurden, können verwendet werden. An die Stelle der Variablen v tritt dann eine beliebige Variablenstruktur.

Eine weitere Konstante, die in der Arbeit eine große Rolle spielt, ist COND. Sie wird für Fallunterscheidungen (if-then-else) verwendet und verfügt über den generischen Typen $\mathsf{bool} \to \alpha \to \alpha \to \alpha$. Auch hier wird mit Hilfe des Parsers eine anschaulichere Notation angeboten. Der Term COND P M N bedeutet "if P then M else N" und wird umgewandelt in

$$P\ \Rightarrow\ M \mid N$$

2.7 Theorembeweisen in HOL

HOL wurde ursprünglich entwickelt, um über die Korrektheit von Hardware zu argumentieren. Die in den vorigen Abschnitten vorgestellte Logik, die im System verwendet wird, ist aber völlig allgemein gehalten, sodass HOL für alle Gebiete eingesetzt werden kann, die mit Logik höherer Ordnung beschrieben werden können. HOL ist eine Weiterentwicklung des LCF Systems, das ursprünglich in Edinburgh konzipiert wurde. Die Logik von HOL wird in der streng-typisierten funktionalen Programmiersprache ML repräsentiert, wodurch die Sicherheit des Theorembeweisens garantiert wird. Terme und Theoreme der Logik werden dabei mit abstrakten Datentypen in ML beschrieben. Die Interaktion mit dem Theorembeweiser findet statt, indem Prozeduren in ML ausgeführt werden, die diese Datentypen benutzen. HOL ist also eine Version von ML, die um eine Sammlung von ML-Funktionen zum Beweisen von Theoremen der Logik höherer Ordnung ergänzt worden ist. Dieses Prinzip ist in Abbildung 2.3 dargestellt.

Abbildung 2.3: Interaktion mit HOL

Da HOL auf einer allgemein einsetzbaren Programmiersprache fußt, kann der Benutzer beliebig komplexe Programme schreiben, um Beweisstrategien zu entwickeln. Aufgrund der Art, in der die Logik in ML eingebettet ist, ist garantiert, dass

solche benutzerdefinierten Beweisstrategien nur logisch einwandfreie Beweisschritte durchführen.

2.7.1 Repräsentation in ML

Die Sicherheit des Theorembeweisens entsteht durch die rigorose Typdisziplin in ML. Theoreme werden in ML durch einen geschützten abstrakten Datentypen thm repräsentiert. Man kann keine Theoreme erzeugen, indem man einfach welche eingibt. Nur gewisse Werte erhalten anfangs den Typ thm. Diese Werte sind die in Tabelle 2.2 auf Seite 17 vorgestellten Axiome der Logik höherer Ordnung. Ansonsten gibt es nur eine Möglichkeit, um Theoreme zu erzeugen. Es werden gewisse ML-Funktionen zur Verfügung gestellt, die Theoreme als Argumente übernehmen und neue Theoreme ergeben. Diese Funktionen entsprechen den in Tabelle 2.3 aufgeführten elementaren Inferenz-Regeln. Der Typüberprüfer in ML sorgt dafür, dass nur solche Werte den Typ thm erhalten, die dadurch erzeugt wurden, dass diese Funktionen entweder auf einen Wert, der ein Axiom repräsentiert, oder auf einen früher erzeugten Wert vom Typ thm angewandt werden. Daraus folgt, dass jedes Theorem in HOL, ausgehend von den Axiomen, mit Hilfe der Inferenz-Regeln erzeugt werden muss. In HOL können daher keine "unrichtigen" Theoreme erzeugt werden.

Neben den elementaren Inferenz-Regeln gibt es eine Reihe abgeleiteter Inferenz-Regeln in HOL. Dies sind ML-Prozeduren, die häufig verwendete Sequenzen der elementaren Regeln durchführen. Damit kann der Benutzer von langwierigen Beweisschritten entlastet werden. Der Code für solche abgeleiteten Regeln in ML kann dabei hinreichend komplex sein. Da immer nur elementare Inferenz-Regeln aufgerufen werden, ist die Sicherheit solcher Beweisschritte garantiert. Unter diesen abgeleiteten Regeln gibt es auch abgeleitete Definitionsregeln. Das sind ML-Prozeduren, die automatisch Beweise durchführen, um kompliziertere Definitionen von den in den Abschnitten 2.4.2 und 2.5.1 vorgestellten Definitionsformen abzuleiten. Dazu gehören rekursive Typendefinitionen, rekursive Konstantendefinitionen über diese Typen und gewisse Arten der Induktionsdefinition [Melh89, Melh92].

2.7.2 Interaktive Beweise

Um die Gültigkeit einer Formel zu beweisen, muss sie mit Hilfe des HOL-Kalküls abgeleitet werden. HOL stellt dazu zwei Verfahren zur Verfügung: *Vorwärtsbeweis* und *Rückwärtsbeweis*. Beim vorwärtsgerichteten Beweisverfahren werden, angefangen bei den Axiomen, sukzessiv weitere Theoreme abgeleitet und dadurch die Menge der bewiesenen Theoreme so lange vergrößert, bis man auch das zu beweisende Theorem erhalten hat. Der Benutzer gibt an, welche Inferenz-Regel angewandt werden soll. Dies kann interaktiv geschehen oder dadurch, dass ein ML-Programm geschrieben wird, das die Regelaufrufe übernimmt. Bei dieser Beweistechnik ergibt sich das Problem, dass man die genauen Details des erforderlichen Beweises zu Anfang kaum

kennt und schwer abschätzen kann, ob die Theoreme, die man in den Zwischenschritten abgeleitet hat, in die richtige Richtung führen.

Beim Rückwärtsbeweis, der auch *zielorientierter Beweis* genannt wird, ist der Ausgangspunkt das zu beweisende Theorem. Dabei wird mit Hilfe spezieller ML-Funktionen, sogenannter *Taktiken*, das Beweisziel in immer einfachere Teilziele zerlegt, bis die Teilziele direkt mit bereits abgeleiteten Theoremen bewiesen werden können. Neben der Zerlegung in Teilziele erzeugt eine Taktik auch einen Vorwärtsbeweis, welcher benutzt wird, um das ursprüngliche Ziel zu beweisen, nachdem die Teilziele bewiesen wurden. Dies ist notwendig, da alle Theoreme letztendlich über einen Vorwärtsbeweis abgeleitet werden müssen. Auch hier entscheidet der Benutzer interaktiv, welche Taktik angewandt werden soll. Das Problem bei dieser Beweistechnik besteht darin, dass man durch Anwenden falscher Taktiken in eine Sackgasse geraten kann, indem man auf widersprüchliche Teilziele stößt, die nicht abgeleitet werden können. In diesem Fall müssen Taktikanwendungen aufgehoben, und ein neuer Beweisweg gesucht werden. Dass weder das vorwärts- noch das rückwärtsgerichtete Beweisen zu einem Algorithmus führen kann, der die Gültigkeit beliebiger Formeln automatisch ableiten kann, ergibt sich bereits aus der Unentscheidbarkeit der Prädikatenlogik höherer Ordnung. Generell kann man aber sagen, dass das rückwärtsgerichtete Verfahren dem Benutzer eher entgegenkommt, da der erforderliche Beweisweg während der Zerlegung in Teilziele besser erahnt werden kann.

Kapitel 3

Grundlagen der Schaltungssynthese und Stand der Technik

In diesem Kapitel sollen zunächst die Grundlagen der digitalen Schaltungssynthese vorgestellt werden. Anschließend soll der Korrektheitsbegriff definiert werden, der in dieser Arbeit verwendet wird. In Abschnitt 3.3 werden verschiedene Methoden vorgestellt, mit denen eine korrekte Schaltungssynthese erreicht werden kann, bevor in Abschnitt 3.4 ein Überblick über den Stand der Technik gegeben wird.

3.1 Schaltungssynthese

Ausgangspunkt des Entwurfs digitaler Schaltungen ist eine abstrakte Spezifikation, die aussagt, *was* die zu realisierende Schaltung zu leisten hat. Diese Spezifikation kann sich aus mehreren Teilen zusammensetzen. Sie beschreibt zum einen den funktionalen Zusammenhang zwischen Ein- und Ausgabe, den die Schaltung realisieren soll. Darüber hinaus können auch Randbedingungen vorgegeben werden, wie etwa die gewünschte Geschwindigkeit der Berechnung, die maximal zu verbrauchende Fläche auf einem Chip oder der maximale Leistungsverbrauch.

Diese Spezifikation wird im Laufe des Entwurfs immer mehr verfeinert, bis schließlich ein Schaltungslayout vorliegt, aus dem ein Chip gefertigt werden kann. Der Entwurf unterteilt sich in mehrere Syntheseschritte, die alle eine Transformation von einer Eingangsbeschreibung auf eine Ausgangsbeschreibung verwirklichen. In der Begriffswelt der Schaltungssynthese wird bei jedem Syntheseschritt eine Spezifikation in eine Implementierung überführt. Diese Implementierung wird anschließend als Spezifikation für den nächsten Syntheseschritt verwendet. Man unterscheidet zwei Arten von Syntheseschritten. *Verfeinerungsschritte* betreffen den Übergang von einer Abstraktionsebene zu einer anderen. Dabei wird der Grad der Abstraktion einer Beschreibung reduziert. Im Gegensatz dazu werden *Optimierungsschritte* innerhalb einer Abstraktionsebene durchgeführt.

In der Schaltungssynthese unterscheidet man normalerweise zwischen fünf verschiedenen Abstraktionsebenen. Die unterste Ebene ist die Layoutebene, bei der eine Platzierung der Objekte und die Verdrahtung der Verbindungen stattfindet. Eine gute Platzierung ist wichtig, da die Verdrahtung zwischen zwei weit entfernt platzierten Objekten sich als ungünstig erweist. Sie würde zusätzliche Fläche benötigen, eine größere Verzögerung und einen erhöhten Leistungsverbrauch bedeuten. Die nächsthöhere Abstraktionsebene ist die Gatterebene, bei der Schaltungen aus Strukturen von elementaren logischen Gattern und Speicherkomponenten beschrieben werden, die über Signalleitungen kommunizieren. Auf der Gatterebene finden Optimierungen statt, um die Anzahl der Gatter und die Länge des kritischen Pfades zu optimieren. Der längste Pfad durch eine kombinatorische Struktur wird als kritischer Pfad bezeichnet. Durch ihn wird die maximal mögliche Taktfrequenz bestimmt. Neben booleschen Optimierungen wird auch eine Technologieabbildung verwirklicht, bei der die Komponenten durch Zellen realisiert werden, die in Form einer Bibliothek gegeben sind. Über der Gatterebene ist die Register-Transfer (RT) Ebene angesiedelt. Hier werden Schaltungen aus Strukturen abstrakter Komponenten wie Funktionseinheiten und Register beschrieben. Die Signalleitungen verfügen ebenfalls über abstrakte Datentypen. Aufgabe der RT-Synthese ist es, eine Schaltungsbeschreibung mit abstrakten Datentypen auf eine Struktur auf der Gatterebene abzubilden. Dazu notwendige Schritte sind die Kodierung von Signalen und Zustandsvariablen. Weitere Syntheseschritte auf der RT-Ebene sind die Zustandsminimierung, Retiming, Reencoding, kombinatorische Optimierungen, etc. Die nächsthöhere Ebene ist die algorithmische Ebene, auf der Schaltungen allein durch die Angabe ihres Verhaltens mittels eines Algorithmus' beschrieben werden. Über der algorithmischen Ebene schließlich befindet sich die Systemebene, die zur Beschreibung nicht nur einzelner Prozesse, sondern ganzer Systeme, die sich aus verschiedenen Prozessen zusammensetzen, verwendet wird. Insbesondere werden auf dieser Ebene auch Systeme betrachtet, die sich aus Hardware- und Software-Komponenten zusammensetzen (HW/SW-Codesign, embedded systems).

Die in dieser Arbeit behandelten Abstraktionsebenen sind die RT-Ebene, die algorithmische Ebene und die Systemebene. Dabei startet die Formale Synthese mit einer Spezifikation auf der algorithmischen bzw. auf der Systemebene, und es wird jeweils eine Implementierung auf der RT-Ebene abgeleitet. Diese Syntheseprozesse werden als High-Level bzw. Systemebenensynthese bezeichnet. Die dabei auftretenden Syntheseschritte werden im Folgenden näher erläutert. Ein Ansatz zur Formalen Synthese auf den unteren Abstraktionsebenen, der die in der vorliegenden Arbeit beschriebene Synthese weiterführt, wird in [Eise99] vorgestellt.

3.1.1 Aufgaben der Synthese auf höheren Abstraktionsebenen

Die High-Level Synthese [GDWL92, Mich94] erzeugt aus einer algorithmischen Schaltungsbeschreibung eine Struktur auf der RT-Ebene. Die Synthese lässt sich im Wesentlichen in vier Teilprobleme unterteilen. In Klammern sind jeweils die

englischen Fachbegriffe aus der Literatur angegeben:

- Zeitliche Einordnung der Operationen (scheduling)
- Bereitstellung von Funktionseinheiten zur Implementierung der Operationen und von Registern zur Speicherung der Zwischenergebnisse (allocation)
- Zuordnung von konkreten Instanzen der bereitgestellten Komponenten an die Operationen und Hilfsvariablen (binding)
- Schnittstellensynthese (interface synthesis)

Zusätzlich kann innerhalb der algorithmischen Ebene vor der Synthese eine Optimierung der Schaltungsbeschreibung durchgeführt werden. Die drei erstgenannten Teilprobleme sind weitestgehend technologieunabhängig und werden unter Berücksichtigung der beiden Optimierungsziele einer hohen Ausführungsgeschwindigkeit und eines geringen Hardware- bzw. Flächenbedarfs durchgeführt. Oft wird eine minimale Ausführungsgeschwindigkeit in Form einer Taktzahl und/oder ein maximaler Flächenbedarf als Randbedingung für die Synthese vorgegeben.

Während der zeitlichen Einordnung werden alle Operationen der algorithmischen Beschreibung einem Kontrollschritt zugewiesen. Diese Kontrollschritte entsprechen den Takten auf der RT-Ebene. Um die Ausführung auf der RT-Ebene möglichst schnell zu machen, sollten möglichst wenige Kontrollschritte eingeführt und somit den Kontrollschritten jeweils möglichst viele Operationen zugewiesen werden. Auf der anderen Seite erhöht sich dadurch der Hardwareaufwand, denn für jede Operation eines Kontrollschrittes muss eine eigene Funktionseinheit bereitgestellt werden. Weiterhin erfordert eine große Anzahl von Funktionseinheiten in den Kontrollschritten die Bereitstellung einer großen Anzahl von Speicherbausteinen (Registern), in denen die Zwischenergebnisse zwischen zwei Kontrollschritten abgelegt werden müssen.

Die Aufgaben der zeitlichen Einordnung, der Bereitstellung und der Zuweisung von Komponenten hängen eng zusammen. Aufgrund der Komplexität dieser Aufgaben setzt man oft Heuristiken zur Entwurfsraumuntersuchung ein, da die Berechnung einer optimalen Lösung in angemessener Zeit oft nicht möglich ist. Führt man als ersten Schritt die Bereitstellung durch, um den Flächenbedarf zu optimieren, muss ein Ressourcen-beschränktes Verfahren zur zeitlichen Einordnung angewandt werden. Kommt es bei dieser zeitlichen Einordnung zur Verletzung etwaiger zeitlicher Randbedingungen, muss die Bereitstellung und anschließend die Einordnung wiederholt werden. Ein typischer Vertreter hierfür ist das Verfahren des "List-based Scheduling". Dabei wird für jede Operation fest vorgegeben, wie viele Instanzen zu ihrer Realisierung bereitgestellt werden sollen. Kann eine Operation aufgrund der Datenabhängigkeiten in einem bestimmten Kontrollschritt ausgeführt werden, ist aber bereits die Anzahl der Instanzen erschöpft, muss diese Operation einem anderen Kontrollschritt zugewiesen werden. Eine höhere Parallelität und eine damit verbundene höhere Ausführungsgeschwindigkeit wird somit einem geringeren

Flächenbedarf untergeordnet. Werden primär zeitliche Randbedingungen vorgegeben, wird ein zeitbeschränktes Einordnungsverfahren verwendet. Typische Vertreter sind der sog. ASAP- (as soon as possible) bzw. der ALAP-Algorithmus (as late as possible). Hier werden die Operationen so früh bzw. so spät wie möglich eingeteilt. Bei diesen zwei Verfahren wird keine Rücksicht auf den resultierenden Hardwareverbrauch genommen. Das Verfahren des "Force-Directed-Scheduling" [PaKn89] hingegen versucht, die Operationen gleichmäßig auf die Kontrollschritte zu verteilen, um die Hardware-Kosten ebenfalls zu minimieren.

Als letzte Aufgabe muss die Schnittstellensynthese durchgeführt werden. In ihr wird aus dem in den vorherigen Schritten erzeugten Zwischenergebnis eine strukturelle Implementierung auf der RT-Ebene erzeugt. Dabei müssen eine Kontrolleinheit und eine Kommunikationsstruktur aufgebaut werden. Die Kommunikationsstruktur ist erforderlich, um die Daten von den Funktionseinheiten zu den Registern und wieder zurück transportieren zu können. Der Aufwand dafür wird stark durch das Zuordnen (Binding) von Registern und Funktionseinheiten bestimmt, da eine schlechte Zuordnung unnötigen Transport verursachen kann. Wird während des Scheduling das sogenannte Chaining – mehrere Operationen werden *nacheinander* in einem Kontrollschritt ausgeführt – zugelassen, muss auch noch eine Kommunikationsstruktur zwischen den zugeordneten Funktionseinheiten realisiert werden.

Die Systemebenensynthese [GVNG94, Mich96, ELLS97] erzeugt aus einer Systembeschreibung, die sich aus mehreren Prozessen zusammensetzt, eine Struktur auf der RT-Ebene. Für die Systemebenensynthese lassen sich drei grundlegende Teilprobleme definieren:

- Systempartitionierung und -zusammenfassung
- Synthese der Prozesse
- Kommunikationssynthese

Durch die Systempartitionierung wird das System-Verhalten auf nebenläufig arbeitende Prozesse verteilt. Es ist aber auch möglich, Prozesse zusammenzufassen. Partitionierung und Zusammenfassung hängen davon ab, ob von einer Einprozessbeschreibung gestartet wird oder ob bereits eine Mehrprozessbeschreibung vorliegt. Das Zusammenfassen von Prozessen hat den Vorteil, dass während der Synthese des neuen Prozesses über die Grenzen der ursprünglichen Prozesse hinweg Optimierungen durchgeführt werden können. Man kann somit zu einer kostengünstigeren Implementierung kommen, die weniger Fläche benötigt, bzw. eine kürzere Abarbeitungszeit hat. Dies wird dadurch noch verstärkt, dass durch die Zusammenfassung der Kommunikationsaufwand zwischen den beteiligten Prozessen beseitigt wird. Die Partitionierung eines Prozesses in mehrere kleinere Prozesse hat dagegen den Vorteil, einen großen, komplexen Prozess in mehrere kleine Prozesse zu zerlegen, um somit die nachfolgende Synthese zu vereinfachen. Darüber hinaus können auf diese Weise Prozesse erzeugt werden, die bereits in einer vorherigen Synthese realisiert wurden und daher wiederverwendet werden können. Des Weiteren ist eine Partitionierung gerade im Hinblick des HW/SW-Codesigns interessant, da ein Prozess in

solche Teile zerlegt werden kann, die in Hardware realisiert werden, und solche, die als Software-Programm ablaufen.

Die Prozesse der sich ergebenden Mehrprozessbeschreibung können durch die bereits vorgestellten Schritte der High-Level Synthese anschließend auf die RT-Ebene überführt werden. Stark damit verknüpft ist die Kommunikationssynthese, welche die für die Kommunikation der Prozesse erforderliche Verbindungsstruktur generiert. Dabei werden gewöhnlich abstrakte Kommunikationskanäle eingeführt, über welche die Prozesse via I/O-Operationen miteinander kommunizieren. Die entsprechenden Kanäle und Schnittstellen müssen dann synthetisiert werden.

3.2 Korrektheit von Schaltungen

Unter der Korrektheit von Schaltungen kann zweierlei verstanden werden: Spezifikationskorrektheit und Implementierungskorrektheit (siehe [Keut96]). Diese Begriffe sollen im Folgenden näher betrachtet werden.

3.2.1 Spezifikationskorrektheit

Beim Begriff der Spezifikationskorrektheit geht es um die Frage, ob die Beschreibung der Schaltung in einer Hardwarebeschreibungssprache eine Spezifikation der Schaltung impliziert. Dieses Problem teilt sich in zwei Teile auf: Zuerst muss eine formale Spezifikation der Schaltung erstellt werden; anschließend muss das Modell in der Hardwarebeschreibungssprache gegenüber dieser Spezifikation verifiziert werden.

Für die Überprüfung der Spezifikationskorrektheit eignen sich Verfahren der Modellprüfung [ClGL96]. Damit können Eigenschaften wie Sicherheit, Lebendigkeit oder Fairness überprüft werden. In dieser Arbeit wird die Spezifikationskorrektheit nicht behandelt, vielmehr wird davon ausgegangen, dass die Spezifikationen korrekt sind.

3.2.2 Implementierungskorrektheit

Die Frage, ob eine erzeugte Implementierung der Spezifikation genügt, die den Ausgangspunkt der Synthese dargestellt hat, führt zum Begriff der Implementierungskorrektheit. Hier wird also die Korrektheit des Syntheseprozesses untersucht. Die vorliegende Arbeit beschäftigt sich ausschließlich mit der Implementierungskorrektheit. Im Weiteren soll unter Korrektheit verstanden werden, dass Implementierung und Spezifikation konsistent sind. Wie bereits in Abschnitt 3.1 erwähnt, wird die durch einen Syntheseschritt gewonnene Implementierung als Spezifikation für den nächsten Syntheseschritt verwendet. Für die zwei Arten von Syntheseschritten, die in Abschnitt 3.1 eingeführt wurden, ergeben sich dabei verschiedene Aussagen. Die Korrektheit eines Verfeinerungsschrittes erfordert den Nachweis eines mathematischen Theorems $\vdash \quad Imp \quad \Rightarrow \quad Spec$, das aussagt, dass die Implementierung die

Spezifikation impliziert. Im Gegensatz dazu muss für einen Optimierungsschritt ein Theorem der Form $\vdash Imp = Spec$ gefunden werden.

Da eine Spezifikation neben dem funktionalen Aspekt auch Randbedingungen wie den Flächenbedarf oder den Zeitverbrauch einschließen kann, müsste der Nachweis der Implementierungskorrektheit all diese Facetten umfassen. Die vorliegende Arbeit soll sich aber auf den Nachweis der Korrektheit des funktionalen Zusammenhangs beschränken, der durch die Spezifikation vorgegeben wird. Dies hat zwei Gründe: Zum einen ist die Überprüfung auf funktionale Korrektheit die bei weitem schwierigste Aufgabe. Zum anderen ist es weder notwendig noch sinnvoll, die Einhaltung von Randbedingungen wie etwa die eines maximalen Flächenbedarfs durch einen mathematischen Beweis zu überprüfen. Das lässt sich einfacher durch nichtformale Methoden erreichen.

Neben der Suche nach einem mathematischen Beweis kann die Implementierungskorrektheit auch durch nichtformale Methoden wie etwa durch Simulation überprüft werden. Wie aber bereits in Kapitel 1 erwähnt, muss eine vollständige Simulation bei der Größe der heutigen Schaltungen in den allermeisten Fällen scheitern. Im Folgenden sollen daher formale Methoden vorgestellt werden, die zur Sicherstellung des *funktionalen* Aspektes der Implementierungskorrektheit verwendet werden können.

3.3 Formale Methoden zur Sicherstellung der Korrektheit

Eine Analyse des Entwurfsprozesses ergibt, dass seine Korrektheit in verschiedenen Phasen garantiert werden kann: *vor*, *während* und *nach* dem Entwurf [KBES96]. Daher können die Ansätze zur Sicherstellung der Korrektheit als *Präsynthese-Verifikation*, *Formale Synthese* und *Postsynthese-Verifikation* klassifiziert werden.

3.3.1 Präsynthese-Verifikation

Unter Präsynthese-Verifikation versteht man dabei den Beweis der Korrektheit des Entwurfsprozesses an sich. Ein auf diese Weise überprüftes Syntheseprogramm liefert immer ein bezüglich der Eingabe korrektes Syntheseergebnis. Dies wird dadurch erreicht, dass das Syntheseprogramm mit Mitteln der Software-Verifikation überprüft wird. Der Korrektheitsbeweis wird dabei ein einziges Mal geführt. Da die gesamte logische Argumentation während der einmaligen Überprüfung des Programms stattfindet und somit der Anwender von der Überprüfung völlig verschont bleibt, wäre diese Methode für den Anwender an sich die optimale. Da aber die Software-Verifikation speziell bei sehr großen Programmen, wie sie Syntheseprogramme gewöhnlich darstellen, sehr kompliziert, wenn nicht zurzeit sogar unmöglich ist, gibt es erst wenige Arbeiten in diesem Bereich.

3.3.2 Postsynthese-Verifikation

Die Postsynthese-Verifikation [Gupt92] ist die heute am meisten eingesetzte formale Methode. Hier wird zunächst ein Syntheseschritt in konventioneller Weise durchgeführt. Anschließend wird die Korrektheit des Syntheseergebnisses überprüft. Ein Vorteil dieser Methode liegt darin, dass sie völlig unabhängig vom Syntheseschritt ist. Es können daher allgemeine Verfahren eingesetzt werden. Dies ist aber gleichzeitig auch ein Nachteil, da nur die Syntheseeingabe und -ausgabe vorliegen, und die Information, wie die Synthese durchgeführt worden ist, nicht länger zur Verfügung steht. Dies führt dazu, dass die Verifikation sehr zeitaufwendig wird und nur eine exzessive Fallunterscheidung zum Ziel führt.

Eine automatische Verifikation lässt sich nur auf unteren Abstraktionsebenen mit Verfahren der Modellprüfung und Tautologieprüfung erreichen. Auf höheren Abstraktionsebenen ist aufgrund der Komplexität eine automatische Verifikation i.Allg. unmöglich. Hier müssen Theorembeweiser eingesetzt werden, die vom Anwender profunde Logikkenntnisse und Erfahrung im Umgang mit diesem Werkzeug verlangen. Um Verfahren der Postsynthese-Verifikation effizienter bzw. auf höheren Abstraktionsebenen überhaupt realisierbar zu machen, muss die interne Steuerinformation des Syntheseprozesses zugänglich gemacht werden. Ferner werden dazu oft nur hinreichende Kriterien für die Korrektheit formuliert und überprüft, anstatt einen durchgängigen Beweis zu führen.

Ein weiteres Problem der Postsynthese-Verifikation liegt darin, dass die für die Synthese oft verwendeten Hardwarebeschreibungssprachen formal schwer zu fassen sind, für die Verifikation aber in eine logische Beschreibung überführt werden müssen.

3.3.3 Formale Synthese

Bei der Formalen Synthese geht es darum, das Syntheseergebnis innerhalb eines allgemeinen Logikkalküls eines Theorembeweisers abzuleiten. In der konventionellen Synthese können beliebige Datenstrukturen zur Repräsentation von Schaltungsbeschreibungen herangezogen werden und es gibt keine Beschränkungen hinsichtlich der Transformationen auf diesen Datenstrukturen. Im Gegensatz dazu werden in der Formalen Synthese Schaltungen durch logische Terme und Formeln beschrieben und es sind nur korrektheitserhaltende, logische Transformationen, die auf allgemeine mathematische Regeln zurückgeführt sind, erlaubt. Dadurch wird die Korrektheit der Synthese implizit garantiert. Als Ergebnis erhält man somit nicht nur eine Implementierung, sondern auch den mathematischen Beweis dafür, dass diese Implementierung korrekt ist.

Die Formale Synthese hat gegenüber der Postsynthese-Verifikation im wesentlichen zwei entscheidende Vorteile. Zum einen leitet das Verfahren eine Implementierung konstruktiv ab. Damit wird die nachträgliche i.Allg. NP-vollständige Suche nach dem Beweis vermieden. Die Formale Synthese hat somit Vorteile bezüglich der Laufzeiteffizienz. Der zweite Vorteil liegt darin, dass generische Schaltungen synthe-

tisiert werden können. Ist etwa die Bitbreite einer Schaltung noch nicht festgelegt, können somit ganze Klassen von korrekten Schaltungen erzeugt werden. Erst später kann dann eine konkrete Bitbreite instantiiert werden. Die Verfahren der Modellprüfung sind nicht in der Lage, generische Schaltungen zu überprüfen. Hier müssten alle erforderlichen Bitbreiten erst instantiiert und dann die entstehenden konkreten Schaltungen geprüft werden. Mit der Größe der Bitbreite nehmen dabei aber der Zustandsraum und der Zeitaufwand exponentiell zu, was die Modellprüfung von Schaltungen mit großen Bitbreiten sehr erschwert, bzw. scheitern lässt.

3.3.4 Vergleich der Formalen Synthese mit dem transformationsbasierten Entwurf

Die Verfahren der Formalen Synthese (aber auch die der Präsynthese-Verifikation) erfüllen das in der Synthesewelt seit längerem propagierte Paradigma der Korrektheit durch Konstruktion ("correctness by construction"). Es gibt neben den Verfahren der Formalen Synthese noch die Ansätze des transformationsbasierten Entwurfs, die ebenfalls von sich behaupten, dieses Paradigma zu erfüllen.

Dazu ist festzustellen, dass in Arbeiten, die ein Verfahren des transformationsbasierten Entwurfs betreffen, zwar ein Korrektheitsbeweis der Transformationen angeboten wird. Einschränkend muss aber gesagt werden, dass den Beweisen oft kein exakter mathematischer Formalismus zugrunde liegt. Darüber hinaus werden die Beweise mit Papier und Bleistift durchgeführt, was bedeutet, dass sie nicht notwendig objektiv richtig sind, sondern lediglich durch Überprüfung, z.B. häufiges Gegenlesen, validiert werden können. Zudem lassen jene Beweise oft ein intuitives Vorgehen [McFa93] sowie eine Beschränkung auf das Überprüfen von Plausibilitätskriterien erkennen und eine durchgängige Beweisführung vermissen. Der wesentliche Kritikpunkt bei den Verfahren des transformationsbasierten Entwurfs betrifft aber die Tatsache, dass nach dem Korrektheitsbeweis die Transformationen durch eine beliebige Software-Implementierung realisiert werden und für diese Programme *kein* Korrektheitsbeweis geführt wird. Jede neue Transformation, die in ein Syntheseprogramm eingeführt wird, erhöht somit das Risiko, dass sich ein Fehler im Programm einschleicht. Dies ist vor allem vor dem Hintergrund zu sehen, dass im transformationsbasierten Entwurf der Kern der Transformationen sehr groß ist, da die Transformationen mit den Schaltungstransformationen identisch sind. Darüber hinaus sind für eine Synthese auf verschiedenen Abstraktionsebenen auch verschiedene Transformationen erforderlich, was die Anzahl der fehlerfrei zu implementierenden Transformationen weiter in die Höhe treibt.

Im Gegensatz dazu ist die Anzahl der elementaren Transformationen bei der Formalen Synthese klein und stabil, da alle Schaltungstransformationen auf allgemeine logische Regeln zurückgeführt sind. Dies führt dazu, dass mit der Aufnahme neuer Schaltungstransformationen in das Syntheseprogramm sich nichts an dessen Korrektheit ändert. Des Weiteren basieren die elementaren Transformationen der allgemeinen Logikkalküle der Formalen Synthese auf wohlbekannten mathematischen Zusammenhängen, die wesentlich besser getestet sind, da sie in verschiedenen Dis-

ziplinen eingesetzt werden. Die Formale Synthese hat darüber hinaus noch zwei weitere positive Konsequenzen. Indem Schaltungstransformationen innerhalb des Theorembeweisers bewiesen werden, erhält man eine höhere Qualität ihrer Beweise. Sie sind objektiv nachvollziehbar; jede Form von Intuition ist eliminiert und der Beweis durchgehend geführt. Ein weiterer Vorteil besteht darin, dass die bewiesene Schaltungstransformation nicht anschließend separat durch ein Softwareprogramm realisiert werden muss, sondern der Korrektheitsbeweis gleichermaßen als Vorlage für die Software-Implementierung für die Transformation dient. Die einzige sicherheitskritische Komponente bei dieser Vorgehensweise ist die Software-Implementierung des Kerns des Theorembeweisers. Nur mit Hilfe dieses Kerns können Theoreme, d.h. Beweise für Transformationen, abgeleitet werden. Da dieser Kern aber (jedenfalls in dem in dieser Arbeit verwendeten Theorembeweiser HOL) sehr klein ist, kann die Formale Synthese als extrem sicher betrachtet werden. All dies erhöht die Zuverlässigkeit der Formalen Synthese in entscheidender Weise und rechtfertigt die Annahme, dass sie ein ungleich vertrauenswürdigeres Fundament darstellt als der transformationsbasierte Entwurf.

Der Vorteil des transformationsbasierten Entwurfs gegenüber der Formalen Synthese besteht allein darin, dass die Synthese effizienter bzw. in kürzerer Zeit durchgeführt werden kann, da die elementaren Transformationen gerade den Schaltungstransformationen entsprechen und in der Formalen Synthese dagegen für jede Schaltungstransformation eine Reihe elementarer Logiktransformationen angewandt werden muss. Da aber aus den oben genannten Gründen die Formale Synthese wesentlich sicherer ist als der transformationsbasierte Entwurf und eigentlich eine zusätzliche Postsynthese-Verifikation für den transformationsbasierten Entwurf erforderlich ist, kann die Formale Synthese im Endeffekt als effizienter betrachtet werden.

3.4 Stand der Technik

Im Folgenden soll ein Überblick über den derzeitigen Stand der Technik gegeben werden. Er beschränkt sich dabei auf Arbeiten, die die Implementierungskorrektheit auf höheren Abstraktionsebenen untersuchen. Der Stand der Technik bezüglich der unteren Abstraktionsebenen ist in [Eise99] beleuchtet.

3.4.1 Verfahren der Präsynthese-Verifikation

Im Bereich der Präsynthese-Verifikation gibt es wegen der oben erwähnten Gründe nur sehr wenige Arbeiten. Das System BEDROC, das an der Cornell University in Ithaca, NY entwickelt wurde, unterstützt eine Präsynthese-Verifikation teilweise [LCAL93, LeAL91]. Mit diesem System werden Verhaltensbeschreibungen in eine Implementierung mit einem programmierbaren Gate-Array (FPGA) überführt. Als Ausgangspunkt dient eine Verhaltensbeschreibung der Schaltung in einer imperativen Programmiersprache namens HardwarePal, die auf der Sprache Pascal

basiert. Diese Beschreibung wird anschließend in ein Zwischenformat übersetzt. Als Zwischenformat dient ein Abhängigkeitsflussgraph. Auf diesem Graph können bestimmte Optimierungen wie Konstantenpropagierung, Eliminierung toten Codes, etc. realisiert werden. Anschließend werden die zeitliche Einordnung und die Bereitstellung/Zuordnung von Hardwarekomponenten iterativ durchgeführt. Die resultierende RT-Beschreibung wird in eine Gatternetzliste überführt und einer Logikminimierung zugeführt. Anschließend wird ein FPGA programmiert. Die Teile des Entwurfsprozesses, die gemäß einer Präsynthese-Verifikation korrekt arbeiten, sind die Übersetzung von HardwarePal in das Zwischenformat, die Optimierungen innerhalb des Zwischenformats sowie die Logikminimierung. Die eigentliche Überführung der Verhaltensbeschreibung in eine Beschreibung auf RT-Ebene ist *nicht* verifiziert.

Eine weitere Arbeit, die sich mit Präsynthese-Verifikation beschäftigt, ist in [NTRG98] beschrieben. Darin wird ein Verfahren zur Verifikation eines Algorithmus' vorgestellt, der die zeitliche Einordnung einer Datenfluss-basierten Schaltungsbeschreibung vornimmt. Dazu werden im Theorembeweiser PVS [OwSR93], der auf Logik höherer Ordnung basiert, Eigenschaften formalisiert und bewiesen. Diese Eigenschaften sind im Wesentlichen die, dass jeder Operation ein Kontrollschritt zugewiesen wird, dass die Datenabhängigkeiten nicht verletzt und dass die Randbedingungen des zulässigen Hardwareaufwands eingehalten werden. Es wird zudem der Beweis erbracht, dass ein formales Modell des Algorithmus' diese drei Eigenschaften erfüllt. Anschließend werden diese Theoreme "sorgfältig" (Zitat) in Überprüfungsanweisungen in der Programmiersprache C++ übersetzt. Somit liefert nach Meinung der Autoren das Softwareprogramm immer ein Ergebnis, bei dem die Operationen in korrekter Weise zeitlich eingeordnet wurden. Einschränkend ist festzustellen, dass sich die Methode nur für reine Datenflüsse einsetzen lässt. Darüber hinaus handelt es sich bei diesem Ansatz streng genommen nicht um eine reine Präsynthese-Verifikation, da das C++-Programm nicht verifiziert wurde. Auch die Überprüfungsroutinen wurden nicht verifiziert, vielmehr wurde nur darauf geachtet, dass diese korrekt implementiert wurden. Ein ähnliches Verfahren derselben Autoren für die Bereitstellung von Registern wird in [NaVe98] vorgestellt.

3.4.2 Verfahren der Postsynthese-Verifikation

Ein Verfahren zur automatischen Postsynthese-Verifikation wird in [MaVe98] beschrieben. Die Verhaltensbeschreibung in VHDL wird dabei zunächst in einen Verhaltensautomaten übersetzt. Dieser besteht aus Zuständen und Transitionen. Es gibt Zuweisungszustände mit nur einer Ausgangstransition und Verzweigungszustände mit zwei Ausgangstransitionen. Die Synthese erzeugt daraus einen Datenpfad und einen Kontrollautomaten, die über Kontrollvariablen kommunizieren. Der Kontrollautomat ist ähnlich aufgebaut wie der Verhaltensautomat, mit dem Unterschied, dass den Zuweisungszuständen lediglich aktive Kontrollsignale zugeordnet sind. Kritische Zustände sind alle Verzweigungs- bzw. Zusammenführungszustände sowie Ein- und Ausgabezustände. Als kritische Variablen werden all diejenigen bezeichnet, die Ein- oder Ausgabevariablen sind, sowie alle lokalen Variablen, deren

Wirkungsweise über kritische Zustände hinweg geht. Die Äquivalenz der beiden Modelle wird durch ein Plausibilitätskriterium überprüft, indem untersucht wird, ob die Werte in den kritischen Variablen der kritischen Zustände des Verhaltensautomaten den Werten in den kritischen Registern der kritischen Zustände des Kontrollers entsprechen. Zu beachten ist die Tatsache, dass während der Synthese kein Code über Verzweigungs- bzw. Schleifenanweisungen verschoben werden darf. Transformationen auf der Verhaltensebene sind nicht erlaubt. Dies ist eine sehr einschränkende Bedingung, da dadurch bereits implizit ein grobes Zeitmodell eingeführt wird. Als Folge ergibt sich, dass die beiden Automaten einen sehr ähnlichen Aufbau haben. Sie unterscheiden sich lediglich dadurch, dass zwischen den kritischen Zuständen evtl. unterschiedlich viele Zwischenzustände vorhanden sind. Die kritischen Zustände werden nicht verändert. Dies macht die Überprüfung sehr einfach. Der Beweis wird im Theorembeweiser PVS erbracht. Dazu werden zunächst die Informationen im Verhaltensautomat, im Datenpfad und im Kontrollautomat als Axiome in PVS übertragen. Dann werden zu beweisende Lemmata aufgestellt. Bei der Formulierung dieser Lemmata muss die Information des Synthesewerkzeugs vorliegen, wie die kritischen Variablen auf Register und wie kritische Verhaltenszustände auf Kontrollzustände abgebildet wurden. Insofern unterscheidet sich dieser Ansatz von allgemeinen Postsynthese-Verifikationsverfahren, bei denen die Syntheseinformation nicht vorliegt. Die Lemmata werden anschließend automatisch durch Termersetzung bewiesen. Bei diesem Ansatz, wie aber auch bei vielen anderen Ansätzen (siehe auch später), muss deutlich darauf hingewiesen werden, dass das Einführen neuer Axiome in den Kalkül eines Theorembeweisers sehr heikel ist, da somit der Kalkül inkonsistent werden kann.

Die Verifikation eines Teilschritts während der High-Level Synthese, nämlich der der zeitlichen Einordnung, wird in [EvHR99] vorgestellt. Die Verhaltensbeschreibung wird zunächst in ein spezielles Zwischenformat namens LLS (Language of Labelled Segments) übersetzt. Nach der zeitlichen Einordnung ergibt sich eine neue LLS-Beschreibung. Das Verifikationsziel besteht darin, zu beweisen, dass die beiden Beschreibungen berechnungsäquivalent sind. Dieser Begriff besagt, dass beide Beschreibungen dasselbe Endergebnis liefern, wenn sie mit denselben Eingabewerten versehen werden. Die Äquivalenz wird aber nur indirekt nachgewiesen. Anstatt beide Beschreibungen zu vergleichen, wird die Ausgangsbeschreibung in eine berechnungsäquivalente Beschreibung transformiert, die bisimilar zur Beschreibung nach der zeitlichen Einordnung ist. Zwei Beschreibungen sind bisimilar, wenn es jeweils zwei bisimilare Segmente gibt. Zwei Segmente sind bisimilar, wenn dieselben Datenoperationen ausgeführt werden und der Kontrollfluss in jeweils bisimilare Segmente überführt wird. Die Transformation der Ausgangsbeschreibung erfolgt durch einfache Transformationsregeln, wie die mögliche Parallelisierung sequentieller Zuweisungen unter Berücksichtigung der Datenabhängigkeiten oder einfacher boolescher Zusammenhänge in bedingten Verzweigungen. Die korrekte Implementierung der Transformationen wird nicht nachgewiesen.

Ein weiterer Ansatz für eine Postsynthese-Verifikation sind die Arbeiten von Mutz und Lock von der Universität Passau. In [Mutz97, LoMu97, LoMM98] wird

deren Verfahren zur Verifikation der High-Level Synthese reiner Datenpfade vorgestellt. Schaltungen auf der algorithmischen sowie auf der RT-Ebene werden in einer Untermenge der Hardwarebeschreibungssprache VHDL beschrieben. Diese VHDL-Beschreibungen werden durch eine formale Verankerung (siehe Abschnitt 4.1) in den Theorembeweiser HOL übersetzt. Ausgehend von der algorithmischen Schaltungsbeschreibung wird in konventioneller Weise eine High-Level Synthese durchgeführt. Wichtig für den Ansatz ist, dass die interne Kontrollinformation des Synthesewerkzeugs, wie die Synthese durchgeführt wurde, der Verifikation zugänglich gemacht wird. Auch hier handelt es sich daher wie bei [MaVe98] um kein reines Postsynthese-Verifikationsverfahren. Für die Überprüfung der Korrektheit der sich durch die Synthese ergebenden RT-Implementierung wird kein durchgehender Beweis geführt, sondern es wird ein intuitives Kriterium aufgestellt. Dieses Kriterium sagt aus, dass die Synthese dann korrekt sei, wenn die Werte der Eingangsvariablen des Datenflusses mit den Initialwerten der ihnen zugeordneten Register übereinstimmen und in den Registern, die den Ausgabevariablen des Datenflusses zugeordnet sind, am Ende der Berechnung die Werte der Ausgabevariablen gespeichert sind. Dies ist ein ähnliches Kriterium wie in den beiden obigen Ansätzen. Die Überprüfung, ob dieses Kriterium erfüllt ist, läuft wie folgt ab: In einem der Synthese vorangehenden Schritt werden mehrere abstrakte Bedingungen formuliert. Es wird dann interaktiv ein allgemeines Theorem in HOL bewiesen, das aussagt, dass bei Erfüllung dieser Bedingungen die Korrektheit nach der eben erwähnten Definition gilt. Dieser Beweis wird einmal geführt. Nach jedem Syntheselauf wird dieses Theorem dann mit einer konkreten Spezifikation und konkreten Syntheseergebnissen instantiiert. Bislang können nur reine Datenflussbeschreibungen mit dieser Methode verifiziert werden. Erste Arbeiten für eine Erweiterung auf Beschreibungen mit Kontrollfluss finden sich in [LoMe98, LoMe99]. Im ersten Artikel wird eine formale Modellierung der Schaltungsbeschreibungen vorgestellt. Im zweiten Artikel werden Korrektheitseigenschaften formuliert. Der schwierigste Teil, der Beweis eines Korrektheitstheorems in HOL und die automatische Verifikation mit diesem Theorem nach der Synthese, steht aber noch aus. Wie die Autoren in [Mutz97] selbst ausführen, hat ihre Methode den entscheidenden Nachteil, dass aufgrund der Plausibilitätsüberprüfung nicht garantiert werden kann, dass die Verifikation die Korrektheit der Synthese immer beweisen kann. Schlägt der Ansatz fehl, müssen allgemeinere, aber ineffizientere Verifikationsmethoden eingesetzt werden.

3.4.3 Verfahren des transformationsbasierten Entwurfs

Mehrere transformationsbasierte Verfahren wurden in den letzten Jahren sowohl im Softwarebereich als auch im Schaltungsentwurf vorgestellt. Im Softwareentwurf haben sich diese Verfahren allerdings noch nicht recht durchsetzen können. Eine Ausnahme gilt lediglich für die automatische Optimierung von Compilern (z.B. [Alle69, AhSU86, ViBT98], siehe auch [Paig97, PePr97] für zukünftige Entwicklungen). Dies liegt vor allem an der enormen Größe von Softwareprogrammen im Gegensatz zu Hardwarebeschreibungen, aber auch an den höheren Anforderungen

bei Hardwaresystemen bezüglich ihrer Zuverlässigkeit und Effizienz. Während man sich oft damit zufrieden gibt, dass ein Softwareprogramm über eine längere Zeit stabil läuft, ist eine fehlerhafte Digitalschaltung nicht akzeptabel. Ein weiterer Grund ist darin zu sehen, dass imperative Programmiersprachen, die den Softwareentwurf aufgrund ihrer Laufzeiteffizienz beherrschen, im Gegensatz zu funktionalen und logischen Programmiersprachen einem transformationsbasierten Entwurf weniger zugänglich sind.

Im CAMAD-System [PeKu94, HaPe95] wird eine automatische High-Level Synthese mit Transformationen für die zeitliche Einordnung und die Bereitstellung von Komponenten durchgeführt. Die Schaltungsbeschreibungen sind in einer Pascalähnlichen Notation gegeben. Um ein Programm zu transformieren, wird es zuerst in eine Repräsentation als zeitbehaftetes Petri-Netz übersetzt. In dieser Darstellung sind der Daten- und der Kontrollfluss getrennt. Sowohl die Übersetzung von Pascal in ein Petri-Netz, als auch die Transformationen innerhalb des Petri-Netzes werden durch Softwareprogramme realisiert, die komplex und sicherheitskritisch sind. Ein expliziter Beweis für die Korrektheit ihrer Implementierungen wird nicht gegeben.

Das TRADES-System [MMEK96, MiRa96, MHMP96, Midd97, Huij98] verwendet einen Ansatz, bei dem die Schaltung durch sogenannte markierte Hypergraphen beschrieben wird. Das Modell ist sehr vage, da es in ihm keine strikte Unterscheidung zwischen algorithmischen und RT-Ebenen-Beschreibungen gibt. Vielmehr wird ziemlich willkürlich zwischen den beiden Repräsentationsmodellen gewechselt. Die Schaltungstransformationen werden interaktiv durch Ersetzung von Teilgraphen ("graph-rewriting") durchgeführt. Die Korrektheit der Transformationen ist intuitiv und informell überprüft. Für einen formalen Beweis sollen sie in Zukunft mit dem Theorembeweiser PVS bewiesen werden, wobei allein schon die Übersetzung von Graphen nach PVS und die interne Repräsentation in PVS noch unklar sind.

Ein transformationsbasierter Ansatz für den Entwurf kontrolldominierter Hardware wird in [GrLS98] vorgestellt. Als Spezifikation auf der algorithmischen Ebene dienen Zeitdiagramme, die in eine Repräsentation in Prozessalgebra überführt werden. Die Korrektheit dieses Schritts wird nicht sichergestellt. Anschließend werden Transformationsregeln angewandt, die auf dem Papier bewiesen wurden, und es wird eine Implementierung erzeugt. Eine Entwurfsraumuntersuchung ist nicht möglich, vielmehr wird die Spezifikation nach einem bestimmten Muster in mehrere Prozesse zerlegt. Jeder Datenflussanteil wird in einen eigenen Prozess abgetrennt, der Kontroller bleibt letztendlich in einem separaten Prozess übrig.

Einen interessanten Ansatz, Programme mittels Transformationen zu synthetisieren, stellt die Arbeit in [JiPB93] dar. Sie entstammt einer Untersuchung von Hoare über einen korrekten Compilerentwurf [HoJS93]. Programme werden dabei durch Transformationen in eine Normalform überführt, und anschließend in eine Gatternetzliste übersetzt. Die Transformationen wurden bewiesen und in einem Compiler realisiert. Wie die Transformationen bewiesen wurden, ob sie ausreichend sind, und ob die Umsetzung in den Compiler korrekt durchgeführt wurde, wird nicht näher erläutert. Nachteilig bei diesem Ansatz ist jedenfalls das Fehlen einer Optimierungsmöglichkeit. Jedes Programm wird in eindeutiger Weise synthetisiert.

3.4.4 Verfahren der Formalen Synthese

Die meisten der in der Literatur bekannten Ansätze der Formalen Synthese beschäftigen sich mit einer Synthese auf unteren Abstraktionsebenen (RT- und Gatterebene). Auf höheren Abstraktionsebenen gibt es bislang nicht sehr viele Ansätze. Hier sind vor allem die Arbeiten von Larsson zu erwähnen. In [Lars94, Lars95] wird ein Verfahren zur Formalen Synthese von Datenflussbeschreibungen vorgestellt. Eine abstrakte Schaltungsbeschreibung wird im Theorembeweiser HOL formalisiert und stufenweise transformiert. Es gibt Transformationen für die zeitliche Einordnung, die Bereitstellung und die Zuordnung von Funktionseinheiten sowie für die Kommunikationssynthese. Eine Schnittstellensynthese wird nicht durchgeführt. Es ergibt sich demnach keine RT-Struktur, sondern eine algorithmische Beschreibung, bei der den zeitlich eingeordneten Operationen konkrete Funktionseinheiten zugeordnet sind. Die Implementierungsbeschreibung verfügt über keine Register. Eine weitere massive Einschränkung besteht darin, dass als Verfahren zur zeitlichen Einordnung implizit der ASAP-Algorithmus verwendet wird. Es besteht keine Möglichkeit für eine (automatische) Entwurfsraumuntersuchung. Außerdem wird die zeitliche Einordnung immer nach der Bereitstellung durchgeführt. Bei der Bereitstellung von Funktionseinheiten kann man zwar unterschiedliche Entscheidungen treffen, dies aber nur interaktiv. Trotz dieser Einschränkungen hat das Verfahren aber gegenüber den bisher vorgestellten Ansätzen den entscheidenden Vorteil, dass die Korrektheit der Implementierung tatsächlich garantiert ist.

Eine weitere Arbeit von Larsson findet sich in [Lars96]. Sie ist unabhängig von dem oben beschriebenen Ansatz, sodass die beiden Verfahren nicht kombiniert werden können. Es werden korrekte Transformationen auf eine Schaltungsbeschreibung innerhalb der algorithmischen Ebene angewandt. Eine Synthese auf die RT-Ebene wird nicht durchgeführt. Im Gegensatz zu [Lars94, Lars95] enthalten die Schaltungsbeschreibungen auch einen Kontrollflussanteil. Die Schaltungsbeschreibungen liegen in einer imperativen Programmiersprache vor. Diese Sprache wird in die Logik des Theorembeweisers HOL eingebettet. Es werden einige Transformationstheoreme aufgestellt, die Termersetzungsregeln darstellen. In den Prämissen der Theoreme sind Beweisverpflichtungen aufgeführt, die erfüllt sein müssen, damit das Transformationstheorem angewandt werden kann. Einige einfache Transformationstheoreme für for-Schleifen wurden bewiesen, allerdings gibt es keine Transformationstheoreme für while-Schleifen. Für die Termersetzung mit den Theoremen musste eine komplizierte Infrastruktur für die Überprüfung der Beweisverpflichtungen geschaffen werden. Dies hängt mit der imperativen Natur der Programmiersprache zusammen, die eine exakte Formalisierung dafür erfordert, was es bedeutet, wenn ein Wert einer Variablen zugewiesen oder eine Variable gelesen wird.

Eine Arbeit, die nur bedingt zur Formalen Synthese gerechnet werden kann, ist in [Raja95a, Raja95b] beschrieben. Es werden wie in [Lars96] Transformationen innerhalb der algorithmischen Ebene beschrieben. Eine Verfeinerung auf die RT-Ebene wird nicht durchgeführt. Die Schaltungsbeschreibungen enthalten lediglich Datenfluss und bedingte Verzweigungen. Schaltungsbeschreibungen, die in Form

eines Abhängigkeitsgraphen gegeben sind, werden in eine Darstellung innerhalb des Theorembeweisers PVS übersetzt. Die Korrektheit dieser Übersetzung wird nicht überprüft. Der Ansatz wird an dieser Stelle aufgeführt, da die Transformationen auf den Schaltungsbeschreibungen innerhalb von PVS bewiesen wurden. Es ist aber zu beachten, dass die Transformationen nicht auf einen allgemeinen Logikkalkül zurückgeführt wurden. Vielmehr wurden einige als intuitiv richtig eingeschätzte Korrektheitseigenschaften als (unbewiesene) Axiome definiert. Die Transformationen wurden dann auf diese Axiome zurückgeführt. Es handelt sich dabei also lediglich um die Überprüfung von Korrektheitseigenschaften basierend auf einem Theorembeweiser. Kritisch ist auch die Frage, ob die axiomatisch eingeführten Korrektheitseigenschaften vollständig, widerspruchsfrei und korrekt sind. Dieser Frage wird nicht nachgegangen.

Auch im Bereich des Softwareentwurfs gibt es einige Arbeiten, die einer Formalen Synthese entsprechen. Zu nennen sind hier vor allem von Wright's Arbeiten [BaWr90, WrSe91, Wrig94a], auf die an dieser Stelle aber nicht näher eingegangen werden soll.

3.4.5 Zusammenfassung

Alle aufgeführten Arbeiten haben die Implementierungskorrektheit auf der algorithmischen Ebene zum Ziel. Vergleichbare Arbeiten, die sich mit einer Synthese auf der Systemebene beschäftigen, sind mir nicht bekannt.

Die Verfahren des transformationsbasierten Entwurfs wurden nur der Vollständigkeit halber aufgeführt. Wie in Abschnitt 3.3.4 erwähnt, können sie die Implementierungskorrektheit nicht garantieren. Die anderen Ansätze haben gemeinsam, dass sie immer nur Teilaufgaben der High-Level Synthese betrachten. Entweder beschränken sich die Autoren auf reine Datenflussbeschreibungen oder es wird nur die Korrektheit von Teilschritten untersucht. Darüber hinaus werden gerade bei den Postsynthese-Verfahren lediglich Plausibilitätskriterien überprüft. Die Arbeit, die dem Anspruch eines garantiert korrekten Syntheseergebnisses am ehesten gerecht werden kann, ist die von Larsson (siehe Abschnitt 3.4.4). Aber auch hier wird keine vollständige Formale Synthese auf der algorithmischen Ebene oder gar auf der Systemebene angeboten. Eine solche Synthese ist daher das Ziel der vorliegenden Arbeit.

Kapitel 4

Konzepte für die Hardwarebeschreibungssprache Gropius

Um eine Synthese digitaler Systeme durchführen zu können, muss als erstes geklärt werden, wie diese Systeme überhaupt beschrieben werden sollen. In der Literatur findet sich dafür eine große Anzahl von Hardwarebeschreibungssprachen. Basierend auf wesentlichen Erkenntnissen und Zielsetzungen, die aus einer kritischen Durchsicht dieser vorhandenen Hardwarebeschreibungssprachen erwachsen sind, wurde eine neue Hardwarebeschreibungssprache mit Namen Gropius* entwickelt. In diesem Kapitel sollen die wesentlichen Konzepte von Gropius vorgestellt werden. In den nachfolgenden Kapiteln 5 und 6 schließlich wird erläutert, wie Schaltungen auf den höheren Abstraktionsebenen in Gropius beschrieben werden.

Statt eine neue Sprache zu definieren, wäre die Alternative gewesen, eine bereits vorhandene Sprache zu erweitern. Die Entscheidung für eine neue Sprache hat aber eine strategische Komponente. Die Hardwarebeschreibungssprache bildet nämlich die Grundlage für die darauf anzuwendenden Syntheseverfahren und somit auch das konzeptionelle Fundament für den in dieser Arbeit entwickelten generellen Entwurfsstil. Nicht zuletzt die Tatsache, dass bestehende Hardwarebeschreibungssprachen die in dieser Arbeit entwickelten Synthesekonzepte nur partiell unterstützen, rechtfertigt die Entscheidung für eine neue Sprache.

Gropius verfügt über die nachfolgend aufgelisteten Eigenschaften, auf die im Weiteren näher eingegangen wird.

- Exakte Semantik
- Mathematische Notation
- Funktionale Schaltungsmodellierung

*Walter Gropius (1883-1969), Begründer des Bauhauses. Im Gegensatz zum damals vorherrschenden Jugendstil mit seinen extravaganten Ausprägungen wurde durch den Stil des Bauhauses der Entwurf auf das Wesentliche konzentriert (*form follows function).*

- Konsistenz und Implementierbarkeit
- Einfache Simulierbarkeit
- Verwendung ebenenspezifischer und aufeinander aufbauender Teilsprachen
- Systematische Wiederverwertbarkeit von Schaltungsbeschreibungen
- Kein Unterschied zwischen externem und internem Format

4.1 Exakte Semantik, mathematische Notation

Das Hauptaugenmerk der formalen Synthese richtet sich auf die Korrektheit des Syntheseprozesses. Unter Korrektheit soll dabei verstanden werden, dass das Ergebnis des Syntheseprozesses (Implementierung) zu der Eingabe des Syntheseprozesses (Spezifikation) in einer bestimmten mathematischen Relation steht. Je nachdem, ob es sich bei dem betrachteten Syntheseschritt um eine Optimierung oder um eine Verfeinerung handelt, ist diese Relation die Äquivalenz oder die Implikation.

Um aber über die Korrektheit der Synthese im mathematischen Sinne sprechen zu können, bedarf es einer mathematisch exakten Grundlage. Dies können die meisten der gängigen Hardwarebeschreibungssprachen wie VHDL [VHDL96] oder Verilog [Cade92] nicht bieten, da ihre Semantik umgangssprachlich definiert ist. Die Sprache Gropius dagegen verfügt über eine mathematisch exakte Semantik. Sie wurde im Theorembeweiser HOL definiert und bildet somit eine Teilmenge der Prädikatenlogik höherer Ordnung. HOL eignet sich besonders gut für eine Formalisierung von Hardwarebeschreibungen [Gord86]. Gropius benutzt dabei eine Notationsform, die einem mathematisch geprägten Denken entgegenkommt.

Es macht auch keinen Sinn, für den einen oder anderen Bereich eine geeignete Teilsprache gängiger umgangssprachlich definierter Hardwarebeschreibungssprachen als Basis zu verwenden. Für eine logische Argumentation über den Verlauf eines Syntheseprozesses ist ohnehin eine Notation in der Logik unabdingbar. Es ist also wenig sinnvoll, während der Synthese zwei verschiedene Notationsformen zu verwenden.

Es gibt mehrere Ansätze, umgangssprachlich definierten Hardwarebeschreibungssprachen nachträglich eine passende mathematisch exakte Semantik zuzuordnen [BGGH92, BGHT91, KlBr95, Pier97, Reet97]. Eine Hardwarebeschreibungssprache formal zu verankern bedeutet jedoch, die Lücke zwischen der Hardwarebeschreibungssprache und der Logik zu schließen. Die Umsetzung der Semantik ist aber nur dann möglich, wenn die umgangssprachliche Definition der Semantik tatsächlich eindeutig ist. Ist dies nicht der Fall (wie z.B. bei VHDL), muss die Umsetzung entweder scheitern, oder man muss sich auf eine Teilsprache zurückziehen, die dann aber evtl. nicht mehr allgemein einsetzbar ist, da andere Benutzer sich eine andere Teilsprache definiert haben könnten. Gerade für VHDL wurden zahlreiche Anstrengungen unternommen, um eine eindeutige formale Semantik zu definieren. Es gibt aber keinen Standard hierfür. Vielmehr wurden mehrere widersprüchliche "Semantiken" von VHDL definiert, was erhebliche Auswirkungen auf den Schaltungsentwurf mit

sich bringt, da VHDL-Beschreibungen infolgedessen von unterschiedlichen Syntheseprogrammen unterschiedlich interpretiert werden.

Wie groß der Aufwand für eine nachträgliche Verankerung ist, hängt nicht zuletzt davon ab, wie nahe die Konstrukte der Hardwarebeschreibungssprache der Logik stehen. Hier stellt die formale Beschreibung imperativer Programmierkonstrukte, wie sie in VHDL vorkommen, ein großes Problem dar. Solche Konstrukte führen – im Gegensatz zu Konstrukten in funktionalen bzw. logischen Programmiersprachen – ausschließlich zu Nebenwirkungen, nämlich der Veränderung des globalen Zustands, d.h. der Veränderung der Werte der beteiligten Objekte. Für eine Verankerung muss also definiert werden, wie jede Anweisung den globalen Zustand verändert. Eine Umsetzung funktionaler Hardwarebeschreibungssprachen ist hierbei vergleichsweise einfach, da wie in Kapitel 2 dargelegt, funktionale Programmiersprachen auf dem λ-Kalkül basieren.

4.2 Funktionale Modellierung, Konsistenz, Simulierbarkeit

Die meisten Ansätze, Schaltungen in der Logik zu formalisieren, verwenden eine relationale Schaltungsmodellierung [Melh93]. Dabei werden Schaltungskomponenten als Relationen zwischen Ein- und Ausgangssignalen modelliert. Die einzelnen Komponenten können anschließend zu relational beschriebenen Strukturen zusammengefasst werden. Der hier vorgestellte Ansatz verwendet eine funktionale Schaltungsmodellierung (die einzige Ausnahme stellt dabei die Beschreibung auf der Systemebene dar, die in Abschnitt 6.1 vorgestellt wird.).

Die Entscheidung, hier vom gängigen Standard abzuweichen, hat mehrere gute Gründe. So muss man sich zum einen vor Augen halten, dass konkret realisierte Digitalschaltungen stets einen funktionalen Zusammenhang zwischen Ein- und Ausgangssignal herstellen. Eine Relation beschreibt lediglich, ob die Signale in der gewünschten Beziehung zueinander stehen. Sie ist jedoch nicht konstruktiv, d.h. es wird nicht beschrieben, wie aus den Eingangswerten die Ausgangswerte abgeleitet werden können.

Ein anderer entscheidender Grund für funktionale Beschreibungen ist die Sicherstellung der Implementierbarkeit. So können bei der relationalen Form der Schaltungsbeschreibung leicht inkonsistente Schaltungen entstehen. Auch dann, wenn die Grundkomponenten richtig formalisiert wurden, sind z.B. durch Schaltungsstrukturen mit verzögerungsfreien Zyklen oder Kurzschlüssen Inkonsistenzen möglich. Inkonsistente Schaltungsbeschreibungen sind jedoch nicht implementierbar. Bei jedem Syntheseschritt findet eine logische Verfeinerung statt. Das Ergebnis könnte eine inkonsistente, nichtimplementierbare Schaltung sein (ex falso quodlibet). Das Resultat einer derartigen Formalen Synthese wäre ein Scheinerfolg: ein korrekter Beweis für eine nichtimplementierbare “Schaltungsbeschreibung”. Da der relationale Ansatz rein syntaktisch auch nichtimplementierbare Beschreibungen zulässt, ist es erforderlich, die Implementierbarkeit explizit zu überprüfen, so dies überhaupt

in effizienter Weise möglich ist. Das Formale-Synthese-System Lambda/Dialog [MaFo91] ist hierfür ein Beispiel. Nach jedem Syntheseschritt wird die Schaltung zur Sicherstellung der Konsistenz auf Kurzschlüsse und verzögerungsfreie Zyklen hin überprüft. Der hier vorgestellte Ansatz unterscheidet sich hiervon grundlegend. Er erzwingt durch syntaktische Beschränkungen implementierbare Schaltungsbeschreibungen, sodass alle syntaktisch zulässigen Beschreibungen auch zugleich konsistente und implementierbare Schaltungsbeschreibungen sind.

Gropius ist durchweg typisiert. Auch dies ist ein wichtiger Beitrag zur Gewährleistung der Konsistenz. Zwar kann es im Softwarebereich durchaus auch interessant sein, mit nichttypisierten Sprachen wie Lisp zu arbeiten; zur Modellierung von Signalleitungen eignen sich Typen mit offenem Wertebereich jedoch nicht. Durch die strenge Typisierung werden Fehler in der Schaltungsbeschreibung bereits zur Compilezeit und nicht erst zur Laufzeit entdeckt. Damit können Fehler in einem viel früheren Stadium des Entwurfs entdeckt werden als bei anderen Hardwarebeschreibungssprachen.

Die in dieser Arbeit verwendeten Schaltungsbeschreibungen sind funktional im Sinne des λ-Kalküls definiert. Die Schaltungen können in einem Interpreter einer funktionalen Programmiersprache wie SML [Paul91] simuliert werden. Dazu müssen in der funktionalen Programmiersprache die in den nächsten Kapiteln vorgestellten Grundfunktionen implementiert werden. Eine Simulation erfolgt durch Evaluierung der Terme. Die Simulation kann ebenenspezifisch auf effiziente Weise ausgeführt werden, und zwar ohne die Schaltungen auf die Gatterebene zurückführen zu müssen. So ist es möglich, auf der RT-Ebene die Simulation mit komplexeren Operationen (arithmetische Einheiten etc.) effizient auszuführen. Auf der algorithmischen Ebene können reine Softwaresimulationen mit den algorithmischen Beschreibungen durchgeführt werden, ohne dabei bereits eine zeitliche Einordnung auf Takte berücksichtigen zu müssen. Auf der Systemebene sind die Beschreibungen in Gropius an Petri-Netze höherer Ordnung [Jens92] angelehnt. Bei Petri-Netzen höherer Ordnung sind die Knoten mit Funktionen versehen, die in der funktionalen Programmiersprache SML implementiert sind. Somit ist die Simulation der einzelnen Prozesse sowie des Gesamtsystems möglich. Man mag sich fragen, warum eine Simulation überhaupt angestrebt wird, nachdem in Abschnitt 1 die Simulation als ungeeignetes Mittel für eine Korrektheitsüberprüfung eingestuft wurde. Tatsächlich bezieht sich diese Aussage aber nur auf die Implementierungskorrektheit. Eine Simulation ist aber auf jeden Fall interessant für die Überprüfung der Spezifikationskorrektheit durch den Entwerfer, z.B. was das Verhalten in bestimmten Situationen angeht.

4.3 Ebenenspezifische Teilsprachen

Es gibt Ansätze, universelle Schaltungsbeschreibungssprachen wie etwa VHDL zu entwerfen, bei denen beabsichtigt ist, mit den gleichen Sprachkonstrukten alle Abstraktionsebenen abzudecken. Die Entwicklung im Synthesebereich deutet je-

doch immer mehr darauf hin, dass sich dieses Konzept nicht durchsetzt. Statt für alle Ebenen in gleicher Weise geeignet zu sein, erweist es sich, dass diese Universalsprachen weder für die eine noch für die andere Ebene gut zu gebrauchen sind. Das oft gebräuchliche Konzept, zu den Universalsprachen ebenenspezifische Teilsprachen mit einer ebenenspezifischen Semantik zu definieren [DeBo95, JePO93, BoSa95, FuMe95, BrFK95], bedeutet letztlich nichts anderes, als das Konzept der Universalsprache aufzugeben. Es zeigt sich immer mehr, dass man an einer dezidierten, für die jeweilige Ebene "effizienten" Darstellung nicht vorbeikommt.

Die Sprache Gropius macht bewusst nicht den Versuch, eine universelle Hardwarebeschreibungssprache darzustellen, sondern ist konzeptionell an die Anforderungen der einzelnen Ebenen angepasst. Die Sprache gliedert sich in vier Teile: Für Schaltungsbeschreibungen auf Systemebene gibt es eine andere Darstellung als für Schaltungsbeschreibungen auf der algorithmischen Ebene, und diese wiederum unterscheiden sich von Schaltungsbeschreibungen auf der RT-Ebene und auf der Gatterebene. Diese vier Teilsprachen von Gropius sollen mit Gropius-3, Gropius-2, Gropius-1 und Gropius-0 bezeichnet werden. Zwar sind diese vier Teilmengen insofern disjunkt, als eine Schaltungsbeschreibung in Gropius immer genau einer der vier Teilsprachen zuzuordnen ist; die vier Teilsprachen bauen jedoch aufeinander auf, wie es in Abbildung 4.1 gezeigt wird.

Dem Benutzer stehen eine Reihe von fest definierten Grundkonstrukten zur Verfügung. Darauf aufbauend kann er sich eigene Funktionen definieren, die er dann zur Schaltungsbeschreibung auf verschiedenen Abstraktionsebenen verwenden kann. Die Beschreibung von Schaltungen auf der Gatterebene stellt das Fundament der Sprache dar. Sie bildet die Grundlage für die restlichen Anteile der Sprache auf der RT-, der algorithmischen und der Systemebene. In dieser Arbeit soll in den Kapiteln 5 und 6 ausführlich auf die Schaltungsbeschreibung auf der algorithmischen und der Systemebene eingegangen werden. Die Schaltungsbeschreibung auf der RT-Ebene wird nur kurz in Abschnitt 5.1.2 angerissen. Eine detaillierte Beschreibung, wie Schaltungen auf der RT- bzw. Gatterebene formalisiert werden, findet sich in [Eise99].

Die Teilsprachen basieren jeweils auf einem möglichst kleinen Kern. Darüber hinausgehende Konstrukte werden aus diesem Kern abgeleitet. Zum einen gestaltet sich dadurch die Synthese einfacher, denn nur für den Kern der Sprache müssen Synthese- und Simulationsverfahren angeboten werden. Zum anderen vereinfacht dieses Konzept das Erlernen der Sprache, denn eine große Anzahl interessanter Konstrukte kann auf einfache Konstrukte zurückgeführt werden, ohne dabei unübersichtlich zu werden. Der Aufwand für die Einarbeitung in konventionelle Hardwarebeschreibungssprachen ist gerade auf höheren Entwurfsebenen nicht unerheblich. In vielen Hardwarebeschreibungssprachen sind sehr viele Grundkonstrukte enthalten, die oft jedoch auch redundant sind. Zur Veranschaulichung sei erwähnt, dass VHDL über mehr als 150 Syntaxregeln verfügt, während es in Gropius nur deren 15 sind (siehe (5.1) auf Seite 50, (5.15) auf Seite 61, (5.34) auf Seite 71 und (6.1) auf Seite 83). Zur Verdeutlichung sei erwähnt, dass in [Lips91] die Sprache VHDL auf knapp 300

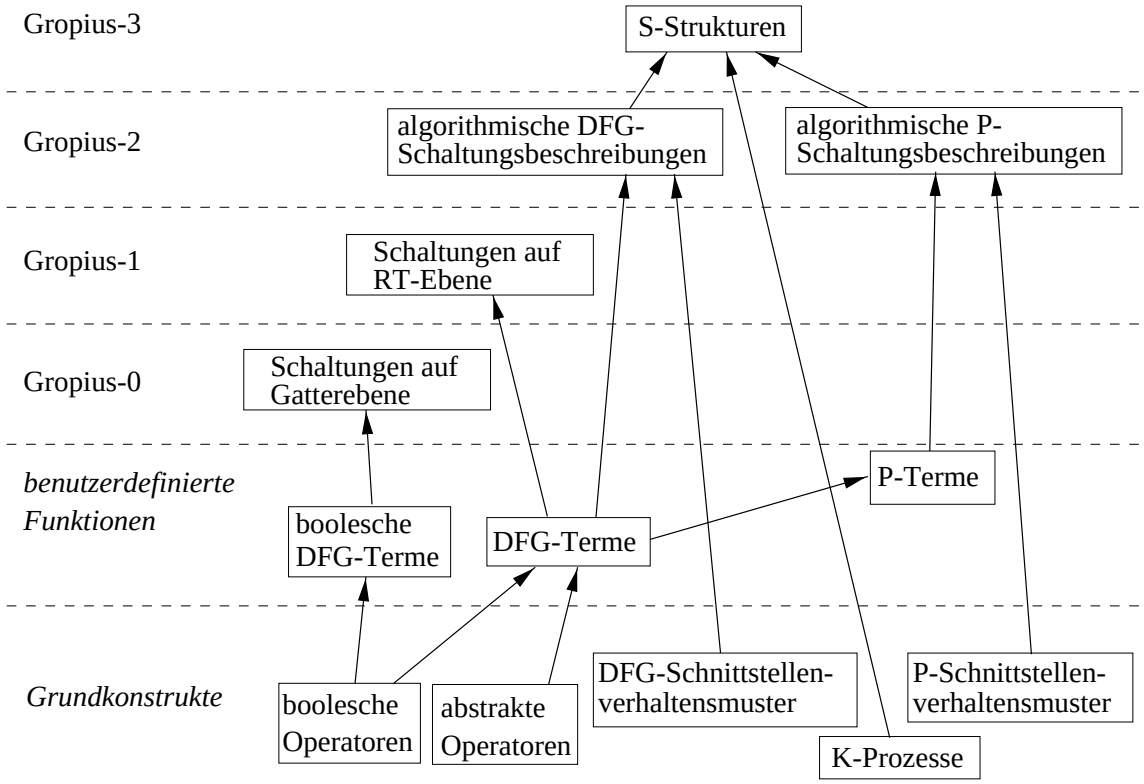


Abbildung 4.1: Aufbau der Sprache Gropius

Seiten umgangssprachlich erklärt wird, während Gropius in dieser Arbeit auf knapp 60 Seiten inklusive der Semantik in der Logik vorgestellt wird.

4.4 Systematische Wiederverwertbarkeit von Schaltungsbeschreibungen

Wegen der Komplexität heutiger Schaltungen und in dem Bemühen um Reduzierung der Entwurfskosten werden große Anstrengungen unternommen, bereits benutzte Schaltungsbeschreibungen für den Entwurf anderer Schaltungen wiederzuverwenden. Dabei ist es von entscheidender Bedeutung, ob und in welchem Maße die verwendete Hardwarebeschreibungssprache Konzepte für eine solche Wiederverwendung vorsieht. Gropius wurde bewusst unter dem Gesichtspunkt entwickelt, die Wiederverwendung von Schaltungsbeschreibungen nicht nur innerhalb einer Abstraktionsebene, sondern auch zwischen den Abstraktionsebenen zu ermöglichen [EiBl98, EiBl99].

Gropius ist eine funktionale Programmiersprache höherer Ordnung, die sehr einfach verwendet werden kann. Der Benutzer kann mit Hilfe der vorhandenen Beschreibungsmittel beliebige Ausdrücke bilden und ihnen einen neuen Namen geben. Durch

eine solche Abkürzung können nicht nur neue Funktionen, sondern auch neue Kontrollstrukturen, die Funktionen höherer Ordnung sind, per definitionem eingeführt werden. Diese neuen Konstrukte stellen eine Erweiterung der Sprache Gropius dar und können in verschiedenen Schaltungsbeschreibungen wiederverwendet werden. Damit wird auch deutlich, warum es keinen Nachteil darstellt, die Sprache auf einem möglichst kleinen Kern zu definieren. Aufgrund der Erweiterungsmöglichkeiten, über die der Benutzer verfügt, ist es für ihn möglich, beliebige Konstrukte zu definieren, die ihm die Beschreibung von Schaltungen erleichtern (siehe dazu auch Abschnitt 5.1.6).

Gropius unterstützt Polymorphie. Die Operanden einer polymorphen Funktion können mehr als einen Typen haben. Solche Funktionen können zur Beschreibung allgemeiner Schaltungen verwendet werden. Durch unterschiedliche Instantiierung der unbestimmten Typen können auf diese Weise unterschiedliche Komponenten erzeugt werden, die auf derselben Beschreibung aufbauen.

Gropius ist eine Programmiersprache höherer Ordnung. Dies bedeutet, dass Funktionen (Schaltungen) als Argumente verwendet werden können, dass Funktionen selbst ein Resultat einer Funktion sein können und dass Funktionen als Variablen gespeichert werden können. Aufgrund der Tatsache, dass in Gropius Schaltungen als Funktionen beschrieben werden, können somit Schaltungen wiederverwendet werden, indem sie als Parameter anderer Schaltungen dienen.

Eine weitere Eigenschaft, die die Wiederverwendung von Schaltungsbeschreibungen erleichtert, ist die systematische Unterstützung von Regularität. Dies wird ausführlich in [Eise99] erläutert. In Gropius gibt es eine elementare generische, polymorphe Struktur, aus der durch Instantiierung die verschiedensten Strukturen abgeleitet werden können.

Wie in Abbildung 4.1 gezeigt, stellen DFG-Terme die Basis sowohl für Beschreibungen auf der Systemebene, als auch für Beschreibungen auf der algorithmischen und der RT-Ebene dar. Damit ermöglicht Gropius eine Wiederverwendung von Beschreibungen auch über Abstraktionsebenen hinweg. Derselbe DFG-Term kann auf verschiedenen Abstraktionsebenen in verschiedenen Kontexten wiederverwendet werden.

Auf der algorithmischen Ebene setzen sich Schaltungsbeschreibungen aus zwei Komponenten zusammen: einer für die funktionale und einer für die zeitliche Beschreibung (siehe Abbildung 4.1). Ein Vorteil dieser strikten Trennung von funktionalem und zeitlichem Verhalten liegt darin, dass funktionale Beschreibungen mit beliebigen zeitlichen Beschreibungen ohne Veränderung kombiniert werden können. Dieser Aspekt wird im nächsten Kapitel noch ausführlicher erläutert.

Schließlich ermöglicht Gropius auf der Systemebene die Wiederverwendung ganzer Prozessbeschreibungen. Durch die Verwendung eines ganz speziellen Kommunikationsschemas auf der Systemebene können bereits existierende Prozesse ohne Veränderung bei der Beschreibung neuer Systeme wiederverwendet werden. Darüber hinaus gibt es sogenannte K-Prozesse (siehe Abbildung 4.1), die ein für allemal definiert wurden und in verschiedenen Systemen eingesetzt werden können.

4.5 Kein Unterschied zwischen externem und internem Format

Konventionelle Syntheseprogramme unterscheiden üblicherweise zwischen interner Schaltungsdarstellung und der Anbindung von Hardwarebeschreibungssprachen. Bei der Synthese muss infolgedessen die vom Benutzer eingegebene Hardwarebeschreibung zunächst in das interne Format konvertiert werden (front-end). Auf dem meist komplizierten internen Format wird dann die eigentliche Schaltungssynthese durchgeführt. Dieses interne Format verfügt oft über komplexe Datenstrukturen, die für Außenstehende nicht mehr zu überblicken sind. Diese Vorgehensweise hat auf der einen Seite zwar den Vorteil einer höheren Effizienz, die Fehleranfälligkeit solcher Programme steigt dadurch aber nicht unerheblich. Das Ergebnis der Berechnung im internen Format wird schließlich durch eine weitere Konvertierung (back-end) zurück in eine Hardwarebeschreibungssprache übersetzt.

Ein wesentlicher Vorteil von Gropius liegt darin, dass es keine Unterscheidung zwischen internem und externem Format mehr gibt, denn die intern wie extern verwendeten Datenstrukturen sind bereits Terme der formalen Sprache Gropius. Die Synthesetransformationen werden direkt auf die vom Benutzer spezifizierten Schaltungsbeschreibungen angewandt. Dadurch entfallen die fehleranfälligen Konvertierungen zwischen interner und externer Darstellung.

Kapitel 5

Schaltungsbeschreibungen auf der algorithmischen Ebene – Gropius-2

Auf der algorithmischen Ebene wird das Verhalten der zu synthetisierenden Schaltung in Form einer algorithmischen Beschreibung dargestellt. Wenn mit einer Schaltungsbeschreibung auf dieser Ebene gestartet wird, ist keinesfalls festgelegt, dass die Schaltung durch Hardware-Komponenten realisiert werden muss. Sie könnte vielmehr genauso gut durch ein Software-Programm oder eine Kombination aus Hardware und Software realisiert werden. Diese Arbeit soll sich jedoch auf die Synthese von Hardware beschränken.

Während auf der RT-Ebene eine genaue zeitliche Zuordnung der Operationen auf einzelne Takte definiert ist, wird auf der algorithmischen Ebene von einer solchen zeitlichen Zuordnung abstrahiert. Dieser Abstraktionsschritt ist insbesondere relevant vor dem Hintergrund komplexer Schaltungen, deren konkretes zeitliches Verhalten flexibel anpassbar sein soll.

Der Übergang von der algorithmischen Ebene zur RT-Ebene wird als High-Level-Synthese bezeichnet. Dieser Übergang ist nicht eindeutig, denn aus einer algorithmischen Beschreibung können i.Allg. sehr verschiedene Implementierungen auf RT-Ebene abgeleitet werden. Da die algorithmische Beschreibung von der zeitlichen Einordnung der verwendeten Operationen abstrahiert, sind während der High-Level-Synthese die Freiheitsgrade groß, die Operationen in verschiedener Weise zeitlich einzuordnen.

Die möglichen Implementierungen unterscheiden sich also vor allem in ihrem zeitlichen Verhalten. Dieses zeitliche Verhalten umfasst mehrere Aspekte. Auf der RT-Ebene wird genau festgelegt, wann die Eingangswerte auf den Signalleitungen gelesen werden, wann welche Operation ausgeführt wird und wann die Ausgangswerte geschrieben werden. Darüber hinaus wird festgelegt, ob und in welcher Weise die Schaltung ihrer Umgebung mitteilt, ob sie gerade eine Berechnung ausführt oder nicht.

Die algorithmische Beschreibung drückt aus, welcher funktionale Zusammenhang

zwischen Eingangswerten und Ausgangswerten der Schaltung besteht. Zeit wird hier nicht betrachtet. Um den Abstraktionsunterschied zwischen algorithmischer und RT-Ebene zu überwinden, ist noch eine zweite Beschreibung notwendig: das Schnittstellenverhalten. Es beschreibt das zeitliche Verhalten der Signale an den Schnittstellen der Schaltung und gibt an, wie die Schaltung mit ihrer Umgebung kommuniziert.

In dem hier vorgestellten Ansatz zur High-Level-Synthese wird zwischen diesen beiden Teilen strikt unterschieden. Sie sind voneinander entkoppelt und können getrennt angegeben werden. In den allermeisten klassischen Ansätzen zur High-Level-Synthese sind algorithmische Beschreibung und Schnittstellenverhaltensbeschreibung jedoch miteinander verwoben. Es ist entweder implizit ein einziges Schnittstellenverhalten vorgegeben, oder das Schnittstellenverhalten kann in beliebiger Weise in die algorithmische Beschreibung integriert werden. In [BRSM97] wird ebenfalls eine Methode vorgestellt, die eine getrennte Beschreibung der beiden Aspekte vorsieht. Die Trennung von funktionalem Verhalten und Schnittstellenverhalten hat mehrere gute Gründe: Abstraktion, Flexibilität, Wiederverwertbarkeit und Verständlichkeit.

Durch die Trennung von funktionalen und zeitlichen Aspekten gelangt man zu einer wirklichen Abstraktion der Schaltungsbeschreibung. Der Entwurf beginnt mit einer rein algorithmischen Beschreibung, ebenso wie im Software-Entwurf. Als algorithmische Beschreibung fungiert immer eine Funktion, die einen Zusammenhang zwischen Ein- und Ausgangswerten herstellt. Zu diesem Zeitpunkt ist die Beschreibung in keiner Weise Hardware-spezifisch. Sie könnte genauso gut Ausgangspunkt für eine Software-Lösung sein. Es gibt weder Signale, noch Zeit, noch Register. In konventionellen Ansätzen ist dies anders. So impliziert etwa eine VHDL-Beschreibung stets ein bestimmtes Zeitmodell. VHDL ist daher für eine abstrakte Beschreibung auf der algorithmischen Ebene gänzlich ungeeignet.

Nach der Programmierung des funktionalen Verhaltens kann der Entwerfer dann von der Software-Sicht zur Hardware-Sicht wechseln, indem er ein Schnittstellenverhalten definiert. Das Schnittstellenverhalten bestimmt das Verhalten der späteren RT-Ebenen-Schaltung, die die Funktion für die algorithmische Beschreibung in einer spezifischen Weise ausführt. Für die Beschreibung des Schnittstellenverhaltens kann er sich dafür unter einer fest vorgegebenen Menge sogenannter Schnittstellenverhaltensmuster eines auswählen. Es ist also insbesondere möglich, denselben Algorithmus für das funktionale Verhalten mit unterschiedlichen Schnittstellenbeschreibungen zu kombinieren. Damit können Algorithmen in flexibler Weise an neue zeitliche Anforderungen angepasst werden. Darüber hinaus wird durch die orthogonale Behandlung von funktionalem Verhalten und Schnittstellenverhalten die Wiederverwertbarkeit von Beschreibungen in systematischer Weise unterstützt. Ein Algorithmus kann ohne Veränderung mit verschiedenen Schnittstellenbeschreibungen kombiniert werden. Sind dagegen die beiden Beschreibungen miteinander verwoben, muss die Beschreibung von Grund auf neu begonnen werden, wenn ein anderes Schnittstellenverhalten nötig ist.

Die Vernetzung von Algorithmus und Schnittstellenverhalten hat auf der ande-

ren Seite den Vorteil, dass sehr komplexe Schnittstellenverhaltensbeschreibungen realisiert werden können, die an die jeweilige Anforderung genau angepasst sind. Andererseits sind solche Beschreibungen oft nur schwer zu beherrschen und zu verstehen. Die in diesem Ansatz vorgestellten Schnittstellenverhaltensmuster können dagegen in genereller Weise eingesetzt werden. Im übrigen können solche Beschreibungen konventioneller Ansätze, bei denen funktionales und zeitliches Verhalten verwoben sind, in gewissem Umfang in Gropius auf der Systemebene repräsentiert werden, wie in Kapitel 6 gezeigt wird.

Außerdem zeigt sich, dass durch die besagte Trennung nicht nur die Verständlichkeit der Beschreibungen gefördert wird, sondern sich auch eine Vereinfachung des Entwurfs sowie des Korrektheitsbeweises ergibt.

Im Folgenden soll zunächst beschrieben werden, wie das funktionale Verhalten in der Hardwarebeschreibungssprache Gropius repräsentiert wird. Anschließend werden die Schnittstellenverhaltensmuster vorgestellt.

5.1 Funktionales Verhalten

In Gropius wird für die Repräsentation des funktionalen Verhaltens zwischen zwei Beschreibungsformen unterschieden. Zum einen werden *DFG-Terme* verwendet, bei denen es sich um einen Spezialfall algorithmischer Beschreibungen handelt. Sie entsprechen azyklischen Datenflussgraphen, deren Ausführung immer terminiert. Zum anderen werden *P-Terme* betrachtet, die zur Beschreibung μ-rekursiver Programme dienen. Die Menge der P-Terme stellt somit eine Programmiersprache dar, mit der beliebige berechenbare Funktionen ausgedrückt werden können, wie dies in herkömmlichen Programmiersprachen wie C oder Pascal möglich ist. Solche Programme können insbesondere im Gegensatz zu denen, die durch DFG-Terme repräsentiert werden, nicht terminieren. DFG-Terme und P-Terme sind klar syntaktisch voneinander getrennt, wobei DFG-Terme die Basis für P-Terme darstellen.

5.1.1 DFG-Terme

DFG-Terme repräsentieren Funktionen. Eine Funktion beschreibt eine Abbildung von einem beliebigen Datentyp α auf einen ebenso beliebigen Datentyp β. In der Notation des λ-Kalküls verwendet man λ-Abstraktionen, um Funktionen zu beschreiben (siehe die Syntaxregel (2.1) auf Seite 6). Die nachfolgende BNF (5.1) beschreibt den syntaktischen Aufbau von DFG-Termen:

Variablenstruktur ::=
    Variable |
    " (" *Variablenstruktur* " , " *Variablenstruktur* ") "

Ausdruck ::=
    Variablenstruktur |
    Konstante |
    " (" *Ausdruck* " , " *Ausdruck* ") " |
    " (" *Ausdruck Ausdruck* ") "

DFG-Term ::=
    " λ " *Variablenstruktur* " . "
    { " let " *Variablenstruktur* " = " *Ausdruck* " in " }
    Ausdruck

(5.1)

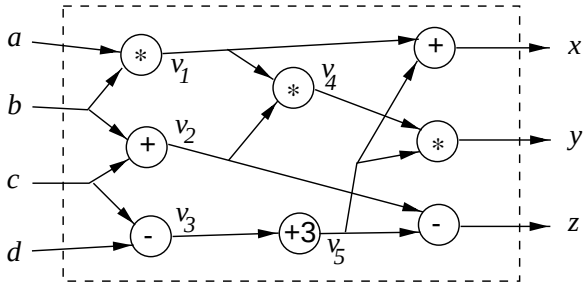
Eine Variablenstruktur besteht aus Variablen und Paaren von Variablen. In HOL gibt es keine n-Tupel, sondern nur Paare, die das kartesische Produkt zweier Terme bilden. n-Tupel werden durch verschachtelte Paare nachgebildet. Dabei gilt die Konvention, dass bei rechtsgeklammerten Paaren die Klammern weggelassen werden können. Für einen Term der Form $(x,(y,z))$ gilt also

$$(x,(y,z)) \;=\; (x,y,z)$$

Dagegen können im Term $((x,y),z)$ keine Klammern entfernt werden. Da HOL streng-typisiert aufgebaut ist, stellt es einen großen Unterschied dar, ob der Term $(x,(y,z))$ oder der Term $((x,y),z)$ vorliegt. Wenn im weiteren Verlauf der Arbeit beliebige n-Tupel $(x_1,\ldots,x_{n-1},x_n)$ verwendet werden, verbirgt sich dahinter also stets eine Variablenstruktur $(x_1,(\ldots,(x_{n-1},x_n)\ldots))$.

Als Parameter einer λ-Abstraktion kann nicht nur eine Variable verwendet werden, wie dies aus der Syntaxregel (2.1) auf Seite 6 hervorgeht. Mit Hilfe der in Abschnitt 2.6.1 vorgestellten Konstanten UNCURRY können auch ganze Variablenstrukturen als Parameter von λ-Abstraktionen dienen. Diese gepaarten λ-Abstraktionen werden durch den Parser des HOL-Systems in verschachtelte gewöhnliche λ-Abstraktionen umgewandelt. Der Parameter der λ-Abstraktion in (5.1) stellt zugleich den Eingang des DFG-Terms dar. Als Eingänge von DFG-Termen können also beliebige Variablenkombinationen verwendet werden, die einen beliebigen Eingangsdatentyp α realisieren.

Ein DFG-Term, wie er in (5.1) definiert ist, bildet eine Variablenstruktur auf einen Ausdruck ab. Er beschreibt eine Ein-Ausgangsfunktion durch eine Komposition der zugrunde liegenden Operationen. Die Operationen entsprechen dabei den Knoten des Datenflussgraphen und werden durch let-Terme aufgelistet. Wie in Abschnitt 2.6.1 beschrieben, stellen let-Terme eine andere Schreibweise für β-Redizes dar, die die Lesbarkeit erhöht. Abbildung 5.1 zeigt ein einfaches Beispiel eines Datenflussgraphen nebst zugehörigem DFG-Term. Der DFG-Term beschreibt eine Funktion vom Typ $(\mathsf{num}\times\mathsf{num}\times\mathsf{num}\times\mathsf{num})\to(\mathsf{num}\times\mathsf{num}\times\mathsf{num})$. Der



(a) Datenflussgraph

$$
\begin{array}{l}
\lambda(a,b,c,d).\\
\quad \mathsf{let}\ v_1 = a * b \quad \mathsf{in}\\
\quad \mathsf{let}\ v_2 = b + c \quad \mathsf{in}\\
\quad \mathsf{let}\ v_3 = c - d \quad \mathsf{in}\\
\quad \mathsf{let}\ v_4 = v_1 * v_2 \quad \mathsf{in}\\
\quad \mathsf{let}\ v_5 = v_3 + 3 \quad \mathsf{in}\\
\quad \mathsf{let}\ x \ = v_1 + v_5 \quad \mathsf{in}\\
\quad \mathsf{let}\ y \ = v_4 * v_5 \quad \mathsf{in}\\
\quad \mathsf{let}\ z \ = v_2 - v_5 \quad \mathsf{in}\\
\quad\ (x,y,z)
\end{array}
$$

(b) DFG-Term

Abbildung 5.1: Beispiel eines DFG-Terms

DFG-Term hat eine Variablenstruktur bestehend aus vier Variablen als Eingang und eine Variablenstruktur mit drei Variablen als Ausgang. Der Datenflussgraph besitzt acht Knoten, die den Teiloperationen der Funktion entsprechen. Jede Operation besteht aus zwei Teilen. Der erste Teil ist eine Variablenstruktur, die den Ausgang der Operation angibt. Der zweite Teil wird durch einen Ausdruck beschrieben, der angibt, wie der Eingang durch die Operation verändert wird. Ein Ausdruck kann gebildet werden aus einer Variablenstruktur, einer Konstante, einem Paar zweier Ausdrücke und aus einer Funktionsanwendung zweier Ausdrücke. Der Ausdruck $(a * b)$ etwa entsteht durch Funktionsanwendung der Konstanten '$*$' auf die Variable a und nochmalige Funktionsanwendung des Ausdruckes $(* a)$ auf die Variable b. Da '$*$' in Infix-Notation verwendet wird, ist der Ausdruck $((* a)\, b)$ äquivalent dem Ausdruck $(a * b)$. Auch wenn in Abbildung 5.1 alle Operationen nur einen Ausgang haben, können grundsätzlich auch Operationen mit mehreren Ausgängen verwendet werden.

Die Reihenfolge, in der die Operationen aufgelistet werden können, ist i.Allg. nicht eindeutig. Durch die Datenabhängigkeiten der Operationen ergibt sich aber eine partielle Ordnung. Die Ausgänge der Teiloperationen des DFG-Terms entsprechen lokalen Variablen, die für nachfolgende Operationen als Eingangsvariablen fungieren können. Zu Beginn stehen nur die Eingänge der Gesamtfunktion zur Verfügung. Durch jede Operation wird die Menge der verwendbaren Eingangsvariablen um die Ausgangsvariablen der Operation erweitert. Eine Operation darf erst dann im DFG-Term durch einen let-Term aufgeführt werden, wenn alle ihre Eingänge bereits eingeführt sind. Da durch die (gepaarte) λ-Abstraktion zu Beginn des DFG-Terms die Eingangsvariablen und durch einen let-Term $\mathsf{let}\ x = y\ \mathsf{in}\ z$ die Ausgangsvariablen x einer Operation gebunden werden, ist dies gleichbedeutend mit der Forderung, dass in einem DFG-Term keine freien Variablen vorkommen dürfen. Zur Veranschaulichung ist in Abbildung 5.2(a) ein korrekter und in Abbildung 5.2(c)

$$
\begin{array}{l}
\lambda(a,b,c,d). \\
\quad \mathsf{let}\ v_2 = b + c \quad \mathsf{in} \\
\quad \mathsf{let}\ v_3 = c - d \quad \mathsf{in} \\
\quad \mathsf{let}\ v_5 = v_3 + 3 \quad \mathsf{in} \\
\quad \mathsf{let}\ z\ = v_2 - v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ v_1 = a * b \quad \mathsf{in} \\
\quad \mathsf{let}\ x\ = v_1 + v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ v_4 = v_1 * v_2 \quad \mathsf{in} \\
\quad \mathsf{let}\ y\ = v_4 * v_5 \quad \mathsf{in} \\
\quad\ (x,y,z)
\end{array}
$$

(a) Richtig

$$
\begin{array}{l}
\lambda(a,b,c,d). \\
\quad \mathsf{let}\ v_1 = a * b \quad \mathsf{in} \\
\quad \mathsf{let}\ v_2 = b + c \quad \mathsf{in} \\
\quad \mathsf{let}\ v_5 = (c - d) + 3 \quad \mathsf{in} \\
\quad \mathsf{let}\ x\ = v_1 + v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ y\ = (v_1 * v_2) * v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ z\ = v_2 - v_5 \quad \mathsf{in} \\
\quad\ (x,y,z)
\end{array}
$$

(b) Richtig

$$
\begin{array}{l}
\lambda(a,b,c,d). \\
\quad \mathsf{let}\ v_1 = a * b \quad \mathsf{in} \\
\quad \mathsf{let}\ v_4 = v_1 * v_2 \quad \mathsf{in} \quad \leftarrow \\
\quad \mathsf{let}\ v_2 = b + c \quad \mathsf{in} \\
\quad \mathsf{let}\ v_3 = c - d \quad \mathsf{in} \\
\quad \mathsf{let}\ v_5 = v_3 + 3 \quad \mathsf{in} \\
\quad \mathsf{let}\ x\ = v_1 + v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ y\ = v_4 * v_5 \quad \mathsf{in} \\
\quad \mathsf{let}\ z\ = v_2 - v_5 \quad \mathsf{in} \\
\quad\ (x,y,z)
\end{array}
$$

(c) Falsch

Abbildung 5.2: DFG-Terme zu Abbildung 5.1

$$
\begin{array}{l}
\lambda(a,b,c,d). \\
\quad (\\
\quad\quad ((a * b) + ((c - d) + 3)), \\
\quad\quad (((a * b) * (b + c)) * ((c - d) + 3)), \\
\quad\quad ((b + c) - ((c - d) + 3)) \\
\quad)
\end{array}
$$

Abbildung 5.3: DFG-Term in Normalform

ein unzulässiger DFG-Term für den in Abbildung 5.1(a) dargestellten Datenflussgraphen angegeben. Im falschen DFG-Term wird die Variable v_2 verwendet, bevor sie durch die entsprechende Operation erzeugt wird.

Nicht nur die Reihenfolge, in der die Operationen aufgeführt werden, ist in der Regel nicht eindeutig zur Beschreibung eines DFG-Terms. Es ist auch möglich, nicht alle Operationen explizit mit ihrem Ausgang als Zwischenergebnis einzuführen, sondern mehrere Operationen zusammenzufassen. In Abbildung 5.2(b) ist ein solcher ebenfalls zulässiger DFG-Term dargestellt. Dieser DFG-Term lässt sich aus dem DFG-Term in Abbildung 5.1(b) gewinnen, indem auf die let-Terme mit den Parametern v_3 und v_4 eine β-Konversion angewandt wird (siehe Abschnitt 2.1.2). Auch hier müssen natürlich die Datenabhängigkeiten beachtet werden, d.h. alle Variablen müssen gebunden sein.

Wird auf alle β-Redizes eine β-Konversion angewandt, erhält man die Normalform des DFG-Terms (siehe Abschnitt 2.1.2). Diese Normalform ist eindeutig für alle DFG-Terme, die denselben Datenflussgraphen repräsentieren. Somit lässt sich aus den drei DFG-Termen in den Abbildungen 5.1(b), 5.2(a) und 5.2(b) derselbe DFG-Term gewinnen. Diese Normalform ist in Abbildung 5.3 gezeigt. In der

Normalform sind alle Operationen zusammengefasst. Es gibt keine Zwischenergebnisse mehr, die mit lokalen Variablen bezeichnet werden. Die Zusammenfassung von Operationen ist von Vorteil, um im DFG-Term Umformungen über die Operationen hinweg durchzuführen. So kann etwa in der Normalform in Abbildung 5.3 der dritte Ausgang umgeformt werden zu $((b + d) + 3)$.

Der Nachteil der Zusammenfassung von Operationen besteht erstens darin, dass die Struktur des Datenflussgraphen schwerer nachvollzogen werden kann. Darüber hinaus werden Zwischenergebnisse, die mehrfach als Eingang anderer Operationen verwendet werden, auch mehrfach aufgeführt. Diese Situation wird noch verschärft, wenn die Ergebnisse nachfolgender Operationen ebenfalls mehrfach als Eingänge verwendet werden. Man spricht dabei von einem Fanout, der größer als eins ist. Der DFG-Term kann somit exponentiell wachsen. Diese beiden Nachteile lassen sich an der Normalform besonders gut nachvollziehen. Diese Nachteile werden vermieden, wenn nur die β-Redizes solcher Operationen ausgewertet werden, deren Ausgänge nur einmal verwendet werden.

Zusammenfassend lässt sich sagen, dass die Formalisierung von DFG-Termen bei weitem nicht eindeutig ist. Der Entwerfer hat viele Freiheitsgrade. Die einzige Bedingung, die für die Beschreibung von DFG-Termen erfüllt werden muss, ist die, dass die Datenabhängigkeiten berücksichtigt sein müssen, was gleichbedeutend mit der Forderung ist, dass der DFG-Term geschlossen sein muss.

Operationen

Die Teiloperationen, die in DFG-Termen verwendet werden, terminieren immer. Somit terminiert auch die Gesamtfunktion, die durch einen DFG-Term repräsentiert wird. Als Operationen können in DFG-Termen nur solche verwendet werden, die zur Realisierung als Hardware-Komponenten geeignet sind. Die Menge solcher Operationen ist daher kleiner als die, die im Software-Entwurf zur Verfügung steht. Operationen mit Zeigern, unendlichen Listen etc. sind für den Hardware-Entwurf nicht geeignet, da hier immer eine feste Verdrahtung und eine statische Anzahl von Komponenten und Signalen vorliegen muss.

Wie in Abbildung 4.1 auf Seite 44 gezeigt, können zur Bildung von DFG-Termen sowohl boolesche als auch abstrakte Operatoren verwendet werden. Die Operatoren werden dabei als Konstanten eingeführt, wie es in Abschnitt 2.4 ausgeführt wurde. Während boolesche Operatoren rein boolesche Eingänge auf rein boolesche Ausgänge abbilden, verfügen abstrakte Operatoren über Ein- und Ausgänge mit beliebigen Datentypen.

Spezielle DFG-Terme

Prinzipiell haben DFG-Terme einen beliebigen Typ $\alpha \rightarrow \beta$. Im Weiteren sollen aber zwei Spezialfälle besondere Beachtung finden, die in Tabelle 5.1 aufgeführt sind. Ein *Elementarblock* sei ein DFG-Term, dessen Ein- und Ausgangsdatentypen übereinstimmen. Eine *Bedingung* produziert auf die Eingabe eines beliebigen Da-

Bezeichner	Typ	Beschreibung
DFG-Term	$\alpha \rightarrow \beta$	Datenflussgraph
Elementarblock	$\alpha \rightarrow \alpha$	spezieller DFG-Term
Bedingung	$\alpha \rightarrow$ bool	spezieller DFG-Term

Tabelle 5.1: DFG-Terme

tentyps eine boolesche Ausgabe. Diese beiden Spezialfälle von DFG-Termen werden zur Beschreibung von P-Termen benötigt (siehe Abschnitt 5.1.4). Im Folgenden werden DFG-Terme immer mit Kleinbuchstaben und P-Terme mit Großbuchstaben bezeichnet.

5.1.2 Einschub: Schaltungsbeschreibungen auf der RT-Ebene – Gropius-1

Das Ergebnis der High-Level Synthese ist eine Struktur auf der RT-Ebene. Schaltungsbeschreibungen auf der RT-Ebene basieren auf DFG-Termen. Daher soll an dieser Stelle kurz erläutert werden, wie Schaltungen auf der RT-Ebene in Gropius repräsentiert werden. Eine ausführliche Beschreibung findet sich in [Eise99].

Schaltungen werden in Gropius-1 durch Automaten dargestellt. Gewöhnlich werden Automaten durch ein 6-Tupel beschrieben, das aus Eingabealphabet, Ausgabealphabet, Zustandsmenge, Ausgangsfunktion, Übergangsfunktion und Anfangszustand besteht. In Gropius dagegen wird ein Automat nur durch ein Paar (f, q) repräsentiert, wobei q der Initialzustand vom Typ σ und f eine zusammengefasste Aus- und Übergangsfunktion ist. Eine solche Funktion f wird dabei durch einen DFG-Term beschrieben, der den Typ $\iota \times \sigma \rightarrow \omega \times \sigma$ besitzt. ι, ω und σ sind die Typen des Eingabealphabets, des Ausgabealphabets und der Zustandsmenge. Aufgrund der Tatsache, dass in Gropius nur streng-typisierte Ausdrücke verwendet werden, ist es nicht notwendig, explizit Ein- und Ausgabealphabet bzw. Zustandsmenge anzugeben. Die Verwendung einer gemeinsamen Aus- und Übergangsfunktion ist deshalb sinnvoll, da es i.Allg. nicht möglich ist, in eindeutiger Weise kombinatorische Operationen entweder der Ausgangs- oder der Übergangsfunktion zuzuordnen.

Durch f und q wird das Verhalten eines Automaten in eindeutiger Weise bestimmt. Durch eine Funktion höherer Ordnung namens automaton wird die Semantik eines Automaten beschrieben, indem das Paar (f, q) auf eine Funktion automaton(f, q) abgebildet wird. Die Funktion automaton(f, q) bildet dabei ein zeitabhängiges Eingangssignal (Typ num $\rightarrow \iota$) auf ein zeitabhängiges Ausgangssignal (Typ num $\rightarrow \omega$) ab. Abbildung 5.4 skizziert, wie sequentielle Schaltungen auf RT-

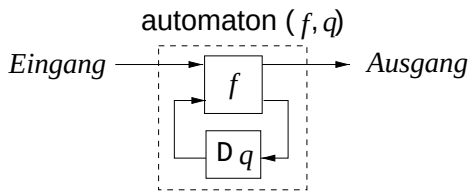


Abbildung 5.4: Schaltung auf RT-Ebene

Ebene, die sich aus einer kombinatorischen Komponente f und einer Speichereinheit (D) mit Initialwert q zusammensetzen, aufgebaut sind. Es ist anzumerken, dass auf der RT-Ebene Zyklen nur über Speicherbausteine geführt werden dürfen. Durch diese syntaktische Begrenzung werden verzögerungsfreie Zyklen ausgeschlossen. Auch Kurzschlüsse werden syntaktisch ausgeschlossen, da in DFG-Termen Kurzschlüsse nicht darstellbar sind. Es werden somit inkonsistente Schaltungsbeschreibungen, wie sie typischerweise bei relationalen Darstellungen entstehen können, a priori vermieden.

5.1.3 Primitive Rekursion

Alle Funktionen, die mit DFG-Termen beschrieben werden können, sind total. Das bedeutet, dass für jede Eingangsbelegung der Funktion ein Ergebnis definiert ist. Die Sprache der DFG-Terme ist sehr einfach. Sie beschreibt eine Menge von Funktionen, die eine echte Untermenge der *loop-berechenbaren* oder *primitiv-rekursiven* Funktionen darstellt. Diese wiederum sind eine echte Untermenge der *while-berechenbaren* oder *μ-rekursiven* Funktionen. In Gropius werden P-Terme dazu verwendet, um μ-rekursive Funktionen zu beschreiben, es gibt keine gesonderte Beschreibungsform für primitiv-rekursive Funktionen. Da alle primitiv-rekursiven Funktionen auch μ-rekursive sind, können durch P-Terme auch primitiv-rekursive Funktionen ausgedrückt werden.

Loop-berechenbare Funktionen erweitern die Klasse der durch DFG-Terme darstellbaren Funktionen um das Konstrukt der *for*-Schleife. Eine *for*-Schleife for i loop a end, wie man sie aus prozeduralen Programmiersprachen kennt, iteriert i-mal eine Funktion a. Bei dem Ausdruck i kann es sich dabei um eine Variable oder eine Konstante handeln. *for*-Schleifen lassen sich nur dann in DFG-Terme durch i-maliges Aufrollen der Schleife überführen, wenn es sich bei i um eine Konstante handelt. *for*-Schleifen mit variablem i stellen eine echte Erweiterung zu DFG-Termen dar.

Die Klasse der primitiv-rekursiven Funktionen ist der Klasse der loop-berechenbaren Funktionen äquivalent. Diese Funktionen lassen sich ineinander überführen. Bei den primitiv-rekursiven Funktionen werden rekursive Unterprogrammaufrufe nach einem gewissen Rekursionsschema verwendet. Das Rekursionsschema über den Datentyp num sieht wie folgt aus: a und f seien primitiv-rekursive Funktionen, dann ist auch die Funktion g primitiv-rekursiv, die nach folgendem

Schema definiert ist:

$$\begin{aligned} g\ 0 &= a \\ g\ (\mathsf{SUC}\ n) &= f\ (g\ n)\ n \end{aligned} \tag{5.2}$$

Zur Berechnung von $(g\ n)$ spielt der Wert n die Rolle des Zählers für die Anzahl der Rekursionsschritte. Bei der Umsetzung in eine loop-berechenbare Funktion entspricht er also dem Schleifenzähler.

In HOL kann ausgehend von den Konstruktoren eines Datentyps ein Theorem abgeleitet werden, das die eindeutige primitive Rekursion über den Datentyp beschreibt. Aufbauend auf diesem Prinzip steht in Gropius für jeden Datentyp typ eine elementare Hilfsfunktion PRIMREC_typ zur Verfügung, die durch Konstantenspezifikation in konsistenzerhaltender Weise automatisch eingeführt wird (siehe [EiSK93]). Bei bestimmten Datentypen stellt diese Funktion gerade das Rekursionsschema für den Datentyp dar. Die Funktion PRIMREC_num etwa ist wie folgt definiert:

$$\begin{aligned} \vdash\ \forall a\ f.\ &\mathsf{PRIMREC\_num}\ 0\ a\ f\ =\ a\ \wedge \\ &\forall n.\ \mathsf{PRIMREC\_num}\ (\mathsf{SUC}\ n)\ a\ f\ =\ f\ (\mathsf{PRIMREC\_num}\ n\ a\ f)\ n \end{aligned} \tag{5.3}$$

Bei anderen Datentypen handelt es sich um eine Analogie, da das Schema zu einer Fallunterscheidung entartet ist. Ein Beispiel ist in Theorem (5.7) auf Seite 58 zu sehen. Die PRIMREC-Funktionen stellen die Basis dar, um die Brücke von den nicht-rekursiven Funktionen zu den primitiv-rekursiven Funktionen zu schlagen.

5.1.4 μ-rekursive Funktionen

Durch Hinzufügen des Konstrukts der for-Schleife zu den durch die DFG-Terme beschreibbaren Algorithmen gelangten wir zu loop-berechenbaren oder primitiv-rekursiven Funktionen. Erweitert man nun die Menge der loop-Algorithmen durch das Konstrukt der while-Schleife, gelangt man zu den while-berechenbaren Funktionen. Die Menge der while-berechenbaren Funktionen ist dabei echt größer als die Menge der loop-berechenbaren Funktionen, da sie auch partielle Funktionen umfasst. Insbesondere sind also alle loop-Algorithmen auch while-Algorithmen.

Auch der Übergang von primitiv-rekursiven Funktionen zu μ-rekursiven Funktionen, die den while-Algorithmen entsprechen, ist möglich. Wie wir gesehen haben, entspricht bei Verwendung des Rekursionsschemas (5.2) bei der Berechnung von $(g\ n)$ der Wert n dem Schleifenzähler in einem loop-Algorithmus. Die Verallgemeinerung zu while-Algorithmen erreicht man, wenn man n solange verändert, bis eine bestimmte Bedingung $(f\ n)$ erfüllt ist. Dazu lässt sich der μ-*Operator* wie folgt definieren:

$$\mu\ f\ =\ \min_n \{n \in \mathbf{N}\ |\ (f\ n)\ =\ \mathsf{T}\ \text{ und für alle } u \leq n \text{ ist } (f\ u) \text{ definiert.}\} \tag{5.4}$$

Die Funktion $(\mu\ f)$ ist undefiniert, falls kein derartiges n existiert. Man schreibt dann $(\mu\ f)\uparrow$.

Erweitert man die Beschreibungsmittel für primitiv-rekursive Funktionen um den μ-Operator, so erhält man die μ-rekursiven Funktionen. Es lässt sich zeigen, dass eine Funktion genau dann μ-rekursiv ist, wenn sie while-berechenbar ist [Goos97c].

Die Menge der μ-rekursiven Funktionen umfasst ebenso wie die Menge der while-berechenbaren Funktionen die Menge aller berechenbaren Funktionen. Es gibt mehrere äquivalente Modelle, um beliebige berechenbare Funktionen zu beschreiben. μ-rekursive und while-berechenbare Funktionen sind zwei davon. Ein weiteres Modell ist die Turing-Maschine, die zu der Menge der turing-berechenbaren Funktionen führt.

Es soll nun vorgestellt werden, wie in Gropius μ-rekursive Funktionen beschrieben werden können. Im Gegensatz zu den durch DFG-Terme beschreibbaren Funktionen, die immer terminieren, kann es bei P-Termen, die μ-rekursive Funktionen beschreiben, passieren, dass sie nicht terminieren. Zur Beschreibung der Funktionswerte von P-Termen wurde der Typoperator partial eingeführt. Der Datentyp (α)partial verfügt über die beiden Konstruktoren Defined : $\alpha \rightarrow (\alpha)$partial und Undefined : (α)partial:

$$\mathsf{partial} \ = \ \mathsf{Defined}\ \text{of}\ \alpha \mid \mathsf{Undefined}$$

Während Funktionen, die durch DFG-Terme beschrieben werden, ganz allgemein den Typ $\alpha \rightarrow \beta$ haben, verfügen P-Terme in Gropius über den Typ $\alpha \rightarrow (\beta)$partial, wobei der Typoperator partial nicht in α und β vorkommen darf. Die Elemente des Typs (β)partial haben entweder den Wert (Defined x), wobei x den Typ β hat, oder der Wert ist Undefined. Wenn eine μ-rekursive Funktion A auf einen Operanden y angewandt wird und das Ergebnis den Funktionswert (Defined x) hat, bedeutet dies, dass die Funktionsanwendung $(A\ y)$ terminiert und dass das Ergebnis gerade x ist. Ist das Ergebnis der Funktionswert Undefined, bedeutet das, dass die Funktionsanwendung $(A\ y)$ nicht terminieren wird. Im Gegensatz zu herkömmlichen Programmiersprachen, bei denen eine μ-rekursive Funktion bei Nichtterminierung keinen Wert zurückgibt, also nur partiell definiert ist, gibt es in Gropius also auch in diesen Fällen den konkreten Funktionswert Undefined. Auch μ-rekursive Funktionen sind deshalb in Gropius total.

Es stellt sich nun die Frage, wie man solche μ-rekursiven Funktionen simulieren kann, denn einen Interpreter, der immer den Funktionswert einer μ-rekursiven Funktion bestimmt, auch wenn diese nicht terminiert, kann es nicht geben (Halteproblem). Ein Interpreter könnte so implementiert werden, dass er immer den Wert (Defined x) einer Funktionsanwendung ermittelt, wenn diese terminiert, und dass er nicht terminiert, wenn der Funktionswert Undefined ist. Eine Verbesserung wäre es, wenn ein Interpreter in möglichst vielen Fällen erkennen würde, dass die Funktionsanwendung nicht terminiert – wenn sich etwa der Wert der Schleifenvariablen von ihrem Abbruchwert entfernt, anstatt ihm näher zu kommen. In solchen Fällen könnte der Interpreter terminieren und den Funktionswert Undefined ausgeben.

P-Terme werden dazu verwendet, beliebige berechenbare *Programme*, sowie *Blöcke* zu repräsentieren. Blöcke werden verwendet, um den inneren Teil eines Programms zu beschreiben. Im Gegensatz zu Programmen, die ganz allgemein über den Typ $\alpha \rightarrow (\beta)\mathsf{partial}$ verfügen, ist der Typ von Blöcken beschränkt auf $\alpha \rightarrow (\alpha)\mathsf{partial}$. Der Grund hierfür sind Schleifen. Um eine Funktion iterieren zu können, müssen Ein- und Ausgangstyp gleich sein. Dieser Zusammenhang ist in Tabelle 5.2 zusammengefasst.

Bezeichner	**Typ**	**Beschreibung**
P-Term	$\alpha \rightarrow (\beta)\mathsf{partial}$	berechenbare Funktion
Programm	$\alpha \rightarrow (\beta)\mathsf{partial}$	P-Term
Block	$\alpha \rightarrow (\alpha)\mathsf{partial}$	spezieller P-Term

Tabelle 5.2: P-Terme

P-Terme repräsentieren beliebige berechenbare Funktionen. Deshalb sind DFG-Terme durch P-Terme darstellbar. Der Übergang von DFG-Termen zu P-Termen geschieht durch die Funktion PARTIALIZE:

$$\vdash \mathsf{PARTIALIZE}\ a\ x\ =\ \mathsf{Defined}\ (a\ x) \tag{5.5}$$

Ein Elementarblock $a : \alpha \rightarrow \alpha$ wird dabei in einen Block $(\mathsf{PARTIALIZE}\ a) : \alpha \rightarrow (\alpha)\mathsf{partial}$ umgewandelt. $(\mathsf{PARTIALIZE}\ a)$ ist eine Funktion, die einen Eingangswert x in den Ausgangswert $(\mathsf{Defined}\ (a\ x))$ abbildet. Da diese Funktion einen DFG-Term auf die Beschreibungsebene von P-Termen überführt, terminiert sie immer. Der Wert Undefined wird nie erreicht.

Bei der Einführung des Datentyps partial in HOL wird automatisch das folgende Theorem abgeleitet, das die Semantik von partial beschreibt.

$$\vdash\ \forall a\ f.\ \exists^1 g.\ (g\ \mathsf{Undefined}\ =\ a)\ \wedge (\forall n.\ g\ (\mathsf{Defined}\ n)\ =\ f\ n) \tag{5.6}$$

Dieses Theorem beschreibt ausgehend von den beiden Konstruktoren, dass die primitive Rekursion über den Datentyp partial eindeutig ist. Ausgehend von diesem Theorem lässt sich nun die entsprechende Funktion PRIMREC_partial definieren:

$$\begin{aligned}\vdash\ \forall a\ f.\ \forall x.\ &\mathsf{PRIMREC\_partial}\ (\mathsf{Defined}\ x)\ f\ a\ =\ (f\ x)\ \wedge \\ &\mathsf{PRIMREC\_partial}\ \mathsf{Undefined}\ f\ a\ =\ a\end{aligned} \tag{5.7}$$

Mit Hilfe von PRIMREC_partial wird nun die Funktion WHILE definiert, die die Basis zur Beschreibung μ-rekursiver Funktionen in Gropius darstellt. Die Semantik

von WHILE selbst basiert auf den Funktionen iota, terminates, mu und power. Diese Funktionen sind im Zusammenhang mit der Entwicklung der Sprache Gropius definiert worden. Sie gehören aber nicht zum Kern der Sprache, sondern dienen lediglich als Hilfsmittel. Alle diese Funktionen werden durch Konstantendefinitionen eingeführt.

$$\vdash \mathsf{iota}\ a\ =\ (\exists^1 x.\ a\ x\ \Rightarrow\ \mathsf{Defined}\ (\varepsilon x.\ a\ x)\ |\ \mathsf{Undefined}) \tag{5.8}$$

$$\vdash \mathsf{terminates}\ (a, n)\ =\ (a\ n)\ \wedge\ (\forall m.\ m < n\ \Rightarrow\ \neg(a\ m)) \tag{5.9}$$

$$\vdash \mathsf{mu}\ a\ =\ \mathsf{iota}\ (\lambda m.\ \mathsf{terminates}\ (a, m)) \tag{5.10}$$

$$\begin{aligned}&\vdash \mathsf{power}\ A\ n\ x\ =\\ &\qquad \mathsf{PRIMREC\_num}\ n\ (\mathsf{Defined}\ x)\\ &\qquad\qquad\qquad (\lambda y\, z.\ \mathsf{PRIMREC\_partial}\ y\ A\ \mathsf{Undefined})\end{aligned} \tag{5.11}$$

$$\begin{aligned}&\vdash \mathsf{WHILE}\ c\ A\ x\ =\\ &\qquad \mathsf{PRIMREC\_partial}\\ &\qquad\qquad (\mathsf{mu}\ (\lambda n.\ \mathsf{PRIMREC\_partial}\ (\mathsf{power}\ A\ n\ x)\ (\lambda y.\ \neg(c\ y))\ \mathsf{F})\ )\\ &\qquad\qquad (\lambda n.\ \mathsf{power}\ A\ n\ x)\\ &\qquad\qquad \mathsf{Undefined}\end{aligned} \tag{5.12}$$

Die Funktion iota überträgt den Hilbert-Operator ε (siehe Abschnitt 2.4) in die Welt des Datentyps (α)partial. Es gibt aber einen entscheidenden Unterschied. Nur wenn es ein eindeutiges x gibt, sodass $(a\ x)$ erfüllt ist, wird der Wert Defined $(\varepsilon x.\ a\ x)$ zurückgegeben. Durch εx wird dabei genau dieses x bezeichnet. Gibt es mehrere Werte x, für die das Prädikat a erfüllt ist, ist das Ergebnis Undefined. Hier unterscheidet sich iota von ε, da der Hilbert-Operator in diesem Fall einen beliebigen dieser Werte herausgreift. Falls das Prädikat nicht erfüllt werden kann, ist das Ergebnis ebenfalls Undefined. Der Hilbert-Operator bezeichnet in diesem Fall einen unbekannten, aber festen Wert.

Die Funktion terminates (a, n) gibt an, dass n der kleinste Wert vom Typ num ist, der das Prädikat a erfüllt. terminates und iota dienen zur Beschreibung der Funktion mu, die an den μ-Operator (5.4) erinnert. Wenn das Prädikat a erfüllbar ist, liefert die Funktion (mu a) den Wert (Defined x), wobei x das kleinste Element des Typs num ist, der a erfüllt. Ist das Prädikat nicht erfüllbar, ist das Ergebnis von (mu a) der Wert Undefined. Die Funktion (mu a) ist also die totale Variante der partiellen Funktion $(\mu\ a)$.

Die Funktion power berechnet die n-fache Anwendung einer μ-rekursiven Funktion $A : \alpha \to (\alpha)$partial. Wenn das Ergebnis einer Funktionsanwendung den Wert (Defined y) hat, wird mit Hilfe der Funktion PRIMREC_partial der Wert y entnommen und dieser anschließend wieder auf die Funktion A angewandt. Das Ergebnis ist Undefined, wenn eine der n Funktionsanwendungen nicht terminiert.

Die Funktion WHILE basiert auf der Funktion mu. Sie hat drei Parameter: eine Bedingung c, einen Block A und den Eingangswert x. Mit Hilfe von mu wird die kleinste Anzahl von Schleifendurchläufen n ermittelt, bis ein Wert (Defined y)

erreicht wird, für den $\neg(c\ y)$ gilt. Anschließend wird der Block A mit Hilfe von power gemäß der ermittelten Anzahl von Schleifendurchläufen n-mal iteriert. Existiert ein solcher Wert n nicht, ist das Ergebnis Undefined.

Die Semantik von WHILE wird klarer, wenn man die folgenden zwei Theoreme betrachtet, die in HOL aus den Theoremen (5.7) bis (5.12) abgeleitet wurden:

$$
\begin{aligned}
&\vdash (\mathsf{WHILE}\ c\ A\ x\ =\ \mathsf{Defined}\ y)\ = \\
&\qquad (\exists t\ s.\ s\ 0\ =\ x\ \ \wedge \\
&\qquad\qquad \forall n.\ n < t\ \Rightarrow\ (A\,(s\ n)\ =\ \mathsf{Defined}\,(s\ (\mathsf{SUC}\ n)))\ \ \wedge \\
&\qquad\qquad \forall n.\ n < t\ \Rightarrow\ c\,(s\ n)\ \ \wedge \\
&\qquad\qquad s\ t\ =\ y\ \ \wedge \\
&\qquad\qquad \neg(c\,(s\ t))\ \ )
\end{aligned}
\tag{5.13}
$$

$$
\begin{aligned}
&\vdash (\mathsf{WHILE}\ c\ A\ x\ =\ \mathsf{Undefined})\ = \\
&\qquad (\exists s.\ s\ 0\ =\ x\ \ \wedge \\
&\qquad\qquad \forall n.\ (A\,(s\ n)\ =\ \mathsf{Defined}\,(s\ (\mathsf{SUC}\ n)))\ \ \wedge \\
&\qquad\qquad \forall n.\ c\,(s\ n)\ \ ) \\
&\qquad \vee \\
&\qquad (\exists t\ s.\ s\ 0\ =\ x\ \ \wedge \\
&\qquad\qquad \forall n.\ n < t\ \Rightarrow\ (A\,(s\ n)\ =\ \mathsf{Defined}\,(s\ (\mathsf{SUC}\ n)))\ \ \wedge \\
&\qquad\qquad \forall n.\ n < t\ \Rightarrow\ c\,(s\ n)\ \ \wedge \\
&\qquad\qquad A\,(s\ t)\ =\ \mathsf{Undefined}\ \ )
\end{aligned}
\tag{5.14}
$$

Theorem (5.13) beschreibt die Situation, wenn eine while-Schleife mit Bedingung c, Schleifenrumpf A und Eingangsbelegung x mit einem gewissen Wert y terminiert. Es existiert dann eine Sequenz s, die zu jedem Schleifendurchlauf den momentan berechneten Wert angibt. Vor dem ersten Durchlauf ($n = 0$) entspricht dieser Wert gerade dem Eingang x. Damit die Schleifenberechnung terminiert, muss bis zu einem gewissen Schleifendurchlauf mit der Nummer t der Schleifenrumpf jedes Mal terminieren und die Bedingung erfüllt sein. Erst nach dem t-ten Schleifendurchlauf ist die Bedingung nicht mehr erfüllt, und der t-te Wert der Sequenz s gibt das Ergebnis an.

Terminiert eine while-Schleife nicht, kann es dafür zwei Gründe geben, wie in Theorem (5.14) gezeigt ist. Entweder terminiert der Schleifenrumpf immer, und die Bedingung c ist ebenfalls immer erfüllt, sodass die Schleife nie abbricht, oder in einem gewissen t-ten Schleifendurchlauf terminiert der Schleifenrumpf A nicht mehr, sodass die Schleife nicht weiter ausgeführt werden kann. Da der Schleifenrumpf A ein Block ist, kann er selbst eine while-Schleife repräsentieren, die bei einer bestimmten Eingangsbelegung nicht terminiert.

Es muss angemerkt werden, dass der Beweis der in der Arbeit vorgestellten Theoreme sehr aufwendig ist und eine intime Kenntnis des Theorembeweisers und seiner Methoden erfordert. Die Schwierigkeit lässt sich vielleicht schon an der aufwendigen Definition der Funktion WHILE erahnen.

5.1.5 Elementare Kontrollstrukturen für P-Terme

Nachdem die Funktion WHILE eingeführt worden ist, mit deren Hilfe beliebige berechenbare Funktionen beschrieben werden können, sollen nun die elementaren Kontrollstrukturen vorgestellt werden, die in Gropius zur Beschreibung von Blöcken und Programmen vorhanden sind. Neben der Funktion PARTIALIZE, die in (5.5) definiert wurde, und der Funktion WHILE gibt es weitere fünf Kontrollstrukturen, um Blöcke zu beschreiben, und eine einzige Kontrollstruktur, um aus einem Block ein Programm zu generieren. Gropius verfügt u.a. über Kontrollstrukturen, wie sie aus herkömmlichen Programmiersprachen bekannt sind. Es ergeben sich die folgenden BN-Formen der Syntax für Blöcke und Programme:

$$
\begin{array}{lll}
\textit{Block} & ::= & \text{“ PARTIALIZE ”}\ \textit{Elementarblock} \ \mid \\
 & & \text{“ WHILE ”}\ \textit{Bedingung Block} \ \mid \\
 & & \textit{Block}\ \text{“ THEN ”}\ \textit{Block} \ \mid \\
 & & \text{“ IFTE ”}\ \textit{Bedingung Block Block} \ \mid \\
 & & \text{“ LOCVAR ”}\ \textit{Konstante Block} \ \mid \\
 & & \text{“ LEFTVAR ”}\ \textit{Block} \ \mid \\
 & & \text{“ RIGHTVAR ”}\ \textit{Block} \\
 & & \\
\textit{Programm} & ::= & \text{“ PROGRAM ”}\ \textit{Konstante Block}
\end{array}
\tag{5.15}
$$

Die Kontrollstrukturen bilden Konstanten, Elementarblöcke, Bedingungen und Blöcke auf Blöcke und Programme ab. Elementarblöcke, Bedingungen und Blöcke sind aber selbst Funktionen. Die Kontrollstrukturen sind also Funktionen von Funktionen, oder Funktionen höherer Ordnung. In Tabelle 5.3 sind alle acht elementaren Kontrollstrukturen mit ihren Typen und einer kurzen Beschreibung zusammengefasst.

Die restlichen sechs Kontrollstrukturen sollen nun vorgestellt werden. Die Funktion THEN ist eine binäre Funktion, die in Infix-Notation verwendet wird. Mit THEN werden zwei Blöcke in einen einzigen Block transformiert, der die Funktionen der beiden Blöcke hintereinander ausführt. Dabei wird aber die Funktion des zweiten Blocks nur dann ausgeführt, wenn der erste Block terminiert. Falls einer der beiden Blöcke nicht terminiert, ist das Gesamtergebnis Undefined. Die Definition von THEN ist wie folgt:

$$
\vdash (A\ \mathsf{THEN}\ B)\ x\ =\ \mathsf{PRIMREC\_partial}\ (A\ x)\ B\ \mathsf{Undefined} \tag{5.16}
$$

Gegeben seien zwei Blöcke A und B und eine Eingangsbelegung x. Zunächst wird A auf x angewandt. Abhängig von der Terminierung wird entweder B auf das Ergebnis angewandt oder der Wert Undefined ausgegeben.

IFTE ist eine Funktion, um bedingte Verzweigungen auszudrücken:

$$
\vdash \mathsf{IFTE}\ c\ A\ B\ x\ =\ (c\ x) \Rightarrow (A\ x) \mid (B\ x) \tag{5.17}
$$

Die Funktion (IFTE c A B) bildet einen Eingangswert x entweder auf $(A\ x)$ oder auf $(B\ x)$ ab, abhängig davon, ob die Bedingung c erfüllt wird oder nicht.

Name	Typ/Beschreibung
PARTIALIZE	$(\alpha \to \alpha) \to \alpha \to (\alpha)\mathsf{partial}$ Umwandlung von Elementarblöcken in Blöcke
WHILE	$(\alpha \to \mathsf{bool}) \to (\alpha \to (\alpha)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$ Schleife
THEN	$(\alpha \to (\alpha)\mathsf{partial}) \to (\alpha \to (\alpha)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$ Serielle Ausführung von Blöcken
IFTE	$(\alpha \to \mathsf{bool}) \to (\alpha \to (\alpha)\mathsf{partial}) \to (\alpha \to (\alpha)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$ Bedingte Verzweigung
LOCVAR	$\beta \to ((\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}) \to \alpha \to (\alpha)\mathsf{partial}$ Einführung einer lokalen Variablen
LEFTVAR	$(\alpha \to (\alpha)\mathsf{partial}) \to (\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}$ Funktionsanwendung auf den ersten Teil des Eingangs
RIGHTVAR	$(\beta \to (\beta)\mathsf{partial}) \to (\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}$ Funktionsanwendung auf den zweiten Teil des Eingangs
PROGRAM	$\beta \to ((\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial}) \to \alpha \to (\beta)\mathsf{partial}$ Umwandlung von Blöcken in Programme

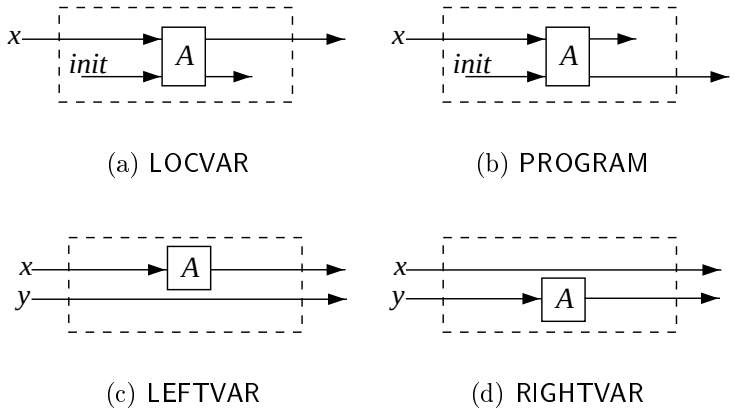
Tabelle 5.3: Elementare Kontrollstrukturen

Die Funktion LOCVAR wird verwendet, um lokale Variablen einzuführen:

$$\begin{aligned} \vdash \mathsf{LOCVAR}\ \mathit{init}\ A\ x\ = \\ \quad \mathsf{PRIMREC\_partial}\ (A\,(x, \mathit{init}))\ (\lambda(y, z).\ \mathsf{Defined}\ y)\ \mathsf{Undefined} \end{aligned} \tag{5.18}$$

Gegeben seien ein beliebiger Initialwert $(\mathit{init} : \beta)$ für die lokale Variable und der Eingangswert $(x : \alpha)$. Der Block $(A : (\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial})$ wird dann zunächst auf das kartesische Produkt (x, init) angewandt. Wenn die Funktionsanwendung mit dem Ergebnis Defined (y, z) terminiert, wird nur der erste Teil als Defined y ausgegeben. Der zweite Teil des Ergebnisses, der der lokalen Variable entspricht, wird weggelassen. Abbildung 5.5(a) veranschaulicht diesen Zusammenhang. Wenn $(A\,(x, \mathit{init}))$ nicht terminiert, ist das Ergebnis Undefined.

Auf ähnliche Weise funktioniert die Kontrollstruktur PROGRAM, mit deren Hilfe Blöcke vom Typ $\alpha \to (\alpha)\mathsf{partial}$ in Programme mit beliebigem Ausgangstyp



(a) LOCVAR (b) PROGRAM

(c) LEFTVAR (d) RIGHTVAR

Abbildung 5.5: Einige Kontrollstrukturen

(β)partial überführt werden können:

$$\begin{aligned}\vdash \mathsf{PROGRAM}\ &\mathit{init}\ A\ x\ = \\ &\mathsf{PRIMREC\_partial}\ (A\,(x, \mathit{init}))\ (\lambda(y, z).\ \mathsf{Defined}\ z)\ \mathsf{Undefined}\end{aligned} \tag{5.19}$$

Dazu wird ein beliebiger Wert ($\mathit{init} : \beta$) angegeben, der den Startwert des Ausgangs anzeigen soll. Die Funktion (PROGRAM $\mathit{init}\ A$) bildet dann zunächst einen Eingangswert x auf die Funktionsanwendung ($A\ (x, \mathit{init})$) ab. Terminiert diese nicht, ist das Ergebnis Undefined. Terminiert sie aber mit dem Ergebnis Defined (y, z), wird diesmal nur die zweite Komponente, die dem Ausgang entspricht, als Defined z ausgegeben. Der Teil y, der dem Eingang entspricht, entfällt. In Abbildung 5.5(b) ist die Wirkungsweise von PROGRAM veranschaulicht.

Die Kontrollstrukturen LEFTVAR und RIGHTVAR werden dazu verwendet, um Funktionsanwendungen nur auf Teile der Eingangsbelegung durchzuführen:

$$\begin{aligned}\vdash \mathsf{LEFTVAR}\ &A\ (x, y)\ = \\ &\mathsf{PRIMREC\_partial}\ (A\ x)\ (\lambda z.\ \mathsf{Defined}\ (z, y))\ \mathsf{Undefined}\end{aligned} \tag{5.20}$$

$$\begin{aligned}\vdash \mathsf{RIGHTVAR}\ &A\ (x, y)\ = \\ &\mathsf{PRIMREC\_partial}\ (A\ y)\ (\lambda z.\ \mathsf{Defined}\ (x, z))\ \mathsf{Undefined}\end{aligned} \tag{5.21}$$

Gegeben sei ein Wertepaar (x, y), das den Eingangswert repräsentiert. Die Funktion LEFTVAR A bildet dann (x, y) auf $(A\ x, y)$ ab. Terminiert $(A\ x)$ mit dem Wert (Defined z), wird als Ergebnis (Defined (z, y)) ausgegeben. Im Falle der Nichtterminierung ist das Ergebnis wiederum Undefined. Die Funktion RIGHTVAR A bildet das Paar (x, y) auf $(x, A\ y)$ ab. Terminiert $(A\ y)$ mit dem Wert (Defined z), wird als Ergebnis (Defined (x, z)) ausgegeben. Die Funktionsweise dieser beiden Kontrollstrukturen ist in Abbildung 5.5(c),(d) skizziert.

Dieser Satz von Kontrollstrukturen stellt einen kleinen Kern dar (siehe Tabelle 5.3), mit dem *alle* berechenbaren Funktionen beschrieben werden können. Der Kern

ist bewusst in dieser knappen Form gehalten, um die Sprache auf wenige wohldefinierte Füße zu stellen.

5.1.6 Abgeleitete Kontrollstrukturen für P-Terme

Aufbauend auf dem Kern der acht elementaren Kontrollstrukturen können – anders als bei vielen herkömmlichen Programmiersprachen wie VHDL oder Verilog – weitere Kontrollstrukturen vom Benutzer eingeführt werden. Eine abgeleitete Kontrollstruktur ist nichts anderes als eine Abkürzung für einen Ausdruck, der sich aus elementaren oder anderen bereits abgeleiteten Kontrollstrukturen zusammensetzt. Aufgrund der Tatsache, dass diese Kontrollstrukturen von den elementaren Kontrollstrukturen abgeleitet sind, erhöhen sie nicht die Ausdrucksmächtigkeit der Sprache, verbessern aber den komfortableren Umgang mit ihr.

Solche abgeleiteten Kontrollstrukturen können in individueller Weise vom Benutzer selbst durch Konstantendefinitionen eingeführt werden. Im Folgenden werden einige mögliche Kontrollstrukturen vorgestellt.

$$\vdash \mathsf{NOP} \;=\; \mathsf{PARTIALIZE}\ (\lambda x.\ x)$$
$$\vdash \mathsf{IFT}\ c\ A\ x \;=\; \mathsf{IFTE}\ c\ A\ \mathsf{NOP}\ x$$

Die Funktion NOP drückt die Identität aus, bei der keine Operation durchgeführt wird. Sie kann in Verbindung mit anderen Kontrollstrukturen verwendet werden, um Spezialfälle für diese Kontrollstrukturen auszudrücken. So wird NOP etwa benutzt bei der Definition der Kontrollstruktur IFT, die von IFTE abgeleitet ist. Im Gegensatz zu IFTE hat IFT keinen "else"-Zweig. Wenn die Bedingung c nicht erfüllt ist, wird der Eingang x also nicht verändert.

$$\begin{aligned}
&\vdash \mathsf{LEFT}\ A\ \mathit{init}\ = \\
&\qquad \mathsf{LOCVAR}\ \mathit{init} \\
&\qquad\qquad \mathsf{PARTIALIZE}\ (\lambda((x,y),h).\ ((x,y),y)) \\
&\qquad\qquad \mathsf{THEN}\ (\mathsf{LEFTVAR}\ A)\ \mathsf{THEN} \\
&\qquad\qquad \mathsf{PARTIALIZE}\ (\lambda((x,y),h).\ ((x,h),h)) \\
&\vdash \mathsf{RIGHT}\ A\ \mathit{init}\ = \\
&\qquad \mathsf{LOCVAR}\ \mathit{init} \\
&\qquad\qquad \mathsf{PARTIALIZE}\ (\lambda((x,y),h).\ ((x,y),x)) \\
&\qquad\qquad \mathsf{THEN}\ (\mathsf{LEFTVAR}\ A)\ \mathsf{THEN} \\
&\qquad\qquad \mathsf{PARTIALIZE}\ (\lambda((x,y),h).\ ((h,y),h))
\end{aligned} \tag{5.22, 5.23}$$

Die Kontrollstrukturen LEFT und RIGHT werden für spezielle Funktionsanwendungen benutzt. Gegeben sei ein Paar (x, y) als Eingang. Mit Hilfe der Funktionen LEFT und RIGHT wird ein Block $(A : (\alpha \times \beta) \to (\alpha \times \beta)\mathsf{partial})$ auf dieses Paar angewandt. Im Gegensatz zu einer normalen Funktionsanwendung sollen zwar beide

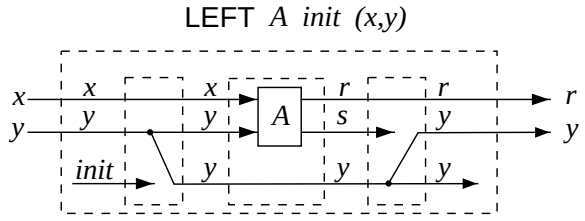


Abbildung 5.6: Wirkungsweise von LEFT

Komponenten für die Funktionsanwendung herangezogen werden, es soll aber eine Komponente ihren Wert nicht verändern. Bei der Kontrollstruktur LEFT ist dies der zweite Teil (y) und bei RIGHT der erste Teil (x) des Eingangs. In Abbildung 5.6 ist die Wirkungsweise für die Kontrollstruktur LEFT dargestellt. Für RIGHT ergäbe sich ein ähnliches Bild. Bei der Funktionsanwendung (LEFT A $init$ (x, y)) wird zuerst eine lokale Variable mit Initialwert ($init : \beta$) eingeführt. Zu Beginn der LOCVAR-Umgebung hat man damit den Zustand $((x, y), init)$. Der lokalen Variablen wird dann sofort der Wert der zweiten Komponente des Eingangs (y) zugewiesen. Dies geschieht in einem Elementarblock, der eine reine Verdrahtung darstellt. Operationen werden dort nicht ausgeführt. Damit ergibt sich der Zustand $((x, y), y)$. Mit Hilfe der Funktion (LEFTVAR A) wird anschließend der Block A auf den ersten Teil (x, y) des Zustands $((x, y), y)$ angewandt. Terminiert diese Funktionsanwendung, ist das Ergebnis Defined (r, s) mit einem gewissen Paar (r, s), das sich gewöhnlich vom Eingang (x, y) der Funktionsanwendung unterscheidet. Man erhält somit den Zustand $((r, s), y)$. Anschließend wird der Wert y, der in der lokalen Variablen zwischengespeichert ist, wieder zurückgeschrieben, indem der Wert s überschrieben wird. Der Zustand ändert sich in $((r, y), y)$. Schließlich wird die LOCVAR-Umgebung wieder verlassen, womit die lokale Variable verlorengeht. Es ergibt sich damit (vorausgesetzt A (x, y) terminiert) als Endergebnis Defined (r, y). Im Falle der Funktionsanwendung (RIGHT A $init$ (x, y)) würde sich als Endergebnis Defined (x, s) ergeben.

Die Kontrollstrukturen ASSOCVAR und SWAPVAR sind Beispiele dafür, wie eine Verdrahtung realisiert werden kann, sodass ein Block A eines bestimmten Typs auf einen gegebenen Eingang angewandt werden kann. Dabei müssen die Teiltypen von A im Typ des Eingangs enthalten sein.

$$
\begin{array}{l}
\vdash \mathsf{ASSOCVAR}\ A\ (i_1, i_2, i_3)\ = \\
\quad \mathsf{LOCVAR}\ (i_1, i_2, i_3) \\
\quad\quad \mathsf{PARTIALIZE}\ (\lambda(((x, y), z), h_1, h_2, h_3).\ (((x, y), z), x, y, z)) \\
\quad\quad \mathsf{THEN}\ (\mathsf{RIGHTVAR}\ A)\ \mathsf{THEN} \\
\quad\quad \mathsf{PARTIALIZE}\ (\lambda(((x, y), z), h_1, h_2, h_3).\ (((h_1, h_2), h_3), h_1, h_2, h_3))
\end{array}
\tag{5.24}
$$

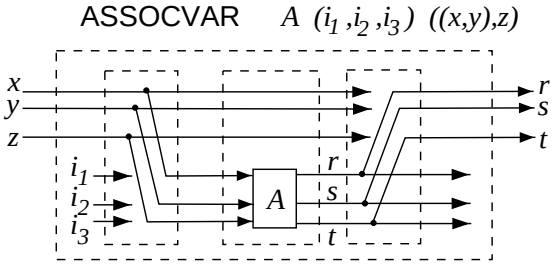


Abbildung 5.7: Wirkungsweise von ASSOCVAR

$$\begin{aligned}\vdash \mathsf{SWAPVAR}\ A\ (i_1, i_2)\ = \\ \quad \mathsf{LOCVAR}\ (i_1, i_2)\ (\ &\mathsf{PARTIALIZE}\ (\lambda((x,y),h_1,h_2).\ ((x,y),y,x)) \\ &\mathsf{THEN}\ (\mathsf{RIGHTVAR}\ A)\ \mathsf{THEN} \\ &\mathsf{PARTIALIZE}\ (\lambda((x,y),h_1,h_2).((h_2,h_1),h_1,h_2))\ )\end{aligned} \tag{5.25}$$

In Abbildung 5.7 ist die Wirkungsweise von ASSOCVAR dargestellt. Gegeben sei ein Eingang mit dem Typ $((\alpha \times \beta) \times \gamma)$. Dieser Eingang soll nun so umgeordnet werden, dass der Block A mit Eingangstyp $(\alpha \times \beta \times \gamma)$ auf ihn angewandt werden kann. Dazu werden drei lokale Hilfsvariablen mit den Initialwerten $(i_1 : \alpha), (i_2 : \beta)$ und $(i_3 : \gamma)$ eingeführt. In einem ersten Schritt werden die drei Komponenten des Eingangs auf diese drei Hilfsvariablen geschrieben. Dann kann auf den rechten Teil des sich ergebenden Zustands die Funktion A angewandt werden. Anschließend wird das Ergebnis, das in den Hilfsvariablen gespeichert ist, wieder in die drei Komponenten des globalen Zustands zurückgeschrieben. Die Funktion SWAPVAR leistet entsprechendes für einen Block A mit Eingangstyp $(\beta \times \alpha)$, der auf einen Eingang vom Typ $(\alpha \times \beta)$ angewandt werden soll. Aus ASSOCVAR und SWAPVAR lassen sich weitere Verdrahtungsstrukturen ableiten, wie in Abschnitt 8.3 vorgestellt wird. Diese Verdrahtungsstrukturen sind notwendig in zweierlei Hinsicht. Erstens dienen sie zur Beschreibung allgemeiner Transformationsschablonen, und zum anderen können bereits beschriebene Blöcke A auf diese Weise ohne Änderung in eine neue Schaltungsbeschreibung übernommen werden. Ergibt sich während der Synthese ein konkreter Block A, können diese Verdrahtungsstrukturen in einfacher Weise beseitigt werden, wie in Abschnitt 7.3.3 gezeigt wird.

Wie zu Beginn dieses Abschnitts erwähnt wurde, sollen mit Hilfe von P-Termen auch primitiv-rekursive Funktionen ausgedrückt werden, auch solche, die durch DFG-Terme nicht darstellbar sind. Im Folgenden werden mit Hilfe der Kontrollstruktur WHILE verschiedene for-Schleifen definiert:

$$\begin{aligned}\vdash \mathsf{FOR\_GEN}\ &min\ step\ A\ = \\ &\mathsf{LOCVAR}\ min \\ &\quad \mathsf{WHILE}\ (\lambda((x,n),h).\ h \leq n) \\ &\qquad A\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ (\lambda((x,n),h).((x,n),h+step)))\end{aligned} \tag{5.26}$$

$$\begin{aligned}
&\vdash \mathsf{FOR_C}\ min\ max\ step\ A\ = \\
&\qquad \mathsf{LOCVAR}\ min \\
&\qquad\qquad \mathsf{WHILE}\ (\lambda(x,n).\, n \leq max) \\
&\qquad\qquad\qquad A\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ (\lambda(x,n).(x, n+step)))
\end{aligned} \tag{5.27}$$

$$\begin{aligned}
&\vdash \mathsf{FOR_PASCAL}\ min\ max\ step\ A\ init\ = \\
&\qquad \mathsf{FOR_C}\ min\ max\ step\ (\mathsf{LEFT}\ A\ init)
\end{aligned} \tag{5.28}$$

$$\begin{aligned}
&\vdash \mathsf{FOR_NFOLD}\ min\ max\ step\ A\ = \\
&\qquad \mathsf{FOR_C}\ min\ max\ step\ (\mathsf{LEFTVAR}\ A)
\end{aligned} \tag{5.29}$$

$$\begin{aligned}
&\vdash \mathsf{FOR_1_NFOLD}\ max\ A\ = \\
&\qquad \mathsf{FOR_NFOLD}\ 1\ max\ 1\ A
\end{aligned} \tag{5.30}$$

Definition (5.26) beschreibt eine allgemeine for-Schleife FOR_GEN. Es wird dabei davon ausgegangen, dass am Eingang der Schleife eine Variablenstruktur (x, n) anliegt, bestehend aus einer Variablen x mit beliebigem Datentyp α und einer Variablen n mit dem Datentyp num. Neben einem Block $(A : (\alpha \times \mathsf{num}) \rightarrow (\alpha \times \mathsf{num})\mathsf{partial})$ besitzt FOR_GEN als weitere Parameter noch zwei Variablen *min* und *step*, die jeweils den Typ num haben. Zu Beginn wird eine lokale Variable mit Initialwert *min* eingeführt. Es wird dann solange eine while-Schleife durchlaufen, bis die lokale Variable größer als die Eingangsvariable n ist. Im Schleifenrumpf wird dabei zunächst die Funktion A ausgeführt, und dann wird explizit die lokale Variable um den Wert *step* erhöht. Handelt es sich bei dem Block A um einen Elementarblock, terminiert die Funktion FOR_GEN immer. Man könnte also meinen, diese Funktion wäre auch durch einen DFG-Term darstellbar. Dies ist aber nicht der Fall. Da die Anzahl der Schleifendurchläufe vom Eingang der Funktion abhängt, ist von vorneherein nicht bekannt, wie oft die Funktion A im Schleifenrumpf ausgeführt wird. Das soll aber nicht bedeuten, dass die Funktion (FOR_GEN *min step* A) nicht determiniert enden würde, da es zu jedem Eingangswert ein bestimmtes Resultat gibt. Bei der abgeleiteten Kontrollstruktur (FOR_GEN *min step* (PARTIALIZE a)) handelt es sich also um die Nachbildung einer primitiv-rekursiven Funktion, die sich mit einem DFG-Term nicht darstellen lässt.

Anders verhält es sich mit der Funktion FOR_C in Definition (5.27). Diese hat einen Eingang nur vom Typ α. Es wird auch hier eine lokale Zählvariable eingeführt, die einen Initialwert *min* erhält. Die Schleife wird dann solange ausgeführt, bis die Zählvariable größer als ein Wert *max* ist. Im Schleifenrumpf wird der Block A ausgeführt und anschließend auch hier die Zählvariable um den Wert *step* erhöht. Wird diese Kontrollstruktur mit einem Block (PARTIALIZE a) parametrisiert, ließe sich die Funktion zumeist auch als DFG-Term darstellen, da die Anzahl der Schleifendurchläufe nicht vom Eingang abhängt. Der Unterschied zu einer Darstellung als DFG-Term bestünde lediglich in einem anderen Ausgangstyp α anstatt (α)partial.

Es muss aber die Einschränkung gemacht werden, dass die Funktion (FOR_C *min max step* (PARTIALIZE a)) tatsächlich nur dann als DFG-Term

realisiert werden kann, wenn die Zählvariable in der Funktion a nicht verändert wird. Im Prinzip lässt dies nämlich die Funktion FOR_C zu, da der Block A den Typ $(\alpha \times \mathsf{num}) \rightarrow (\alpha \times \mathsf{num})\mathsf{partial}$ besitzt, also auch Einfluss auf den Wert der Zählvariablen nimmt. Man kann deshalb davon sprechen, dass FOR_C eine for-Schleife mit der Semantik der Programmiersprache C realisiert. In C hat die Veränderung des Wertes der Zählvariablen im Schleifenrumpf einen Einfluss auf die späteren Schleifendurchläufe. Die Anzahl der Schleifendurchläufe lässt sich also in der Funktion FOR_C i.Allg. nicht vorher bestimmen.

Um garantiert immer eine bestimmte Anzahl von Schleifendurchläufen zu haben, kann dagegen die in (5.28) definierte Kontrollstruktur FOR_PASCAL verwendet werden. Sie unterscheidet sich darin von der Kontrollstruktur FOR_C, dass die Einschränkung gemacht wird, dass nicht ein beliebiger Block A, sondern nur ein Block (LEFT A h) eingesetzt werden kann. Wie oben dargelegt, wird durch Anwendung der Funktion (LEFT A h) auf einen Eingang (x, n) nur der Wert von x verändert. n hat zwar einen Einfluß auf den zukünftigen Wert von x, wird aber selbst nicht verändert (siehe auch Abbildung 5.6). Mit dieser Maßnahme wird also eine unzulässige Veränderung der Zählvariablen verhindert, die somit garantiert pro Schleifendurchlauf nur um den Wert *step* erhöht wird. Die Funktion FOR_PASCAL bildet somit eine for-Schleife in der Semantik der Programmiersprache Pascal nach. In Pascal wird garantiert, dass zu Beginn des i-ten Schleifendurchlaufs die Zählvariable den Wert $(min + i * step)$ hat.

Die zwei Strukturen FOR_C und FOR_PASCAL gehen beide von einem Block A mit dem Typ $(\alpha \times \mathsf{num}) \rightarrow (\alpha \times \mathsf{num})\mathsf{partial}$ aus. Hat man einen solchen Block A einmal beschrieben, kann er ohne Änderung in diesen Kontrollstrukturen wiederverwendet werden.

Wird in der Kontrollstruktur FOR_C anstelle eines beliebigen Blocks der Ausdruck (LEFTVAR A) verwendet, gelangt man zu einer weiteren semantischen Variation einer for-Schleife, wie sie in Definition (5.29) gezeigt ist. In dem Block (LEFTVAR A) ist es nicht möglich, die Zählvariable n zu verwenden. LEFTVAR sorgt dafür, dass die Funktion $(A : \alpha \rightarrow (\alpha)\mathsf{partial})$ nur auf den ersten Teil des Zustands angewandt wird (vergleiche auch Abbildung 5.5(c)). Die Schleife FOR_NFOLD entspricht also lediglich einer mehrfachen Anwendung derselben Funktion A auf den Eingang x der Schleife. Wie oft die Funktion A angewandt wird, hängt von den Parametern min, max und $step$ ab. Die Anzahl der Schleifendurchläufe ergibt sich zu

$$\Big((max - min)\ \mathsf{div}\ step\Big)\ +\ 1$$

In Definition (5.30) ist schließlich noch ein Spezialfall der Funktion FOR_NFOLD eingeführt. Mit Hilfe der Struktur FOR_1_NFOLD wird die Funktion A $(max+1)$-mal ausgeführt. Sie entspricht damit der in (5.11) auf Seite 59 definierten Hilfsfunktion power A $(max + 1)$.

Natürlich können auf ähnliche Weise noch verschiedene for-Schleifen definiert werden, beispielsweise solche, bei denen die Zählvariable erniedrigt wird. Der Fantasie des Anwenders sind hier keine Grenzen gesetzt. Aber nicht nur for-Schleifen

können in beliebiger Weise eingeführt werden, sondern jede Art von Schleifen. Sie können alle auf die Grundfunktion WHILE zurückgeführt werden. So kann z.B. auch die an die repeat-Schleife herkömmlicher Programmiersprachen angelehnte Funktion REPEAT definiert werden:

$$\vdash \mathsf{LOOP}\ A\ c\ B\ =\ A\ \mathsf{THEN}\ (\mathsf{WHILE}\ c\ (B\ \mathsf{THEN}\ A)) \tag{5.31}$$

$$\vdash \mathsf{REPEAT}\ A\ c\ =\ \mathsf{LOOP}\ A\ c\ \mathsf{NOP} \tag{5.32}$$

Die Struktur LOOP realisiert eine Schleife, bei der eine Bedingung c zwischen den zwei Teilen A und B ihres Rumpfes abgefragt wird. Ersetzt man den zweiten Block B durch die Identitätsfunktion NOP, erhält man die Semantik der repeat-Schleife. Es lässt sich leicht beweisen, dass der Ausdruck (NOP THEN A) äquivalent dem Ausdruck A ist. Bei der Funktion REPEAT wird somit der Block A mindestens einmal ausgeführt.

5.1.7 Ein Beispielprogramm

Zur Veranschaulichung, wie mit Hilfe der vorgestellten Kontrollstrukturen Programme in Gropius beschrieben werden können, ist in Abbildung 5.8(a) ein Beispielprogramm in der imperativen Programmiersprache Pascal und in Abbildung 5.8(b) ein entsprechendes Programm in der funktionalen Schreibweise von Gropius dargestellt. Es handelt sich dabei um ein Programm zur Berechnung der n-ten Fibonacci-Zahl. Dieses Programm verwendet einen schnellen Algorithmus, der einen Aufwand von $O(\log_2 n)$ hat. Der naive Algorithmus, bei dem immer die beiden letzten Fibonacci-Zahlen addiert werden, hat linearen Aufwand.

Es soll nun das Gropius-Programm fib näher erläutert werden. fib besitzt den Typ num $\rightarrow$ (num)partial. Die Eingangsvariable vom Typ num bezeichnet die Nummer der gewünschten Fibonacci-Zahl. Durch das Konstrukt (PROGRAM 1) wird eine Ausgabevariable vom Typ num mit dem Initialwert 1 eingeführt. Über diese Ausgabevariable wird nach der Berechnung die Fibonacci-Zahl ausgegeben. Die Ausgangsvariable entspricht dabei der lokalen Variable $y1$ des Pascal-Programms. Die Funktion (PROGRAM 1) wird angewandt auf einen Block vom Typ num$\times$num $\rightarrow$ (num $\times$ num)partial. Mit Hilfe des Konstrukts (LOCVAR $(1,1,0,0)$) werden innerhalb dieses Blocks vier lokale Variablen eingeführt, die den lokalen Variablen $a1, a2, y2$ und m des Pascal-Programms entsprechen. Folglich haben alle DFG-Terme des Blocks, auf den die Funktion (LOCVAR $(1,1,0,0)$) angewandt wird, den Eingangstyp ((num $\times$ num) $\times$ num $\times$ num $\times$ num $\times$ num). Während in der Pascal-Notation die Initialwerte für $y1, a1, a2$ und $y2$ angegeben sind, ist der Initialwert für m nicht spezifiziert. Da in Gropius für alle Variablen, die mit PROGRAM oder LOCVAR eingeführt werden, ein Initialwert angegeben werden muss, wird in diesem Fall ein Standard-Wert für den entsprechenden Typ verwendet. Für eine Variable vom Typ num sei dies 0, für eine vom Typ bool F, usw.

Die Bezeichnung der Variablen innerhalb der DFG-Terme ist im Gropius-Programm fib entsprechend der Bezeichnung im Pascal-Programm FIB mit

```
FUNCTION FIB
(var n : int) : int;
VAR a1, a2, y1, y2, m : int;
VAR r, s : int;
BEGIN
 a1 := 1; a2 := 1;
 y1 := 1;
 y2 := 0;
 m := n div 2 + 1;
 WHILE (m <> 0) DO
 BEGIN
  IF(odd m)
   THEN BEGIN
    r := y1; y1 := y1 * a1 + y2 * a2;
    y2 := r * a2 + y2 * (a1 + a2);
    m := m - 1
   END
   ELSE BEGIN
    s := a1; a1 := a1 * a1 + a2 * a2;
    a2 := s * a2 + a2 * (s + a2);
    m := m div 2;
   END;
 END;
 IF(odd n)
  THEN
   RETURN y2
  ELSE
   RETURN y1
END;
```

(a) Pascal

```
fib =
PROGRAM 1
 LOCVAR (1, 1, 0, 0)
  PARTIALIZE (λ((n, y1), a1, a2, y2, m).
       ((n, y1), a1, a2, y2, (n DIV 2) + 1))
  THEN
  WHILE (λ((n, y1), a1, a2, y2, m).¬(m = 0))
   PARTIALIZE (λ((n, y1), a1, a2, y2, m).
        let c = ODD m in
        let m1 = m − 1 in
        let m2 = m DIV 2 in
        let m3 = MUX(c, m1, m2) in
        let x = a1 + a2 in
        let x1 = MUX(c, y1, a1) in
        let x2 = MUX(c, y2, a2) in
        let x3 = x1 ∗ a1 in
        let x4 = x2 ∗ a2 in
        let x5 = x3 + x4 in
        let x6 = x1 ∗ a2 in
        let x7 = x2 ∗ x in
        let x8 = x6 + x7 in
        let y1′ = MUX(c, x5, y1) in
        let a1′ = MUX(c, a1, x5) in
        let a2′ = MUX(c, a2, x8) in
        let y2′ = MUX(c, x8, y2) in
        ((n, y1′), a1′, a2′, y2′, m3))
  THEN
  PARTIALIZE (λ((n, y1), a1, a2, y2, m).
       ((n, MUX(ODD n, y2, y1)), a1, a2, y2, m))
```

(b) Gropius

Abbildung 5.8: Programm zur Berechnung der n-ten Fibonacci-Zahl

$n, y1, a1, a2, y2$ und m gewählt. Es wäre nicht notwendig, dieselbe Variable in verschiedenen DFG-Termen mit demselben Namen zu bezeichnen. Da jeder DFG-Term einen geschlossenen Ausdruck darstellt, der durch α-Konversion (siehe Abschnitt 2.1.2) verändert werden kann, könnten die Variablen in verschiedenen DFG-Termen unterschiedliche Namen tragen. Eine solche Vorgehensweise würde aber die Lesbarkeit von Programmen erheblich erschweren. Obwohl es also syntaktisch nicht unbedingt erforderlich ist, soll die Regel gelten, dass eine Variable, deren Sichtbarkeitsbereich sich über mehrere DFG-Terme erstreckt, in diesen DFG-Termen auch in derselben Weise bezeichnet werden soll.

Die Funktion MUX bezeichnet eine Fallunterscheidung innerhalb eines DFG-Terms. Sie hat dieselbe Semantik wie die in Abschnitt 2.6.1 eingeführte Konstante COND:

$$\mathsf{MUX}\ (c, a, b) \ = \ \mathsf{COND}\ c\ a\ b \ = \ c \ \Rightarrow \ a \mid b \tag{5.33}$$

5.2 Schnittstellenverhalten

Bei der Beschreibung des Schnittstellenverhaltens muss prinzipiell unterschieden werden, ob es sich bei der algorithmischen Beschreibung des funktionalen Verhaltens um einen DFG-Term oder um einen P-Term handelt. Die Ausführung von DFG-Termen terminiert immer. DFG-Terme können durch geeignete Implementierungen auf der RT-Ebene in einer festen Anzahl von Takten ausgeführt werden. Die Anzahl der Takte kann vom Entwerfer zu Beginn der Synthese als Randbedingung vorgegeben werden, oder der Entwerfer benutzt eine Variable, die im Laufe der Synthese instantiiert wird. Anders verhält es sich bei Schnittstellenverhaltensmustern für P-Terme. Hier ist eine Angabe der benötigten Takte nicht möglich, da die Dauer der Funktionsausführung von vorneherein nicht feststeht, ja sogar unendlich sein kann, wenn die Ausführung nicht terminiert.

In Gropius sind sowohl verschiedene generische DFG- als auch P-Schnittstellenverhaltensmuster definiert. Als Parameter werden den P-Schnittstellenverhaltensmustern P-Terme, den DFG-Schnittstellenverhaltensmustern DFG-Terme sowie zusätzlich auch die Anzahl der benötigten Takte übergeben.

Schaltungsbeschreibungen auf der algorithmischen Ebene, die Startpunkt für die Synthese sein sollen, müssen sowohl das funktionale als auch das Schnittstellenverhalten umfassen. Es ergibt sich demnach der folgende syntaktische Aufbau für Schaltungsbeschreibungen auf der algorithmischen Ebene (siehe auch Abbildung 4.1 auf Seite 44):

A-Schnittstelle ::=
“ (” *Variable* { “ , ” *Variable* } “) ”

Algorithmische DFG-Schaltungsbeschreibung ::=
DFG-Schnittstellenverhaltensmuster (5.34)
“ (” *DFG-Term* “ , ” *Taktzahl* “) ” *A-Schnittstelle*

Algorithmische P-Schaltungsbeschreibung ::=
P-Schnittstellenverhaltensmuster *Programm* *A-Schnittstelle*

Die Schnittstellen der algorithmischen Schaltungsbeschreibungen weisen neben den Dateneingängen und Datenausgängen der zu realisierenden Funktion, die durch einen DFG-Term bzw. P-Term gegeben ist, auch noch mehrere Kontrollsignale auf, die den zeitlichen Ablauf gegenüber der Umgebung steuern. Es werden z.B. Signale benötigt, um eine Berechnung anzustoßen oder um eine Berechnung abzubrechen.

Bei der Kommunikation mit der Umwelt unterscheiden sich Schaltungen, die aus DFG-Termen abgeleitet wurden, von solchen, die auf P-Termen basieren. Da aus DFG-Termen abgeleitete Schaltungen den DFG-Term immer in einer konstanten Anzahl von Takten ausführen, ist es nicht unbedingt notwendig, den Berechnungszustand der Schaltung nach außen zu signalisieren. Bei Schaltungen, die auf P-Termen basieren, ist dies hingegen unerlässlich.

Im Gegensatz zum generellen Konzept von Gropius, das eine funktionale Schaltungsbeschreibung vorsieht, wird die Semantik der Schnittstellenverhaltensmuster in HOL relational beschrieben. Es ist anzumerken, dass diese Spezifikationen nicht simulierbar sind, da sie im Ganzen eine relationale Beziehung zwischen Signalen und keinen funktionalen Zusammenhang zwischen Ein- und Ausgabe herstellen. Außerdem sind die Spezifikationen bewusst partiell. Es würde keinen Sinn machen, während der Berechnungsphase zu jedem Zeitpunkt den Wert am Ausgangs-Signal anzugeben, solange das Ergebnis noch nicht vorliegt. Eine solche Vorgabe würde nur die Freiheitsgrade im weiteren Syntheseverlauf einschränken, ohne einen Vorteil zu liefern. Jeder dieser Spezifikationen wird jedoch eine Implementierung in Form einer sequentiellen Schaltung auf der RT-Ebene gegenübergestellt (siehe Abschnitt 7). Diese stellt wiederum eine funktionale, simulierbare Beschreibung dar.

Im Folgenden sollen zunächst die Schnittstellenverhaltensmuster für P-Terme vorgestellt werden. Da DFG-Terme in P-Terme überführt werden können, lassen sich darauf aufbauend in einfacher Weise auch Schnittstellenverhaltensmuster für DFG-Terme definieren.

5.2.1 Schnittstellenverhaltensmuster für P-Terme

Die Schnittstellenverhaltensmuster werden durch Konstantendefinitionen eingeführt. In Abbildung 5.9 ist die formale Definition eines Schnittstellenverhaltensmusters für P-Terme mit Namen P_IFC_RESET_START angegeben*. Eine beispielhafte schematische Beschreibung des Verhaltens, das sich durch P_IFC_RESET_START für ein gewisses P ergibt, ist in Abbildung 5.10(a) gezeigt. Die Funktion P_IFC_RESET_START beschreibt eine Relation zwischen den Schnittstellensignalen *input*, *reset*, *start*, *ready* und *output* bez. eines Programms P. Die fünf Signale *input*, *reset*, *start*, *ready* und *output* beschreiben jeweils eine Sequenz von Werten unterschiedlichen Typs. Jedem Zeitpunkt wird ein Wert zugewiesen. Die Zeit wird dabei durch den Typ num der natürlichen Zahlen repräsentiert. Die Signale *reset*, *start* und *ready* haben den Typ $\mathsf{num} \rightarrow \mathsf{bool}$, während *input* vom Typ $\mathsf{num} \rightarrow \alpha$ und *output* vom Typ $\mathsf{num} \rightarrow \beta$ sind. Die fünf Signale entsprechen genau den Signalen an der Schnittstelle der entsprechenden Schaltung auf RT-Ebene.

Die Signale *input* und *output* entsprechen dem Eingang bzw. dem Ausgang des Programms P. Die Signale *reset*, *start* und *ready* werden durch das Schnittstellenverhaltensmuster zur Kommunikation mit der Umwelt zusätzlich eingeführt. *start* dient dazu, eine Berechnung anzustoßen, mit Hilfe des *ready*-Signals zeigt der Prozess an, ob er gerade eine Berechnung ausführt, und *reset* wird verwendet, um eine Berechnung abzubrechen.

Die Schaltung, die durch das Schnittstellenverhaltensmuster P_IFC_RESET_START und den P-Term P beschrieben wird, verfügt über einen internen Zustand, der durch das Signal *ready* angezeigt wird. Hat *ready* den Wert

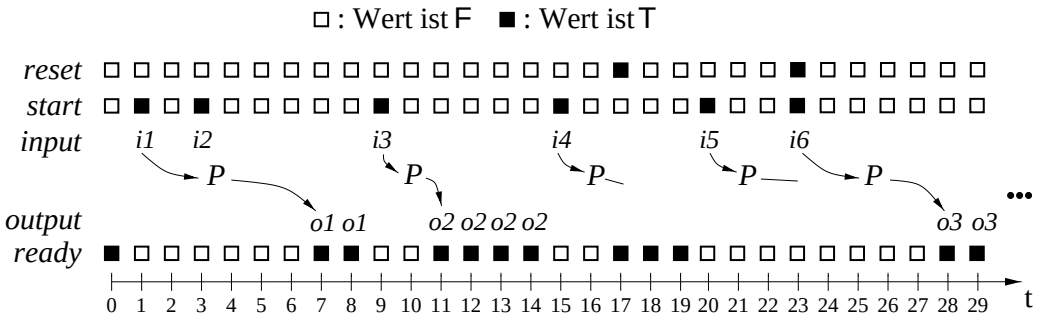
*Im weiteren Verlauf der Arbeit wird in den Formeln, wie in Abbildung 5.9 gezeigt, Einrücken verwendet, um Klammerungen auszudrücken.

$$
\begin{array}{ll}
\vdash \mathsf{P\_IFC\_RESET\_START} = & (1)\\
\quad \lambda P.\lambda(input, reset, start, output, ready). & (2)\\
\qquad \neg(start\ 0) \ \Rightarrow\ ready\ 0 & (3)\\
\qquad \wedge & (4)\\
\qquad \forall t. & (5)\\
\qquad\quad (ready\ t\ \wedge\ \neg(start\,(t+1))) \ \Rightarrow & (6)\\
\qquad\qquad ready\,(t+1)\ \wedge\ (output\,(t+1)\ =\ output\ t) & (7)\\
\qquad\quad \wedge & (8)\\
\qquad\quad (reset\ t\ \wedge \neg(start\ t)) \ \Rightarrow\ ready\ t & (9)\\
\qquad\quad \wedge & (10)\\
\qquad\quad (((t=0)\ \vee\ ready\,(t-1)\ \vee\ reset\ t)\ \wedge\ start\ t) \ \Rightarrow & (11)\\
\qquad\qquad \mathsf{PRIMREC\_partial}\ (P\ (input\ t)) & (12)\\
\qquad\qquad \big(\lambda y. & (13)\\
\qquad\qquad\ \exists m. & (14)\\
\qquad\qquad\quad \forall n.\,(n<m\ \wedge\ (\forall p.\,p<n\ \Rightarrow\ \neg(reset\,(t+p+1)))) \ \Rightarrow & (15)\\
\qquad\qquad\qquad\quad \neg(ready(t+n)) & (16)\\
\qquad\qquad\quad \wedge & (17)\\
\qquad\qquad\quad (\forall p.\,p<m\ \Rightarrow\ \neg(reset\,(t+p+1))) \ \Rightarrow & (18)\\
\qquad\qquad\qquad\quad ((output\,(t+m)=y)\ \wedge\ ready\,(t+m))\ \big) & (19)\\
\qquad\qquad & (20)\\
\qquad\qquad \big(\forall m. & (21)\\
\qquad\qquad\quad (\forall n.\,n<m\ \Rightarrow\ \neg(reset\,(t+n+1))) \ \Rightarrow & (22)\\
\qquad\qquad\qquad \neg(ready\,(t+m))\ \big) &
\end{array}
$$

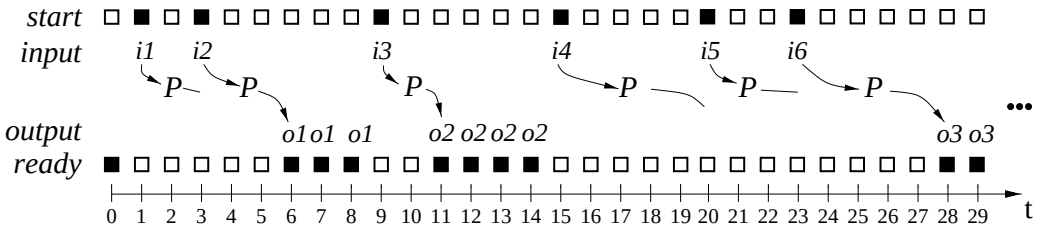
Abbildung 5.9: Formale Definition des Schnittstellenverhaltensmusters P_IFC_RESET_START

T, ist die Schaltung im Ruhezustand, beim Wert F ist sie dagegen im aktiven Zustand. Zum Zeitpunkt 0 hat *ready* den Wert T, dies allerdings nur, wenn zu diesem Zeitpunkt kein Startsignal stattfindet, wie es in Zeile 3 beschrieben ist. Befindet sich die Schaltung im Ruhezustand und wird im nächsten Zeitpunkt kein Startsignal gegeben (Zeile 6), wird die Schaltung auch im nächsten Zeitpunkt im Ruhezustand sein, und am Ausgang wird der letzte Wert gehalten (Zeile 7). Dies ist in Abbildung 5.10(a) etwa zum Zeitpunkt 12 der Fall. Durch das Signal *reset* kann die Schaltung in den Ruhezustand überführt werden, aber nur dann, wenn nicht gleichzeitig das *start*-Signal aktiviert wird (Zeile 9 und Zeitpunkt 17).

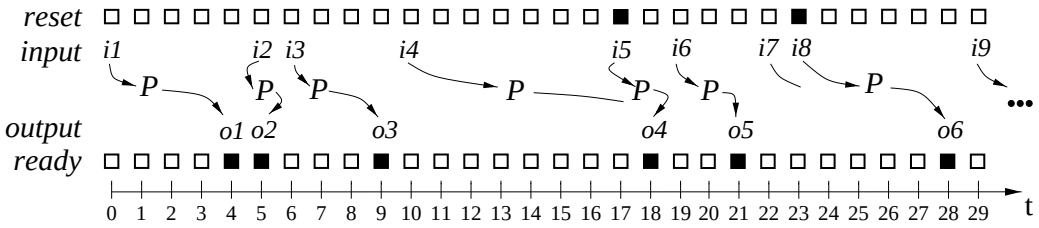
Falls *reset* und *start* gleichzeitig aktiv sind, setzt sich das Signal *start* durch, wie aus Zeile 11, bzw. Zeitpunkt 23 ersichtlich ist. Ansonsten kann nur dann eine Berechnung gestartet werden, wenn die Schaltung zum vorherigen Zeitpunkt im Ruhezustand war (Zeitpunkte 1, 9, 15, 20) oder wenn das Startsignal zum Zeitpunkt 0 kommt. Es kann also insbesondere alleine durch Aktivieren des Signals *start* im bereits aktiven Zustand eine laufende Berechnung nicht abgebrochen werden, wie in Abbildung 5.10(a) zum Zeitpunkt 3 veranschaulicht wird. Wenn eine Berechnung



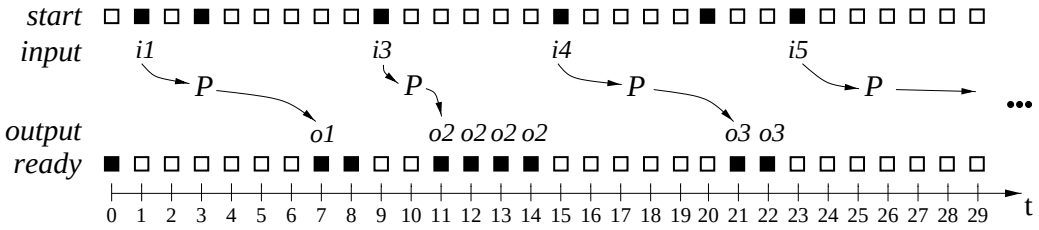
(a) P_IFC_RESET_START



(b) P_IFC_START



(c) P_IFC_CYCLE_1



(d) P_IFC_NO_RESET

Abbildung 5.10: Verschiedene Schnittstellenverhaltensmuster I

zum Zeitpunkt t gestartet wurde, gibt es zwei Alternativen:

- Die Funktionsanwendung $P\,(input\ t)$ terminiert mit einem gewissen Resultat Defined y (Zeilen 13-19)

- Die Funktionsanwendung terminiert nicht und das Ergebnis ist Undefined (Zeilen 20-22)

Im zweiten Fall befindet sich die Schaltung solange im aktiven Zustand (Zeile 22), bis sie durch Setzen des *reset*-Signals (Zeile 21) in den Ruhezustand überführt oder durch gleichzeitiges Setzen von *reset* und *start* eine neue Berechnung angestoßen wird. Solange *ready* den Wert F hat und *reset* nicht aktiviert worden ist, kann keine neue Berechnung gestartet werden.

Im ersten Fall ist die Schaltung im aktiven Zustand ($ready = \mathsf{F}$), bis die Berechnung durchgeführt ist und *reset* nicht gesetzt wurde (Zeilen 15,16 und Zeitpunkte 1-6, 9-10, 23-27). Die Dauer der Berechnung des Ausdrucks $P\,(input\ t)$ kann i.Allg. nicht vorab bestimmt werden, da sie nicht von P, sondern vom Eingang abhängt. Sie ist deshalb nicht festgelegt. Dies wird durch den Ausdruck "$\exists m$" in Zeile 14 beschrieben. Wenn bis zum Ende der Berechnung *reset* nicht aktiviert worden ist (Zeile 18), liegt schließlich zum Zeitpunkt $t + m$ am Ausgangssignal *output* das Ergebnis y an, und die Schaltung wechselt in den Ruhezustand (Zeile 19 und Zeitpunkte 7, 11, 28).

Es muss angemerkt werden, dass das Schnittstellenverhaltensmuster das Schnittstellenverhalten der entsprechenden Schaltung auf der RT-Ebene beschreibt. Das bedeutet, dass der Wertebereich für P-Terme in den Wertebereich, der auf der RT-Ebene zulässig ist, umkodiert werden muss. Auf der RT-Ebene gibt es nicht den Datentyp partial und damit nicht die Werte Defined y und Undefined. Vielmehr wird durch das Schnittstellenverhaltensmuster P_IFC_RESET_START und auch die anderen Schnittstellenverhaltensmuster, die später vorgestellt werden sollen, der Wert Defined y auf der RT-Ebene so interpretiert, dass nach einer gewissen Zeit am Ausgangssignal der Wert y anliegt und das Signal *ready* gesetzt ist. Der Wert Undefined wird dahingehend ausgedrückt, dass das Signal *ready* unendlich lange den Wert F hat. Rein praktisch kann der Wert Undefined aber nicht anhand nur der Signalwerte *output* und *ready* ermittelt werden, da diese nicht unendlich lange beobachtet werden können. Wird eine Berechnung (notwendigerweise) abgebrochen, kann man nur sagen, dass sie bis zu diesem Zeitpunkt nicht beendet wurde. Sie hätte aber zu einem späteren Zeitpunkt terminieren können. Das Schnittstellenverhaltensmuster P_IFC_RESET_START drückt genau dies aus, indem von den Signalwerten her kein Unterschied gemacht wird, ob durch das *reset*-Signal die Berechnung vor ihrer Terminierung abgebrochen wird oder ob die Berechnung nie terminieren würde.

Ausgehend von dem Schnittstellenverhaltensmuster P_IFC_RESET_START können mehrere andere Schnittstellenverhaltensmuster definiert werden, indem die

Eingangskontrollsignale *reset* und *start* in geeigneter Weise beschaltet werden:

$$\vdash \mathsf{P\_IFC\_START}\ P\ (input, start, output, ready) = \mathsf{P\_IFC\_RESET\_START}\ P\ (input, start, start, output, ready) \tag{5.35}$$

$$\vdash \mathsf{P\_IFC\_CYCLE\_1}\ P\ (input, reset, output, ready) = \mathsf{P\_IFC\_RESET\_START}\ P\ (input, reset, (\lambda t.\mathsf{T}), output, ready) \tag{5.36}$$

$$\vdash \mathsf{P\_IFC\_CYCLE\_2}\ P\ (input, output, ready) = \mathsf{P\_IFC\_RESET\_START}\ P\ (input, (\lambda t.\mathsf{F}), (\lambda t.\mathsf{T}), output, ready) \tag{5.37}$$

$$\vdash \mathsf{P\_IFC\_NO\_RESET}\ P\ (input, start, output, ready) = \mathsf{P\_IFC\_RESET\_START}\ P\ (input, (\lambda t.\mathsf{F}), start, output, ready) \tag{5.38}$$

Das Schnittstellenverhaltensmuster P_IFC_START ergibt sich dadurch, dass in P_IFC_RESET_START das *reset*-Signal durch das *start*-Signal ersetzt wird. Die Folge ist ein völlig verändertes Schnittstellenverhalten, wie es in Abbildung 5.10(b) veranschaulicht ist. Eine Berechnung kann nun durch Setzen des *start*-Signals abgebrochen werden, indem eine neue Berechnung gestartet wird.

Setzt man das Signal *start* für alle Zeiten auf den Wert T, ergibt sich das Verhalten, das durch das Schnittstellenverhaltensmuster P_IFC_CYCLE_1 beschrieben wird. Beginnend mit dem Zeitpunkt 0 wird eine Berechnung gestartet und nach deren Beendigung sofort im nächsten Zeitpunkt eine neue begonnen. Dieses zyklische Verhalten kann durch Setzen des *reset*-Signals unterbrochen werden. Wenn *reset* den Wert T hat, wird ein neuer Zyklus begonnen. Dieses Verhalten ist in Abbildung 5.10(c) schematisiert.

Eine weitere Variante ergibt sich bei den Schnittstellenverhaltensmustern P_IFC_CYCLE_2 und P_IFC_NO_RESET. Hier wird für alle Zeiten das *reset*-Signal ignoriert. Aufgrund des Terminierungsproblems ist klar, dass diese beiden Schnittstellenverhaltensmuster nur in solchen Fällen angewandt werden sollten, in denen der verwendete P-Term bekanntermaßen terminiert und zwar für alle erdenklichen Eingaben. In Abbildung 5.10(d) ist das Schnittstellenverhaltensmuster P_IFC_NO_RESET skizziert. Wenn die Funktionsanwendung $P\ (input\ 23)$ nicht terminieren würde, bestünde keine Möglichkeit mehr, die Berechnung abzubrechen und eine neue zu starten, es sei denn, die Stromversorgung würde unterbrochen und die Schaltung von externer Seite in den Initialzustand gebracht, indem das den internen Zustand repräsentierende Register mit dem Wert T geladen wird.

Bei der Definition des Schnittstellenverhaltensmusters P_IFC_RESET_START wurde von einem dominanten *start*-Signal ausgegangen. Bei gleichzeitigem Setzen der Signale *reset* und *start* wird nicht nur eine evtl. noch laufende Berechnung abgebrochen, sondern eine neue Berechnung wird umgehend begonnen. Genauso ist es natürlich denkbar, ein dominantes *reset*-Signal einzusetzen. In Abbildung 5.11 ist ein solches Schnittstellenverhaltensmuster mit dem Namen P_IFC_RESET definiert. Es unterscheidet sich darin von dem in Abbildung 5.9 definierten Schnittstellenver-

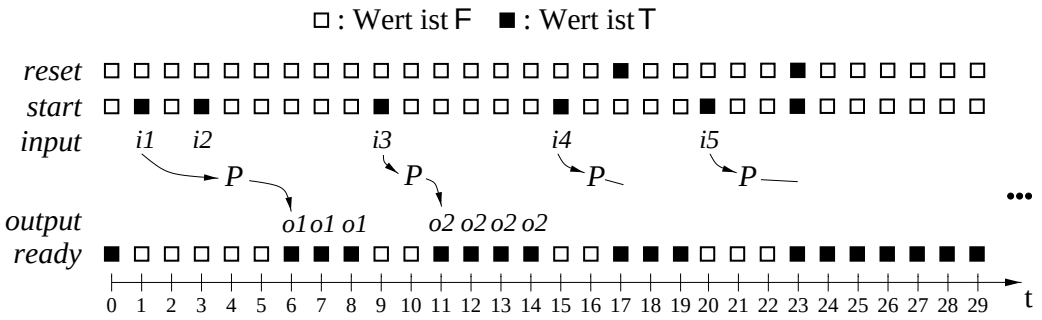
$$
\begin{array}{ll}
\vdash \mathsf{P\_IFC\_RESET} \;= & (1)\\
\quad \lambda P.\lambda(input, reset, start, output, ready). & (2)\\
\quad\quad \neg(start\ 0) \;\Rightarrow\; ready\ 0 & (3)\\
\quad\quad \wedge & (4)\\
\quad\quad \forall t. & (5)\\
\quad\quad\quad (ready\ t \;\wedge\; \neg(start\,(t+1))) \;\Rightarrow & (6)\\
\quad\quad\quad\quad (ready\,(t+1) \;\wedge\; (output\,(t+1) \;=\; output\ t)) & (7)\\
\quad\quad\quad \wedge & (8)\\
\quad\quad\quad (reset\ t \;\Rightarrow\; ready\ t) & (9)\\
\quad\quad\quad \wedge & (10)\\
\quad\quad\quad (((t=0) \;\vee\; ready\,(t-1)) \;\wedge\; start\ t \;\wedge\; \neg(reset\ t)) \;\Rightarrow & (11)\\
\quad\quad\quad\quad \mathsf{PRIMREC\_partial}\ (P\ (input\ t)) & (12)\\
\quad\quad\quad\quad \big(\lambda y. & (13)\\
\quad\quad\quad\quad\ \exists m. & (14)\\
\quad\quad\quad\quad\quad \forall n.\,(n < m \;\wedge\; (\forall p.\,p < n \;\Rightarrow\; \neg(reset\,(t+p+1)))) \;\Rightarrow & (15)\\
\quad\quad\quad\quad\quad\quad \neg(ready(t+n)) & (16)\\
\quad\quad\quad\quad\quad \wedge & (17)\\
\quad\quad\quad\quad\quad (\forall p.\,p < m \;\Rightarrow\; \neg(reset\,(t+p+1))) \;\Rightarrow & (18)\\
\quad\quad\quad\quad\quad\quad (output\,(t+m) = y) \;\wedge\; ready\,(t+m)\big) & (19)\\
\quad\quad\quad\quad & (20)\\
\quad\quad\quad\quad \big(\forall m. & (21)\\
\quad\quad\quad\quad\quad (\forall n.\,n < m \;\Rightarrow\; \neg(reset\,(t+n+1))) \;\Rightarrow & (22)\\
\quad\quad\quad\quad\quad\quad \neg(ready\,(t+m))\big) &
\end{array}
$$

Abbildung 5.11: Formale Definition des Schnittstellenverhaltensmusters P_IFC_RESET

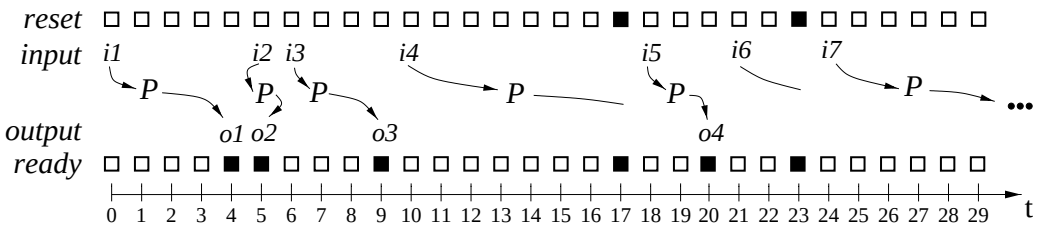
haltensmuster P_IFC_RESET_START, dass in jedem Fall die Schaltung durch Setzen von *reset* in den Ruhezustand überführt wird (siehe Zeile 9). Als Konsequenz kann eine Berechnung nur dann gestartet werden, wenn die Schaltung im Ruhezustand ist, das *start*-Signal gesetzt wird und *reset* den Wert F hat, wie es in Zeile 11 beschrieben ist. Im weiteren Verlauf unterscheidet sich dieses Schnittstellenverhaltensmuster nicht von P_IFC_RESET_START. In Abbildung 5.12(a) ist das Verhalten, das durch P_IFC_RESET beschrieben wird, schematisch dargestellt. Von diesem Schnittstellenverhaltensmuster ausgehend können nun wieder andere Schnittstellenverhaltensmuster abgeleitet werden. Es ist aber nur das Schnittstellenverhaltensmuster P_IFC_CYCLE_3 sinnvoll, bei dem das Signal *start* immer den Wert T hat.

$$
\begin{array}{l}
\vdash \mathsf{P\_IFC\_CYCLE\_3}\ P\ (input, reset, output, ready) \;=\\
\qquad \mathsf{P\_IFC\_RESET}\ P\ (input, reset, (\lambda t.\mathsf{T}), output, ready)
\end{array}
\tag{5.39}
$$

Dieses Schnittstellenverhaltensmuster ist in Abbildung 5.12(b) veranschaulicht. Es unterscheidet sich darin von P_IFC_CYCLE_1, dass bei der Unterbrechung eines Zyklus' durch *reset* der neue Zyklus einen Zeitpunkt später aufgenommen wird, da zunächst die Schaltung in den Ruhezustand versetzt wird.



(a) P_IFC_RESET



(b) P_IFC_CYCLE_3

Abbildung 5.12: Verschiedene Schnittstellenverhaltensmuster II

Es ist dagegen nicht sinnvoll, in P_IFC_RESET das *reset*-Signal mit *start* gleichzusetzen. Die Konsequenz wäre nämlich, dass niemals eine Berechnung gestartet werden könnte. Wenn das *reset*-Signal immer den Wert F erhalten würde, bekäme man dieselben zwei Schnittstellenverhaltensmuster P_IFC_CYCLE_2 und P_IFC_NO_RESET, die schon von P_IFC_RESET_START abgeleitet wurden.

Auch wenn der Unterschied zwischen P_IFC_RESET_START und P_IFC_RESET sehr klein ist, soll dieses Schnittstellenverhaltensmuster dennoch an dieser Stelle eingeführt werden, da es die Grundlage für das Kommunikationsschema auf der Systemebene darstellt (siehe Abschnitte 6.2 und 8.1), bei dem ebenfalls ein dominantes *reset*-Signal eingesetzt wird.

5.2.2 Schnittstellenverhaltensmuster für DFG-Terme

Anhand der Schnittstellenverhaltensmuster für P-Terme lassen sich in einfacher Weise äquivalente Schnittstellenverhaltensmuster für DFG-Terme ableiten. Abbildung 5.13 zeigt beispielsweise die Definition des Schnittstellenverhaltensmusters DFG_IFC_RESET, das an P_IFC_RESET_START angelehnt ist. Eine schematische Darstellung des Schnittstellenverhaltensmusters DFG_IFC_RESET findet sich in Abbildung 5.14.

Anstelle eines P-Terms bekommt diese Funktion ein Paar, bestehend aus einem

$$
\begin{array}{ll}
\vdash \mathsf{DFG\_IFC\_RESET} \;=\; & (1)\\
\quad \lambda(f,m).\lambda(input, reset, start, output, ready). & (2)\\
\quad\quad \neg(start\ 0) \;\Rightarrow\; ready\ 0 & (3)\\
\quad\quad \wedge & (4)\\
\quad\quad \forall t. & (5)\\
\quad\quad\quad (ready\ t \;\wedge\; \neg(start\,(t+1))) \;\Rightarrow & (6)\\
\quad\quad\quad\quad ready\,(t+1) \;\wedge\; (output\,(t+1) \;=\; output\ t) & (7)\\
\quad\quad\quad \wedge & (8)\\
\quad\quad\quad (reset\ t \;\Rightarrow\; ready\ t) & (9)\\
\quad\quad\quad \wedge & (10)\\
\quad\quad\quad ((((t=0) \;\vee\; ready\,(t-1)) \;\wedge\; start\ t \;\wedge\; \neg(reset\ t)) \;\Rightarrow & (11)\\
\quad\quad\quad\quad \forall n.\,(n<m \;\wedge\; (\forall p.\,p<n \;\Rightarrow\; \neg(reset\,(t+p+1)))) \;\Rightarrow & (12)\\
\quad\quad\quad\quad\quad \neg(ready(t+n)) & (13)\\
\quad\quad\quad\quad \wedge & (14)\\
\quad\quad\quad\quad (\forall p.\,p<m \;\Rightarrow\; \neg(reset\,(t+p+1))) \;\Rightarrow & (15)\\
\quad\quad\quad\quad\quad ((output\,(t+m) = f\,(input\ t)) \;\wedge\; ready\,(t+m))\,\big) & (16)
\end{array}
$$

Abbildung 5.13: Formale Definition des Schnittstellenverhaltensmusters DFG_IFC_RESET

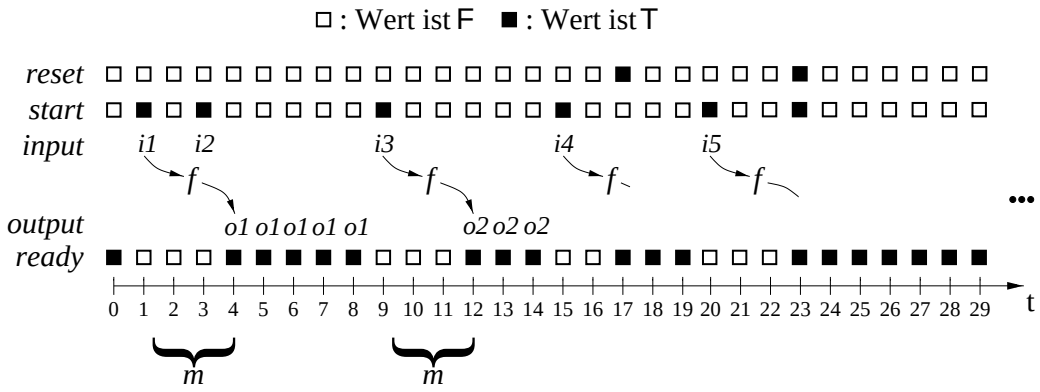


Abbildung 5.14: Schematische Darstellung von DFG_IFC_RESET

DFG-Term f und einer natürlichen Zahl m, übergeben (Zeile 2). Zusätzlich wird der Teil der Definition angepasst, der die Berechnungsphase beschreibt. Ersetzt man ab Zeile 12 in Abbildung 5.9 den Ausdruck PRIMREC_partial $(P\,(input\ t))\ (\lambda y.\,A[y])\ B$ durch den Ausdruck PRIMREC_partial $((\mathsf{PARTIALIZE}\ f)\,(input\ t))\ (\lambda y.\,A[y])\ B$, so ergibt sich mit Hilfe der Definitionen (5.5) und (5.7) auf Seite 58 gerade $A[f\,(input\ t)]$. Das existenzquantifizierte m im Ausdruck A lässt sich ebenfalls beseitigen, da es für alle Berechnungen mit dem Eingang m der Gesamtfunktion instantiiert wird. Es ergeben sich schließlich die Zeilen 12-16 in Abbildung 5.13.

Neben DFG_IFC_RESET ließe sich auch ein Schnittstellenverhaltensmuster DFG_IFC_RESET_START definieren, das dem im vorherigen Abschnitt vorgestell-

ten Schnittstellenverhaltensmuster P_IFC_RESET_START entspricht. Ausgehend von diesen Schnittstellenverhaltensmustern ließen sich dann weitere Schnittstellenverhaltensmuster definieren, worauf an dieser Stelle aber verzichtet werden soll. Im Gegensatz zu den P-Schnittstellenverhaltensmustern sind im Falle der DFG-Terme die Schnittstellenverhaltensmuster, die über kein *reset*-Signal mehr verfügen, auf jeden Fall unkritisch, da die Berechnungen immer nach derselben Zeit terminieren. Eine weitere Besonderheit liegt darin, dass bei den Schnittstellenverhaltensmustern ohne *reset*-Signal das *ready*-Signal nicht mehr unbedingt erforderlich ist, da bekannt ist, wann die Schaltung eine Berechnung beendet hat.

Kapitel 6

Schaltungsbeschreibungen auf der Systemebene – Gropius-3

Ausgehend von Beschreibungen auf der algorithmischen Ebene werden durch die Synthese Schaltungen erzeugt, die immer nur einzelne Programme ausführen. Für die Beschreibungen komplexerer Systeme, die in verstärktem Maße Berücksichtigung finden, ist das oft nicht mehr ausreichend. Solche komplexen Systeme sind nur noch dann überschaubar und handhabbar, wenn sie als Strukturen nebenläufiger, algorithmisch beschriebener Prozesse modelliert werden. In dem vorliegenden Ansatz kommunizieren dabei die beteiligten Prozesse mittels eines festgelegten Kommunikationsschemas. Dieses Kommunikationsschema ist angelehnt an Petri-Netzen höherer Ordnung [Jens92]. Daten wandern dabei wie Marken zwischen den Prozessen, und Prozesse können erst dann ihre Funktion evaluieren, wenn die Eingangsdaten anliegen. Insbesondere können Datenpakete nicht zwischen zwei Prozessen verlorengehen, indem ein Prozess sie liefert und ein anderer vergisst, sie abzuholen.

Diese Strukturen auf der Systemebene, die S-Strukturen genannt werden, zeichnen sich durch einen besonderen Aufbau aus. Im nächsten Abschnitt soll der Aufbau der S-Strukturen erläutert werden, bevor in Abschnitt 6.2 das spezielle Kommunikationsschema vorgestellt wird. Abschnitt 6.3 behandelt spezielle Prozesse namens K-Prozesse, die für die Verteilung der Datenpakete zuständig sind. S-Strukturen dienen zur Beschreibung des gemischt funktional/zeitlichen Verhaltens auf der Systemebene. In Abschnitt 6.4 wird eine Beispielstruktur vorgestellt, die sich aus einer Vielzahl von bis dahin vorgestellten Prozessen zusammensetzt. In Abschnitt 6.5 wird die funktionale Semantik von S-Strukturen eingeführt, die für Transformationen auf S-Strukturen wichtig ist. Anschließend wird in Abschnitt 6.6 gezeigt, wie mit Hilfe der K-Prozesse andere Kommunikationsschemata definiert werden können. In Abschnitt 6.7 wird eine Beschreibungsmethode für solche Einprozessbeschreibungen angegeben, die neben funktionalen auch zeitliche Aspekte aufweisen und daher auf der algorithmischen Ebene nicht ausgedrückt werden können. Schließlich wird in Abschnitt 6.8 eine Methode vorgestellt, um inkonsistente Schaltungsbeschreibungen auf der Systemebene zu vermeiden.

6.1 S-Strukturen

Strukturen auf der Systemebene zeichnen sich dadurch aus, dass die Prozesse über Kanäle kommunizieren. Diese Kanäle haben dabei einen fest vorgeschriebenen Aufbau, der in Abbildung 6.1 dargestellt ist.

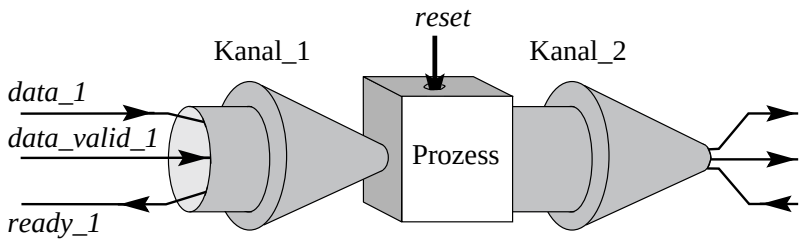


Abbildung 6.1: Schnittstelle eines Prozesses mit einem Eingangs- und einem Ausgangskanal

Alle Kanäle bestehen aus drei Leitungen: einer Datenleitung *data*, die über einen beliebigen Datentypen α verfügt, sowie zwei booleschen Kontrollleitungen *data_valid* und *ready*. Es ist zu beachten, dass das Signal *ready* in entgegengesetzter Richtung zu *data* und *data_valid* verläuft. Die Richtung des gesamten Kanals definiert sich über die Richtung des Signals *data*. Neben den Kanälen existiert außerdem ein globales *reset*-Signal (siehe Abbildung 6.1). Dieses *reset*-Signal wird an alle beteiligten Prozesse als Eingang weitergeleitet.

Aufgrund des bidirektionalen Aufbaus der Kanäle zeichnen sich Strukturen auf der Systemebene dadurch aus, dass die Kanäle jeweils nur einen Ausgang eines Prozesses mit genau einem Eingang eines anderen Prozesses verbinden, dass sie also einen Fanout von 1 haben. Sie unterscheiden sich dadurch von Strukturen auf der RT-Ebene, bei denen ein Ausgang einer Komponente an mehrere Eingänge anderer Komponenten angelegt werden darf (Fanout ≥ 1). Ohne diese Beschränkung käme es zu Kurzschlüssen, falls mehrere *ready*-Leitungen zu einer einzigen zusammengeführt würden. Die drei Teilleitungen treten nur gebündelt auf und werden als ein Kanal den Prozessen zugeführt. Ausserhalb der Prozesse kann auf die Teilsignale nicht explizit zugegriffen werden.

Um S-Strukturen zu beschreiben, können drei Arten von Prozessen verwendet werden: DFG-Term-basierte Prozesse, P-Term-basierte Prozesse und sogenannte K-Prozesse, auf die in Abschnitt 6.3 näher eingegangen wird. Die DFG- und P-Term-basierten Prozesse sind im Wesentlichen nichts anderes als bestimmte Schaltungsbeschreibungen auf der algorithmischen Ebene in Gropius-2. Sie unterscheiden sich aber dadurch von den in Kapitel 5 vorgestellten Schaltungsbeschreibungen, dass ihnen ganz spezielle Schnittstellenverhaltensmuster (P_IFC_SYSTEM, DFG_IFC_SYSTEM) zugrunde liegen, die im nächsten Abschnitt vorgestellt werden.

Für die algorithmisch beschriebenen Prozesse, die auf DFG- oder P-Termen basieren, gilt in Bezug auf ihren syntaktischen Aufbau, dass sie jeweils nur einen

Eingangs- und einen Ausgangskanal besitzen. Demgegenüber können K-Prozesse über eine beliebige Anzahl von Ein- und Ausgangskanälen verfügen. Der syntaktische Aufbau von S-Strukturen ergibt sich durch die nachfolgende BNF:

S-Schnittstelle ::=
" (" " reset " { " , " *Kanal* } ") "

DFG-Prozess ::=
" DFG_IFC_SYSTEM " " (" *DFG-Term* " , " *Taktzahl* ") " *S-Schnittstelle*

P-Prozess ::=
" P_IFC_SYSTEM " *Programm S-Schnittstelle*

K-Prozess ::=
Name { *Variable* } *S-Schnittstelle* (6.1)

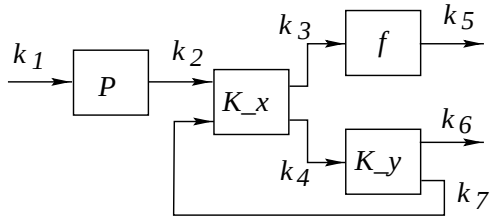
S-Prozess ::=
P-Prozess | *DFG-Prozess* | *K-Prozess* | *Prozess-Name S-Schnittstelle*

S-Struktur ::=
" ∃ " { *Kanal* } " . "
S-Prozess { " ∧ " *S-Prozess* }

S-Struktur-Definition ::=
Prozess-Name S-Schnittstelle " = " *S-Struktur*

Zur Beschreibung von S-Strukturen ist in (6.1) die Möglichkeit vorgesehen, S-Strukturen einen Namen zu geben und diese als neuen Prozess zu betrachten. Auf diese Weise können S-Strukturen auch hierarchisch beschrieben werden. Dabei *kann* eine S-Struktur mit dem Namen $S1$ eine S-Struktur $S2$ *aufrufen*, wenn die Definition von $S1$ einen Aufruf der S-Struktur $S2$ unmittelbar enthält oder in der Definition von $S1$ ein Aufruf einer S-Struktur S vorkommt, die $S2$ aufrufen kann. Rekursivität ist verboten: keine S-Struktur kann sich selbst aufrufen.

Man erkennt, dass S-Strukturen in Gropius-3 im Gegensatz zur üblichen Beschreibungsweise in Gropius relational beschrieben werden. Dies liegt daran, dass es auf der Systemebene sinnvoll und erforderlich ist, Zyklen zuzulassen. Zyklen können aber in funktionaler Weise nicht beschrieben werden, wodurch eine relationale Beschreibung erforderlich wird. Diese relationale Beschreibung von S-Strukturen orientiert sich an der üblichen relationalen Beschreibung von Schaltungsstrukturen, wie sie beispielsweise in [Melh93] vorgestellt wird. Interne Leitungen werden durch existenzquantifizierte Variablen beschrieben und die beteiligten Komponenten werden durch eine Konjunktion aufgelistet. Die externen Leitungen werden durch freie Variablen beschrieben, die implizit allquantifiziert sind. In Abbildung 6.2(b) wird zur



(a) Prozessstruktur

$$\begin{aligned}&\exists k_2\, k_3\, k_4\, k_7.\\ &\quad \mathsf{P\_IFC\_SYSTEM}\ P\ (\mathsf{reset}, k_1, k_2) \quad \wedge\\ &\quad \mathsf{K\_IFC\_x}\ (\mathsf{reset}, k_2, k_7, k_3, k_4) \quad \wedge\\ &\quad \mathsf{DFG\_IFC\_SYSTEM}\ (f, n)\ (\mathsf{reset}, k_3, k_5) \quad \wedge\\ &\quad \mathsf{K\_IFC\_y}\ (\mathsf{reset}, k_4, k_6, k_7)\end{aligned}$$

(b) S-Struktur

Abbildung 6.2: Beispiel einer S-Struktur

Verdeutlichung die formale Beschreibung der Prozessstruktur in Abbildung 6.2(a) gezeigt.

Die Folge einer relationalen Beschreibung ist, dass Inkonsistenzen in der Schaltungsbeschreibung nicht von vorneherein ausgeschlossen sind (siehe Abschnitt 4.2). Deshalb müssen mögliche Inkonsistenzen in S-Strukturen aufgespürt und evtl. beseitigt werden. Diese Problemstellung wird in Abschnitt 6.8 behandelt.

6.1.1 Verhaltenssemantik für S-Strukturen

Für einen DFG-, P-, oder K-Prozess P wird die Verhaltenssemantik $[\![P]\!]$ definiert als eine Funktion der höheren Ordnung, die alle freien Variablen von P enthält und einfach die Anwendung von P bedeutet: $[\![P]\!] \equiv P$.

Die Verhaltenssemantik eines Prozesses $Sname(\bar{a})$ einer definierten S-Struktur $Sname$ mit der Definition $Sname(\bar{x}) = SStrukt$ wird mittels

$$[\![Sname(\bar{a})]\!] \ \equiv\ (\lambda \bar{x}.\ [\![SStrukt]\!])\ \bar{a}$$

eingeführt. Die Verhaltenssemantik einer S-Struktur $S = \exists\ \bar{b}.\ S_1 \wedge \ldots \wedge S_n$ wird durch

$$[\![S]\!] \ \equiv\ \exists\ \bar{b}.\ \bigwedge_{i=1}^{n} [\![S_i]\!]$$

festgelegt. Man nennt zwei S-Strukturen S_1 und S_2 *verhaltensäquivalent*, wenn sie die gleiche Verhaltenssemantik aufweisen:

$$S_1 \cong S_2 \equiv ([\![S_1]\!] = [\![S_2]\!])$$

6.2 Kommunikationsschema auf der Systemebene

Auf der Systemebene kommunizieren die Prozesse in einer fest vorgegebenen Art und Weise. Aus diesem Grunde gibt es auf dieser Abstraktionsebene für die Beschreibung

von P- bzw. DFG-Prozessen nur zwei Schnittstellenverhaltensmuster: eines für P-Terme und eines für DFG-Terme. Beide haben jeweils genau einen Eingangs- und einen Ausgangskanal.

Das Kommunikationsschema lehnt sich an die Funktionsweise von Petri-Netzen höherer Ordnung an, bei denen jede Marke Daten enthält. Jedem Prozess entspricht eine Transition mit gewissen Plätzen am Ausgang. In Analogie zu Petri-Netzen "feuern" die Prozesse, was bedeutet, dass die Eingangsmarken abgezogen werden, auf ihren Daten eine Berechnung durchgeführt wird und anschließend das Ergebnis in Form einer neuen Marke ausgegeben wird. Das Kommunikationsschema unterscheidet sich aber in einem entscheidenden Punkt von der Funktionsweise von Petri-Netzen. Während Petri-Netze nichtdeterministisch (und meist auch nichtdeterminiert) sein können, verhalten sich S-Strukturen stets deterministisch.

Die Kommunikation und damit auch die Markenübergabe zwischen den Prozessen beruht auf einem "Handshake"-Protokoll. Gegeben seien zwei Prozesse A und B und ein Kanal zwischen A und B (in Abbildung 6.1 könnte man sich dazu an der Quelle von Kanal_1 einen Prozess A vorstellen; der abgebildete Prozess wäre dann B). Prozess A teilt seinem Nachfolger B über das Signal *data_valid* mit, ob er über das Signal *data* eine Marke an B überträgt. Andererseits signalisiert Prozess B über *ready* an A, ob er zur Aufnahme einer Marke bereit ist. Immer wenn in einem Kanal *data_valid* und *ready* gleichzeitig wahr sind, findet die Kommunikation statt, d.h. eine Marke wird auf *data* von A nach B übertragen. Das globale Signal *reset* unterbricht bei Aktivierung alle Berechnungen in den Prozessen und überführt alle Prozesse in ihren Initialzustand.

In Abbildung 6.3 ist die formale Definition des Schnittstellenverhaltensmusters P_IFC_SYSTEM angegeben, das in Kombination mit P-Termen verwendet wird. In Abbildung 6.4(a) ist eine beispielhafte schematische Darstellung zu sehen.

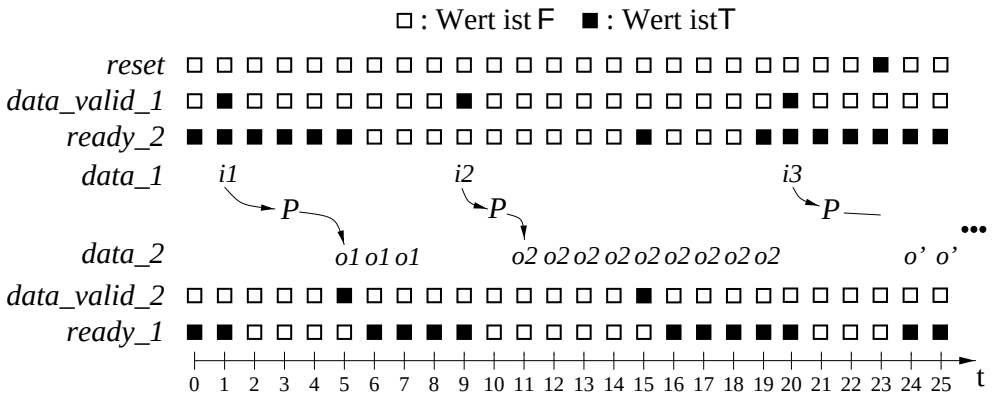
Das Schnittstellenverhaltensmuster P_IFC_SYSTEM ist eine Funktion mit mehreren Parametern. Der erste Parameter ist ein Programm P (siehe Zeile 2 in Abbildung 6.3). Die weiteren Parameter sind das globale Signal *reset* sowie die zwei Kanäle, die sich aus den Schnittstellensignalen *data_1*, *data_valid_1* und *ready_1* bzw. *data_2*, *data_valid_2* und *ready_2* zusammensetzen (Zeile 3). Um Ein- und Ausgangskanal unterscheiden zu können, wurden an die Teilsignale die Endungen *_1* bzw. *_2* angefügt. P_IFC_SYSTEM beschreibt eine Relation zwischen den Schnittstellensignalen bez. des Programms P.

Der Prozess, der sich durch die Kombination von P_IFC_SYSTEM und einem beliebigen Programm ergibt, verfügt über einen internen Zustand, der mittels *ready_1* angegeben wird. Zu Beginn hat dieser Zustand den Wert T (Zeile 4 in Abbildung 6.3 bzw. Zeitpunkt 0 in Abbildung 6.4(a)), womit angezeigt wird, dass der Prozess zur Aufnahme einer Marke bereit ist. Solange keine Berechnung auf einer Marke durchgeführt wird, d.h. *ready_1* und *data_valid_1* sind nicht gleichzeitig aktiviert, bleibt der Prozess im aufnahmebereiten Zustand und das *data_2*-Signal hält seinen letzten Wert (Zeilen 7-9, bzw. Zeitpunkte 6-8, 16-19, 24-25).

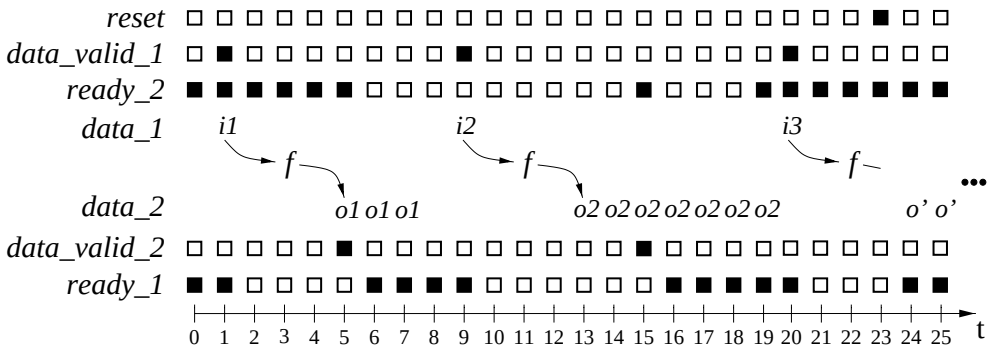
Wenn eine Berechnung gestartet wird (Zeile 10, bzw. Zeitpunkte 1,9,20), können, wie auch bei den in Abschnitt 5.2.1 erläuterten Schnittstellenverhaltensmustern für

$$
\begin{array}{ll}
\vdash \mathsf{P\_IFC\_SYSTEM} = & (1)\\
\quad \lambda P. & (2)\\
\quad \lambda\,(reset, (data\_1, data\_valid\_1, ready\_1),\ (data\_2, data\_valid\_2, ready\_2)). & (3)\\
\quad\quad ready\_1\ 0 \quad \wedge & (4)\\
\quad\quad \forall t. & (5)\\
\quad\quad\quad reset\ t\ \Rightarrow\ (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)) \quad \wedge & (6)\\
\quad\quad\quad (ready\_1\ t\ \wedge\ \neg(data\_valid\_1\ t))\ \Rightarrow & (7)\\
\quad\quad\quad\quad (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t) \wedge & (8)\\
\quad\quad\quad\quad\quad (data\_2\ t\ =\ data\_2\ (t-1))) \quad \wedge & (9)\\
\quad\quad\quad (ready\_1\ t\ \wedge\ data\_valid\_1\ t)\ \Rightarrow & (10)\\
\quad\quad\quad\quad \mathsf{PRIMREC\_partial}\ (P\ (data\_1\ t)) & (11)\\
\quad\quad\quad\quad \big(\lambda y. & (12)\\
\quad\quad\quad\quad\ \exists m. & (13)\\
\quad\quad\quad\quad\quad \forall n. & (14)\\
\quad\quad\quad\quad\quad\quad (n < m\ \wedge\ (\forall p.\ p \le n\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (15)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+n+1)) \wedge & (16)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_2\ (t+n)) & (17)\\
\quad\quad\quad\quad\quad \wedge & (18)\\
\quad\quad\quad\quad\quad \forall s. & (19)\\
\quad\quad\quad\quad\quad\quad (\forall p.\ p \le m+s\ \Rightarrow\ \neg(reset\ (t+p)))\ \Rightarrow & (20)\\
\quad\quad\quad\quad\quad\quad\quad (ready\_2\ (t+m+s) \wedge & (21)\\
\quad\quad\quad\quad\quad\quad\quad\quad (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_2\ (t+m+p))))\ \Rightarrow & (22)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad ready\_1\ (t+m+s+1) \wedge & (23)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad data\_valid\_2\ (t+m+s) \wedge & (24)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (data\_2\ (t+m+s) = y) & (25)\\
\quad\quad\quad\quad\quad\quad\quad \wedge & (26)\\
\quad\quad\quad\quad\quad\quad\quad (\forall p.\ p \le s\ \Rightarrow\ \neg(ready\_2\ (t+m+p)))\ \Rightarrow & (27)\\
\quad\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+m+s+1)) \wedge & (28)\\
\quad\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_2\ (t+m+s)) \wedge & (29)\\
\quad\quad\quad\quad\quad\quad\quad\quad (data\_2\ (t+m+s) = y)\big) & (30)\\
\quad\quad\quad\quad \big(\forall m. & (31)\\
\quad\quad\quad\quad\quad\ (\forall n.\ n \le m\ \Rightarrow\ \neg(reset\ (t+n)))\ \Rightarrow & (32)\\
\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+m+1)) \wedge & (33)\\
\quad\quad\quad\quad\quad\quad \neg(data\_valid\_2\ (t+m))\big) & (34)
\end{array}
$$

Abbildung 6.3: Formale Definition des Schnittstellenverhaltensmusters P_IFC_SYSTEM für die Systemebene



(a) P_IFC_SYSTEM



(b) DFG_IFC_SYSTEM

Abbildung 6.4: Schematische Darstellung der Schnittstellenverhaltensmuster für die Systemebene

P-Terme, zwei Fälle auftreten:

I: Die Funktionsanwendung $P\ (data\_1\ t)$ terminiert mit einem gewissen Resultat Defined y (Zeilen 12-30)

II: Die Funktionsanwendung terminiert nicht und das Ergebnis ist Undefined (Zeilen 31-34)

Im zweiten Fall bleiben *ready_1* und *data_valid_2* solange F, bis mit Hilfe des *reset*-Signals die Berechnung abgebrochen wird (siehe Zeile 6 und Zeitpunkt 23). Terminiert die Funktionsanwendung dagegen, bleiben *ready_1* und *data_valid_2* zumindest während der Berechnungsdauer F, sofern die Berechnung nicht durch *reset* abgebrochen wird (Zeilen 14-17, Zeitpunkte 2-4, 10, 21-22). Die Dauer für die Funktionsauswertung eines P-Terms hängt vom Eingang $(data\_1\ t)$ ab und ist deshalb nicht festgelegt ("$\exists m$" in Zeile 13). Wenn die Berechnung $P\ (data\_1\ t)$ beendet ist, können zwei verschiedene Fälle auftreten:

Ia: Der Nachfolgeprozess ist am Ende der Berechnung aufnahmebereit: *ready_2* $(t+m)$, Zeile 21 mit $s=0$, Zeitpunkt 5

Ib: Der Nachfolgeprozess ist am Ende der Berechnung *nicht* aufnahmebereit: $\neg$(*ready_2* $(t+m)$), Zeile 27 mit $s>0$, Zeitpunkt 11

Im ersten Fall wird das Resultat sofort an den Nachfolger weitergeleitet. Dazu wird *data_valid_2* gesetzt, und der Nachfolger liest über *data_2* die neue Marke ein. Einen Takt später ist *ready_1* wieder T und der betrachtete Prozess somit wieder aufnahmebereit (Zeilen 21-25 mit $s=0$, Zeitpunkte 5-6). Ist dagegen der Nachfolger am Ende der Berechnung selber noch beschäftigt, wird das Ergebnis solange gehalten, bis der Nachfolger aufnahmebereit ist, sofern der Vorgang nicht durch *reset* abgebrochen wird (Zeile 20; es ist möglich, dass der Nachfolger selbst ein P-Prozess ist, der mit einer nicht terminierenden Berechnung beschäftigt ist). In dieser Zeit bleibt *ready_1* F und somit auch der betrachtete Prozess nicht aufnahmebereit (Zeilen 27-30, Zeitpunkte 11-14). Sobald der Nachfolger aufnahmebereit ist (Zeile 21 mit $s>0$, Zeitpunkt 15), wird *data_valid_2* gesetzt, die Marke wird übertragen, und einen Takt später geht der Prozess wieder in den aufnahmebereiten Zustand über (Zeilen 21-25 mit $s>0$, Zeitpunkte 15-16).

Beim Kommunikationsschema auf der Systemebene wird ebenso wie beim Schnittstellenverhaltensmuster P_IFC_RESET, das in Abschnitt 5.2.1 vorgestellt wurde, ein dominantes *reset*-Signal verwendet (siehe Zeile 6 in Abbildung 6.3). Das bedeutet, dass unabhängig von den anderen Signalen durch Aktivieren von *reset* alle laufenden Berechnungen abgebrochen werden. In Abschnitt 5.2.1 wurde als Alternative zu P_IFC_RESET das Schnittstellenverhaltensmuster P_IFC_RESET_START eingeführt, welches über ein dominantes *start*-Signal verfügt. Ein solches Verhalten wäre aber auf der Systemebene unerwünscht. Wäre das Schnittstellenverhaltensmuster P_IFC_SYSTEM mit einem dominanten *data_valid_1*-Signal definiert, d.h. bei gleichzeitiger Aktivierung von *data_valid_1* und *reset* würde sich *data_valid_1* durchsetzen und eine neue Berechnung starten, ergäbe sich ein nichtdeterminiertes Verhalten. Dazu betrachte man wieder zwei Prozesse mit einem gemeinsamen Kanal. Würde im ersten Prozess gerade eine Berechnung laufen, die gerade in dem Augenblick terminiert, in dem *reset* aktiviert wird, würde der zweite Prozess anstatt in den Initialzustand zu gehen, mit einer Berechnung beginnen – vorausgesetzt er ist gerade im aufnahmebereiten Zustand. Da bei P-Prozessen der Zeitpunkt der Terminierung in der Regel unbekannt ist, wäre nicht vorhersehbar, welche Prozesse durch das Setzen von *reset* in den Initialzustand gehen würden.

In Abbildung 6.5 ist die formale Definition für das entsprechende Schnittstellenverhaltensmuster für DFG-Terme angegeben. Analog zu den in Abschnitt 5.2.2 vorgestellten Schnittstellenverhaltensmustern erwartet DFG_IFC_SYSTEM anstelle eines P-Terms ein Paar bestehend aus DFG-Term f und natürlicher Zahl m. Der Unterschied zwischen DFG_IFC_SYSTEM und P_IFC_SYSTEM besteht darin, dass die Funktionsanwendungen f (*data_1* t) immer terminieren und dies immer innerhalb einer bestimmten Taktzahl m, wenn die Berechnung nicht vorher durch *reset* abgebrochen wurde. DFG_IFC_SYSTEM lässt sich aus P_IFC_SYSTEM gewinnen, indem in

$$
\begin{array}{ll}
\vdash \mathsf{DFG\_IFC\_SYSTEM} = & (1)\\
\quad \lambda(f, m). & (2)\\
\quad \lambda\,(\mathit{reset}, (\mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}),\ (\mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2})). & (3)\\
\qquad \mathit{ready\_1}\ 0 \quad \wedge & (4)\\
\qquad \forall t. & (5)\\
\qquad\quad \mathit{reset}\ t\ \Rightarrow\ (\mathit{ready\_1}\ (t+1)\ \wedge\ \neg(\mathit{data\_valid\_2}\ t)) \quad \wedge & (6)\\
\qquad\quad (\mathit{ready\_1}\ t\ \wedge\ \neg(\mathit{data\_valid\_1}\ t))\ \Rightarrow & (7)\\
\qquad\qquad (\mathit{ready\_1}\ (t+1)\ \wedge\ \neg(\mathit{data\_valid\_2}\ t) \wedge & (8)\\
\qquad\qquad\quad (\mathit{data\_2}\ t\ =\ \mathit{data\_2}\ (t-1))) \quad \wedge & (9)\\
\qquad\quad (\mathit{ready\_1}\ t\ \wedge\ \mathit{data\_valid\_1}\ t)\ \Rightarrow & (10)\\
\qquad\qquad \forall n. & (11)\\
\qquad\qquad\quad (n < m\ \wedge\ (\forall p.\ p \leq n\ \Rightarrow\ \neg(\mathit{reset}\ (t+p))))\ \Rightarrow & (12)\\
\qquad\qquad\qquad \neg(\mathit{ready\_1}\ (t+n+1)) \wedge & (13)\\
\qquad\qquad\qquad \neg(\mathit{data\_valid\_2}\ (t+n)) & (14)\\
\qquad\qquad \wedge & (15)\\
\qquad\qquad \forall s. & (16)\\
\qquad\qquad\quad (\forall p.\ p \leq m+s\ \Rightarrow\ \neg(\mathit{reset}\ (t+p)))\ \Rightarrow & (17)\\
\qquad\qquad\qquad (\mathit{ready\_2}\ (t+m+s) \wedge & (18)\\
\qquad\qquad\qquad\quad (\forall p.\ p < s\ \Rightarrow\ \neg(\mathit{ready\_2}\ (t+m+p))))\ \Rightarrow & (19)\\
\qquad\qquad\qquad\quad \mathit{ready\_1}\ (t+m+s+1) \wedge & (20)\\
\qquad\qquad\qquad\quad \mathit{data\_valid\_2}\ (t+m+s) \wedge & (21)\\
\qquad\qquad\qquad\quad (\mathit{data\_2}\ (t+m+s) = f\ (\mathit{data\_1}\ t)) & (22)\\
\qquad\qquad\qquad \wedge & (23)\\
\qquad\qquad\qquad (\forall p.\ p \leq s\ \Rightarrow\ \neg(\mathit{ready\_2}\ (t+m+p)))\ \Rightarrow & (24)\\
\qquad\qquad\qquad\quad \neg(\mathit{ready\_1}\ (t+m+s+1)) \wedge & (25)\\
\qquad\qquad\qquad\quad \neg(\mathit{data\_valid\_2}\ (t+m+s)) \wedge & (26)\\
\qquad\qquad\qquad\quad (\mathit{data\_2}\ (t+m+s) = f\ (\mathit{data\_1}\ t)) & (27)
\end{array}
$$

Abbildung 6.5: Formale Definition des Schnittstellenverhaltensmusters DFG_IFC_SYSTEM für die Systemebene

Abbildung 6.3 ab Zeile 11 der Ausdruck PRIMREC_partial $(P\,(\mathit{data\_1}\ t))\ (\lambda y.\ A[y])\ B$ durch den Ausdruck PRIMREC_partial $((\mathsf{PARTIALIZE}\ f)\,(\mathit{data\_1}\ t))\ (\lambda y.\ A[y])\ B$ ersetzt und mit Hilfe der Definitionen (5.5) und (5.7) auf Seite 58 umgeformt wird. Abbildung 6.4(b) zeigt beispielhaft in schematischer Weise einen Zeitverlauf für die Kombination von DFG_IFC_SYSTEM mit einem gewissen DFG-Term f und der Taktzahl 4.

6.3 K-Prozesse

Um den Markenfluss und die Kommunikation zwischen P- und DFG-Prozessen zu steuern, werden sogenannte K-Prozesse angeboten. Sie können als eine Art

"glue logic" angesehen werden, um Datenpakete zu verteilen, zusammenzuführen, zu synchronisieren etc. Es gibt acht elementare K-Prozesse: Double, Join, Split, Synchronize, Fork, Choose, Counter und Sink. In Abbildung 6.6 sind sie schematisch dargestellt.

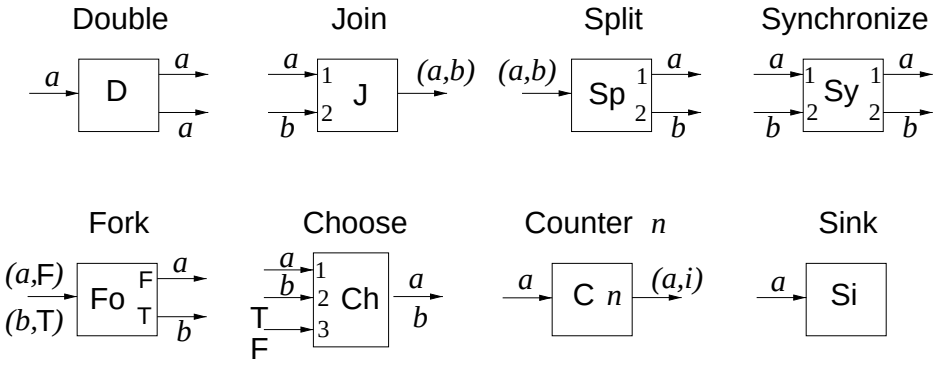


Abbildung 6.6: Elementare K-Prozesse

Die K-Prozesse unterscheiden sich von den DFG- und P-Prozessen, indem sie über mehrere Eingangs- und Ausgangskanäle verfügen können und indem sie nur über ihr Schnittstellenverhalten definiert sind. Dieses Schnittstellenverhalten genügt dem im vorigen Abschnitt vorgestellten markenbasierten Kommunikationsschema. In den K-Prozessen werden die Marken in bestimmter Weise weitergereicht; es werden aber keine Berechnungen auf ihnen durchgeführt. Aus diesem Grunde ergibt sich die Definition eines K-Prozesses nur aus einer Beschreibung, die eine Relation zwischen den Teilsignalen der beteiligten Kanäle und dem *reset*-Signal ausdrückt. Im Folgenden werden die formalen Definitionen der elementaren K-Prozesse vorgestellt.

6.3.1 Der K-Prozess Join

Der K-Prozess Join wird dazu verwendet, um zwei separate Marken zu einer einzigen zusammenzufassen. Damit verbunden ist eine Synchronisierung des Markenflusses. Die beiden Marken werden zu i.Allg. unterschiedlichen Zeitpunkten eingelesen und in Abhängigkeit von der Aufnahmebereitschaft des Nachfolgeprozesses wird die zusammengefasste Marke zu einem bestimmten Zeitpunkt ausgegeben. Der Datentyp der beiden einzulesenden Marken ist beliebig, aber fest. Zwei Marken vom Typ α bzw. β werden zu einer Marke vom Typ $(\alpha \times \beta)$ zusammengefasst.

Abbildung 6.7 zeigt die formale Definition des K-Prozesses Join. Die Konstante K_IFC_JOIN beschreibt eine Relation zwischen den Teilsignalen der Eingangskanäle (*Kanal_1*, *Kanal_2*) und denen des Ausgangskanals (*Kanal_3*) sowie dem *reset*-Signal. Wie jeder S-Prozess ist Join zu Beginn im aufnahmebereiten Zustand (Zeile 3). In diesen wird er durch Aktivieren von *reset* in jedem Fall zurückgeführt (Zeilen 5,6). Ist Join aufnahmebereit, erhält aber über beide Eingangskanäle keine Marke, ist er auch im nächsten Takt aufnahmebereit und gibt auch keine Marke an den

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_JOIN} = \lambda\ (reset, (data\_1, data\_valid\_1, ready\_1), (data\_2, data\_valid\_2, ready\_2), & (1)\\
\qquad (data\_3, data\_valid\_3, ready\_3)). & (2)\\
\quad ready\_1\ 0 \wedge ready\_2\ 0 \quad \wedge & (3)\\
\quad \forall t. & (4)\\
\qquad reset\ t \Rightarrow & (5)\\
\qquad\quad (ready\_1\ (t+1) \wedge ready\_2\ (t+1) \wedge \neg(data\_valid\_3\ t)) \quad \wedge & (6)\\
\qquad (ready\_1\ t \wedge ready\_2\ t \wedge \neg(data\_valid\_1\ t) \wedge \neg(data\_valid\_2\ t)) \Rightarrow & (7)\\
\qquad\quad (ready\_1\ (t+1) \wedge ready\_2\ (t+1) \wedge \neg(data\_valid\_3\ t) \wedge & (8)\\
\qquad\quad\ (data\_3\ t = data\_3\ (t-1))) \quad \wedge & (9)\\
\qquad (ready\_1\ t \wedge ready\_2\ t \wedge data\_valid\_1\ t) \Rightarrow & (10)\\
\qquad\quad \forall m.\ (\forall p.\ p \leq m \Rightarrow \neg(reset\ (t+p)))) \Rightarrow & (11)\\
\qquad\quad\ (data\_valid\_2\ (t+m) \wedge (\forall p.\ p < m \Rightarrow \neg(data\_valid\_2\ (t+p)))) \Rightarrow & (12)\\
\qquad\quad\ \forall s.\ (\forall p.\ p \leq s \Rightarrow \neg(reset\ (t+m+p)))) \Rightarrow & (13)\\
\qquad\quad\ (ready\_3\ (t+m+s) \wedge (\forall p.\ p < s \Rightarrow \neg(ready\_3\ (t+m+p)))) \Rightarrow & (14)\\
\qquad\quad\ ready\_1\ (t+m+s+1) \wedge & (15)\\
\qquad\quad\ ready\_2\ (t+m+s+1) \wedge & (16)\\
\qquad\quad\ data\_valid\_3\ (t+m+s) \wedge & (17)\\
\qquad\quad\ (data\_3\ (t+m+s) = (data\_1\ t, data\_2\ (t+m))) & (18)\\
\qquad\quad\ \wedge & (19)\\
\qquad\quad\ (\forall p.\ p \leq s \Rightarrow \neg(ready\_3\ (t+m+p))) \Rightarrow & (20)\\
\qquad\quad\ \neg(ready\_1\ (t+m+s+1)) \wedge & (21)\\
\qquad\quad\ \neg(ready\_2\ (t+m+s+1)) \wedge & (22)\\
\qquad\quad\ \neg(data\_valid\_3\ (t+m+s)) \wedge & (23)\\
\qquad\quad\ (data\_3\ (t+m+s) = (data\_1\ t, data\_2\ (t+m))) & (24)\\
\qquad\quad \wedge & (25)\\
\qquad\quad (\forall p.\ p \leq m \Rightarrow \neg(data\_valid\_2\ (t+p))) \Rightarrow & (26)\\
\qquad\quad\ \neg(ready\_1\ (t+m+1)) \wedge & (27)\\
\qquad\quad\ ready\_2\ (t+m+1) \wedge & (28)\\
\qquad\quad\ \neg(data\_valid\_3\ (t+m)) \wedge & (29)\\
\qquad\quad\ (\mathsf{FST}(data\_3\ (t+m) = data\_1\ t)) & (30)\\
\quad \wedge & (31)\\
\quad (ready\_1\ t \wedge ready\_2\ t \wedge data\_valid\_2\ t) \Rightarrow & (32)\\
\qquad \forall m.\ (\forall p.\ p \leq m \Rightarrow \neg(reset\ (t+p)))) \Rightarrow & (33)\\
\qquad\quad (data\_valid\_1\ (t+m) \wedge (\forall p.\ p < m \Rightarrow \neg(data\_valid\_1\ (t+p)))) \Rightarrow & (34)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s \Rightarrow \neg(reset\ (t+m+p)))) \Rightarrow & (35)\\
\qquad\quad\ (ready\_3\ (t+m+s) \wedge (\forall p.\ p < s \Rightarrow \neg(ready\_3\ (t+m+p)))) \Rightarrow & (36)\\
\qquad\quad\ ready\_1\ (t+m+s+1) \wedge & (37)\\
\qquad\quad\ ready\_2\ (t+m+s+1) \wedge & (38)\\
\qquad\quad\ data\_valid\_3\ (t+m+s) \wedge & (39)\\
\qquad\quad\ (data\_3\ (t+m+s) = (data\_1\ (t+m), data\_2\ t)) & (40)\\
\qquad\quad\ \wedge & (41)\\
\qquad\quad\ (\forall p.\ p \leq s \Rightarrow \neg(ready\_3\ (t+m+p))) \Rightarrow & (42)\\
\qquad\quad\ \neg(ready\_1\ (t+m+s+1)) \wedge & (43)\\
\qquad\quad\ \neg(ready\_2\ (t+m+s+1)) \wedge & (44)\\
\qquad\quad\ \neg(data\_valid\_3\ (t+m+s)) \wedge & (45)\\
\qquad\quad\ (data\_3\ (t+m+s) = (data\_1\ (t+m), data\_2\ t)) & (46)\\
\qquad\quad \wedge & (47)\\
\qquad\quad (\forall p.\ p \leq m \Rightarrow \neg(data\_valid\_1\ (t+p))) \Rightarrow & (48)\\
\qquad\quad\ ready\_1\ (t+m+1) \wedge & (49)\\
\qquad\quad\ \neg(ready\_2\ (t+m+1)) \wedge & (50)\\
\qquad\quad\ \neg(data\_valid\_3\ (t+m)) \wedge & (51)\\
\qquad\quad\ (\mathsf{SND}(data\_3\ (t+m) = data\_1\ t)) & (52)
\end{array}
$$

Abbildung 6.7: Formale Definition von K_IFC_JOIN

Nachfolgeprozess weiter. Ferner wird in diesem Fall am Ausgang der letzte Wert gehalten (Zeilen 7-9). Bis hierher unterscheidet sich Join kaum von P- bzw. DFG-Prozessen. Der für den Prozess Join charakteristische Teil der Spezifikation, der die Weitergabe von Marken beschreibt, beginnt mit Zeile 10. In den Zeilen 10 bis 30 wird der Ablauf beschrieben, wenn beide Eingangskanäle bereit sind (die *ready*-Signale haben den Wert T) und im ersten Kanal eine Marke eingelesen wird. Die Zeilen 32 bis 52 behandeln den Fall, dass bei aufnahmebereiten Eingangskanälen im zweiten Kanal eine Marke eingelesen wird. Die beiden Fälle beschreiben die folgenden Möglichkeiten:

1. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Sobald der Nachfolger aufnahmebereit ist, wird die zusammengefasste Marke an den Nachfolger ausgegeben.

2. Der Vorgänger-Prozess, der mit Join über den zweiten (ersten) Kanal verbunden ist, liefert zu keinem Zeitpunkt eine Marke. Eine mögliche Erklärung dafür ist, dass er als P-Prozess mit einer nichtterminierenden Berechnung beschäftigt ist. Die Konsequenz ist, dass Join selbst nie eine Marke ausgeben kann.

3. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Da der Nachfolger aber zu keinem Zeitpunkt aufnahmebereit ist, kann die zusammengefasste Marke nicht weitergegeben werden.

4. Bevor über den zweiten (ersten) Kanal eine Marke eingelesen werden kann, wird der Vorgang durch *reset* gestoppt.

5. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Bevor die zusammengefasste Marke weitergegeben werden kann, wird ein Reset ausgelöst.

Im Folgenden soll kurz der Fall erläutert werden, bei dem zunächst über den ersten Kanal eine Marke eingelesen wird (Zeilen 10-30). Der Ablauf im anderen Fall, dass zuerst eine Marke über *Kanal_2* aufgenommen wird, ergibt sich entsprechend. Wird zu einem Zeitpunkt t über den ersten Kanal eine Marke eingelesen, liegen an den Ausgangssignalen von Join die folgenden Werte an, solange über den zweiten Kanal keine Marke eingelesen wird (Zeilen 26-30): *ready_1* ist F, was bedeutet, dass über *Kanal_1* keine weitere Marke eingelesen werden kann; *ready_2* ist T, da *Kanal_2* noch auf eine Eingabe wartet; *data_valid_3* ist F, da noch keine zusammengefasste Marke ausgegeben werden kann. Am Signal *data_3* liegt bereits der erste Teil an, der gerade (*data_1* t) ist. Der zweite Teil von *data_3* ist unbestimmt. Wird zu einem gewissen Zeitpunkt $(t + m)$ eine Marke über den zweiten Kanal eingelesen (Zeile 12), muss gewartet werden, bis der Nachfolger bereit ist, d.h. bis *ready_3* den Wert T annimmt. Existiert ein solcher Zeitpunkt nicht, tritt Möglichkeit Nummer zwei ein. Die Zeilen 20-24 beschreiben die Situation des Wartens: neben *ready_1* hat nun auch *ready_2* den Wert F, da keine neuen Marken eingelesen werden dürfen, solange die zusammengefasste alte Marke noch nicht weitergegeben wurde. Auch

data_valid_3 hat immer noch den Wert F. Das Signal *data_3* verfügt jetzt auch über einen konkreten zweiten Anteil: *data_2* $(t + m)$. Ist der Nachfolger zu einem gewissen Zeitpunkt $(t + m + s)$ aufnahmebereit (Zeile 14), wird die Marke weitergegeben (*data_valid_3* hat den Wert T), und einen Takt später sind beide Eingangskanäle wieder aufnahmebereit. Gibt es aber einen solchen Zeitpunkt nicht, tritt die oben skizzierte dritte Möglichkeit ein, dass die zusammengefasste Marke nicht weitergegeben werden kann. Ist die Bedingung in Zeile 11 bzw. die in Zeile 13 nicht erfüllt, kommt es zu einem vorzeitigen Reset (Möglichkeiten vier bzw. fünf).

6.3.2 Die K-Prozesse Double und Split

Mit Hilfe des K-Prozesses Double wird eine Marke von einem Vorgängerprozess aufgenommen, dupliziert und an zwei Nachfolgeprozesse weitergereicht. Je nach Aufnahmebereitschaft der Nachfolger kann die Weitergabe zu unterschiedlichen Zeitpunkten erfolgen. Der Prozess Split ist der inverse Prozess zu Join. Sein Verhalten unterscheidet sich kaum von dem des Prozesses Double. Der einzige Unterschied besteht in der Ausgabe über die beiden Ausgangskanäle. Während Double an beiden Ausgängen die gleiche Marke weitergibt, leitet Split an den Ausgängen jeweils einen Teil einer gepaarten Marke weiter. Abbildung 6.8 zeigt die formalen Definitionen von Double und Split. Sie unterscheiden sich lediglich in den Zeilen 18-19, 25-26, 34-35, 41-42 sowie 48-49.

Die Konstanten K_IFC_DOUBLE bzw. K_IFC_SPLIT beschreiben eine Relation zwischen den Teilsignalen des Eingangskanals (*Kanal_1*) und denen der zwei Ausgangskanäle (*Kanal_2*, *Kanal_3*) sowie dem *reset*-Signal. Ab Zeile 10 in Abbildung 6.8 wird beschrieben, wie Double bzw. Split eine eingehende Marke verarbeiten. Die Art und Weise, wie die Prozesse diese Marke weitergeben, hängt ganz allein von den beiden Nachfolgern (und selbstverständlich *reset*) ab. Ein Nachfolger erhält dabei seine Marke, sobald er selbst aufnahmebereit ist. Es gibt neun unterschiedliche Möglichkeiten:

1. Beide Marken werden an die Nachfolger gleichzeitig ausgegeben.
2. Der erste Nachfolger erhält seine Marke, bevor der zweite Nachfolger seine bekommt.
3. Der zweite Nachfolger erhält seine Marke, bevor der erste Nachfolger seine bekommt.
4. Beide Nachfolger sind mit nicht terminierenden Berechnungen beschäftigt und bekommen ihre Marken daher nie.
5. Der erste Nachfolger erhält seine Marke, während der zweite Nachfolger seine nie bekommt.
6. Der zweite Nachfolger erhält seine Marke, während der erste Nachfolger seine nie bekommt.

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_DOUBLE} & \\
\vdash \mathsf{K\_IFC\_SPLIT} & = \lambda\ (reset, (data\_1, data\_valid\_1, ready\_1), \quad (1)\\
& \qquad (data\_2, data\_valid\_2, ready\_2), (data\_3, data\_valid\_3, ready\_3)). \quad (2)\\
ready\_1\ 0\ \wedge & \quad (3)\\
\forall t. & \quad (4)\\
\quad reset\ t\ \Rightarrow & \quad (5)\\
\quad\quad (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge\ \neg(data\_valid\_3\ t))\ \wedge & \quad (6)\\
\quad (ready\_1\ t\ \wedge\ \neg(data\_valid\_1\ t))\ \Rightarrow & \quad (7)\\
\quad\quad (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge\ \neg(data\_valid\_3\ t)\ \wedge & \quad (8)\\
\quad\quad\ (data\_2\ t\ =\ data\_2\ (t-1))\ \wedge\ (data\_3\ t\ =\ data\_3\ (t-1)))\ \wedge & \quad (9)\\
\quad (ready\_1\ t\ \wedge\ data\_valid\_1\ t)\ \Rightarrow & \quad (10)\\
\quad\quad \forall s.\ (\forall p.\ p \le s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & \quad (11)\\
\quad\quad\quad (ready\_2\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_2\ (t+p))\ \wedge\ \neg(ready\_3\ (t+p))))\ \Rightarrow & \quad (12)\\
\quad\quad\quad\quad \forall m.\ (\forall p.\ p \le m\ \Rightarrow\ \neg(reset\ (t+s+p))))\ \Rightarrow & \quad (13)\\
\quad\quad\quad\quad\quad (ready\_3\ (t+s+m)\ \wedge\ (\forall p.\ p < m\ \Rightarrow\ \neg(ready\_3\ (t+s+p))))\ \Rightarrow & \quad (14)\\
\quad\quad\quad\quad\quad\quad ready\_1\ (t+s+m+1)\ \wedge & \quad (15)\\
\quad\quad\quad\quad\quad\quad \mathsf{MUX}(m=0, data\_valid\_2\ (t+s), \neg(data\_valid\_2\ (t+s+m)))\ \wedge & \quad (16)\\
\quad\quad\quad\quad\quad\quad data\_valid\_3\ (t+s+m)\ \wedge & \quad (17)\\
\quad\quad\quad\quad\quad\quad \begin{array}{l} data\_2\ (t+s+m) = \\ data\_3\ (t+s+m) = \end{array} \mathsf{Double:} \begin{array}{l} data\_1\ t \\ data\_1\ t \end{array} \mathsf{Split:} \begin{array}{l} \mathsf{FST}(data\_1\ t)\ \wedge \\ \mathsf{SND}(data\_1\ t) \end{array} & \quad (18)\text{–}(19)\\
\quad\quad\quad\quad\quad\quad \wedge & \quad (20)\\
\quad\quad\quad\quad\quad\quad (\forall p.\ p \le m\ \Rightarrow\ \neg(ready\_3\ (t+s+p)))\ \Rightarrow & \quad (21)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+s+m+1))\ \wedge & \quad (22)\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{MUX}(m=0, data\_valid\_2\ (t+s), \neg(data\_valid\_2\ (t+s+m)))\ \wedge & \quad (23)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+s+m))\ \wedge & \quad (24)\\
\quad\quad\quad\quad\quad\quad\quad \begin{array}{l} data\_2\ (t+s+m) = \\ data\_3\ (t+s+m) = \end{array} \mathsf{Double:} \begin{array}{l} data\_1\ t \\ data\_1\ t \end{array} \mathsf{Split:} \begin{array}{l} \mathsf{FST}(data\_1\ t)\ \wedge \\ \mathsf{SND}(data\_1\ t) \end{array} & \quad (25)\text{–}(26)\\
\quad\quad\quad\quad \wedge & \quad (27)\\
\quad\quad\quad\quad (ready\_3\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_2\ (t+p))\ \wedge\ \neg(ready\_3\ (t+p))))\ \Rightarrow & \quad (28)\\
\quad\quad\quad\quad\quad \forall m.\ (\forall p.\ p \le m\ \Rightarrow\ \neg(reset\ (t+s+p))))\ \Rightarrow & \quad (29)\\
\quad\quad\quad\quad\quad\quad (ready\_2\ (t+s+m)\ \wedge\ (\forall p.\ p < m\ \Rightarrow\ \neg(ready\_2\ (t+s+p))))\ \Rightarrow & \quad (30)\\
\quad\quad\quad\quad\quad\quad\quad ready\_1\ (t+s+m+1)\ \wedge & \quad (31)\\
\quad\quad\quad\quad\quad\quad\quad data\_valid\_2\ (t+s+m)\ \wedge & \quad (32)\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{MUX}(m=0, data\_valid\_3\ (t+s), \neg(data\_valid\_3\ (t+s+m)))\ \wedge & \quad (33)\\
\quad\quad\quad\quad\quad\quad\quad \begin{array}{l} data\_2\ (t+s+m) = \\ data\_3\ (t+s+m) = \end{array} \mathsf{Double:} \begin{array}{l} data\_1\ t \\ data\_1\ t \end{array} \mathsf{Split:} \begin{array}{l} \mathsf{FST}(data\_1\ t)\ \wedge \\ \mathsf{SND}(data\_1\ t) \end{array} & \quad (34)\text{–}(35)\\
\quad\quad\quad\quad\quad\quad \wedge & \quad (36)\\
\quad\quad\quad\quad\quad\quad (\forall p.\ p \le m\ \Rightarrow\ \neg(ready\_2\ (t+s+p)))\ \Rightarrow & \quad (37)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+s+m+1))\ \wedge & \quad (38)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_2\ (t+s+m))\ \wedge & \quad (39)\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{MUX}(m=0, data\_valid\_3\ (t+s), \neg(data\_valid\_3\ (t+s+m)))\ \wedge & \quad (40)\\
\quad\quad\quad\quad\quad\quad\quad \begin{array}{l} data\_2\ (t+s+m) = \\ data\_3\ (t+s+m) = \end{array} \mathsf{Double:} \begin{array}{l} data\_1\ t \\ data\_1\ t \end{array} \mathsf{Split:} \begin{array}{l} \mathsf{FST}(data\_1\ t)\ \wedge \\ \mathsf{SND}(data\_1\ t) \end{array} & \quad (41)\text{–}(42)\\
\quad\quad\quad\quad \wedge & \quad (43)\\
\quad\quad\quad\quad (\forall p.\ p \le s\ \Rightarrow\ \neg(ready\_2\ (t+p))\ \wedge\ \neg(ready\_3\ (t+p))))\ \Rightarrow & \quad (44)\\
\quad\quad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge & \quad (45)\\
\quad\quad\quad\quad\quad \neg(data\_valid\_2\ (t+s))\ \wedge & \quad (46)\\
\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+s))\ \wedge & \quad (47)\\
\quad\quad\quad\quad\quad \begin{array}{l} data\_2\ (t+s+m) = \\ data\_3\ (t+s+m) = \end{array} \mathsf{Double:} \begin{array}{l} data\_1\ t \\ data\_1\ t \end{array} \mathsf{Split:} \begin{array}{l} \mathsf{FST}(data\_1\ t)\ \wedge \\ \mathsf{SND}(data\_1\ t) \end{array} & \quad (48)\text{–}(49)
\end{array}
$$

Abbildung 6.8: Formale Definitionen von K_IFC_DOUBLE und K_IFC_SPLIT

7. Beide Nachfolger bekommen ihre Marken nicht, da die Weitergabe durch *reset* abgebrochen wurde.

8. Der erste Nachfolger erhält seine Marke, der Zweite wegen *reset* nicht.

9. Der zweite Nachfolger erhält seine Marke, der erste wegen *reset* nicht.

Die Zeilen 12-26 in Abbildung 6.8 beschreiben die Situation, in der der erste Nachfolger zu einem Zeitpunkt $(t+s)$ aufnahmebereit ist und dann auch seine Marke erhält. Gibt es einen Zeitpunkt $(t+s+m)$, an dem auch der andere Nachfolger bereit ist, tritt entweder die erste Möglichkeit ein $(m=0)$ oder die zweite $(m>0)$. Gibt es diesen Zeitpunkt nicht, entsteht Möglichkeit Nummer fünf. Die Möglichkeiten drei und sechs sind in den Zeilen 28-42 beschrieben, wobei auch hier nochmals die erste Möglichkeit vorkommt. In den Zeilen 44-49 ist beschrieben, was geschieht, solange keiner der Nachfolger aufnahmebereit ist: Double bzw. Split selbst sind beschäftigt, keine Marke wird weitergegeben, die Marken liegen aber bereits an den Ausgängen an. Gibt es keinen Zeitpunkt s, in dem die Bedingung in Zeile 44 nicht erfüllt ist, tritt die oben erwähnte vierte Möglichkeit ein. Die siebte Möglichkeit tritt ein, wenn es einen Zeitpunkt s gibt, in dem die Bedingung in Zeile 11 nicht erfüllt ist. Ist die Bedingung in Zeile 13 bzw. 29 für ein gewisses m nicht erfüllt, tritt der achte bzw. neunte Fall ein.

6.3.3 Der K-Prozess Synchronize

Bei dem Prozess Join werden zwei Marken zumeist unterschiedlichen Datentyps zu in der Regel unterschiedlichen Zeitpunkten eingelesen und zusammengefasst. Im Gegensatz dazu führt der K-Prozess Synchronize eine andere Form der Synchronisation von Markenflüssen durch. Auch hier werden zwei Marken, die i.Allg. einen unterschiedlichen Datentyp haben, zu gewöhnlich unterschiedlichen Zeitpunkten eingelesen. Sie werden aber nicht zusammengefasst, sondern bleiben unverändert. Die beiden Marken werden erst dann *gleichzeitig* weitergegeben, wenn *beide* Nachfolge-Prozesse aufnahmebereit sind.

Abbildung 6.9 zeigt die formale Definition von Synchronize. Die Konstante K_IFC_SYNCHRONIZE beschreibt eine Relation zwischen den Teilsignalen der Eingangskanäle (*Kanal_1*, *Kanal_2*) und denen der Ausgangskanäle (*Kanal_3*, *Kanal_4*) sowie dem *reset*-Signal. Der für den Prozess Synchronize charakteristische Teil der Spezifikation, der die Weitergabe von Marken beschreibt, beginnt mit Zeile 10. Ähnlich wie bei Join beschreibt der Teil von Zeile 10 bis 36, wie zunächst eine Marke über *Kanal_1* eingelesen wird. In den Zeilen 38 bis 64 steht der entsprechende Teil beschrieben, bei dem zunächst über *Kanal_2* eine Marke aufgenommen wird. Für jeden dieser beiden Fälle gibt es wieder eine Reihe von Möglichkeiten, wie sich der Baustein Synchronize verhalten kann:

1. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Sobald beide Nachfolger aufnahmebereit sind, werden die beiden Marken an die entsprechenden Nachfolger ausgegeben.

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_SYNCHRONIZE} = \lambda\ (reset, (data\_1, data\_valid\_1, ready\_1), (data\_2, data\_valid\_2, ready\_2), & (1)\\
\quad (data\_3, data\_valid\_3, ready\_3), (data\_4, data\_valid\_4, ready\_4)). & (2)\\
\quad ready\_1\ 0\ \wedge ready\_2\ 0 \quad \wedge & (3)\\
\quad \forall t. & (4)\\
\quad\quad reset\ t\ \Rightarrow & (5)\\
\quad\quad\quad (ready\_1\ (t+1)\ \wedge\ ready\_2\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge\ \neg(data\_valid\_3\ t)) \quad \wedge & (6)\\
\quad\quad (ready\_1\ t\ \wedge\ ready\_2\ t\ \wedge\ \neg(data\_valid\_1\ t)\ \wedge\ \neg(data\_valid\_2\ t))\ \Rightarrow & (7)\\
\quad\quad\quad (ready\_1\ (t+1)\ \wedge\ ready\_2\ (t+1)\ \wedge\ \neg(data\_valid\_3\ t)\ \wedge\ \neg(data\_valid\_4\ t)\ \wedge & (8)\\
\quad\quad\quad\ (data\_3\ t\ =\ data\_3\ (t-1))\ \wedge\ (data\_4\ t\ =\ data\_4\ (t-1))) \quad \wedge & (9)\\
\quad\quad (ready\_1\ t\ \wedge\ ready\_2\ t\ \wedge\ data\_valid\_1\ t)\ \Rightarrow & (10)\\
\quad\quad\quad \forall m.\ (\forall p.\ p \leq m\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (11)\\
\quad\quad\quad\quad (data\_valid\_2\ (t+m)\ \wedge\ (\forall p.\ p < m\ \Rightarrow\ \neg(data\_valid\_2\ (t+p))))\ \Rightarrow & (12)\\
\quad\quad\quad\quad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+m+p))))\ \Rightarrow & (13)\\
\quad\quad\quad\quad\quad\quad (ready\_3\ (t+s)\ \wedge\ ready\_4\ (t+s)\ \wedge & (14)\\
\quad\quad\quad\quad\quad\quad\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_3\ (t+p)\ \wedge\ ready\_4\ (t+p))))\ \Rightarrow & (15)\\
\quad\quad\quad\quad\quad\quad\quad ready\_1\ (t+m+s+1)\ \wedge & (16)\\
\quad\quad\quad\quad\quad\quad\quad ready\_2\ (t+m+s+1)\ \wedge & (17)\\
\quad\quad\quad\quad\quad\quad\quad data\_valid\_3\ (t+m+s)\ \wedge & (18)\\
\quad\quad\quad\quad\quad\quad\quad data\_valid\_4\ (t+m+s)\ \wedge & (19)\\
\quad\quad\quad\quad\quad\quad\quad (data\_3\ (t+m+s) = data\_1\ t)\ \wedge & (20)\\
\quad\quad\quad\quad\quad\quad\quad (data\_4\ (t+m+s) = data\_2\ (t+m)) & (21)\\
\quad\quad\quad\quad\quad\quad \wedge & (22)\\
\quad\quad\quad\quad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_3\ (t+m+p)\ \wedge\ ready\_4\ (t+m+p)))\ \Rightarrow & (23)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+m+s+1))\ \wedge & (24)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_2\ (t+m+s+1))\ \wedge & (25)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+m+s))\ \wedge & (26)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_4\ (t+m+s))\ \wedge & (27)\\
\quad\quad\quad\quad\quad\quad\quad (data\_3\ (t+m+s) = data\_1\ t)\ \wedge & (28)\\
\quad\quad\quad\quad\quad\quad\quad (data\_4\ (t+m+s) = data\_2\ (t+m)) & (29)\\
\quad\quad\quad\quad\quad \wedge & (30)\\
\quad\quad\quad\quad\quad (\forall p.\ p \leq m\ \Rightarrow\ \neg(data\_valid\_2\ (t+p)))\ \Rightarrow & (31)\\
\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+m+1))\ \wedge & (32)\\
\quad\quad\quad\quad\quad\quad ready\_2\ (t+m+1)\ \wedge & (33)\\
\quad\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+m))\ \wedge & (34)\\
\quad\quad\quad\quad\quad\quad \neg(data\_valid\_4\ (t+m))\ \wedge & (35)\\
\quad\quad\quad\quad\quad\quad (data\_3\ (t+m) = data\_1\ t) & (36)\\
\quad\quad \wedge & (37)\\
\quad\quad (ready\_1\ t\ \wedge\ ready\_2\ t\ \wedge\ data\_valid\_2\ t)\ \Rightarrow & (38)\\
\quad\quad\quad \forall m.\ (\forall p.\ p \leq m\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (39)\\
\quad\quad\quad\quad (data\_valid\_1\ (t+m)\ \wedge\ (\forall p.\ p < m\ \Rightarrow\ \neg(data\_valid\_1\ (t+p))))\ \Rightarrow & (40)\\
\quad\quad\quad\quad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+m+p))))\ \Rightarrow & (41)\\
\quad\quad\quad\quad\quad\quad (ready\_3\ (t+s)\ \wedge\ ready\_4\ (t+s)\ \wedge & (42)\\
\quad\quad\quad\quad\quad\quad\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_3\ (t+p)\ \wedge\ ready\_4\ (t+p))))\ \Rightarrow & (43)\\
\quad\quad\quad\quad\quad\quad\quad ready\_1\ (t+m+s+1)\ \wedge & (44)\\
\quad\quad\quad\quad\quad\quad\quad ready\_2\ (t+m+s+1)\ \wedge & (45)\\
\quad\quad\quad\quad\quad\quad\quad data\_valid\_3\ (t+m+s)\ \wedge & (46)\\
\quad\quad\quad\quad\quad\quad\quad data\_valid\_4\ (t+m+s)\ \wedge & (47)\\
\quad\quad\quad\quad\quad\quad\quad (data\_3\ (t+m+s) = data\_1\ (t+m))\ \wedge & (48)\\
\quad\quad\quad\quad\quad\quad\quad (data\_4\ (t+m+s) = data\_2\ t) & (49)\\
\quad\quad\quad\quad\quad\quad \wedge & (50)\\
\quad\quad\quad\quad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_3\ (t+m+p)\ \wedge\ ready\_4\ (t+m+p)))\ \Rightarrow & (51)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_1\ (t+m+s+1))\ \wedge & (52)\\
\quad\quad\quad\quad\quad\quad\quad \neg(ready\_2\ (t+m+s+1))\ \wedge & (53)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+m+s))\ \wedge & (54)\\
\quad\quad\quad\quad\quad\quad\quad \neg(data\_valid\_4\ (t+m+s))\ \wedge & (55)\\
\quad\quad\quad\quad\quad\quad\quad (data\_3\ (t+m+s) = data\_1\ (t+m))\ \wedge & (56)\\
\quad\quad\quad\quad\quad\quad\quad (data\_4\ (t+m+s) = data\_2\ t) & (57)\\
\quad\quad\quad\quad\quad \wedge & (58)\\
\quad\quad\quad\quad\quad (\forall p.\ p \leq m\ \Rightarrow\ \neg(data\_valid\_1\ (t+p)))\ \Rightarrow & (59)\\
\quad\quad\quad\quad\quad\quad ready\_1\ (t+m+1)\ \wedge & (60)\\
\quad\quad\quad\quad\quad\quad \neg(ready\_2\ (t+m+1))\ \wedge & (61)\\
\quad\quad\quad\quad\quad\quad \neg(data\_valid\_3\ (t+m))\ \wedge & (62)\\
\quad\quad\quad\quad\quad\quad \neg(data\_valid\_4\ (t+m))\ \wedge & (63)\\
\quad\quad\quad\quad\quad\quad (data\_4\ (t+m) = data\_2\ t) & (64)
\end{array}
$$

Abbildung 6.9: Formale Definition von K_IFC_SYNCHRONIZE

2. Der Vorgänger-Prozess, der mit Synchronize über den zweiten (ersten) Kanal verbunden ist, liefert zu keinem Zeitpunkt eine Marke. Eine mögliche Erklärung dafür ist, dass er als P-Prozess mit einer nichtterminierenden Berechnung beschäftigt ist. Die Konsequenz ist, dass Synchronize selbst nie eine Marke ausgeben kann, auch die nicht, die bereits eingelesen wurde.

3. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Einer der beiden Nachfolger (oder beide) wird aber zu keinem Zeitpunkt aufnahmebereit. Als Folge werden beide Marken nie ausgegeben, auch die Marke nicht, die an den evtl. aufnahmebereiten Nachfolger weitergegeben werden soll.

4. Bevor über den zweiten (ersten) Kanal eine Marke eingelesen werden kann, wird der Vorgang durch *reset* gestoppt.

5. Im zweiten (ersten) Kanal wird (gleichzeitig oder später) ebenfalls eine Marke eingelesen. Bevor die Marken weitergegeben werden können, wird ein Reset ausgelöst.

Die Definition der Konstanten K_IFC_SYNCHRONIZE und damit die oben beschriebenen Möglichkeiten lassen sich sehr einfach anhand der Definition von K_IFC_JOIN nachvollziehen. Der Unterschied besteht darin, dass drei Signale, die dem zweiten Ausgangskanal *Kanal_4* entsprechen, hinzugekommen sind. Dabei gilt, dass die Signale *data_valid_3* und *data_valid_4* immer den gleichen Wert haben, da die Marken gleichzeitig ausgegeben werden. Außerdem wird nicht nur auf die Aufnahmebereitschaft eines Kanals, sondern auf die zweier Kanäle geachtet. Dabei ist aber unerheblich, welcher von beiden Kanälen zuerst bereit ist. Erst wenn es einen Zeitpunkt $(t + s)$ gibt, an dem erstmals beide bereit sind (Zeilen 14-15, bzw. 42-43), können die Marken weitergegeben werden.

6.3.4 Der K-Prozess Fork

Anders als bei den bisher vorgestellten K-Prozessen Double, Join, Split und Synchronize, bei denen immer alle Ein- und Ausgangskanäle an der Markenübertragung beteiligt sind, ist dies bei Fork und dem im nächsten Abschnitt vorgestellten Prozess Choose nicht der Fall. Des Weiteren unterscheiden sich diese K-Prozesse von den vorherigen vier darin, dass die Teilsignale *data* der Kanäle nicht einen beliebigen, aber festen Typen α haben dürfen, sondern die Wahl des Datentyps teilweise eingeschränkt ist.

Der Prozess Fork hat einen Ein- und zwei Ausgangskanäle. Über das Signal *data_1* des Eingangskanals *Kanal_1* werden dabei Daten des Typs $(\alpha \times \mathsf{bool})$ übertragen, während beide Ausgangskanäle *Kanal_2* und *Kanal_3* Marken des Typs α übertragen. Die Wirkungsweise von Fork ist wie folgt: wird eine Marke eingelesen und hat ihre zweite Komponente den Wert F, wird die erste Komponente über den Ausgang *Kanal_2* weitergegeben. Hat die Marke als zweiten Anteil den Wert T, wird der erste Teil über *Kanal_3* ausgegeben.

Die genaue Definition findet sich in Abbildung 6.10. Die Konstante K_IFC_FORK beschreibt eine Relation zwischen den Teilsignalen des Eingangskanals (*Kanal_1*) und denen der Ausgangskanäle (*Kanal_2*, *Kanal_3*) sowie dem *reset*-Signal.

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_FORK} = \lambda\ (reset, (data\_1, data\_valid\_1, ready\_1), & (1)\\
\qquad (data\_2, data\_valid\_2, ready\_2), (data\_3, data\_valid\_3, ready\_3)). & (2)\\
\quad ready\_1\ 0\ \ \wedge & (3)\\
\quad \forall t. & (4)\\
\qquad reset\ t\ \Rightarrow & (5)\\
\qquad\quad (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge\ \neg(data\_valid\_3\ t))\ \ \wedge & (6)\\
\qquad (ready\_1\ t\ \wedge\ \neg(data\_valid\_1\ t))\ \Rightarrow & (7)\\
\qquad\quad (ready\_1\ (t+1)\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge\ \neg(data\_valid\_3\ t)\ \wedge & (8)\\
\qquad\quad\ (data\_2\ t\ =\ data\_2\ (t-1))\ \wedge\ (data\_3\ t\ =\ data\_3\ (t-1)))\ \ \wedge & (9)\\
\qquad (ready\_1\ t\ \wedge\ data\_valid\_1\ t\ \wedge\ \neg(\mathsf{SND}(data\_1\ t)))\ \Rightarrow & (10)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (11)\\
\qquad\quad\ (ready\_2\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_2\ (t+p))))\ \Rightarrow & (12)\\
\qquad\quad\quad ready\_1\ (t+s+1)\ \wedge & (13)\\
\qquad\quad\quad data\_valid\_2\ (t+s)\ \wedge & (14)\\
\qquad\quad\quad \neg(data\_valid\_3\ (t+s))\ \wedge & (15)\\
\qquad\quad\quad (data\_2\ (t+s) = \mathsf{FST}(data\_1\ t))\ \wedge & (16)\\
\qquad\quad\quad (data\_3\ (t+s) = \mathsf{FST}(data\_1\ t)) & (17)\\
\qquad\quad\ \wedge & (18)\\
\qquad\quad\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_2\ (t+p)))\ \Rightarrow & (19)\\
\qquad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge & (20)\\
\qquad\quad\quad \neg(data\_valid\_2\ (t+s))\ \wedge & (21)\\
\qquad\quad\quad \neg(data\_valid\_3\ (t+s))\ \wedge & (22)\\
\qquad\quad\quad (data\_2\ (t+s) = \mathsf{FST}(data\_1\ t))\ \wedge & (23)\\
\qquad\quad\quad (data\_3\ (t+s) = \mathsf{FST}(data\_1\ t)) & (24)\\
\qquad \wedge & (25)\\
\qquad (ready\_1\ t\ \wedge\ data\_valid\_1\ t\ \wedge\ \mathsf{SND}(data\_1\ t))\ \Rightarrow & (26)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (27)\\
\qquad\quad\ (ready\_3\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_3\ (t+p))))\ \Rightarrow & (28)\\
\qquad\quad\quad ready\_1\ (t+s+1)\ \wedge & (29)\\
\qquad\quad\quad \neg(data\_valid\_2\ (t+s))\ \wedge & (30)\\
\qquad\quad\quad data\_valid\_3\ (t+s)\ \wedge & (31)\\
\qquad\quad\quad (data\_2\ (t+s) = \mathsf{FST}(data\_1\ t))\ \wedge & (32)\\
\qquad\quad\quad (data\_3\ (t+s) = \mathsf{FST}(data\_1\ t)) & (33)\\
\qquad\quad\ \wedge & (34)\\
\qquad\quad\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_3\ (t+p)))\ \Rightarrow & (35)\\
\qquad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge & (36)\\
\qquad\quad\quad \neg(data\_valid\_2\ (t+s))\ \wedge & (37)\\
\qquad\quad\quad \neg(data\_valid\_3\ (t+s))\ \wedge & (38)\\
\qquad\quad\quad (data\_2\ (t+s) = \mathsf{FST}(data\_1\ t))\ \wedge & (39)\\
\qquad\quad\quad (data\_3\ (t+s) = \mathsf{FST}(data\_1\ t)) & (40)
\end{array}
$$

Abbildung 6.10: Formale Definition von K_IFC_FORK

Der für den Prozess Fork charakteristische Teil der Spezifikation, der die Weitergabe von Marken beschreibt, beginnt mit Zeile 10. In dem Teil von Zeile 10 bis Zeile 24 wird beschrieben, wie eine Marke eingelesen wird, deren zweite Komponente den

Wert F hat. In den Zeilen 26 bis 40 folgt die Beschreibung für eine Marke mit T als zweiter Komponente. Es gibt auch hier wieder eine Reihe von möglichen Abläufen bei der Markenübertragung:

1. Nachdem die Marke (x, F) bzw. (x, T) eingelesen wurde, ist der erste bzw. zweite Nachfolger nach einer gewissen Zeit aufnahmebereit und der erste Teil x der Marke wird über den entsprechenden Kanal ausgegeben. Die Aufnahmebereitschaft des zweiten bzw. ersten Nachfolgers spielt dabei keine Rolle.

2. Nachdem die Marke (x, F) bzw. (x, T) eingelesen wurde, ist der erste bzw. zweite Nachfolger zu keinem Zeitpunkt aufnahmebereit, und es wird keine Marke ausgegeben.

3. Nachdem die Marke (x, F) bzw. (x, T) eingelesen wurde, wird ein Reset ausgelöst, bevor die Marke x weitergegeben werden kann.

6.3.5 Der K-Prozess Choose

Ebenso wie bei Fork sind bei dem K-Prozess Choose nicht immer alle Kanäle bei der Übertragung einer Marke beteiligt. Außerdem existiert auch hier eine gewisse Einschränkung, was den Datentyp der Marken anbelangt. Choose hat drei Eingangskanäle (*Kanal_1*, *Kanal_2*, *Kanal_3*) und einen Ausgangskanal (*Kanal_4*). Während über *Kanal_1*, *Kanal_2* und *Kanal_4* Marken eines beliebigen, aber festen Datentyps α übertragen werden, kann *Kanal_3* nur boolesche Marken aufnehmen. Eine weitere Besonderheit des Prozesses Choose stellt die Tatsache dar, dass im aufnahmebereiten Zustand des Prozesses nicht alle Eingangskanäle gleichzeitig aufnahmebereit sind. Während *Kanal_3* in diesem Fall immer Marken aufnehmen kann, gilt dies immer nur für *Kanal_1* oder *Kanal_2*, aber nicht für beide gleichzeitig. Tabelle 6.1 zeigt in einer knappen Übersicht das Verhalten des Prozesses Choose.

Die exakte Definition des Prozesses Choose findet sich in Abbildung 6.11. Die Konstante K_IFC_CHOOSE beschreibt eine Relation zwischen den Teilsignalen der Eingangskanäle und denen des Ausgangskanals sowie dem *reset*-Signal. Im Folgenden soll diese Beschreibung näher erläutert werden:

Wie bereits erwähnt, ist im aufnahmebereiten Zustand des Prozesses entweder *Kanal_1* oder *Kanal_2* bereit, eine Marke aufzunehmen. Zu Beginn (Zeitpunkt 0) ist immer *Kanal_1* aufnahmebereit, wie es in Zeile 4 beschrieben ist. Bei Auslösen eines Resets gelangt der Prozess wieder in diesen Initialzustand, d.h. *Kanal_1* und *Kanal_3* sind aufnahmebereit, *Kanal_2* ist es nicht (Zeilen 6,7). Ist *Kanal_1* aufnahmebereit und kommt keine Marke über diesen Kanal in den Prozess und kommt entweder keine Marke oder eine Marke vom Wert T über *Kanal_3* (Zeilen 8,9), dann bleibt *Kanal_1* aufnahmebereit, es wird keine Marke über *Kanal_4* ausgegeben und *data_4* hält seinen letzten Wert (Zeilen 12, 13). Dies entspricht der Darstellung in den Reihen 5 und 6 in Tabelle 6.1. Dasselbe Ergebnis erhält man, wenn *Kanal_2* zwar aufnahmebereit ist, aber keine Marke aufnimmt und die Marke F über *Kanal_3* eingelesen wird (Zeilen 10,11, bzw. Reihe 10).

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_CHOOSE} = \lambda\ (reset, (data\_1, data\_valid\_1, ready\_1), (data\_2, data\_valid\_2, ready\_2), & (1)\\
\qquad (data\_3, data\_valid\_3, ready\_3), & (2)\\
\qquad (data\_4, data\_valid\_4, ready\_4)). & (3)\\
\quad ready\_1\ 0\ \wedge\ \neg(ready\_2\ 0)\ \wedge\ ready\_3\ 0\quad \wedge & (4)\\
\quad \forall t. & (5)\\
\qquad reset\ t\ \Rightarrow & (6)\\
\qquad\quad (ready\_1\ (t+1)\ \wedge\ \neg(ready\_2\ (t+1))\ \wedge\ ready\_3\ (t+1)\ \wedge\ \neg(data\_valid\_4\ t))\quad \wedge & (7)\\
\qquad ((ready\_1\ t\ \wedge\ \neg(ready\_2\ t)\ \wedge\ ready\_3\ t\ \wedge\ \neg(data\_valid\_1\ t)\ \wedge & (8)\\
\qquad\quad (\neg(data\_valid\_3\ t)\ \vee\ data\_3\ t))\ \vee & (9)\\
\qquad\quad (\neg(ready\_1\ t)\ \wedge\ ready\_2\ t\ \wedge\ ready\_3\ t\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge & (10)\\
\qquad\quad data\_valid\_3\ t\ \wedge\ \neg(data\_3\ t)))\ \Rightarrow & (11)\\
\qquad\quad (ready\_1\ (t+1)\ \wedge\ \neg(ready\_2\ (t+1))\ \wedge\ ready\_3\ (t+1)\ \wedge\ \neg(data\_valid\_4\ t)\ \wedge & (12)\\
\qquad\quad (data\_4\ t = data\_4\ (t-1)))\quad \wedge & (13)\\
\qquad ((ready\_1\ t\ \wedge\ \neg(ready\_2\ t)\ \wedge\ ready\_3\ t\ \wedge\ \neg(data\_valid\_1\ t)\ \wedge & (14)\\
\qquad\quad data\_valid\_3\ t\ \wedge\ \neg(data\_3\ t)\ \wedge\ \neg(reset\ t))\ \vee & (15)\\
\qquad\quad (\neg(ready\_1\ t)\ \wedge\ ready\_2\ t\ \wedge\ ready\_3\ t\ \wedge\ \neg(data\_valid\_2\ t)\ \wedge & (16)\\
\qquad\quad (\neg(data\_valid\_3\ t)\ \vee\ data\_3\ t)\ \wedge\ \neg(reset\ t)))\ \Rightarrow & (17)\\
\qquad\quad (\neg(ready\_1\ (t+1))\ \wedge\ ready\_2\ (t+1)\ \wedge\ ready\_3\ (t+1)\ \wedge\ \neg(data\_valid\_4\ t)\ \wedge & (18)\\
\qquad\quad (data\_4\ t = data\_4\ (t-1)))\quad \wedge & (19)\\
\qquad (ready\_1\ t\ \wedge\ \neg(ready\_2\ t)\ \wedge\ ready\_3\ t\ \wedge\ data\_valid\_1\ t\ \wedge & (20)\\
\qquad\quad (\neg(data\_valid\_3\ t)\ \vee\ \neg(data\_3\ t)))\ \Rightarrow & (21)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (22)\\
\qquad\quad\quad (ready\_4\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_4\ (t+p))))\ \Rightarrow & (23)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ ready\_2\ (t+s+1)\ \wedge\ ready\_3\ (t+s+1)\ \wedge & (24)\\
\qquad\quad\quad\quad data\_valid\_4\ (t+s)\ \wedge\ (data\_4\ (t+s) = data\_1\ t) & (25)\\
\qquad\quad\quad \wedge & (26)\\
\qquad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_4\ (t+p)))\ \Rightarrow & (27)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ \neg(ready\_3\ (t+s+1))\ \wedge & (28)\\
\qquad\quad\quad\quad \neg(data\_valid\_4\ (t+s))\ \wedge\ (data\_4\ (t+s) = data\_1\ t) & (29)\\
\qquad \wedge & (30)\\
\qquad (\neg(ready\_1\ t)\ \wedge\ ready\_2\ t\ \wedge\ ready\_3\ t\ \wedge\ data\_valid\_2\ t\ \wedge & (31)\\
\qquad\quad (\neg(data\_valid\_3\ t)\ \vee\ data\_3\ t))\ \Rightarrow & (32)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (33)\\
\qquad\quad\quad (ready\_4\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_4\ (t+p))))\ \Rightarrow & (34)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ ready\_2\ (t+s+1)\ \wedge\ ready\_3\ (t+s+1)\ \wedge & (35)\\
\qquad\quad\quad\quad data\_valid\_4\ (t+s)\ \wedge\ (data\_4\ (t+s) = data\_2\ t) & (36)\\
\qquad\quad\quad \wedge & (37)\\
\qquad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_4\ (t+p)))\ \Rightarrow & (38)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ \neg(ready\_3\ (t+s+1))\ \wedge & (39)\\
\qquad\quad\quad\quad \neg(data\_valid\_4\ (t+s))\ \wedge\ (data\_4\ (t+s) = data\_2\ t) & (40)\\
\qquad \wedge & (41)\\
\qquad (ready\_1\ t\ \wedge\ \neg(ready\_2\ t)\ \wedge\ ready\_3\ t\ \wedge\ data\_valid\_1\ t\ \wedge & (42)\\
\qquad\quad data\_valid\_3\ t\ \wedge\ data\_3\ t)\ \Rightarrow & (43)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (44)\\
\qquad\quad\quad (ready\_4\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_4\ (t+p))))\ \Rightarrow & (45)\\
\qquad\quad\quad\quad ready\_1\ (t+s+1)\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ ready\_3\ (t+s+1)\ \wedge & (46)\\
\qquad\quad\quad\quad data\_valid\_4\ (t+s)\ \wedge\ (data\_4\ (t+s) = data\_1\ t) & (47)\\
\qquad\quad\quad \wedge & (48)\\
\qquad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_4\ (t+p)))\ \Rightarrow & (49)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ \neg(ready\_3\ (t+s+1))\ \wedge & (50)\\
\qquad\quad\quad\quad \neg(data\_valid\_4\ (t+s))\ \wedge\ (data\_4\ (t+s) = data\_1\ t) & (51)\\
\qquad \wedge & (52)\\
\qquad (\neg(ready\_1\ t)\ \wedge\ ready\_2\ t\ \wedge\ ready\_3\ t\ \wedge\ data\_valid\_2\ t\ \wedge & (53)\\
\qquad\quad data\_valid\_3\ t\ \wedge\ \neg(data\_3\ t))\ \Rightarrow & (54)\\
\qquad\quad \forall s.\ (\forall p.\ p \leq s\ \Rightarrow\ \neg(reset\ (t+p))))\ \Rightarrow & (55)\\
\qquad\quad\quad (ready\_4\ (t+s)\ \wedge\ (\forall p.\ p < s\ \Rightarrow\ \neg(ready\_4\ (t+p))))\ \Rightarrow & (56)\\
\qquad\quad\quad\quad ready\_1\ (t+s+1)\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ ready\_3\ (t+s+1)\ \wedge & (57)\\
\qquad\quad\quad\quad data\_valid\_4\ (t+s)\ \wedge\ (data\_4\ (t+s) = data\_2\ t) & (58)\\
\qquad\quad\quad \wedge & (59)\\
\qquad\quad\quad (\forall p.\ p \leq s\ \Rightarrow\ \neg(ready\_4\ (t+p)))\ \Rightarrow & (60)\\
\qquad\quad\quad\quad \neg(ready\_1\ (t+s+1))\ \wedge\ \neg(ready\_2\ (t+s+1))\ \wedge\ \neg(ready\_3\ (t+s+1))\ \wedge & (61)\\
\qquad\quad\quad\quad \neg(data\_valid\_4\ (t+s))\ \wedge\ (data\_4\ (t+s) = data\_2\ t) & (62)
\end{array}
$$

Abbildung 6.11: Formale Definition von K_IFC_CHOOSE

Zustand	Marken-eingabe	Marken-ausgabe	Neuer Zustand
Kanal_1, *Kanal_3* bereit	*Kanal_1*: x	*Kanal_4*: x	*Kanal_2*, *Kanal_3* bereit
	Kanal_1: x, *Kanal_3*: F	*Kanal_4*: x	*Kanal_2*, *Kanal_3* bereit
	Kanal_1: x, *Kanal_3*: T	*Kanal_4*: x	*Kanal_1*, *Kanal_3* bereit
	Kanal_3: F	–	*Kanal_2*, *Kanal_3* bereit
	Kanal_3: T	–	*Kanal_1*, *Kanal_3* bereit
	–	–	*Kanal_1*, *Kanal_3* bereit
Kanal_2, *Kanal_3* bereit	*Kanal_2*: x	*Kanal_4*: x	*Kanal_2*, *Kanal_3* bereit
	Kanal_2: x, *Kanal_3*: F	*Kanal_4*: x	*Kanal_1*, *Kanal_3* bereit
	Kanal_2: x, *Kanal_3*: T	*Kanal_4*: x	*Kanal_2*, *Kanal_3* bereit
	Kanal_3: F	–	*Kanal_1*, *Kanal_3* bereit
	Kanal_3: T	–	*Kanal_2*, *Kanal_3* bereit
	–	–	*Kanal_2*, *Kanal_3* bereit

Tabelle 6.1: Kurze Erläuterung des Verhaltens von Choose

Ist *Kanal_1* aufnahmebereit, liest aber keine Marke ein, und wird über *Kanal_3* die Marke F eingelesen und wird kein Reset ausgelöst (Zeilen 14,15), dann wird *Kanal_2* aufnahmebereit, es wird keine Marke über *Kanal_4* ausgegeben, und *data_4* hält seinen letzten Wert (Zeilen 18, 19, bzw. Reihe 4). Dasselbe Ergebnis erhält man, wenn *Kanal_2* zwar aufnahmebereit ist, aber keine Marke aufnimmt, und entweder keine Marke oder T über *Kanal_3* eingelesen wird sowie kein Reset ausgelöst wird (Zeilen 16,17, bzw. Reihen 11,12).

Bis jetzt wurde jeweils über *Kanal_1* bzw. *Kanal_2* keine Marke eingelesen. Wird dagegen über *Kanal_1* eine Marke eingelesen und wird über *Kanal_3* keine Marke oder F eingelesen (Zeilen 20,21) geschieht folgendes: Solange der Nachfolger nicht aufnahmebereit ist, bleiben alle Eingangskanäle nicht aufnahmebereit; es wird keine Marke über *Kanal_4* ausgegeben und das Signal *data_4* hält die über *Kanal_1* eingelesene Marke (Zeilen 27-29). Gibt es einen Zeitpunkt $(t+s)$, an dem der Nachfolger aufnahmebereit wird, wird die Marke über *Kanal_4* ausgegeben, und einen Takt später sind *Kanal_2* und *Kanal_3* wieder aufnahmebereit. Dies funktioniert aber nur unter der Bedingung, dass bis zum Zeitpunkt $(t+s)$ kein Reset ausgelöst wird (Zeile 22). Dieses Verhalten entspricht den Reihen 1 und 2 in Tabelle 6.1. In den Zeilen 31-40, 42-51 bzw. 53-62 stehen die Teile beschrieben, die den Reihen 7 und 9, 3 bzw. 8 entsprechen.

6.3.6 Der K-Prozess Counter

Alle bisher vorgestellten K-Prozesse sind so aufgebaut, dass eine Konstante mit einer S-Schnittstelle parametrisiert wird. Laut der BNF für S-Strukturen (6.1) auf

Seite 83 können der Konstanten aber zusätzlich auch ein oder mehrere Variablen als Parameter übergeben werden. Der K-Prozess Counter ist der einzige elementare K-Prozess, bei dem dies der Fall ist. Ihm wird zusätzlich eine Variable n vom Typ num übergeben. Counter verfügt über einen Ein- und einen Ausgangskanal. Die Marke x, die über *Kanal_1* eingelesen wird, wird intern umgewandelt in eine Marke (x, i), wobei $i \leq n$ gilt. Die nächste Marke y, die zu einem späteren Zeitpunkt eingelesen wird, wird versehen mit $i+1$, bzw. mit 0, falls $i = n$. Der Prozess merkt sich demnach bis zum Eintreffen einer neuen Marke, mit welchem Wert die vorherige Marke versehen wurde. Über ein solches Gedächtnis verfügen DFG-Prozesse nicht. Aus diesem Grund kann Counter nicht durch einen DFG-Prozess realisiert werden, was man zunächst vermuten könnte. Die formale Definition ist dargestellt in Abbildung 6.12. Die Konstante K_IFC_COUNTER beschreibt eine Relation zwischen der

$$
\begin{array}{ll}
\vdash \mathsf{K\_IFC\_COUNTER} = \lambda n.\lambda\ (\mathit{reset}, (\mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}), & (1)\\
\qquad\qquad (\mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2})). & (2)\\
\quad \mathit{ready\_1}\ 0 \ \wedge & (3)\\
\quad (\neg(\mathit{data\_valid\_1}\ 0) \Rightarrow (\mathsf{SND}(\mathit{data\_2}\ t) = n)) \ \wedge & (4)\\
\quad \forall t. & (5)\\
\quad\quad \mathit{reset}\ t \Rightarrow & (6)\\
\quad\quad\quad (\mathit{ready\_1}\ (t+1) \wedge \neg(\mathit{data\_valid\_2}\ t) \wedge (\mathsf{SND}(\mathit{data\_2}\ 0) = n)) \ \wedge & (7)\\
\quad\quad (\mathit{ready\_1}\ t \wedge \neg(\mathit{data\_valid\_1}\ t)) \Rightarrow & (8)\\
\quad\quad\quad (\mathit{ready\_1}\ (t+1) \wedge \neg(\mathit{data\_valid\_2}\ t) \wedge (\mathit{data\_2}\ t = \mathit{data\_2}\ (t-1))) \ \wedge & (9)\\
\quad\quad (\mathit{ready\_1}\ t \wedge \mathit{data\_valid\_1}\ t \Rightarrow & (10)\\
\quad\quad\quad \forall s.\ (\forall p.\ p \leq s \Rightarrow \neg(\mathit{reset}\ (t+p)))) \Rightarrow & (11)\\
\quad\quad\quad\quad (\mathit{ready\_2}\ (t+s) \wedge (\forall p.\ p < s \Rightarrow \neg(\mathit{ready\_2}\ (t+p)))) \Rightarrow & (12)\\
\quad\quad\quad\quad\quad \mathit{ready\_1}\ (t+s+1) \wedge & (13)\\
\quad\quad\quad\quad\quad \mathit{data\_valid\_2}\ (t+s) \wedge & (14)\\
\quad\quad\quad\quad\quad (\mathit{data\_2}\ (t+s) = & (15)\\
\quad\quad\quad\quad\quad\quad \mathsf{let}\ z = \mathsf{SND}(\mathit{data\_2}\ (t-1))\ \mathsf{in} & (16)\\
\quad\quad\quad\quad\quad\quad (\mathit{data\_1}\ t, \mathsf{MUX}(t = 0 \vee z = n, 0, z+1))) \ \wedge & (17)\\
\quad\quad\quad\quad (\forall p.\ p \leq s \Rightarrow \neg(\mathit{ready\_2}\ (t+p))) \Rightarrow & (18)\\
\quad\quad\quad\quad\quad \neg(\mathit{ready\_1}\ (t+s+1)) \wedge & (19)\\
\quad\quad\quad\quad\quad \neg(\mathit{data\_valid\_2}\ (t+s)) \wedge & (20)\\
\quad\quad\quad\quad\quad (\mathit{data\_2}\ (t+s) = & (21)\\
\quad\quad\quad\quad\quad\quad \mathsf{let}\ z = \mathsf{SND}(\mathit{data\_2}\ (t-1))\ \mathsf{in} & (22)\\
\quad\quad\quad\quad\quad\quad (\mathit{data\_1}\ t, \mathsf{MUX}(t = 0 \vee z = n, 0, z+1))) & (23)
\end{array}
$$

Abbildung 6.12: Formale Definition von K_IFC_COUNTER

Variablen n, den Teilsignalen des Eingangskanals und denen des Ausgangskanals sowie dem *reset*-Signal. Zu Beginn ist der Prozess aufnahmebereit (Zeile 3), und am Ausgangssignal *data_2* ist der Wert der zweiten Komponente gerade n, wenn nicht sofort eine Marke übergeben wird (Zeile 4). Wird ein Reset ausgelöst, bekommt der zweite Teil von *data_2* n zugewiesen und der Prozess geht auf jeden Fall in den aufnahmebereiten Zustand über (Zeilen 6,7). Wird im aufnahmebereiten Zustand keine Marke eingelesen, bleibt der Prozess aufnahmebereit und *data_2* hält seinen letzten Wert (Zeilen 8,9). Wird eine Marke eingelesen (Zeile 10), wird der Eingangskanal gesperrt, solange der Nachfolger nicht aufnahmebereit ist, und es wird keine

Marke ausgegeben. Außerdem wird die eingelesene Marke mit einem bestimmten Wert versehen: Ist der Zeitpunkt der Markeneingabe gerade 0 oder ist der Wert z der zweiten Komponente von *data_2* gerade n, wird die Marke mit dem Wert 0 versehen, anderenfalls mit dem Wert $z + 1$ (Zeilen 18-23). Wird der Nachfolger zu einem gewissen Zeitpunkt $(t + s)$ aufnahmebereit, wird die veränderte Marke ausgegeben und *Kanal_1* wird wieder bereit zur Aufnahme von Marken (Zeilen 12-17). Die Marke kann aber wieder nur dann ausgegeben werden, wenn bis zum Zeitpunkt $(t + s)$ kein Reset ausgelöst wird (Zeile 11).

6.3.7 Der K-Prozess Sink

Der Prozess Sink stellt einen ganz speziellen K-Prozess dar, der nur über einen Eingangskanal verfügt. Sinn und Zweck von Sink ist es, eine Senke für Marken zu realisieren. Da eine Marke nur dann aus einem Prozess über einen Kanal ausgegeben werden kann, wenn das *ready*-Signal des betreffenden Kanals den Wert T hat, kann ein solcher Kanal, über den eine Marke ausgegeben werden soll, die nicht mehr gebraucht wird, nicht einfach im Leerlauf gelassen werden. Der Prozess Sink stellt sicher, dass zu jedem Zeitpunkt das Signal *ready* in diesem Kanal den Wert T hat (siehe die Definition in Abbildung 6.13). Eine Marke kann somit über einen Kanal, der den Ausgang eines beliebigen Prozesses mit Sink verbindet, den entsprechenden Prozess verlassen und in Sink eingelesen werden, wo sie anschließend verschwindet.

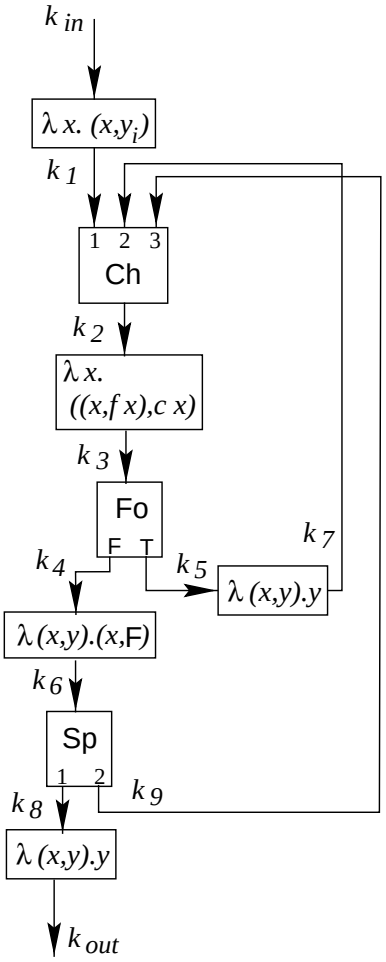
$$\vdash \mathsf{K\_IFC\_SINK} = \lambda(reset, (data, data\_valid, ready)). \quad (1)$$
$$\forall t.\ ready\ t \quad (2)$$

Abbildung 6.13: Formale Definition von K_IFC_SINK

6.4 Eine Beispielstruktur

Alle Arten von elementaren Prozessen, die auf der Systemebene Verwendung finden, wurden in den vorigen Abschnitten eingeführt. Um zu demonstrieren, wie mit Hilfe dieser Prozesse Systeme beschrieben werden können, ist in Abbildung 6.14(b) eine S-Struktur angegeben, die neben einigen K-Prozessen auch DFG-Prozesse enthält. Diese S-Struktur entspricht der in Abbildung 6.14(a) schematisch gezeigten Prozessstruktur. Das Verhalten der S-Struktur entspricht dem eines P-Prozesses, der eine while-Schleife realisiert, wie er in Abbildung 6.14(c) gezeigt ist. Damit soll aber nicht der Eindruck erweckt werden, S-Strukturen ließen sich prinzipiell durch einen P-Prozess ausdrücken.

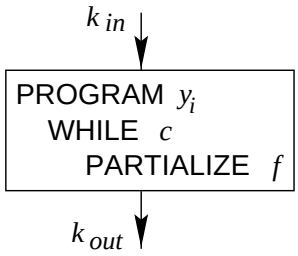
Es soll nun veranschaulicht werden, dass die S-Struktur tatsächlich gerade eine while-Schleife realisiert. Wird über den Kanal k_{in} eine Marke x übertragen, wird sie zunächst umgewandelt in ein Paar bestehend aus x und dem Wert y_i, der den Initialwert des Ausgangs darstellt. Die Umwandlung der Marke geschieht in einem



(a) Prozessstruktur

$\exists k_1\ k_2\ k_3\ k_4\ k_5\ k_6\ k_7\ k_8\ k_9.$
DFG_IFC_SYSTEM $(\lambda x.(x, y_i), n)$
$(\mathsf{reset}, k_{in}, k_1)$ $\wedge$
K_IFC_CHOOSE $(\mathsf{reset}, k_1, k_7, k_9, k_2)$ $\wedge$
DFG_IFC_SYSTEM $(\lambda x.((x, f\ x), c\ x), n)$
$(\mathsf{reset}, k_2, k_3)$ $\wedge$
K_IFC_FORK $(\mathsf{reset}, k_3, k_4, k_5)$ $\wedge$
DFG_IFC_SYSTEM $(\lambda(x, y).(x, \mathsf{F}), n)$
$(\mathsf{reset}, k_4, k_6)$ $\wedge$
DFG_IFC_SYSTEM $(\lambda(x, y).y, n)$
$(\mathsf{reset}, k_5, k_7)$ $\wedge$
K_IFC_SPLIT $(\mathsf{reset}, k_6, k_8, k_9)$ $\wedge$
DFG_IFC_SYSTEM $(\lambda(x, y).y, n)$
$(\mathsf{reset}, k_8, k_{out})$

(b) S-Struktur



(c) Äquivalenter P-Prozess

Abbildung 6.14: S-Struktur einer while-Schleife

ersten DFG-Prozess. Die neue Marke wird über k_1 an den Prozess Choose weitergegeben. Zu Beginn sind der erste und dritte Eingang von Choose aufnahmebereit (siehe Abschnitt 6.3.5). Da nur über den ersten Eingangskanal eine Marke eingegeben wird, wird nach Reihe 1 in Tabelle 6.1 die Marke über den Ausgangskanal k_2 ausgegeben und die Aufnahmebereitschaft der Eingangskanäle wechselt. k_1 ist ab sofort gesperrt, während k_7 aufnahmebereit wird. k_9 bleibt aufnahmebereit. Aus diesem Grunde kann keine neue Marke von außen in den Zyklus eintreten.

Die Marke, die über k_2 ausgegeben wurde, wird nun in einen DFG-Prozess weitergegeben, in dem sowohl die Schleifenbedingung als auch der Schleifenrumpf berechnet wird. Hat der Rumpf der while-Schleife nicht die Form (PARTIALIZE f), sondern ist er ein beliebiger P-Term, wird die Struktur etwas komplexer. Neben den Werten $(c\ x)$ und $(f\ x)$ wird auch noch der alte Wert x ausgegeben, der benötigt wird, wenn die Schleifenbedingung nicht erfüllt ist. Hat die Schleifenbedingung den Wert T ergeben, wird der erste Teil der Marke in k_3 mittels des Fork-Prozesses über

den Kanal k_5 weitergegeben. Dort läuft die neue Marke in einen weiteren DFG-Prozess, der nur den zweiten Teil der Marke weitergibt. Der erste Teil, der dem alten Wert vor dem Schleifenrumpf entspricht, fällt weg. Da der Kanal k_7 aufnahmebereit ist, kann die Marke in den Prozess Choose eingelesen werden und die Schleife wird nochmals ausgeführt. Hat dagegen die Schleifenbedingung den Wert F ergeben, läuft der erste Teil der Marke $((x, f\ x), c\ x)$ über k_4 in einen DFG-Prozess, der das Paar $(x, f\ x)$ umwandelt in (x, F). Da die Schleifenbedingung nicht erfüllt ist, muss auf den alten Wert zurückgegriffen werden, und der Wert $(f\ x)$ wird nicht benötigt. Die Marke (x, F) läuft daraufhin in einen Split-Prozess. Der erste Teil x wird dabei über k_8 ausgegeben und läuft in einen DFG-Prozess. In diesem DFG-Prozess wird der zweite Teil, der dem Ausgang entspricht, herausgefiltert. Der erste Teil, der dem Eingang entspricht, wird weggelassen. Die beiden DFG-Prozesse zu Beginn und am Ende der S-Struktur realisieren die Kontrollstruktur PROGRAM des äquivalenten P-Prozesses, die in (5.19) auf Seite 63 definiert wurde. Der zweite Teil F der Marke in k_6 wird über k_9 ausgegeben. Wird aber der Wert F über den dritten Eingangskanal in den Prozess Choose eingelesen, wechselt nach Reihe 10 in Tabelle 6.1 die Aufnahmebereitschaft der Eingangskanäle. Ab sofort sperrt der zweite Eingangskanal, während der erste und der dritte aufnahmebereit sind. Es kann damit über k_1 eine neue Marke in den Zyklus aufgenommen werden, und eine neue Berechnung kann beginnen.

Es ist anzumerken, dass Choose der einzige Prozess ist, der einen Zyklus sinnvoll schließen kann. Da in Gropius nur reaktive Systeme beschrieben werden, befindet sich zu Beginn keine Marke in dem Zyklus. Damit unterscheiden sich S-Strukturen prinzipiell von Petri-Netzen. Choose ist der einzige Prozess, der eine Marke übertragen kann, wenn nur ein Eingangskanal eine Marke einliest. Bei den Prozessen Join und Synchronize, die über zwei Eingangskanäle verfügen, ist dies nicht der Fall. Würden diese Prozesse einen Zyklus schließen, könnte keine Marke in den Zyklus aufgenommen werden. Dies ist darin begründet, dass sich zu Beginn keine Marke im Zyklus befindet und diese Prozesse nur feuern können, wenn sowohl eine Marke über den Kanal übertragen wird, der ausserhalb des Zyklus' liegt, als auch eine Marke über den zyklusinternen Kanal übertragen wird. Das kann aber nie der Fall sein. Rein syntaktisch ist es zwar möglich, einen Zyklus mit Join oder Synchronize zu beschreiben, es käme aber aus diesem Grunde zu einer Verklemmung.

6.5 Funktionale Semantik von S-Strukturen

So wie bei Einprozessbeschreibungen auf der algorithmischen Ebene eine klare Trennung zwischen funktionaler Beschreibung und zeitlicher Einordnung vorgenommen wird, existiert bis jetzt eine solche Trennung auf der Systemebene nur für die einzelnen Prozesse. P- bzw. DFG-Prozesse setzen sich aus P- bzw. DFG-Termen und je einem speziellen Schnittstellenverhaltensmuster zusammen. K-Prozesse existieren im Wesentlichen nur aus einer zeitlichen Beschreibung, das funktionale Verhalten ist bei ihnen trivial. Im Gegensatz zur Trennung bei den einzelnen Prozessen exi-

stiert eine solche Trennung aber nicht für eine S-Struktur aus mehreren Prozessen. S-Strukturen realisieren nur eine gemischt funktional/zeitliche Beschreibung. Durch diese Beschreibung wird einer Struktur mehrerer Prozesse ein ganz konkreter Markenfluss zugewiesen. Jeder Prozess auf einem Pfad kann dabei theoretisch eine Marke aufnehmen. Auf einem Pfad mit n S-Prozessen können also je nach Eingabe des Gesamtsystems zu einem Zeitpunkt n Marken im Umlauf sein. Eine mögliche Analogie zu dieser Situation stellt eine n-stufige Pipeline eines Prozessors dar. Auch hier können n Vorgänge gleichzeitig ausgeführt werden.

Eine genaue Festlegung des Markenflusses ist aber oft gar nicht wünschenswert. Bei der anfänglichen Beschreibung von Systemen steht oft vor allem die Funktionalität im Vordergrund. Es werden einige Prozesse beschrieben und zusammengesetzt, die eine gewisse Gesamtfunktion des Systems realisieren sollen. Das genaue zeitliche Verhalten des Markenflusses spielt dabei oft noch keine Rolle. Von dieser anfänglichen Prozessbeschreibung ausgehend können dann weitere Prozesse eingeführt werden, oder einzelne Prozesse können partitioniert werden, ohne die Funktionalität zu verändern (siehe auch Abschnitt 8.3). Auf der Basis der S-Strukturen würde sich durch diese Maßnahme aber das zeitliche Verhalten völlig verändern. Um bei der obigen Analogie zu bleiben, würde dabei eine n-stufige Pipeline in eine m-stufige Pipeline umgewandelt. Eine Äquivalenz zwischen der initialen und der umgewandelten Systembeschreibung auf der Ebene einer genauen zeitlichen Einordnung des Markenflusses lässt sich damit nicht nachweisen.

Es ist also wünschenswert, nicht nur gemischt funktional/zeitliche Beschreibungen von Systemen zur Verfügung zu haben, sondern auch Beschreibungen, die nur die funktionale Seite des Systems ausdrücken. Nur mit Hilfe einer solchen Beschreibung lässt sich der Übergang von einer Systembeschreibung zu einer anderen in einer formalen Weise darstellen.

Zur Beschreibung des rein funktionalen Aspektes werden die funktionale Semantik und die Relationen der funktionalen Äquivalenz sowie der funktionalen Implikation der S-Strukturen definiert. Diese Äquivalenz bzw. Implikation soll so eingeführt werden, dass aus der Verhaltensäquivalenz bzw. logischen Implikation zweier S-Strukturen deren funktionale Äquivalenz bzw. Implikation folgt. Die funktionale Semantik einer S-Struktur ist eine boolesche Funktion, die als Parameter solche Signale enthält, welche den Datentyp (α)partial haben. Hintergrund sind dafür die P-Terme, die unter Umständen nicht terminieren können und deren Wertebereich sich über (α)partial erstreckt. Der Wert Undefined wird dabei so interpretiert, als liege am Eingang des betrachteten S-Prozesses kein gültiges Signal an, d.h. es wird keine Marke in den Prozess eingelesen. Der Wert (Defined x) dagegen sagt aus, dass am Eingang eine Marke mit dem Wert x anliegt.

Zur Definition der funktionalen Semantik werden neue Hilfskonstrukte eingeführt, die die in (5.7) auf Seite 58 definierte Funktion PRIMREC_partial benutzen:

$$\begin{aligned} &\vdash \mathsf{APPLY1}\ A\ x\ =\ \mathsf{PRIMREC\_partial}\ x\ A\ \mathsf{Undefined} \\ &\vdash \mathsf{APPLY2}\ A\ x\ =\ \mathsf{PRIMREC\_partial}\ x\ A\ (\mathsf{Undefined}, \mathsf{Undefined}) \end{aligned} \tag{6.2}$$

Ist der Wert des Eingangs x Undefined, liefert die Funktion (APPLY1 A) ebenso den Wert Undefined zurück. Im Falle eines Eingangswertes (Defined r) ist das Ergebnis (A r). Die Funktion APPLY2 dient zur Beschreibung von Prozessen mit zwei Ausgängen. Während die Funktion APPLY1 den Ausgangsdatentyp (β)partial hat, liefert APPLY2 Werte des Typs $((\beta)\text{partial}, (\gamma)\text{partial})$ zurück. Mit Hilfe dieser Funktionen kann nun für jeden der auf der Systemebene betrachteten Prozesse P eine relationale Beschreibung des funktionalen Zusammenhangs als funktionale Semantik $\| P \|$ eingeführt werden:

$$
\begin{aligned}
&\| \text{P\_IFC\_SYSTEM } A \ \| \ (x, y) \equiv (y \ = \ \text{APPLY1 } A \ x) \\
&\| \text{DFG\_IFC\_SYSTEM } f \ \| \ (x, y) \ \equiv (y \ = \ \text{APPLY1 } (\lambda r.\text{Defined}(f \ r)) \ x) \\
&\| \text{K\_IFC\_DOUBLE} \ \| \ (x, y_1, y_2) \ \equiv \\
&\qquad (y_1, y_2) \ = \ \text{APPLY2 } (\lambda r.(\text{Defined } r, \text{Defined } r) \ x \\
&\| \text{K\_IFC\_JOIN} \ \| \ (x_1, x_2, y) \ \equiv \\
&\qquad y \ = \ \text{APPLY1 } (\lambda r.\text{APPLY1 } (\lambda s.\text{Defined}(r, s)) \ x_2) \ x_1 \\
&\| \text{K\_IFC\_SPLIT} \ \| \ (x, y_1, y_2) \ \equiv \\
&\qquad (y_1, y_2) \ = \ \text{APPLY2 } (\lambda(r, s).(\text{Defined } r, \text{Defined } s) \ x \\
&\| \text{K\_IFC\_SYNCHRONIZE} \ \| \ (x_1, x_2, y_1, y_2) \ \equiv \\
&\qquad (y_1, y_2) \ = \ \text{APPLY2 } (\lambda r.\text{APPLY2 } (\lambda s.(\text{Defined } r, \text{Defined } s)) \ x_2) \ x_1 \\
&\| \text{K\_IFC\_FORK} \ \| \ (x, y_1, y_2) \ \equiv \\
&\qquad (y_1, y_2) \ = \ \text{APPLY2 } (\lambda(r, s).\text{MUX}(s, \ (\text{Undefined}, \text{Defined } r), \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{Defined } r, \text{Undefined}))) \ x \\
&\| \text{K\_IFC\_CHOOSE} \ \| \ (x_1, x_2, x_3, y) \ \equiv \\
&\qquad \exists r. \ (y \ = \ \text{MUX}(r, \text{APPLY1 Defined } x_1, \text{APPLY1 Defined } x_2)) \\
&\| \text{K\_IFC\_COUNTER } n \ \| \ (x, y) \ \equiv \\
&\qquad \exists m. \ m \leq n \ \wedge \ (y \ = \ \text{APPLY1 } (\lambda r. \ \text{Defined}(r, \text{MUX}(m < n, m + 1, 0))) \ x) \\
&\| \text{K\_IFC\_SINK} \ \| \ x \ \equiv \text{T}
\end{aligned}
$$

Die funktionale Semantik eines P-Prozesses mit zugrunde liegendem P-Term A besteht gerade in der Auswertung der Funktion A. Liegt allerdings der Wert Undefined und somit keine Marke am Eingang an, ist der Wert am Ausgang ebenfalls Undefined, es wird also auch keine Marke ausgegeben. Ähnlich verhält es sich mit der funktionalen Semantik eines DFG-Prozesses. Der zugrunde liegende DFG-Term f wird ausgewertet, wobei das Ergebnis zusätzlich in den Wertebereich des Typs (α)partial überführt wird. Die funktionale Semantik der K-Prozesse Double, Join, Split, Synchronize und Fork besteht im Wesentlichen aus einer Abfrage, ob am Eingang eine Marke anliegt. Falls ja, wird diese Marke gemäß der Definition des K-Prozesses an den Ausgang weitergegeben. Falls nein, ist der Wert an allen Ausgängen Undefined, um anzuzeigen, dass keine Marke ausgegeben wird.

Die Angabe einer funktionalen Semantik für die K-Prozesse Choose und Counter gestaltet sich schwieriger, da ihr funktionales Verhalten von der Vorgeschichte

abhängt. Die Übergangsfunktion ist in beiden Fällen abhängig vom Wert einer Variablen, die den inneren Zustand des Prozesses repräsentiert. Dieser Wert ist aber nicht bekannt; es ist lediglich garantiert, dass ein solcher Wert existiert. Beim Prozess Choose repräsentiert die Variable r den Zustand, ob der erste oder zweite Eingangskanal aufnahmebereit ist. Hat r den Wert T, kann über den ersten Kanal eine Marke eingelesen werden. Im anderen Falle ist nach der Definition von Choose der zweite Kanal aufnahmebereit (Multiplexerfunktion). Der dritte Eingang hat auf die Übergangsfunktion keinen Einfluss, sondern lediglich auf den zukünftigen Wert des inneren Zustands. Die Übergangsfunktion des Prozesses Counter ist abhängig von einer Variablen m, die angibt, mit welchem Wert die vorherige Marke versehen wurde. Der Wert dieser Variablen bewegt sich zwischen 0 und n, der genaue Wert ist aber unbekannt. Je nachdem welchen Wert m bei der Aufnahme einer Marke x hat, wird diese Marke in $(x, m+1)$ oder in $(x, 0)$ umgewandelt.

Einen Sonderfall stellt der Prozess Sink dar. Da dieser Prozess über keinen Ausgang verfügt, kann keine funktionale Ein-/Ausgangsbeziehung aufgestellt werden. Die funktionale Semantik wird einfach durch die Konstante T beschrieben, da dieser Prozess immer eine Marke aufnehmen kann. In einer relationalen Beschreibung einer S-Struktur auf Basis der funktionalen Semantik kann dieser Prozess einfach durch boolesche Umformung eliminiert werden, da die S-Prozesse in einer Konjunktion aufgelistet werden (siehe unten).

Die funktionale Semantik eines Prozesses $Sname(\bar{a})$ einer definierten S-Struktur $Sname$ mit der Definition $Sname(\bar{x}) = SStrukt$ wird ähnlich der Verhaltenssemantik mittels

$$\| \, Sname(\bar{a}) \, \| \equiv \; (\lambda \bar{x}. \;\; \| \, SStrukt \, \|) \; \bar{a}$$

eingeführt. Die funktionale Semantik einer S-Struktur $S = \exists \, \bar{b}. \; S_1 \wedge \ldots \wedge S_n$ wird festgelegt durch

$$\| \, \mathcal{S} \, \| \equiv \; \exists \, \bar{b}. \; \bigwedge_{i=1}^{n} \| \, S_i \, \|$$

Man nennt zwei S-Strukturen S_1 und S_2 *funktional äquivalent*, wenn sie die gleiche funktionale Semantik haben:

$$S_1 \approx S_2 \;\; \equiv \;\; (\, \| \, S_1 \, \| = \| \, S_2 \, \|)$$

Die *funktionale Implikation* $S_1 \Longrightarrow S_2$ bedeutet einfach die übliche logische Implikation für die funktionalen Semantiken der entsprechenden S-Strukturen:

$$S_1 \Longrightarrow S_2 \;\; \equiv \;\; (\, \| \, S_1 \, \| \Rightarrow \| \, S_2 \, \| \,)$$

Die funktionale Äquivalenz (funktionale Implikation) der S-Strukturen ist korrekt in dem Sinne, dass aus der Verhaltensäquivalenz (logischen Implikation) zweier S-Strukturen deren funktionale Äquivalenz (funktionale Implikation) folgt:

$$(S_1 \cong S_2) \Rightarrow (S_1 \approx S_2) \quad \text{bzw.} \quad (S_1 \Rightarrow S_2) \Rightarrow (S_1 \Longrightarrow S_2)$$

6.6 Realisierung anderer Kommunikationsschemata

Das in Abschnitt 6.2 vorgestellte Kommunikationsschema sieht vor, dass algorithmisch beschriebene Prozesse wie DFG- und P-Prozesse immer nur ein Datenpaket geliefert bekommen, darauf ihre Funktion anwenden und anschließend ein Datenpaket als Resultat an den Nachfolger weitergeben. Dieses Kommunikationsschema ist sehr restriktiv, da es z.B. nicht erlaubt, mehrere Datenpakete hintereinander einzulesen oder auszugeben.

Die in Abschnitt 6.3 vorgestellten K-Prozesse können aber verwendet werden, um genau dieses Problem zu lösen. Sie können zu komplexeren Gebilden zusammengefasst werden, um das Schnittstellenverhalten für die algorithmischen Prozesse zu verändern. Auf diese Weise können andere Kommunikationsschemata gewonnen werden.

Abbildung 6.15 zeigt dafür zwei Beispiele. In Abbildung 6.15(a) werden durch die gezeigte Struktur drei Marken, die nacheinander eingelesen werden, zu einer einzigen Marke zusammengefasst. Dieses kombinierte Datenpaket kann anschließend einem DFG- oder P-Prozess zugeführt werden, sodass er seine Funktion auf allen drei Marken gleichzeitig evaluieren kann. Abbildung 6.15(b) zeigt den umgekehrten Vorgang. Hier werden die Teile einer kombinierten Marke nacheinander ausgelesen. Die Strukturen in Abbildung 6.15 sind für jeweils drei Marken ausgelegt. Die Vorgänge lassen sich aber für beliebig viele Marken verallgemeinern. Für n Marken müssten dazu jeweils die grau unterlegten Teilstrukturen $(n-3)$-mal an den unterbrochenen Stellen der Gesamtstruktur wiederholt werden. Werden die unterbrochenen Stellen ohne Einfügen der Teilstrukturen geschlossen, erhält man jeweils gerade die erforderliche Struktur, um drei Marken umzuwandeln. Anmerkung zu Abbildung 6.15(a): Aus Darstellungsgründen war es notwendig, den DFG-Prozess g einzuführen. Es ist leicht zu erkennen, dass dieser Prozess nach Einfügen der für n Marken erforderlichen Teilstrukturen, bzw. nach Schließen der unterbrochenen Verbindung für drei Marken, mit dem vorangehenden DFG-Prozess f zu einem einzigen DFG-Prozess mit der Funktion $\lambda(x,h).(x,(h-1)=0)$ verschmolzen werden kann.

Es soll nur kurz die Wirkungsweise der in Abbildung 6.15(a) gezeigten Struktur erläutert werden. Die Wirkungsweise der in Abbildung 6.15(b) dargestellten Struktur lässt sich leicht anhand der in den vorigen Abschnitten erläuterten Wirkungsweisen der K-Prozesse nachvollziehen. Die erste Marke x_0 wird durch Counter mit dem Wert 0 versehen. Der anschließende DFG-Prozess prüft, ob die zweite Komponente der eingehenden Marke gleich 0 ist. Dies ist hier der Fall. Es wird die Marke $((x_0,0),\mathsf{T})$ ausgegeben. Der erste Teil $(x_0,0)$ dieser Marke wird über den oberen Ausgangskanal von Fork weitergegeben. Split trennt nun diese Marke auf, wobei 0 in die Senke läuft und verschwindet und x_0 über den ersten Eingangskanal in Join eingelesen wird. Der Prozess Join kann aber noch nicht feuern, da am zweiten Eingangskanal noch keine Marke eingetroffen ist. Die nächste Marke x_1 erhält von Counter den Wert 1. Der DFG-Prozess wandelt diese Marke um in

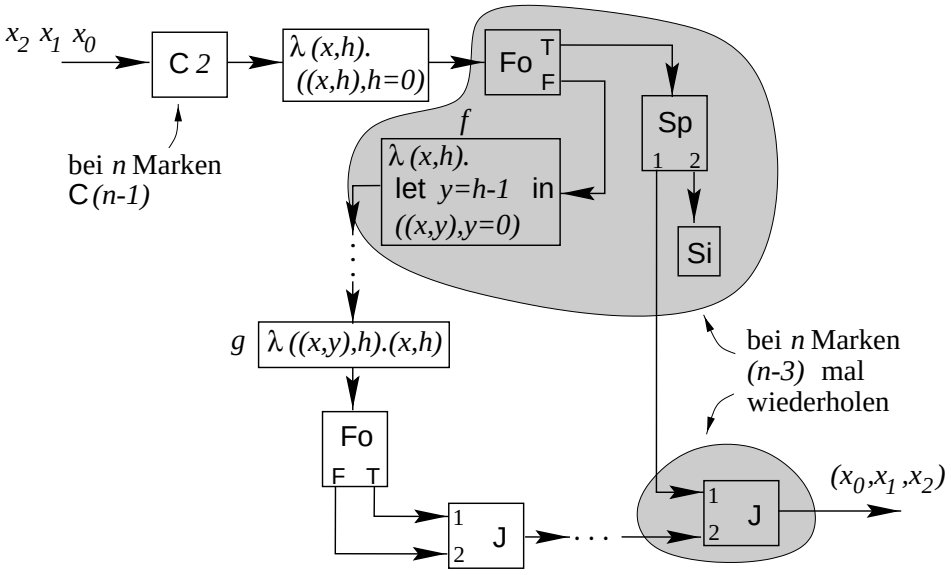
$((x_1, 1), \mathsf{F})$, weshalb Fork den ersten Teil über den unteren Ausgang weitergibt. Die Marke $(x_1, 1)$ läuft dann in einen weiteren DFG-Prozess. Wie bereits ausgeführt, wird dieser DFG-Prozess f mit dem DFG-Prozess g vereinigt. Der resultierende DFG-Prozess dekrementiert die zweite Komponente und prüft sie anschließend auf Gleichheit mit 0. Es entsteht die Marke (x_1, T), deren erster Teil mittels Fork über den ersten Eingang von Join eingelesen wird. Auch dieser Join-Prozess kann noch nicht feuern, da die zweite Eingangsmarke noch fehlt. Kommt nun die dritte Marke x_2, gelangt sie schließlich an den zweiten Eingang des Join-Prozesses, der schon die Marke x_1 aufgenommen hat. Nun kann dieser Prozess feuern und die Marke (x_1, x_2) erreicht den zweiten Join-Baustein, der bereits x_0 aufgenommen hat. Ist der nachfolgende Prozess, der in Abbildung 6.15(a) nicht gezeigt ist, aufnahmebereit, kann dieser Join-Prozess feuern und die Marke (x_0, x_1, x_2) ausgeben.

6.7 Beschreibungen mit funktionalem und zeitlichem Anteil

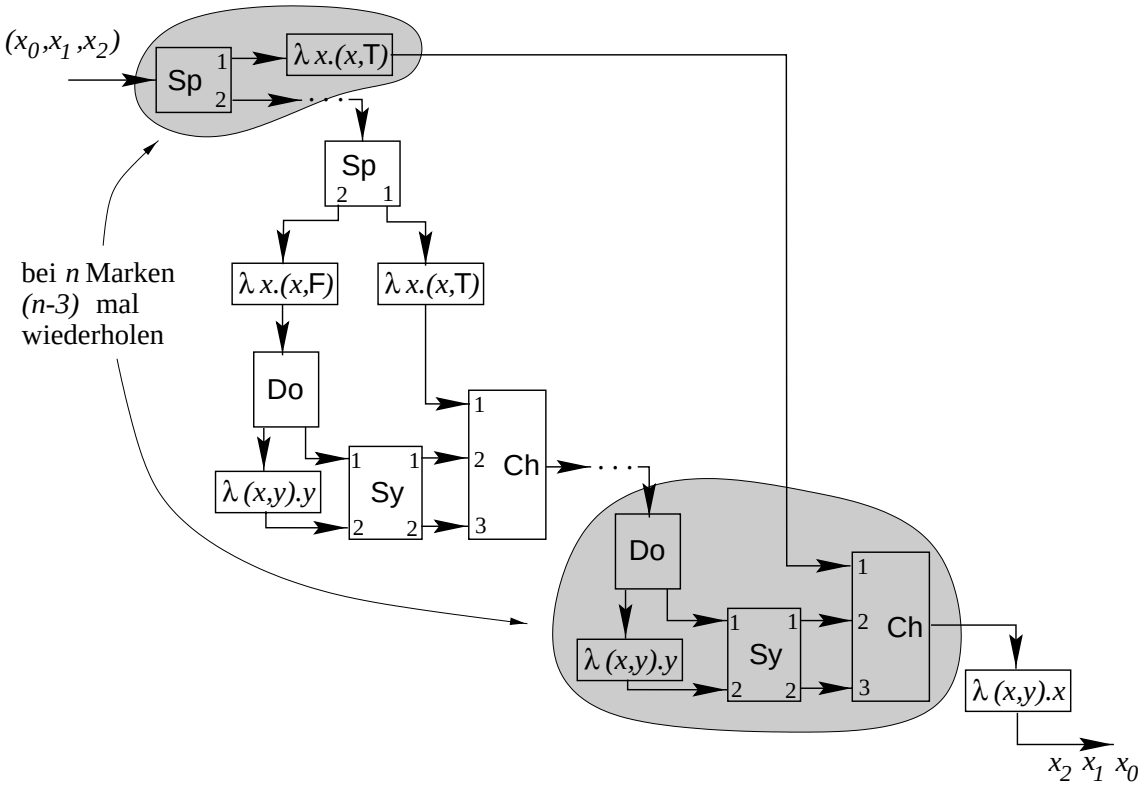
Wie bereits in Abschnitt 4.4 und zu Beginn von Kapitel 5 erwähnt, wird in Gropius eine strikte Trennung zwischen funktionalen und zeitlichen Aspekten einer algorithmischen Schaltungsbeschreibung unterschieden. Dies hat zur Folge, dass gewisse Beschreibungen, die über beide Anteile verfügen sollen, auf der algorithmischen Ebene nicht repräsentiert werden können, da dort nur Einprozessbeschreibungen berücksichtigt werden (siehe Kapitel 5).

Es ist aber dennoch möglich, solche Beschreibungen in Gropius auszudrücken. Dies geschieht dadurch, dass man die Beschreibung als Mehrprozessstruktur auf die Systemebene überträgt. Die ursprüngliche Beschreibung muss dazu in rein funktionale und rein zeitliche Anteile unterteilt werden. Die funktionalen Anteile können dann durch DFG- oder P-Prozesse beschrieben werden, während die zeitlichen Anteile durch K-Prozesse ausgedrückt werden. Abbildung 6.16 zeigt ein Beispiel. Abbildung 6.16(a) zeigt einen Teil einer VHDL-Beschreibung, die funktionale und zeitliche Aspekte miteinander kombiniert. Eine derartige Beschreibung ist in Gropius auf der algorithmischen Ebene nicht möglich. Abbildung 6.16(b) zeigt eine einfache S-Struktur auf der Systemebene, die das gewünschte Verhalten nachbildet. Sie realisiert die beiden funktionalen Anteile der VHDL-Beschreibung in zwei verschiedenen DFG-Prozessen. Zunächst wird die Funktion f des ersten DFG-Prozesses auf die Marke x angewandt. Man erhält die Marke y. Der Synchronize-Prozess sorgt nun dafür, dass der Markenfluss solange verzögert wird, bis die Marke *signal* an seinem ersten Eingang erscheint. Ist diese Marke bereits vor Eintreffen von y da, wird der Markenfluss nicht verzögert. Sind beide Marken eingetroffen, werden sie gleichzeitig weitergegeben, sobald der zweite DFG-Prozess aufnahmebereit ist. Die Marke *signal* läuft dabei in eine Senke – vorausgesetzt sie wird nicht mehr benötigt. Die Marke y durchläuft den zweiten DFG-Prozess und wird in die Marke z umgewandelt.

Eine zweite, etwas komplexere S-Struktur, die ebenfalls die VHDL-Beschreibung nachbildet, ist in Abbildung 6.16(c) dargestellt. Hier werden die zwei Funktionen



(a) Sammeln seriell eintreffender Marken



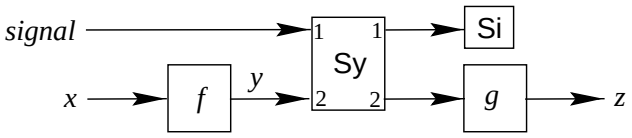
(b) Erzeugen seriell abgegebener Marken

Abbildung 6.15: Verändern des Markenflusses (Beispiele mit drei Marken)

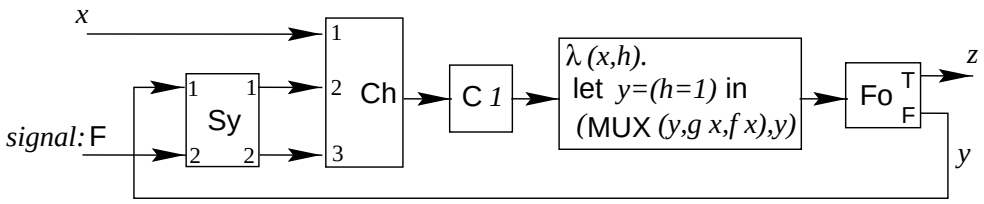
VHDL:

$y := f\ x;$
$wait\ until\ (signal\ =\ '1');$
$z := g\ y;$

(a) VHDL-Beschreibung



(b) S-Struktur mit zwei DFG-Prozessen



(c) S-Struktur mit einem DFG-Prozess

Abbildung 6.16: Beispiel einer Beschreibung mit funktionalem und zeitlichem Anteil

f und g in einem einzigen DFG-Prozess ausgeführt. Die Kombination der drei K-Prozesse Choose, Counter und Fork wirkt dabei wie eine for-Schleife. Zu Beginn ist der erste Eingangskanal von Choose aufnahmebereit, sodass die Marke x weitergegeben werden kann. Als Folge wird der erste Kanal gesperrt und der zweite ist aufnahmebereit. Die Marke x durchläuft dann den Prozess Counter, der sie mit dem Wert 0 versieht. In dem DFG-Prozess wird nun geprüft, ob der zweite Anteil der einkommenden Marke gleich 1 ist. Ist dies der Fall, wird die Funktion g auf den ersten Anteil der Marke angewandt, im anderen Fall die Funktion f. Das Ergebnis stellt den ersten Teil der resultierenden Marke dar, der Wert der Bedingung den Zweiten. Die Marke $(x, 0)$ wird also umgewandelt in die Marke $(f\ x, \mathsf{F})$. Der Prozess Fork sorgt nun dafür, dass die Marke $(f\ x)$ über den unteren Ausgangskanal ausgegeben und über den ersten Eingang in den Synchronize-Baustein aufgenommen wird. Über den zweiten Eingangskanal von Synchronize läuft eine Marke ein, wenn das Signal *signal* aktiviert worden ist. Diese Marke hat den Wert F. Der Markenfluss wird also an dieser Stelle solange verzögert, bis *signal* aktiv wird. Ist auch die zu *signal* gehörende Marke von Synchronize aufgenommen worden, werden die beiden Marken an die Eingänge zwei und drei von Choose weitergereicht. Dies hat zur Folge, dass die Marke $(f\ x)$ weitergegeben wird und der erste Eingangskanal von Choose wieder aufnahmebereit wird. $(f\ x)$ erhält in Counter den Wert 1, und daher wird in dem DFG-Prozess die Funktion g auf $(f\ x)$ angewandt. Anschließend wird die Marke $z = (g(f\ x))$ vom Fork-Prozess über seinen oberen Ausgangskanal als Ergebnis ausgegeben.

6.8 Vermeidung inkonsistenter Schaltungsbeschreibungen

Einen sehr wichtigen Gesichtspunkt bei der Formalen Synthese stellt, wie auch schon in Abschnitt 4.2 erwähnt, die Tatsache dar, dass Schaltungsbeschreibungen auf keinen Fall Inkonsistenzen enthalten dürfen. Eine inkonsistente Schaltungsbeschreibung ist nicht als konkrete Schaltung auf einem Chip realisierbar. Der formale Beweis, dass die Implementierung korrekt ist, wäre nur ein Trugschluss.

Auf den unteren Abstraktionsebenen wird rein syntaktisch gewährleistet, dass nur tatsächlich implementierbare Schaltungen beschrieben werden. Dies wird durch die funktionale Modellierung erreicht. Auf der Systemebene wird aber im Gegensatz dazu eine relationale Schaltungsmodellierung verwendet, wie sie in Abschnitt 6.1 vorgestellt wurde. Dadurch werden Inkonsistenzen nicht a priori ausgeschlossen; vielmehr müssen die Schaltungsbeschreibungen explizit überprüft werden. Es gibt zwei Gründe, die zu inkonsistenten und damit widersprüchlichen Beschreibungen führen können:

- Kurzschlüsse
- verzögerungsfreie Zyklen

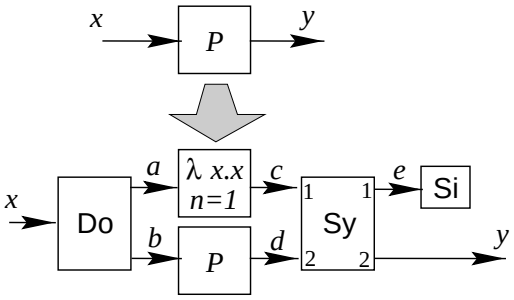
Kurzschlüsse können relativ leicht in S-Strukturen aufgespürt werden. Wie bereits in Abschnitt 6.1 erwähnt, darf jeder Kanal nur einen Prozessausgang mit jeweils einem Prozesseingang verbinden. Diese Forderung lässt sich lokal für jeden Prozess überprüfen.

Anders sieht es bei verzögerungsfreien Zyklen aus. Ein verzögerungsfreier Zyklus liegt dann vor, wenn es einen geschlossenen Pfad gibt, der nur solche Prozesse einschließt, die den Markenfluss normalerweise nicht verzögern oder von denen nicht bekannt ist, ob sie den Markenfluss verzögern. Die überwiegende Mehrzahl der S-Prozesse fällt in diese Kategorie. Alle K-Prozesse können sich verzögerungsfrei verhalten. Wenn der oder die Nachfolger aufnahmebereit sind, werden die Marken sofort weitergeleitet. Auch alle P-Prozesse sind potentiell verzögerungsfrei. Dies liegt darin begründet, dass über eine Terminierung der zugrunde liegenden P-Terme nichts bekannt ist, wie in Kapitel 5 ausgeführt wurde. Daher ist insbesondere auch in der Regel nicht bekannt, ob der P-Prozess in null Takten terminiert und die Marke ohne Verzögerung weitergibt. Die einzige Ausnahme stellen die DFG-Prozesse dar, die immer terminieren und die mit einer bestimmten Taktzahl parametrisiert sind. Ist diese Taktzahl aber explizit zu null gewählt, verzögern auch diese Prozesse i.Allg. nicht (die Verzögerung hängt schließlich auch von der Aufnahmebereitschaft des nachfolgenden Prozesses ab).

Durch eine Pfadanalyse muss ermittelt werden, ob ein potentiell verzögerungsfreier Zyklus existiert. Ist dies der Fall, gibt es die folgenden Möglichkeiten, eine für alle Fälle garantierte Verzögerung sicherzustellen:

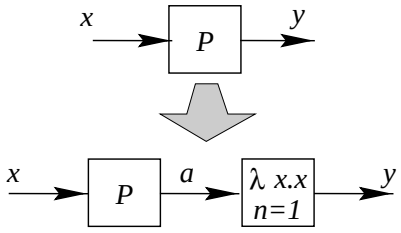
Gibt es in dem Zyklus einen DFG-Prozess mit variabler Taktzahl, deren konkreter Wert erst im Laufe der Synthese bestimmt werden soll, muss dafür gesorgt

werden, dass diese Taktzahl mindestens den Wert eins erhält.



$$
\begin{aligned}
&\vdash \mathsf{P\_IFC\_SYSTEM}\ P\ (\mathsf{reset}, x, y)\\
&\quad \approx\\
&\exists a\ b\ c\ d\ e.\\
&\quad \mathsf{K\_IFC\_DOUBLE}\ (\mathsf{reset}, x, a, b)\ \wedge\\
&\quad \mathsf{DFG\_IFC\_SYSTEM}\ (\lambda x.x, 1)\ (\mathsf{reset}, a, c)\ \wedge\\
&\quad \mathsf{P\_IFC\_SYSTEM}\ P\ (\mathsf{reset}, b, d)\ \wedge\\
&\quad \mathsf{K\_IFC\_SYNCHRONIZE}\ (\mathsf{reset}, c, d, e, y)\ \wedge\\
&\quad \mathsf{K\_IFC\_SINK}\ (\mathsf{reset}, e)
\end{aligned}
$$

(a) Besseres Zeitverhalten



$$
\begin{aligned}
&\vdash \mathsf{P\_IFC\_SYSTEM}\ P\ (\mathsf{reset}, x, y)\\
&\quad \approx\\
&\exists a.\\
&\quad \mathsf{P\_IFC\_SYSTEM}\ P\ (\mathsf{reset}, x, a)\ \wedge\\
&\quad \mathsf{DFG\_IFC\_SYSTEM}\ (\lambda x.x, 1)\ (\mathsf{reset}, a, y)
\end{aligned}
$$

(b) Weniger zusätzliche Fläche

Abbildung 6.17: Umwandeln eines P-Prozesses in eine nicht verzögerungsfreie Struktur

Gibt es diese Möglichkeit nicht, muss ein beteiligter Prozess durch eine der in Abbildung 6.17 dargestellten Strukturen ersetzt werden. Abbildung 6.17(a) zeigt auf der linken Seite, wie ein P-Prozess durch eine Struktur ersetzt werden kann, die den Markenfluss um mindestens einen Takt verzögert. Verzögert der P-Prozess von sich aus bereits, wird keine zusätzliche Verzögerung eingeführt. Der eingeführte DFG-Prozess verändert die Marke nicht, sondern verzögert sie nur um einen Takt. Auf der rechten Seite von Abbildung 6.17(a) ist das entsprechende Theorem dargestellt, das die funktionale Äquivalenz zwischen den beiden Strukturen beschreibt. Da sich das zeitliche Verhalten ändert, ist ein Theorem auf Basis der Verhaltensäquivalenz nicht beweisbar. Abbildung 6.17(b) zeigt eine zweite Variante, die immer eine zusätzliche Verzögerung um einen Takt erzeugt. Der Vorteil dieser Struktur liegt darin, dass der zusätzliche Hardwareaufwand geringer ist. Sie ist dann von Vorteil, wenn das zeitliche Verhalten im Gegensatz zum Flächenbedarf des Systems nicht kritisch ist.

Kapitel 7

Schaltungssynthese auf der algorithmischen Ebene

Ausgangspunkt für die Synthese auf der algorithmischen Ebene, die als High-Level Synthese bezeichnet wird, ist eine algorithmische Schaltungsbeschreibung, die sowohl die Funktionalität der zu synthetisierenden Schaltung als auch deren zeitliche Einordnung in ihre Umwelt berücksichtigt. In diesem Ansatz wird dabei unterschieden zwischen Beschreibungen, deren funktionaler Anteil über keinen Kontrollfluss verfügt (algor. DFG-Schaltungsbeschreibungen) und solchen, die auf Kontrollfluss basieren (algor. P-Schaltungsbeschreibungen). Beide Klassen von Schaltungsbeschreibungen eignen sich als Startpunkt für einen Syntheseprozess.

Das Kapitel beginnt mit der Erläuterung der generellen Vorgehensweise bei diesem Ansatz der Formalen Synthese: der Aufteilung von Entwurfsraumuntersuchung und Transformation. Diese Methode ist nicht auf die Synthese auf der algorithmischen Ebene beschränkt, sondern auch auf die Synthese auf der Systemebene anwendbar. Anschließend wird auf die Synthese von DFG- bzw. P-Term-basierten Beschreibungen eingegangen.

7.1 Aufteilung in Entwurfsraumuntersuchung und Transformation

Um die Komplexität eines Formale-Syntheseprogramms beherrschen zu können, wird in diesem Ansatz die Formale Synthese dergestalt durchgeführt, dass eine strikte Trennung zwischen der Entwurfsraumuntersuchung und der logischen Schaltungstransformation vorgenommen wird, wie es in Abbildung 7.1 dargestellt ist.

Durch diese Aufteilung wird es möglich, die Entwurfsraumuntersuchung außerhalb der Logik in effizienter Weise durchzuführen. Die dabei gewonnene Steuerinformation, die aussagt, wie der Syntheseschritt durchgeführt werden soll, wird anschließend der logischen Transformation als Parameter zugeführt. Diese Transformation erzeugt durch Anwendung einer Reihe von elementaren mathematischen Regeln innerhalb des Theorembeweisers HOL das Syntheseergebnis. Außerdem wird

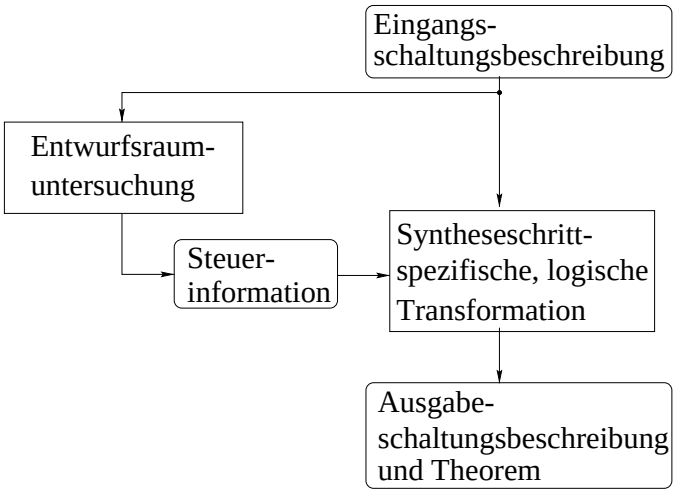


Abbildung 7.1: Entwurfsraumuntersuchung ↔ Schaltungstransformation

ein Theorem erzeugt, das aussagt, dass das Synthesergebnis zur Syntheseeingabe in einer bestimmten Relation steht. Bei Optimierungstransformationen ist diese Relation die Äquivalenz, während es bei Verfeinerungsschritten die Implikation ist. Es ist dabei möglich, die logische Transformation im Hintergrund ablaufen zu lassen, während schon die Steuerinformation für den nächsten Syntheseschritt berechnet wird. Wird im Laufe der logischen Transformation ein Fehler entdeckt, muss aber die Steuerinformation neu berechnet werden.

Es zeigt sich, dass die Aufteilung zwischen Entwurfsraumuntersuchung und Transformation auf viele Syntheseschritte angewandt werden kann. Dabei muss für jeden Syntheseschritt eine eigene logische Transformation angeboten werden, die aufgrund der Steuerinformation den Syntheseschritt durchführt. Es ist anzumerken, dass diese logische Transformation nur vom Syntheseschritt, aber nicht von der Entwurfsraumuntersuchungsmethode abhängt.

Der entscheidende Vorteil dieses Ansatzes liegt auf der Hand. Es werden dadurch zwei wichtige Kriterien für den Entwurf erfüllt: Kosteneffizienz und Korrektheit des Syntheseergebnisses. Die Entwurfsraumuntersuchung sorgt dafür, dass möglichst die bei einer vorgegebenen Kostenfunktion günstigste Lösung ausgesucht wird. Indem die Entwurfsraumuntersuchung außerhalb der Logik stattfindet, kann das große Repertoire an in der Literatur vorhandenen Synthesealgorithmen eingesetzt werden. Auf der anderen Seite wird die Korrektheit der Synthesetransformation dadurch garantiert, dass sie im Theorembeweiser HOL durchgeführt wird. Aufgrund dieser Tatsache können falsche Syntheseergebnisse *nicht* erzeugt werden, auch dann nicht, wenn die Steuerinformation fehlerhaft ist, z.B. wenn die Datenabhängigkeiten durch sie verletzt würden. In diesem Fall wird die logische Transformation an irgendeinem Punkt in HOL scheitern und anstelle eines falschen Ergebnisses wird eine Fehlermeldung produziert.

Wie aus dem vorigen Kapitel deutlich wird, sind die Aufgaben, die während der

High-Level Synthese durchgeführt werden müssen, sehr komplex und bedingen einander in starkem Maße. Oft werden daher die einzelnen Aufgaben wie etwa die zeitliche Einteilung und die Bereitstellung von Komponenten iterativ angegangen, um die gegenseitigen Wechselbeziehungen zu berücksichtigen. Dieses Vorgehen stellt für die Trennung von Entwurfsraumuntersuchung und logischer Transformation keine Einschränkung dar. Auch wenn die logischen Transformationen in einer bestimmten Reihenfolge angewandt werden müssen, können die entsprechenden Steuerinformationen zu beliebigen Zeitpunkten berechnet werden. Dieser Zusammenhang ist in Abbildung 7.2 dargestellt. Es ist sowohl möglich, für jede einzelne Transformation die Steuerinformation einzeln zu bestimmen, als auch vorab für mehrere Transformationsschritte in der Logik eine Reihe von Steuerinformationen zu berechnen.

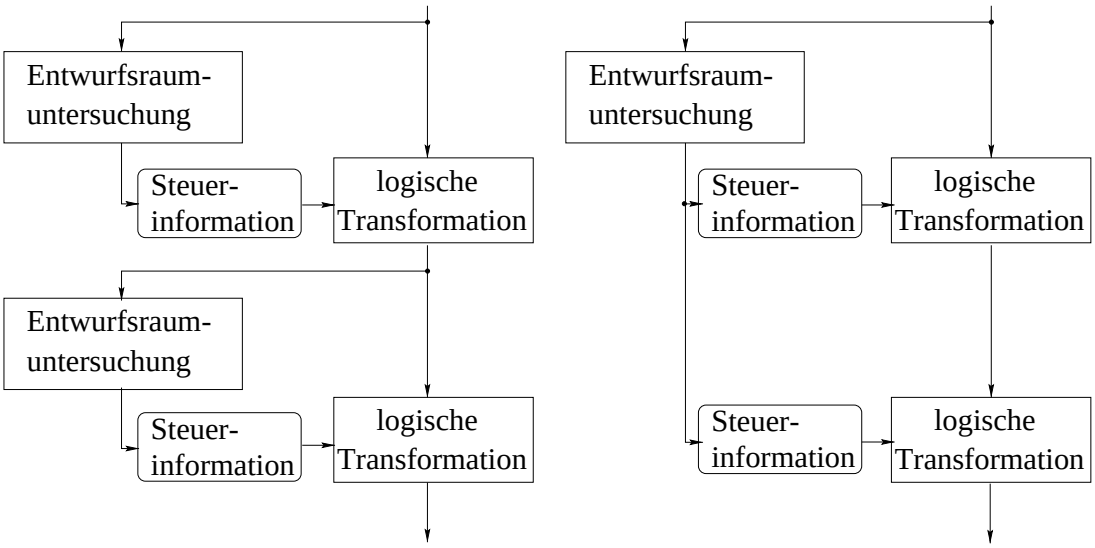


Abbildung 7.2: Möglichkeiten für die Aufteilung

7.2 Synthese von algorithmischen DFG-Schaltungsbeschreibungen

Die Synthese von DFG-Term-basierten Schaltungsbeschreibungen läuft so ab, dass zunächst auf dem zugrunde liegenden DFG-Term die Schritte der zeitlichen Einordnung der Operationen, der Bereitstellung und Zuordnung von Komponenten sowie die Kommunikationssynthese durchgeführt werden. Anschließend wird die sich dadurch ergebende Schaltungsbeschreibung, die sich aus dem veränderten DFG-Term und einem ausgewählten Schnittstellenverhaltensmuster zusammensetzt, durch eine Schnittstellensynthese mittels Anwendung eines Implementierungstheorems in eine Struktur auf der RT-Ebene überführt.

Zur Veranschaulichung der Synthese soll ein beispielhafter DFG-Term betrachtet werden, der eine Polynomdivision durchführt:

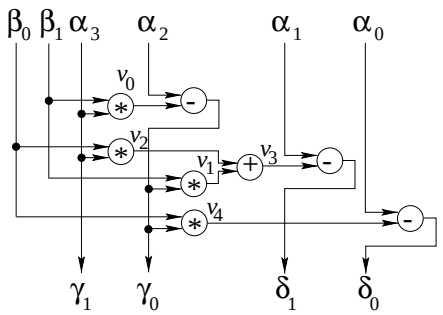
$$\frac{\sum_{i=0}^{p+q} \alpha_i \, x^i}{\sum_{i=0}^{p} \beta_i \, x^i} = \sum_{i=0}^{q} \gamma_i \, x^i + \frac{\sum_{i=0}^{p-1} \delta_i \, x^i}{\sum_{i=0}^{p} \beta_i \, x^i} \tag{7.1}$$

Gegeben seien die Koeffizienten α_i des Zählers und β_i des Nenners. Die Koeffizienten γ_i des Ergebnisses sowie δ_i des Restes sollen berechnet werden. Um die Berechnung zu vereinfachen, sei angenommen, dass der Nenner bez. β_p normalisiert sei ($\beta_p = 1$). Nach einigen algebraischen Umformungen erhält man die folgenden zwei Formeln für die gewünschten Koeffizienten:

$$\gamma_i = \alpha_{i+p} - \sum_{k=i+1}^{min\{i+p,q\}} \beta_{i+p-k} \cdot \gamma_k \qquad i = 0 \ldots q \tag{7.2}$$

$$\delta_j = \alpha_j - \sum_{k=0}^{min\{j,q\}} \beta_{j-k} \cdot \gamma_k \qquad j = 0 \ldots p-1 \tag{7.3}$$

Im Folgenden sei $p = 2$ und $q = 1$ angenommen. Man erhält damit den in Abbildung 7.3 dargestellten Datenflussgraph bzw. DFG-Term. Zu beachten ist, dass der Koeffizient γ_q immer identisch ist mit α_{p+q}.



(a) Datenflussgraph

$$\begin{array}{l}
\lambda(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1). \\
\quad \mathsf{let}\ v_0 = \beta_1 * \alpha_3\ \mathsf{in} \\
\quad \mathsf{let}\ \gamma_0 = \alpha_2 - v_0\ \mathsf{in} \\
\quad \mathsf{let}\ v_1 = \beta_1 * \gamma_0\ \mathsf{in} \\
\quad \mathsf{let}\ v_2 = \beta_0 * \alpha_3\ \mathsf{in} \\
\quad \mathsf{let}\ v_3 = v_2 + v_1\ \mathsf{in} \\
\quad \mathsf{let}\ \delta_1 = \alpha_1 - v_3\ \mathsf{in} \\
\quad \mathsf{let}\ v_4 = \beta_0 * \gamma_0\ \mathsf{in} \\
\quad \mathsf{let}\ \delta_0 = \alpha_0 - v_4\ \mathsf{in} \\
\quad (\gamma_0, \alpha_3, \delta_0, \delta_1)
\end{array}$$

(b) DFG-Term

Abbildung 7.3: DFG-Term der Polynomdivision mit $p = 2$ und $q = 1$

7.2.1 Zeitliche Einordnung, Bereitstellung, Zuordnung, Kommunikationssynthese

Die Transformation von DFG-Termen für diese vier Syntheseschritte beruht jeweils auf der bereits in den Abschnitten 2.1.2 und 5.1.1 erwähnten Normalisierung von λ-Ausdrücken. Der allgemeine Algorithmus dafür lautet wie folgt:

1. Der DFG-Term f wird durch rückwärtige gepaarte η-Konversion (siehe Abschnitt 2.1.2) in den äquivalenten DFG-Term $\lambda(x_1, x_2, \ldots, x_m).f\ (x_1, x_2, \ldots, x_m)$ überführt.

2. Gibt es in f $\circ$-Operationen, so werden diese durch Termersetzung mit ihrer Definition expandiert: $h \circ g = (\lambda x.\ h(g\ x))$

3. Es werden wo immer möglich (gepaarte) β-Konversionen durchgeführt.

Basierend auf dieser Normalisierung lässt sich für die Transformation von DFG-Termen ein generelles Schema angeben:

1. Ausgehend von einem DFG-Term f wird außerhalb der Logik nach dem in Abschnitt 7.1 vorgestellten Verfahren eine Steuerinformation für die Transformation bestimmt.

2. Dieser Steuerinformation zufolge wird ein neuer DFG-Term f' erstellt.

3. f und f' werden daraufhin normalisiert, wobei die zwei Theoreme $\vdash f = \hat{f}$ und $\vdash f' = \hat{f}$ entstehen (die Normalform ist bei richtiger Steuerinformation in beiden Fällen dieselbe).

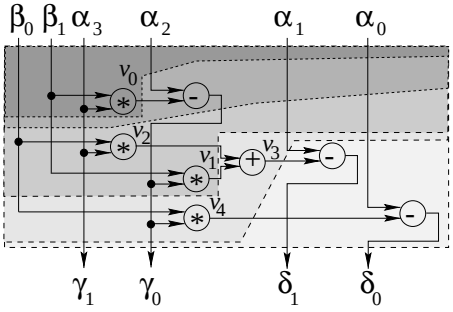
4. Die beiden Theoreme werden vereint zum Theorem $\vdash f = f'$ durch Ausnützung der Symmetrie- und Transitivitätseigenschaft der Äquivalenzrelation.

Dieses generelle Schema für die Transformation von DFG-Termen hat einen entscheidenden Nachteil. DFG-Terme, deren Zwischenergebnisse mehrfach verwendet werden (wie etwa die Variable γ_0 in Abbildung 7.3) werden während der β-Konversion mehrfach dupliziert. Da solche β-Redizes auch verschachtelt sein können, kann es geschehen, dass die Termgröße während der Normalisierung exponentiell anwächst und damit auch der Zeitbedarf für die Transformation.

Das generelle Transformationsschema ist vergleichbar mit einer Post-Synthese-Verifikation, bei der nicht die Kenntnis mit eingeht, *wie* der Syntheseschritt durchgeführt wurde. Im Folgenden soll daher für jeden Syntheseschritt eine *spezifische* Transformation vorgestellt werden, die genau diese Kenntnis ausnützt. Auch diese spezifischen Transformationen basieren auf der oben vorgestellten Methode, wobei jedoch die Normalisierung an den jeweiligen Syntheseschritt angepasst ist.

Zeitliche Einordnung

Während dieses Syntheseschritts wird ein DFG-Term f umgewandelt in eine Komposition f' von n DFG-Termen, wobei n die Anzahl der dabei eingeführten Kontrollschritte ist. In Abbildung 7.4 ist dies für den oben eingeführten DFG-Term zur Berechnung einer Polynomdivision beispielhaft dargestellt. Der neue DFG-Term ergibt sich dabei zu $f' = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1$.



(a) Datenflussgraph nach zeitlicher Einordnung

$\lambda(v_3, v_4, \gamma_0, \alpha_0, \alpha_1, \alpha_3).$
  let $\delta_1 = \alpha_1 - v_3$ in
  let $\delta_0 = \alpha_0 - v_4$ in
    $(\gamma_0, \alpha_3, \delta_0, \delta_1)$
$\circ$ $\lambda(v_2, v_1, \gamma_0, \alpha_0, \alpha_1, \alpha_3, \beta_0).$
  let $v_3 = v_2 + v_1$ in
  let $v_4 = \beta_0 * \gamma_0$ in
    $(v_3, v_4, \gamma_0, \alpha_0, \alpha_1, \alpha_3)$
$\circ$ $\lambda(\gamma_0, \alpha_0, \alpha_1, \alpha_3, \beta_0, \beta_1).$
  let $v_1 = \beta_1 * \gamma_0$ in
  let $v_2 = \beta_0 * \alpha_3$ in
    $(v_2, v_1, \gamma_0, \alpha_0, \alpha_1, \alpha_3, \beta_0)$
$\circ$ $\lambda(v_0, \alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1).$
  let $\gamma_0 = \alpha_2 - v_0$ in
    $(\gamma_0, \alpha_0, \alpha_1, \alpha_3, \beta_0, \beta_1)$
$\circ$ $\lambda(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1).$
  let $v_0 = \beta_1 * \alpha_3$ in
    $(v_0, \alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1)$

(b) Komposition von DFG-Termen

Abbildung 7.4: DFG-Term der Polynomdivision mit $p = 2$ und $q = 1$ nach zeitlicher Einordnung

Anstatt den ursprünglichen DFG-Term f und den durch die Steuerinformation eines Einordnungsalgorithmus' gewonnenen neuen DFG-Term f' auf einmal zu normalisieren, wird bei der spezifischen Transformation der Normalisierungsschritt $(n-1)$-mal durchgeführt. Bei jedem dieser Schritte werden nur die Variablen durch β-Konversion expandiert, die dem betrachteten Kontrollschritt zugeordnet sind. Diese Vorgehensweise wird durch Abbildung 7.5 veranschaulicht.

Der Unterschied der Laufzeiten zwischen der Anwendung des allgemeinen Transformationsschemas und der spezifischen Transformation ist in Abbildung 7.6 dargestellt. (Die Experimente wurden dabei auf einer SUN UltraCreator mit Solaris 5.5.1 und 196 MB Hauptspeicher durchgeführt.) Beide Transformationen wurden auf eine Reihe von DFG-Termen für die Polynomdivision angewandt. Bei der Transformation nach dem generellen Schema kommt es zu einem exponentiellen Anwachsen sowohl des Zeit- wie auch des Speicherbedarfs. Größere DFG-Terme konnten aufgrund des Speichermangels nicht mehr transformiert werden. Auf der anderen Seite sieht man, dass mittels der dem Syntheseschritt angepassten Transformation auch größere DFG-Terme in angemessener Zeit transformiert werden können.

$\vdash f = f_5 \circ f_{5_r}$	bei Normalisierung nur β-Konversion von	δ_1, δ_0
$\vdash f_{5_r} = f_4 \circ f_{4_r}$	"	v_3, v_4
$\vdash f_{4_r} = f_3 \circ f_{3_r}$	"	v_1, v_2
$\vdash f_{3_r} = f_2 \circ f_1$	"	γ_0
$\Rightarrow \quad \vdash f = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1$		

Abbildung 7.5: Vorgehen bei der spezifischen Transformation für die zeitliche Einordnung

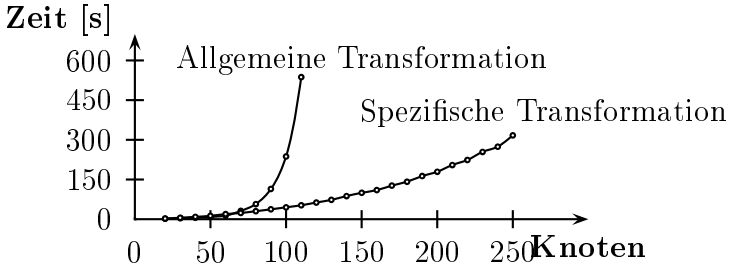


Abbildung 7.6: Vergleich der Transformationen ($p = 5, q = 1, 2, \ldots$)

Bereitstellung und Zuordnung von Registern

Bei der Bereitstellung von Registern wird die Anzahl und Art der Register bestimmt, die benötigt werden, um die Zwischenergebnisse zwischen zwei Kontrollschritten zu speichern. Beim Zuordnen wird für jeden Kontrollschritt eine Abbildung zwischen den Registern und den Hilfsvariablen, in denen die Zwischenergebnisse abgelegt sind, bestimmt.

Diese beiden Transformationen haben eines gemeinsam. Sie haben nur einen Einfluss auf die Schnittstelle zwischen zwei Kontrollschritten. Die Schnittstellen, die vor diesen Syntheseschritten durch beliebige Variablen bezeichnet waren, werden nun durch die konkreten Bezeichnungen der bereitgestellten Register ausgetauscht. In der diesen Syntheseschritten angepassten Transformation werden die Schnittstellen nacheinander ausgetauscht, anstatt dies auf einmal durchzuführen. Das allgemeine Transformationsschema wird also nicht auf den Term f' angewandt, der sich durch die zeitliche Einordnung ergeben hat, sondern $(n-1)$-mal auf einen Teilterm $(f_{i+1} \circ f_i)$, wie es in Abbildung 7.7 dargestellt ist. Die Schreibweise $f_{i'}$ soll dabei andeuten, dass die Eingangsschnittstelle des DFG-Terms f_i ausgetauscht wurde. $f_{'i}$ bezeichnet eine ausgetauschte Ausgangsschnittstelle und $f_{'i'}$ steht für den Term, der sich ergibt, wenn beide Schnittstellen des DFG-Terms f_i ausgetauscht wurden. Zusätzlich muss bei der Transformation $2(n-2)$-mal das Assoziativitätsgesetz für den $\circ$-Operator angewandt werden, um die jeweiligen Schnittstellen austauschen zu können.

Für das laufende Beispiel müssen sieben Register $r_1, \ldots, r_7$ bereitgestellt werden.

$\vdash f_5 \circ f_4 = f_{5'} \circ f_{'4}$	bei Normalisierung nur β-Konversion von	$\delta_1, \delta_0, v_3, v_4$
$\vdash f_{'4} \circ f_3 = f_{'4'} \circ f_{'3}$	"	v_3, v_4, v_1, v_2
$\vdash f_{'3} \circ f_2 = f_{'3'} \circ f_{'2}$	"	v_1, v_2, γ_0
$\vdash f_{'2} \circ f_1 = f_{'2'} \circ f_{'1}$	"	γ_0, v_0
$\Rightarrow \quad \vdash f = f_{5'} \circ f_{'4'} \circ f_{'3'} \circ f_{'2'} \circ f_{'1}$		

Abbildung 7.7: Vorgehen bei der spezifischen Transformation für die Bereitstellung/Zuordnung von Registern

Nach der Transformation ergibt sich der in Abbildung 7.8 dargestellte DFG-Term. In jedem DFG-Term drücken dabei die Namen $r_1, \ldots, r_7$ die Werte in den Registern vor der Funktionsauswertung aus, und die Namen $r'_1, \ldots, r'_7$ die Werte in den Registern danach.

$$
\begin{aligned}
&(\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).\\
&\quad \mathsf{let}\ \delta_1 = r_3 - r_5\ \mathsf{in}\ \mathsf{let}\ \delta_0 = r_2 - r_4\ \mathsf{in}\\
&\quad \mathsf{let}\gamma_0 = r_1\ \mathsf{in}\ \mathsf{let}\ \alpha_3 = r_6\ \mathsf{in}\ (\gamma_0, \alpha_3, \delta_0, \delta_1))\\
&\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).\\
&\quad \mathsf{let}\ r'_5 = r_7 + r_4\ \mathsf{in}\ \mathsf{let}\ r'_4 = r_5 * r_1\ \mathsf{in}\ \mathsf{let}\ r'_1 = r_1\ \mathsf{in}\\
&\quad \mathsf{let}\ r'_2 = r_2\ \mathsf{in}\ \mathsf{let}\ r'_3 = r_3\ \mathsf{in}\ \mathsf{let}\ r'_6 = r_6\ \mathsf{in}\ (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7))\\
&\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).\\
&\quad \mathsf{let}\ r'_4 = r_7 * r_1\ \mathsf{in}\ \mathsf{let}\ r'_7 = r_5 * r_6\ \mathsf{in}\ \mathsf{let}\ r'_1 = r_1\ \mathsf{in}\ \mathsf{let}\ r'_2 = r_2\ \mathsf{in}\\
&\quad \mathsf{let}\ r'_3 = r_3\ \mathsf{in}\ \mathsf{let}\ r'_6 = r_6\ \mathsf{in}\ \mathsf{let}\ r'_5 = r_5\ \mathsf{in}\ (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7))\\
&\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).\\
&\quad \mathsf{let}\ r'_1 = r_4 - r_1\ \mathsf{in}\ \mathsf{let}\ r'_2 = r_2\ \mathsf{in}\ \mathsf{let}\ r'_3 = r_3\ \mathsf{in}\ \mathsf{let}\ r'_6 = r_6\ \mathsf{in}\\
&\quad \mathsf{let}\ r'_5 = r_5\ \mathsf{in}\ \mathsf{let}\ r'_7 = r_7\ \mathsf{in}\ (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7))\\
&\circ\ (\lambda(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1).\\
&\quad \mathsf{let}\ r'_1 = \beta_1 * \alpha_3\ \mathsf{in}\ \mathsf{let}\ r'_2 = \alpha_0\ \mathsf{in}\ \mathsf{let}\ r'_3 = \alpha_1\ \mathsf{in}\ \mathsf{let}\ r'_4 = \alpha_2\ \mathsf{in}\\
&\quad \mathsf{let}\ r'_6 = \alpha_3\ \mathsf{in}\ \mathsf{let}\ r'_5 = \beta_0\ \mathsf{let}\ r'_7 = \beta_1\ \mathsf{in}\ (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7))
\end{aligned}
$$

Abbildung 7.8: DFG-Term nach Registerbereitstellung und -zuordnung

Bereitstellung und Zuordnung von Funktionseinheiten

In diesem Syntheseschritt wird eine zusammengefasste funktionale Einheit FU eingeführt. Sie enthält die Funktionseinheiten, durch die die Operationen jedes Kontrollschrittes realisiert werden. Die Einheit FU wird mittels eines β-Redex beschrieben:

$$\mathsf{let}\ \mathsf{FU} = g\ \mathsf{in}\ (f_n \circ \cdots \circ f_1)$$

In dieser Schreibweise ist g ein DFG-Term, und jeder DFG-Term f_i enthält genau einen einzigen FU-Operator.

Die Transformation hat nur Einfluss auf die einzelnen DFG-Terme f_i. Aus diesem Grunde kann die Transformation in der Weise durchgeführt werden, dass jeder der n DFG-Terme einzeln umgeformt wird, anstatt die Normalisierung wieder auf den gesamten DFG-Term f anzuwenden (siehe Abbildung 7.9).

$$\vdash f_{5'} = f_{5'}{}_{\mathsf{FU}} \quad \cdots \quad \vdash f_{'1} = f_{'1}{}_{\mathsf{FU}}$$

$$\Rightarrow \quad \vdash f = \mathsf{let\ FU\ =\ } g \mathsf{\ in\ } (f_{5'}{}_{\mathsf{FU}} \circ f_{'4'}{}_{\mathsf{FU}} \circ f_{'3'}{}_{\mathsf{FU}} \circ f_{'2'}{}_{\mathsf{FU}} \circ f_{'1}{}_{\mathsf{FU}})$$

Abbildung 7.9: Vorgehen bei der spezifischen Transformation für die Bereitstellung/Zuordnung von Funktionseinheiten

Für unser Beispiel der Polynomdivision mit $p = 2$ und $q = 1$ ergibt sich der in Abbildung 7.10 dargestellte Term. Es wurde eine Funktionseinheit FU bereitgestellt, die sich aus vier Einheiten zusammensetzt: zwei Multiplizierer MULT, ein Subtrahierer SUB und eine Multifunktionseinheit ADD_SUB, die sowohl addieren als auch subtrahieren kann. Diese Multifunktionseinheit hat neben den beiden Dateneingängen noch einen zusätzlichen Kontrolleingang. Hat dieser Eingang den Wert T, wird addiert, im anderen Fall subtrahiert. Die Bereitstellung dieser Multifunktionseinheit hat den Vorteil, dass stattdessen nicht ein zusätzlicher Subtrahierer und ein Addierer bereitgestellt werden müssen. Die Funktionseinheit FU hat 15 Eingänge und zehn Ausgänge. Während auf den ersten neun Eingängen Berechnungen ausgeführt werden können, werden die restlichen sechs Eingänge nur durchgeschleift. Es muss angemerkt werden, dass der Term in Abbildung 7.10 entgegen der Darstellung in Abschnitt 5.1.1 über freie Variablen verfügt. Diese freien Variablen I_i bezeichnen aber einfach nur beliebige Werte an manchen Eingängen der FU. Es macht in diesem Stadium der Synthese keinen Sinn, diese Werte zu konkretisieren. In Abschnitt 5.1.1 wurden keine freien Variablen zugelassen, um sicherzustellen, dass es keine Verletzung der Datenabhängigkeiten gibt. Durch die hier vorkommenden freien Variablen kann dies nicht geschehen. Sie stellen einfach nur Platzhalter dar, die im weiteren Verlauf der Synthese instantiiert werden können.

Kommunikationssynthese

Nachdem in den vorherigen Schritten die Register und die zusammengefasste Funktionseinheit FU eingeführt wurden, muss nun die Kommunikationsstruktur von FU zu den Registern und zurück festgelegt werden. Dazu wird der Term in die folgende Form umgewandelt:

$$(g^n \circ \mathsf{FU} \circ h^n) \circ \cdots \circ (g^1 \circ \mathsf{FU} \circ h^1) \tag{7.4}$$

Jede Teilfunktion, die einen Kontrollschritt bezeichnet, wird umgewandelt in eine Komposition von drei Funktionen, wobei die Funktionseinheit FU jeweils in der

$$
\begin{array}{l}
\mathsf{let\ FU\ =} \\
\quad \lambda(I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, I_{13}, I_{14}, I_{15}). \\
\qquad \mathsf{let}\ O_1 \ = \ I_4\ \mathsf{ADD\_SUB}\ I_1\ I_2\ I_3\ \mathsf{in} \\
\qquad \mathsf{let}\ O_2 \ = \ I_4\ \mathsf{MULT}\ I_5\ \mathsf{in} \\
\qquad \mathsf{let}\ O_3 \ = \ I_6\ \mathsf{MULT}\ I_7\ \mathsf{in} \\
\qquad \mathsf{let}\ O_4 \ = \ I_8\ \mathsf{SUB}\ I_9\ \mathsf{in} \\
\qquad \mathsf{let}\ O_5 \ = \ I_{10}\ \mathsf{in\ let}\ O_6 \ = \ I_{11}\ \mathsf{in\ let}\ O_7 \ = \ I_{12}\ \mathsf{in} \\
\qquad \mathsf{let}\ O_8 \ = \ I_{13}\ \mathsf{in\ let}\ O_9 \ = \ I_{14}\ \mathsf{in\ let}\ O_{10} \ = \ I_{15}\ \mathsf{in} \\
\quad (O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8, O_9, O_{10}) \\
\mathsf{in} \\
(\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7). \\
\qquad \mathsf{let}\ (\delta_0, O_2, O_3, \delta_1, \gamma_0, \alpha_3, O_7, O_8, O_9, O_{10}) \ = \\
\qquad\quad \mathsf{FU}\ (\mathsf{F}, r_2, r_4, I_4, I_5, I_6, I_7, r_3, r_5, r_1, r_6, I_{12}, I_{13}, I_{14}, I_{15})\ \mathsf{in} \\
\qquad (\gamma_0, \alpha_3, \delta_0, \delta_1)) \\
\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7). \\
\qquad \mathsf{let}\ (r'_5, r'_4, O_3, O_4, r'_1, r'_2, r'_3, r'_6, O_9, O_{10}) \ = \\
\qquad\quad \mathsf{FU}\ (\mathsf{T}, r_7, r_4, r_5, r_1, I_6, I_7, I_8, I_9, r_1, r_2, r_3, r_6, I_{14}, I_{15})\ \mathsf{in} \\
\qquad (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \\
\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7). \\
\qquad \mathsf{let}\ (O_1, r'_4, r'_7, O_4, r'_1, r'_2, r'_3, r'_6, r'_5, O_{10}) \ = \\
\qquad\quad \mathsf{FU}\ (I_1, I_2, I_3, r_7, r_1, r_5, r_6, I_8, I_9, r_1, r_2, r_3, r_6, r_5, I_{15})\ \mathsf{in} \\
\qquad (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \\
\circ\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7). \\
\qquad \mathsf{let}\ (O_1, O_2, O_3, r'_1, r'_2, r'_3, r'_6, r'_5, r'_7, O_{10}) \ = \\
\qquad\quad \mathsf{FU}\ (I_1, I_2, I_3, I_4, I_5, I_6, I_7, r_4, r_1, r_2, r_3, r_6, r_5, r_7, I_{15})\ \mathsf{in} \\
\qquad (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \\
\circ\ (\lambda(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1). \\
\qquad \mathsf{let}\ (O_1, r'_1, O_3, O_4, r'_2, r'_3, r'_4, r'_6, r'_5, r'_7) \ = \\
\qquad\quad \mathsf{FU}\ (I_1, I_2, I_3, \beta_1, \alpha_3, I_6, I_7, I_8, I_9, \alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1)\ \mathsf{in} \\
\qquad (r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7))
\end{array}
$$

Abbildung 7.10: DFG-Term nach Bereitstellung und Zuordnung der Funktionseinheiten

Mitte erscheint. h^1 bezeichnet die Verdrahtung zwischen dem Eingang und FU, g^n drückt die Verdrahtung zwischen FU und dem Ausgang aus, und die restlichen Funktionen g^i und h^i bezeichnen die Verdrahtung zwischen FU und den bereitgestellten Registern und umgekehrt. Auch diese Transformation kann jeweils auf den einzelnen DFG-Termen durchgeführt werden. Aus diesem Grunde und weil sich das Ergebnis durch die Zuordnung der Funktionseinheit ergibt, wird sie sinnvollerweise zusammen mit der Transformation zur Bereitstellung und Zuordnung der Funktionseinheit durchgeführt.

Abbildung 7.11 zeigt den entsprechenden Term für das fortlaufende Beispiel. Dabei wurde wieder die Funktionseinheit FU durch einen β-Redex dargestellt.

$$
\begin{array}{l}
\mathsf{let\ FU\ =} \\
\quad \lambda(I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, I_{13}, I_{14}, I_{15}). \\
\qquad \mathsf{let}\ O_1 \ = \ \mathsf{ADD\_SUB}\ I_1\ I_2\ I_3\ \mathsf{in} \\
\qquad \mathsf{let}\ O_2 \ = \ I_4\ \mathsf{MULT}\ I_5\ \mathsf{in} \\
\qquad \mathsf{let}\ O_3 \ = \ I_6\ \mathsf{MULT}\ I_7\ \mathsf{in} \\
\qquad \mathsf{let}\ O_4 \ = \ I_8\ \mathsf{SUB}\ I_9\ \mathsf{in} \\
\qquad \mathsf{let}\ O_5 \ = \ I_{10}\ \mathsf{in\ let}\ O_6 \ = \ I_{11}\ \mathsf{in\ let}\ O_7 \ = \ I_{12}\ \mathsf{in} \\
\qquad \mathsf{let}\ O_8 \ = \ I_{13}\ \mathsf{in\ let}\ O_9 \ = \ I_{14}\ \mathsf{in\ let}\ O_{10} \ = \ I_{15}\ \mathsf{in} \\
\quad (O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8, O_9, O_{10}) \\
\mathsf{in} \\
((\lambda(\delta_0, O_2, O_3, \delta_1, \gamma_0, \alpha_3, O_7, O_8, O_9, O_{10}).(\gamma_0, \alpha_3, \delta_0, \delta_1)) \circ \mathsf{FU} \circ \\
\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).(\mathsf{F}, r_2, r_4, I_4, I_5, I_6, I_7, r_3, r_5, r_1, r_6, I_{12}, I_{13}, I_{14}, I_{15}))) \\
\circ \\
((\lambda(r'_5, r'_4, O_3, O_4, r'_1, r'_2, r'_3, r'_6, O_9, O_{10}).(r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \circ \mathsf{FU} \circ \\
\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).(\mathsf{T}, r_7, r_4, r_5, r_1, I_6, I_7, I_8, I_9, r_1, r_2, r_3, r_6, I_{14}, I_{15}))) \\
\circ \\
((\lambda(O_1, r'_4, r'_7, O_4, r'_1, r'_2, r'_3, r'_6, r'_5, O_{10}).(r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \circ \mathsf{FU} \circ \\
\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).(I_1, I_2, I_3, r_7, r_1, r_5, r_6, I_8, I_9, r_1, r_2, r_3, r_6, r_5, I_{15}))) \\
\circ \\
((\lambda(O_1, O_2, O_3, r'_1, r'_2, r'_3, r'_6, r'_5, r'_7, O_{10}).(r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \circ \mathsf{FU} \circ \\
\ (\lambda(r_1, r_2, r_3, r_4, r_5, r_6, r_7).(I_1, I_2, I_3, I_4, I_5, I_6, I_7, r_4, r_1, r_2, r_3, r_6, r_5, r_7, I_{15}))) \\
\circ \\
((\lambda(O_1, r'_1, O_3, O_4, r'_2, r'_3, r'_4, r'_6, r'_5, r'_7).(r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7)) \circ \mathsf{FU} \circ \\
\ (\lambda(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1).(I_1, I_2, I_3, \beta_1, \alpha_3, I_6, I_7, I_8, I_9, \alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \beta_1)))
\end{array}
$$

Abbildung 7.11: DFG-Term nach Kommunikationssynthese

7.2.2 Schnittstellensynthese

In Abschnitt 5.2.2 wurde vorgestellt, wie Schnittstellenverhaltensmuster für DFG-Terme definiert werden können. Diese Schnittstellenverhaltensmuster können dann mit einem beliebigen DFG-Term zu einer algorithmischen DFG-Schaltungsbeschreibung kombiniert werden. Für jedes dieser Schnittstellenverhaltensmuster wird zusätzlich ein korrektes Implementierungsmuster auf der RT-Ebene in Form eines vorbewiesenen Theorems zur Verfügung gestellt. Jedes dieser Implementierungstheoreme geht davon aus, dass der DFG-Term in folgender Form vorliegt:

$$(g^n \circ h^n) \circ (\mathsf{list\_o}\ (\mathsf{MAP}\ (\lambda(p,q).p \circ \mathsf{FU} \circ q)\ L)) \circ (g^1 \circ h^1) \tag{7.5}$$

Diese Form lässt sich in einfacher Weise aus (7.4) gewinnen. L bezeichnet dabei eine Liste mit den inneren Verdrahtungen g^i und h^i: $L = [(g^{n-1}, h^{n-1}), \ldots, (g^2, h^2)]$. Diese Form impliziert, dass der ursprüngliche DFG-Term f während des Syntheseschritts der zeitlichen Einordnung in mindestens zwei Teile zerlegt wurde (dann

wäre $L = [\,]$). Der triviale Fall, dass sich während der Einordnung nur ein einziger Kontrollschritt ergibt, soll hier nicht näher betrachtet werden.

Die Funktionen list_o und MAP sind nicht Teil der Beschreibungssprache Gropius. Sie sind durch primitive Rekursion über Listen definiert, wie in Abbildung 7.12 beschrieben ist. Die Funktion CONS hängt ein einzelnes Element vorne an eine

$$
\begin{array}{rcl}
\mathsf{list\_o}\ [\,] & = & (\lambda x.x) \\
\mathsf{list\_o}\ (\mathsf{CONS}\ h\ t) & = & (\mathsf{list\_o}\ t)\ \circ\ h \\
 & & \\
\mathsf{MAP}\ f\ [\,] & = & [\,] \\
\mathsf{MAP}\ f\ (\mathsf{CONS}\ h\ t) & = & \mathsf{CONS}\ (f\ h)\ (\mathsf{MAP}\ f\ t) \\
 & & \\
\mathsf{APPEND}\ [\,]\ l & = & l \\
\mathsf{APPEND}\ (\mathsf{CONS}\ h\ t)\ l & = & \mathsf{CONS}\ h\ (\mathsf{APPEND}\ t\ l) \\
 & & \\
\mathsf{LENGTH}\ [\,] & = & 0 \\
\mathsf{LENGTH}\ (\mathsf{CONS}\ h\ t) & = & 1\ +\ (\mathsf{LENGTH}\ t)
\end{array}
$$

Abbildung 7.12: Listenfunktionen

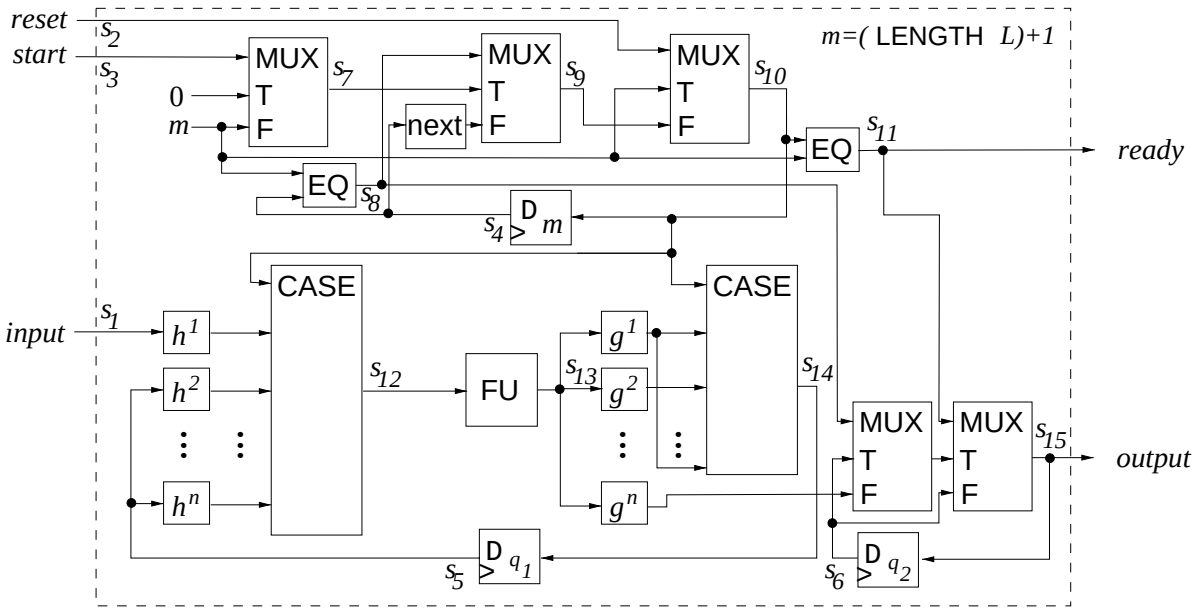
Liste an. Die Funktion list_o beschreibt eine Komposition von Funktionen, die alle in einer Liste aufgeführt sind. Ist die Liste leer, entspricht sie gerade der Identität $(\lambda x.x)$. Im anderen Fall, wenn die Liste sich mindestens aus einer Funktion h und einer Restliste t zusammensetzt, ergibt sich die Komposition aus h mit der Funktion (list_o t). Es ist dabei zu beachten, dass der Typ aller Funktionen in der Liste L gleich sein muss. Die Funktion MAP wendet eine Funktion f auf jedes Element einer Liste an.

Die Tatsache, dass die drei Funktionen list_o, CONS und MAP keinen Teil der Sprache Gropius darstellen, ist kein Widerspruch. Sie beziehen sich nur auf Listen und werden dazu verwendet, eine allgemeine Schablone zu beschreiben. Während der Synthese werden die allgemeinen Listen mit konkreten Listen instantiiert. Durch Termersetzung mit ihren in Abbildung 7.12 gezeigten Definitionen können diese drei Hilfsfunktionen beseitigt werden. Es ergibt sich dann ein reiner Gropius-Term.

Abbildung 7.13(a) zeigt als Beispiel die Gropius-Beschreibung des Implementierungsmusters DFG_IMP_RESET, das das Pendant zu dem Schnittstellenverhaltensmuster DFG_IFC_RESET darstellt. Die RT-Struktur ist in Abbildung 7.13(b) schematisch dargestellt. q_1 ist ein beliebiger Initialwert in den bereitgestellten Registern, während q_2 ein beliebiger Anfangswert in dem Register ist, das den Ausgangswert hält. Der Ausdruck (CASE L i) greift das i-te Element einer Liste L heraus. Die Funktionen APPEND und LENGTH sind in Abbildung 7.12 beschrieben. APPEND kettet zwei Listen hintereinander und LENGTH liefert die Länge einer Liste. enum und next sind Funktionen über einen Aufzählungsdatentyp. Die Kardinalität eines solchen Aufzählungsdatentyps lässt sich beliebig wählen. Ein Aufzählungsdatentyp

```
DFG_IMP_RESET (input, reset, start, output, ready, FU, g^n, h^n, L, g^1, h^1) =
    ∃q1 q2.
    (λt. (output t, ready t)) =
      automaton
        ( (λ((s1, s2, s3), (s4, s5, s6)).
            let s7  = MUX (s3, enum ((LENGTH L) + 2) 0, (LENGTH L) + 1) in
            let s8  = EQ ((LENGTH L) + 1, s4) in
            let s9  = MUX (s8, s7, INC next ((LENGTH L) + 2) s4) in
            let s10 = MUX (s2, (LENGTH L) + 1, s9) in
            let s11 = EQ (s10, (LENGTH L) + 1) in
            let s12 = CASE (APPEND (CONS (h^1 s1) (MAP (λx.(SND x) s5) L))
                                   [h^n s5] )
                           s10 in
            let s13 = FU s12 in
            let s14 = CASE (APPEND (CONS (g^1 s13) (MAP (λx.(FST x) s13) L))
                                   [g^1 s13] )
                           s10 in
            let s15 = MUX (s11, MUX (s8, s6, g^n s13), s6) in
            ((s15, s11), (s10, s14, s15)))
          , ((LENGTH L) + 1, q1, q2) )
        (λt. (input t, reset t, start t))
```

(a) Beschreibung in Gropius-1



(b) Schematische Darstellung

Abbildung 7.13: Implementierungsmuster DFG_IMP_RESET

mit Kardinalität n wird als enum_n bezeichnet. Der Wertebereich dieses Datentyps erstreckt sich von 0 bis zu $n-1$. Die Funktion enum gibt den i-ten Wert dieses Wertebereichs zurück. Die Funktion next ermittelt den Nachfolger eines Wertes. Die Semantik der Funktionen enum und next ist dabei wie folgt definiert:

$$\begin{aligned} \mathsf{enum}\ n\ m &= \mathsf{MUX}(m < n, m, 0) \\ \mathsf{next}\ n\ x &= \mathsf{MUX}(\mathsf{SUC}\ x\ <\ n, \mathsf{SUC}\ x, 0) \end{aligned}$$

Die Implementierung lässt sich in zwei Teile unterteilen. Im oberen Teil erkennt man einen Modulo-m-Zähler, der als Kontroller fungiert. Er steuert im unteren Teil der Implementierung über die CASE-Bausteine den Datenfluss.

Das Implementierungstheorem (7.6) schließlich drückt aus, dass das Implementierungsmuster DFG_IMP_RESET die Spezifikation aus einem DFG-Term (der in der Form (7.5) vorliegt) und dem Schnittstellenverhaltensmuster DFG_IFC_RESET erfüllt.

$$\begin{array}{l} \vdash\ \forall\, \mathsf{FU}\ g^n\ h^n\ L\ g^1\ h^1. \\ \quad \mathsf{DFG\_IMP\_RESET} \\ \quad\quad (\mathit{input}, \mathit{reset}, \mathit{start}, \mathit{output}, \mathit{ready}, \\ \quad\quad \mathsf{FU}, g^n, h^n, L, g^1, h^1) \\ \quad \Rightarrow \\ \quad \mathsf{DFG\_IFC\_RESET} \\ \quad\quad ((g^n \circ \mathsf{FU} \circ h^n) \circ (\mathsf{list\_o}\ (\mathsf{MAP}\ (\lambda(p,q).p \circ \mathsf{FU} \circ q)\ L)) \circ (g^1 \circ \mathsf{FU} \circ h^1), \\ \quad\quad (\mathsf{LENGTH}\ L) + 2) \\ \quad\quad (\mathit{input}, \mathit{reset}, \mathit{start}, \mathit{output}, \mathit{ready}) \end{array} \tag{7.6}$$

Dieses Theorem wird während der Schnittstellensynthese instantiiert. Somit erhält man die gewünschte korrekte Struktur auf RT-Ebene.

7.3 Synthese von algorithmischen P-Schaltungsbeschreibungen

Viele in der Literatur vorgestellte Syntheseverfahren beschränken sich auf reine Datenflussgraphen, die den DFG-Termen entsprechen. Erst zögerlich setzen sich auch Verfahren mit gemischtem Daten- und Kontrollfluss (P-Terme) durch. Dabei wird aber oft versucht, die Synthese auf bestehende Verfahren für Datenflüsse zurückzuführen. Ausgehend von einer Beschreibung, die in Form eines Kontroll-/Datenflussgraphen (CDFG) vorliegt, werden zunächst die Zyklen aufgeschnitten, um dann entlang des Graphen mehrere Kontrollschritte einzuführen. Auf diese Weise wird der Graph in kleinere, azyklische Programmstücke zerlegt, die jeweils einem Takt entsprechen. Im System Amical [Amic96] etwa wird dabei von "macrocycles" gesprochen. Es ist aber zu beachten, dass die Anzahl der zu betrachtenten Teilgraphen exponentiell mit der Anzahl der Kontrollstrukturen anwächst. Auf den einzelnen Teilgraphen werden dann die zeitliche Einordnung (Einführung sogenannter "microcycles" im System Amical), die Bereitstellung und die Zuordnung durchgeführt.

Anschließend werden die so bearbeiteten Teilgraphen wieder zusammengeführt, wodurch ein getrennter Datenpfad und eine symbolische Zustandsübergangstabelle entsteht. Danach wird durch eine Steuerwerkssynthese die Kontrolleinheit erzeugt.

7.3.1 Vorgehensweise

Der hier vorgestellte Ansatz zur Synthese von algorithmischen P-Schaltungsbeschreibungen unterscheidet sich von dem vorstehend beschriebenen Vorgehen grundsätzlich. Die Vorgehensweise in den konventionellen Syntheseansätzen wie z.B. Amical führt zu wesentlichen Beschränkungen. Beispiele für Transformationen, die über solche Verfahren hinausgehen, sind das Aufrollen von Schleifen oder das Heraus-/Hereinziehen von Schleifenrümpfen. In diesem Ansatz wird die Synthese nicht auf die von reinen Datenflussgraphen zurückgeführt, sondern die Schaltungsbeschreibung bleibt immer geschlossen und die RT-Ebenenstruktur wird mit Hilfe von Programmtransformationen erreicht, die auf vorbewiesenen Theoremen basieren.

Die High-Level Synthese wird in vier Schritten durchgeführt. Im ersten Schritt wird das gegebene Programm, das die funktionale Seite der algorithmischen P-Schaltungsbeschreibung repräsentiert, in ein äquivalentes, aber optimiertes Programm transformiert. Anschließend wird dieses Programm in ein äquivalentes überführt, das eine einzige Schleife besitzt. In diesen beiden Schritten wird implizit eine zeitliche Einordnung der Operationen und die Bereitstellung und Zuordnung von Registern durchgeführt. Im dritten Schritt folgt die Schnittstellensynthese, indem eine RT-Ebenenstruktur durch Anwenden eines Implementierungstheorems erzeugt wird. Im letzten Schritt schließlich werden innerhalb der kombinatorischen Komponente der RT-Schaltung die Funktionseinheiten bereitgestellt und zugeordnet. Dieser Schritt funktioniert in ähnlicher Weise wie der in Abschnitt 7.2.1 vorgestellte Vorgang zur Bereitstellung und Zuordnung von Funktionseinheiten für DFG-Terme und soll daher im Folgenden nicht näher vorgestellt werden. Die Tatsache, dass die Bereitstellung und Zuordnung von Funktionseinheiten erst im letzten Schritt durchgeführt werden, stellt keine Einschränkung dar. Vielmehr wird in diesem Schritt lediglich die logische Transformation durchgeführt. Die Information, welche und wie viele Funktionseinheiten zur Verfügung stehen, geht bereits in den ersten Syntheseschritt mit ein.

Im Folgenden werden zunächst die beiden ersten Schritte, die Transformation des Programms in eine Form mit nur einer Schleife, vorgestellt.

7.3.2 Transformation in Ein-Schleifen-Form (ESF)

Die wesentliche Idee bei der Synthese von P-Termen ist die Transformation eines gegebenen Programms in ein äquivalentes Programm, das in ESF vorliegt. Die ESF ist ein spezielles Muster für Programme und hat folgendes Aussehen:

$$\mathsf{PROGRAM}\ out\_init\ (\mathsf{LOCVAR}\ var\_init\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))) \tag{7.7}$$

Die Ausdrücke *out_init* bzw. *var_init* bezeichnen beliebige Konstanten. Sie repräsentieren den Initialwert der Ausgabevariablen bzw. den Initialwert von lokalen Variablen. Der Ausdruck c steht für eine beliebige Bedingung (DFG-Term mit boolescher Ausgabe) und a repräsentiert einen beliebigen Elementarblock (DFG-Term mit gleichem Ein- und Ausgabetyp).

Die Motivation, ein Programm in eine ESF zu transformieren, liegt darin begründet, dass Hardware-Implementierungen sich prinzipiell wie eine einzige while-Schleife verhalten, die immer denselben Block ausführt. Der Übergang von einer algorithmischen Beschreibung mit einer einzigen while-Schleife in eine RT-Ebenenstruktur ist somit naheliegend.

Jedes Programm kann in eine äquivalente ESF überführt werden. Dies folgt aus dem Satz von KLEENE, der dabei von einer Normalform μ-rekursiver Programme spricht [Goos97c]. Es ist dabei aber zu beachten, dass es für ein gegebenes Programm nicht eine einzige eindeutige ESF gibt. Vielmehr lässt sich ein Programm in eine unendliche Anzahl äquivalenter ESF's transformieren.

Man könnte einwenden, dass durch eine Transformation eines Programms in eine ESF der Aufwand für die Abarbeitung des Programms zunimmt, da anstatt kleiner lokaler Programmteile immer der gesamte Schleifenrumpf durchlaufen werden muss. Dazu muss zweierlei angemerkt werden: Zum einen liegt diese Transformation in der Natur einer Hardware-Implementierung, da auf RT-Ebene die Schaltung ohnehin durch einen Automaten beschrieben wird. Der Unterschied besteht darin, dass in dieser Arbeit die Transformation bereits auf der algorithmischen Ebene und nicht erst beim Übergang auf die RT-Ebene stattfindet. Zum anderen nimmt der Aufwand nur insofern zu, als im Rumpf der ESF zusätzliche Multiplexer durchlaufen werden müssen, die die auszuführenden Operationen bestimmen. Der Schleifenrumpf eines Programms in ESF verhält sich wie eine Case-Anweisung, wie sie in Abbildung 7.14 dargestellt ist. Abhängig vom Kontrollzustand werden gewisse Operationen

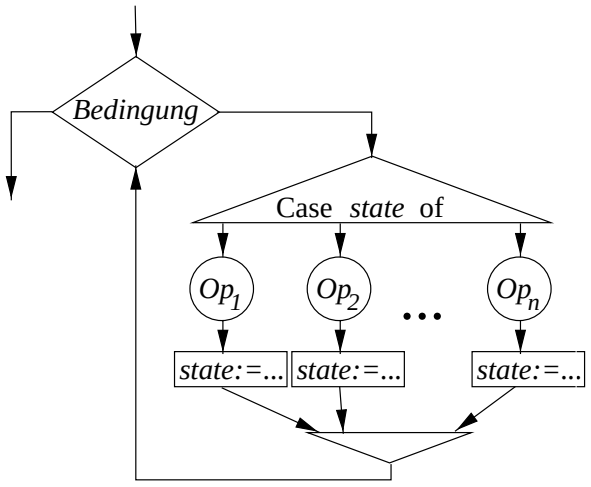
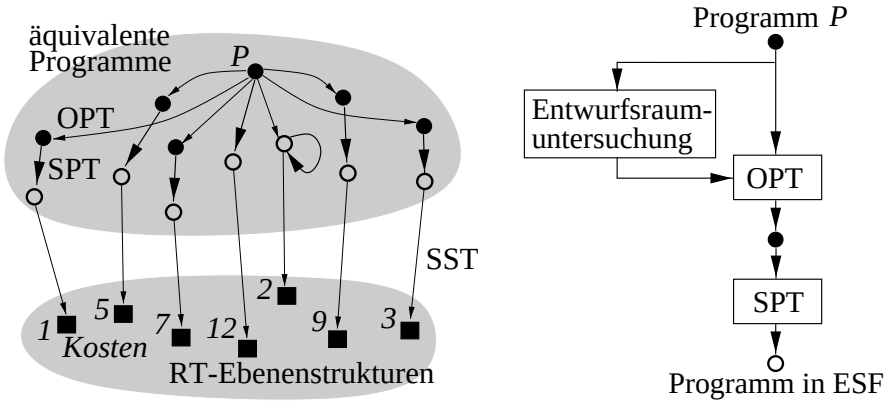


Abbildung 7.14: Wirkungsweise der ESF

während eines Durchlaufs durch den Schleifenrumpf ausgeführt und anschließend wird der neue Kontrollzustand ermittelt. Ein Schleifenrumpfdurchlauf entspricht somit einem Kontrollschritt. Wenn das Programm in ESF durch die Schnittstellensynthese in eine RT-Ebenenstruktur überführt worden ist (siehe Abschnitt 7.3.5), entspricht jeder Kontrollschritt einem Takt. Die Kosten der Hardware-Implementierung hinsichtlich Ausführungsgeschwindigkeit und Flächenbedarf richten sich also danach, welche Operationen innerhalb welches Kontrollschrittes ausgeführt werden. Jedes Programm in ESF entspricht somit einer RT-Implementierung mit bestimmten Hardwarekosten. Es ist also die Aufgabe der High-Level Synthese, ausgehend von einem gegebenen Programm dies in ein äquivalentes Programm in ESF zu transformieren, bei dem die Operationen in der Weise im Schleifenrumpf angeordnet sind, dass das ESF-Programm einer für eine gegebene Kostenfunktion optimalen RT-Implementierung entspricht.

Die Transformation eines gegebenen Programms läuft in zwei Stufen ab, wie es in Abbildung 7.15 dargestellt ist. Zunächst wird das gegebene Programm durch



(a) Ermittelung der Kosten

(b) Einbettung nichtformaler Methoden

Abbildung 7.15: Zweistufige Transformation in ESF

eine Transformation, die Optimierungs-Programm-Transformation (OPT) genannt wird, in ein äquivalentes, aber optimiertes Programm umgewandelt. Anschließend wird durch eine sogenannte Standard-Programm-Transformation (SPT) dieses Programm in ein äquivalentes Programm in ESF transformiert. In Abbildung 7.15(a) ist außerdem noch die anschließende Schnittstellensynthese-Transformation (SST) skizziert. Während SPT und SST eindeutig sind, kann die OPT in beliebiger Weise durchgeführt werden. Die Eindeutigkeit von SPT und SST führt dazu, dass ausgehend von einem optimierten Programm die anschließend entstehenden Hardware-Kosten abgeschätzt werden können (siehe Abbildung 7.15(a)). Während der OPT muss also in geeigneter Weise der Entwurfsraum untersucht werden. Dies geschieht im Rahmen der bereits in Abschnitt 7.1 vorgestellten Aufteilung in Entwurfsrau-

muntersuchung und Transformation. Bei der Entwurfsraumuntersuchung können bekannte Methoden aus dem Compiler-Bau eingesetzt werden, welche die entstehenden Hardware-Kosten bestimmen, während die eigentliche Transformation der OPT im Theorembeweiser abläuft und damit die Korrektheit garantiert. Im Folgenden soll nun zuerst auf die SPT näher eingegangen werden, bevor die OPT genauer betrachtet wird.

7.3.3 Standard-Programm-Transformation (SPT)

Im Theorembeweiser HOL wurde für die SPT ein Satz von Transformationstheoremen bewiesen. Diese Transformationstheoreme sind ausreichend, um *alle* Programme, die in Gropius beschrieben werden können, in ein äquivalentes Programm in ESF zu transformieren. Obwohl es unendlich viele ESF's für ein gegebenes Programm gibt, wird durch die SPT ein Programm in eine eindeutige ESF überführt. Dabei werden die Operationen in eindeutiger Weise im Schleifenrumpf der ESF angeordnet, was einer eindeutigen zeitlichen Einordnung entspricht. Damit verbunden ist eine eindeutige Zuweisung von Hardware-Kosten, wie bereits oben erwähnt.

Die Theoreme stellen allgemeine Transformationsschablonen dar. Während der Synthese wird eine Termersetzung mit ihnen durchgeführt, indem sie mit einem konkreten Term instantiiert werden. Die Termersetzung mit diesem Satz an Theoremen terminiert immer und läuft automatisch ab.

Die Theoreme der SPT teilen sich in zwei Gruppen. Mit den Theoremen der ersten Gruppe werden die Kontrollstrukturen eines Programms schrittweise entfernt und stattdessen boolesche Hilfsvariablen eingeführt, die die entsprechende Kontrollinformation enthalten. Mit den Theoremen der zweiten Gruppe werden diese lokalen Variablen herausgezogen und zusammengefasst, so dass schließlich in der ESF alle lokalen Variablen vor der resultierenden while-Schleife eingeführt werden (siehe (7.7) auf Seite 129). Es ist aber nicht so, dass zunächst nur Theoreme der ersten und anschließend solche der zweiten Gruppe angewandt werden. Vielmehr wird je nach momentanem Aussehen des Programms ein Theorem der ersten oder der zweiten Gruppe für die Termersetzung verwendet. Zur Begriffsvereinfachung soll im Folgenden ein Block der Form (PARTIALIZE a) als *Elementarrumpf* bezeichnet werden. Des Weiteren soll eine while-Schleife, deren Rumpf aus einem Elementarrumpf besteht, *Elementarschleife* genannt werden. Die Transformation eines gegebenen Programms läuft "von innen nach außen" ab, d.h. es werden die Teilterme ersetzt, die sich aus Elementarrümpfen und/oder Elementarschleifen zusammensetzen.

Es werden nun zunächst die Theoreme der ersten Gruppe vorgestellt. Die Theoreme (7.8) bis (7.10) spielen dabei eine Sonderrolle. Mit (7.8) werden zwei Elementarrümpfe zusammengefasst, die hintereinander ausgeführt werden, während mit (7.9) zwei Elementarrümpfe, die einander ausschließen, zusammengefasst werden. Es werden dabei im Gegensatz zu den anderen Theoremen keine Hilfsvariablen eingeführt, da sich am Kontrollzustand nichts ändert. Theorem (7.10) dient dazu, eine lokale Variable in einen Elementarrumpf hineinzuziehen und damit nach außen hin verschwinden zu lassen. Anmerkung: Bei allen Theoremen erscheint auf der linken

Seite jeweils als DFG-Term nur eine Funktion f, aber keine λ-Abstraktion. Nach Abschnitt 2.1.2 gilt aber aufgrund der η-Konversion ($f = \lambda x.f\ x$).

$$\begin{aligned}\vdash\ &(\mathsf{PARTIALIZE}\ a)\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ b)\ = \\ &\quad \mathsf{PARTIALIZE}\ (\lambda x.\ \mathsf{let}\ x1\ =\ (a\ x)\ \mathsf{in}\ (b\ x1)\ )\end{aligned} \tag{7.8}$$

$$\begin{aligned}\vdash\ &\mathsf{IFTE}\ c\ (\mathsf{PARTIALIZE}\ a)\ (\mathsf{PARTIALIZE}\ b)\ = \\ &\quad \mathsf{PARTIALIZE}\ (\lambda x.\mathsf{MUX}(c\ x, a\ x, b\ x))\end{aligned} \tag{7.9}$$

$$\begin{aligned}\vdash\ &\mathsf{LOCVAR}\ i\ (\mathsf{PARTIALIZE}\ a)\ = \\ &\quad \mathsf{PARTIALIZE}\ (\lambda x.\mathsf{let}\ (x', y)\ =\ a(x, i)\ \mathsf{in}\ x')\end{aligned} \tag{7.10}$$

Die nächsten beiden Theoreme beschreiben, wie eine Sequenz aus einem Elementarrumpf und einer Elementarschleife, bzw. umgekehrt, zu einer einzigen Elementarschleife umgewandelt werden kann.

$$\begin{aligned}\vdash\ &(\mathsf{PARTIALIZE}\ a)\ \mathsf{THEN}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ b))\ = \\ &\quad \mathsf{LOCVAR\ F} \\ &\quad\ \ \mathsf{WHILE}\ (\lambda(x, h).\ c\ x\ \vee\ \neg h) \\ &\quad\ \ \ \ \mathsf{PARTIALIZE}\ (\lambda(x, h).\ (\mathsf{MUX}\ (h, b\ x, a\ x), \mathsf{T}))\end{aligned} \tag{7.11}$$

$$\begin{aligned}\vdash\ &(\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ b)\ = \\ &\quad \mathsf{LOCVAR\ F} \\ &\quad\ \ \mathsf{WHILE}\ (\lambda(x, h).\ \neg h) \\ &\quad\ \ \ \ \mathsf{PARTIALIZE}\ (\lambda(x, h).\ \mathsf{MUX}\ (c\ x, (a\ x, \mathsf{F}), (b\ x, \mathsf{T})))\end{aligned} \tag{7.12}$$

Zur besseren Veranschaulichung ist in Abbildung 7.16 das Theorem (7.11) schematisch dargestellt. Die LOCVAR-Umgebung auf der rechten Seite des Theorems ist durch die beiden DFG-Terme außerhalb der Schleife abgebildet. In den beiden

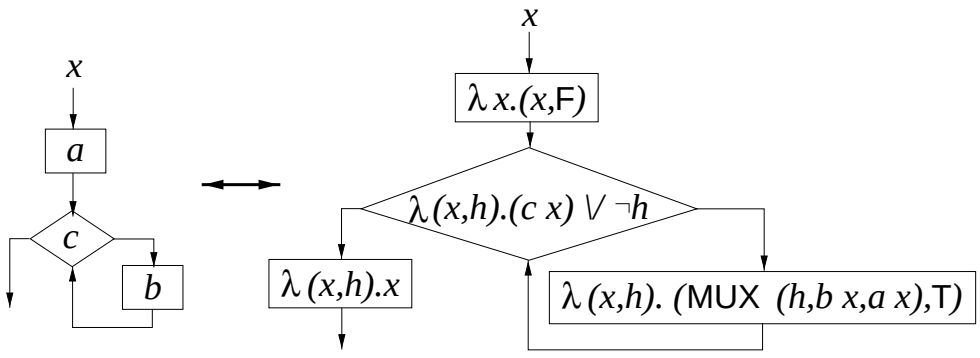


Abbildung 7.16: Bildhafte Darstellung von Theorem (7.11)

Theoremen (7.11) und (7.12) wird jeweils eine lokale boolesche Hilfsvariable mit Initialwert F eingeführt. In (7.11) sorgt die Kontrollvariable dafür, dass die Schleife mindestens einmal ausgeführt wird. Dies wird durch die Bedingung erreicht, die

wahr wird, wenn die Hilfsvariable den Wert F hat. Im ersten Schleifendurchlauf sorgt der Multiplexer dafür, dass der Elementarblock a ausgeführt wird. Gleichzeitig ändert sich der Wert der Kontrollvariablen zu T; diesen Wert behält sie auch in den folgenden Schleifenrumpfdurchläufen. Nach dem ersten Schleifendurchlauf spielt die Hilfsvariable bei der Auswertung der Schleifenbedingung keine Rolle mehr und die Schleife verhält sich wie die Schleife auf der linken Seite des Theorems. Auch in Theorem (7.12) nimmt die Kontrollvariable Einfluss auf die Schleifenbedingung, aber in anderer Form. Zu Beginn ist die Bedingung erfüllt. Im Schleifenrumpf wird infolge des Ergebnisses der alten Schleifenbedingung c der Elementarblock a bzw. der Elementarblock b ausgeführt. Ist c erfüllt, verhält sich die Schleife demnach wie die alte Schleife, und die Hilfsvariable behält den Wert F, wodurch die Schleife nicht verlassen wird. Erst wenn c nicht mehr erfüllt ist, wird b ausgeführt, und die Hilfsvariable ändert ihren Wert zu T. Dies hat zur Folge, dass die Schleife anschließend verlassen wird.

Die zwei folgenden Theoreme widmen sich der Sequenz zweier Elementarschleifen, bzw. der Verschachtelung zweier Schleifen, deren innere eine Elementarschleife ist.

$$\begin{aligned}
&\vdash (\mathsf{WHILE}\ c_1\ (\mathsf{PARTIALIZE}\ a))\ \mathsf{THEN}\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ b)) \ = \\
&\quad \mathsf{LOCVAR\ F} \\
&\qquad \mathsf{WHILE}\ (\lambda(x,h).\ c_2\ x\ \vee\ \neg h) \\
&\qquad\quad \mathsf{PARTIALIZE}\ (\lambda(x,h). \\
&\qquad\qquad\qquad \mathsf{MUX}\ (h, (b\ x, \mathsf{T}), \mathsf{MUX}(c_1\ x, (a\ x, \mathsf{F}), (x, \mathsf{T}))))
\end{aligned} \tag{7.13}$$

$$\begin{aligned}
&\vdash \mathsf{WHILE}\ c_1\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ a)) \ = \\
&\quad \mathsf{LOCVAR\ F} \\
&\qquad \mathsf{WHILE}\ (\lambda(x,h).\ c_1\ x\ \vee\ h) \\
&\qquad\quad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ \mathsf{MUX}\ (c_2\ x, (a\ x, \mathsf{T}), (x, \mathsf{F})))
\end{aligned} \tag{7.14}$$

In Theorem (7.13) wird analog zu (7.11) und (7.12) die Kontrollstruktur THEN beseitigt, indem eine lokale Kontrollvariable mit Initialwert F eingeführt wird. Aufgrund des Ausdrucks $(\neg h)$ in der neuen Bedingung wird damit die neue Elementarschleife mindestens einmal ausgeführt. Der erste Multiplexer im Schleifenrumpf wählt im ersten Durchlauf das Ergebnis des zweiten Multiplexers aus. Dieser übernimmt der Bedingung c_1 zufolge entweder das Ergebnis des Elementarblocks a, und dabei wird die Hilfsvariable auf F gelassen, oder die Hilfsvariable wird auf T gesetzt. Die Schleife verhält sich in diesem Falle, solange die Hilfsvariable den Wert F hat, wie die erste Schleife auf der linken Seite des Theorems. Ist die Bedingung c_1 nicht mehr erfüllt, wird also die erste Schleife verlassen, hängt die Bedingung der neuen Schleife nicht mehr von der Hilfsvariablen ab und die Schleife verhält sich nun wie die zweite Schleife auf der linken Seite. Auch in Theorem (7.14) wird eine Kontrollvariable mit Anfangswert F eingesetzt. Dadurch wird die Kontrollstruktur WHILE beseitigt. Die Bedingung der neuen Schleife hängt beim ersten Durchlauf nicht von der Kontrollvariablen ab, sondern nur von der Bedingung c_1 der alten äußeren Schleife. Ist diese

erfüllt, wird im Rumpf abhängig vom Ergebnis der Bedingung c_2 der alten inneren Schleife der Elementarblock a ausgeführt und die lokale Variable auf T gesetzt, oder die lokale Variable bleibt auf F. Ist c_2 erfüllt und damit der Wert der Hilfsvariablen T, bleibt die Bedingung der neuen Schleife wahr, bis c_2 nicht mehr gilt. In diesem Falle wechselt die Kontrollvariable wieder auf F und ein neuer Durchlauf hängt wieder vom Resultat der Bedingung c_1 ab.

Die Theoreme (7.15), (7.16) und (7.17) beschreiben, wie eine bedingte Verzweigung mit einem Elementarrumpf in einem Zweig und mit einer Elementarschleife im anderen Zweig, bzw. je einer Elementarschleife in den Zweigen, umgewandelt werden kann in eine Sequenz aus Elementarrumpf und Elementarschleife. Diese Sequenz kann anschließend mit Theorem (7.11) in eine einzige Elementarschleife umgewandelt werden.

$$
\begin{aligned}
&\vdash \mathsf{IFTE}\ c_1\ (\mathsf{PARTIALIZE}\ a)\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ b)) \ = \\
&\quad \mathsf{LOCVAR\ F} \\
&\qquad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ \mathsf{MUX}\ (c_1\ x, (a\ x, \mathsf{F}), (x, \mathsf{T}))) \\
&\qquad \mathsf{THEN} \\
&\qquad \mathsf{WHILE}\ (\lambda(x,h).\ c_2\ x\ \wedge\ h)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (b\ x, \mathsf{T})))
\end{aligned}
\tag{7.15}
$$

$$
\begin{aligned}
&\vdash \mathsf{IFTE}\ c_1\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ a))\ (\mathsf{PARTIALIZE}\ b) \ = \\
&\quad \mathsf{LOCVAR\ F} \\
&\qquad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ \mathsf{MUX}\ (c_1\ x, (x, \mathsf{T}), (b\ x, \mathsf{F}))) \\
&\qquad \mathsf{THEN} \\
&\qquad \mathsf{WHILE}\ (\lambda(x,h).\ c_2\ x\ \wedge\ h)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (a\ x, \mathsf{T})))
\end{aligned}
\tag{7.16}
$$

$$
\begin{aligned}
&\vdash \mathsf{IFTE}\ c_1\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ a))\ (\mathsf{WHILE}\ c_3\ (\mathsf{PARTIALIZE}\ b)) \ = \\
&\quad \mathsf{LOCVAR\ F} \\
&\qquad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ (x, c_1\ x)) \\
&\qquad \mathsf{THEN} \\
&\qquad \mathsf{WHILE}\ (\lambda(x,h).\ (c_2\ x\ \wedge\ h)\ \vee\ (c_3\ x\ \wedge\ \neg h)) \\
&\qquad\quad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ (\mathsf{MUX}(h, a\ x, b\ x), h))
\end{aligned}
\tag{7.17}
$$

Zur Beseitigung der Kontrollstruktur IFTE wird wieder jeweils eine Kontrollvariable mit Anfangswert F eingeführt. In (7.15) entsteht eine Sequenz, wobei zunächst in einem Elementarblock geprüft wird, ob die Verzweigungsbedingung c_1 erfüllt ist. Ist dies der Fall, wird der Elementarblock a aus dem ersten Zweig ausgeführt, und die Kontrollvariable bleibt unverändert. Dies führt dazu, dass die anschließende Elementarschleife wegen des Ausdrucks $(c_2\ x\ \wedge\ h)$ in ihrer Bedingung nicht ausgeführt wird. Ist dagegen c_1 nicht erfüllt, ändert sich im Elementarblock nur der Wert der Kontrollvariablen auf T. Dadurch wird gewährleistet, dass sich die anschließende Elementarschleife wie die Elementarschleife im zweiten Zweig der IFTE-Umgebung verhält, da sich ihre Bedingung gerade zu $(c_2\ x)$ reduziert. Theorem (7.16) beschreibt die analoge Situation, bei der die Zweige in der IFTE-Umgebung vertauscht sind. Theorem (7.17) ist etwas komplizierter. Es entsteht eine Sequenz, bei der

zunächst die Hilfsvariable auf den Wert des Ergebnisses der Verzweigungsbedingung c_1 gesetzt wird. Die Bedingung der anschließenden Elementarschleife setzt sich aus zwei Teilen zusammen, die eine Disjunktion bilden. Einer dieser Teile reduziert sich aufgrund des neuen Wertes der Kontrollvariablen zu F. Ist c_1 erfüllt, ist es der zweite Teil, im anderen Fall der erste Teil. Im Rumpf der neuen Elementarschleife wird abhängig vom Wert der Kontrollvariablen und damit vom Ergebnis von c_1 entweder der Rumpf a der Schleife im ersten Zweig von IFTE ausgeführt oder der Rumpf b der Schleife im zweiten Zweig.

Die Theoreme (7.18) bis (7.23) zeigen, wie die Kontrollstrukturen LEFTVAR bzw. RIGHTVAR beseitigt werden können. Da die Transformation eines Programms in die ESF im Syntaxbaum von unten nach oben abläuft, müssen jeweils nur die folgenden drei Fälle betrachtet werden. LEFTVAR bzw. RIGHTVAR wird angewandt auf einen Elementarrumpf, eine Elementarschleife oder eine lokale Variable mit Elementarschleife. Alle Blöcke, auf die LEFTVAR bzw. RIGHTVAR angewandt werden können, können durch die bereits vorgestellten und die noch vorzustellenden Theoreme in eine dieser drei Formen überführt werden.

$$\vdash \mathsf{LEFTVAR}\ (\mathsf{PARTIALIZE}\ a)\ =\ \mathsf{PARTIALIZE}\ (\lambda(x,h).\ (a\ x,h)) \tag{7.18}$$

$$\vdash \mathsf{LEFTVAR}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))\ = \\ \quad \mathsf{WHILE}\ (\lambda(x,h).\ c\ x)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).\ (a\ x,h))) \tag{7.19}$$

$$\begin{aligned}\vdash\ & \mathsf{LEFTVAR}\ (\mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)))\ = \\ & \quad \mathsf{LOCVAR}\ i \\ & \quad\ \mathsf{WHILE}\ (\lambda((x,h_1),h_2).\ c\ (x,h_2)) \\ & \quad\quad \mathsf{PARTIALIZE}\ (\lambda((x,h_1),h_2). \\ & \quad\quad\quad\quad \mathsf{let}\ (r,s)\ =\ a\ (x,h_2)\ \mathsf{in}\ ((r,h_1),s))\end{aligned} \tag{7.20}$$

$$\vdash \mathsf{RIGHTVAR}\ (\mathsf{PARTIALIZE}\ a)\ =\ \mathsf{PARTIALIZE}\ (\lambda(h,x).\ (h,a\ x)) \tag{7.21}$$

$$\vdash \mathsf{RIGHTVAR}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))\ = \\ \quad \mathsf{WHILE}\ (\lambda(h,x).\ c\ x)\ (\mathsf{PARTIALIZE}\ (\lambda(h,x).\ (h,a\ x))) \tag{7.22}$$

$$\begin{aligned}\vdash\ & \mathsf{RIGHTVAR}\ (\mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)))\ = \\ & \quad \mathsf{LOCVAR}\ i \\ & \quad\ \mathsf{WHILE}\ (\lambda((h_1,x),h_2).\ c\ (x,h_2)) \\ & \quad\quad \mathsf{PARTIALIZE}\ (\lambda((h_1,x),h_2). \\ & \quad\quad\quad\quad \mathsf{let}\ (r,s)\ =\ a\ (x,h_2)\ \mathsf{in}\ ((h_1,r),s))\end{aligned} \tag{7.23}$$

Die Kombination von LEFTVAR mit einem Elementarrumpf führt zu einem etwas anderen Elementarrumpf, bei dem sich der Typ geändert hat, da nun eine zusätzliche Eingangsvariable berücksichtigt werden muss, die durch LEFTVAR ausgeblendet war. Analog wird die Kombination von LEFTVAR mit einer Elementarschleife umgeformt. Kommt zusätzlich eine lokale Variable ins Spiel, wird es etwas komplizierter. In

Theorem (7.20) setzt sich der Eingang der Schleifenbedingung und des Schleifenrumpfes auf der rechten Seite nunmehr aus drei Anteilen zusammen. Die Variable h_2 entspricht dabei der lokalen Variablen, deren Initialwert i ist. Da die LOCVAR-Umgebung nun außen steht, entspricht der lokalen Variablen nach Definition (5.18) auf Seite 62 der zweite Teil der Variablenstruktur $((x, h_1), h_2)$ der inneren DFG-Terme. Die Variable h_1 entspricht der Variablen, die zuvor durch LEFTVAR ausgeblendet war. Die DFG-Terme c und a müssen also auf das Paar (x, h_2) angewandt werden. Die Theoreme, mit deren Hilfe RIGHTVAR beseitigt wird, sind ähnlich aufgebaut, sodass sie nicht näher erläutert werden müssen.

Damit ist die erste Gruppe der SPT-Theoreme komplett. Es werden nun die Theoreme der zweiten Gruppe vorgestellt, die zum Herausziehen der lokalen Variablen verwendet werden. Mit Hilfe des Theorems (7.24) können zwei "verschachtelte" lokale Variablen zu einer einzigen lokalen Variablen zusammengefasst werden. Es wird dabei angenommen, dass der Block, auf den die innere lokale Variable angewandt wird, durch die anderen Theoreme bereits in eine Elementarschleife transformiert wurde. Der Initialwert der neuen lokalen Variablen ergibt sich aus dem kartesischen Produkt der beiden ursprünglichen Initialwerte. Es muss dabei beachtet werden, dass sich damit der Typ der inneren Elementarschleife von $(((\alpha \times \beta) \times \gamma) \to ((\alpha \times \beta) \times \gamma)\mathsf{partial})$ verändert zu $((\alpha \times \beta \times \gamma) \to (\alpha \times \beta \times \gamma)\mathsf{partial})$.

$$
\begin{aligned}
\vdash\ & \mathsf{LOCVAR}\ i_1\ (\mathsf{LOCVAR}\ i_2\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))) \ = \\
& \quad \mathsf{LOCVAR}\ (i_1, i_2) \\
& \quad\ \ \mathsf{WHILE}\ (\lambda(x, y, z).\ c\ ((x, y), z)) \\
& \quad\quad \mathsf{PARTIALIZE}\ (\lambda(x, y, z). \\
& \qquad\qquad\qquad \mathsf{let}\ ((x', y'), z') \ = \ a\ ((x, y), z)\ \mathsf{in}\ (x', y', z')\ )
\end{aligned}
\tag{7.24}
$$

Das Theorem (7.25) dient dazu, eine lokale Variable aus einer Schleife herauszuziehen. Es wird dabei angenommen, dass der Teil des Schleifenrumpfes, auf den die lokale Variable angewandt wird, durch die obigen Theoreme bereits in eine Elementarschleife umgewandelt wurde.

$$
\begin{aligned}
\vdash\ & \mathsf{WHILE}\ c_1\ (\mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ c_2\ (\mathsf{PARTIALIZE}\ a))) \ = \\
& \quad \mathsf{LOCVAR}\ i \\
& \quad\ \ \mathsf{LOCVAR}\ \mathsf{F} \\
& \quad\quad \mathsf{WHILE}\ (\lambda((x, h_1), h_2).\ c_1\ x\ \vee\ h_2) \\
& \quad\quad\quad \mathsf{PARTIALIZE}\ (\lambda((x, h_1), h_2). \\
& \qquad\qquad\qquad\qquad \mathsf{MUX}\ (c_2\ (x, h_1), (a\ (x, h_1), \mathsf{T}), ((x, i), \mathsf{F})))
\end{aligned}
\tag{7.25}
$$

Beim Herausziehen der lokalen Variablen mit Initialwert i muss eine weitere lokale Variable mit Initialwert F eingeführt werden. Diese lokale Variable, die in den DFG-Termen mit h_2 bezeichnet wird, gibt an, ob die innere Schleife ausgeführt wird oder nicht. Zunächst hat sie keinen Einfluß auf die neue Schleifenbedingung. Ist die ursprünglich äußere Schleifenbedingung c_1 erfüllt, wird im Schleifenrumpf überprüft,

ob die alte innere Schleifenbedingung c_2 erfüllt ist. Ist dies der Fall, wird der alte innere Schleifenrumpf a ausgeführt und die boolesche Hilfsvariable wechselt ihren Wert auf T, wodurch die neue Schleifenbedingung erfüllt wird. Ist die Bedingung c_2 einmal nicht mehr erfüllt, wird der Wert der booleschen Kontrollvariablen wieder auf F gesetzt und der Wert der ursprünglich vorhandenen lokalen Variablen wird wieder auf ihren Anfangswert i gesetzt. In Abhängigkeit der Bedingung c_1 kann dann ein neuer Schleifendurchlauf beginnen. Da Theorem (7.25) nicht nur eine lokale Variable herauszieht, sondern auch eine boolesche Kontrollvariable einführt, indem eine der while-Schleifen beseitigt wird, ist das Theorem nicht eindeutig einer der beiden Gruppen von SPT-Theoremen zuzuordnen. In der Tat stellt dieses Theorem auch das Komplexeste der SPT-Theoreme dar und der Beweis ist sehr aufwendig.

Die letzten Theoreme dienen dazu, lokale Variablen aus einer Sequenz oder einer bedingten Verzweigung herauszuziehen.

$$\vdash (\mathsf{LOCVAR}\ i\ A)\ \mathsf{THEN}\ B\ = \\ \qquad \mathsf{LOCVAR}\ i\ (A\ \mathsf{THEN}\ (\mathsf{LEFTVAR}\ B)) \tag{7.26}$$

$$\vdash B\ \mathsf{THEN}\ (\mathsf{LOCVAR}\ i\ A)\ = \\ \qquad \mathsf{LOCVAR}\ i\ ((\mathsf{LEFTVAR}\ B)\ \mathsf{THEN}\ A) \tag{7.27}$$

$$\vdash \mathsf{IFTE}\ c\ (\mathsf{LOCVAR}\ i\ A)\ B\ = \\ \qquad \mathsf{LOCVAR}\ i\ (\mathsf{IFTE}\ (\lambda(x,h).\ c\ x)\ A\ (\mathsf{LEFTVAR}\ B)) \tag{7.28}$$

$$\vdash \mathsf{IFTE}\ c\ B\ (\mathsf{LOCVAR}\ i\ A)\ = \\ \qquad \mathsf{LOCVAR}\ i\ (\mathsf{IFTE}\ (\lambda(x,h).\ c\ x)\ (\mathsf{LEFTVAR}\ B)\ A) \tag{7.29}$$

Die Blöcke B in den Theoremen (7.26) bis (7.29), die nicht über die lokale Variable mit Initialwert $(i : \beta)$ verfügen, haben den Typ $(\alpha \to (\alpha)\mathsf{partial})$ und werden durch die Theoreme in eine LEFTVAR-Umgebung eingebettet. Damit werden sie nur auf den linken Teil des Zustands, der den Typ $(\alpha \times \beta)$ hat, angewandt. Der Wert der lokalen Variablen bleibt dadurch unberührt. Diese LEFTVAR-Umgebungen können mit Hilfe der Theoreme (7.18) bis (7.20) beseitigt werden. Der Block B muss dazu aber noch jeweils in eine entsprechende Form transformiert werden.

Optimierung der SPT

Nun sind alle Theoreme vorgestellt, mit deren Hilfe jedes Programm in eine ESF überführt werden kann. Aus Optimierungsgründen hinsichtlich der Anzahl der Kontrollvariablen in der ESF sollen im Folgenden aber noch weitere Theoreme vorgestellt werden.

Es ist zu beachten, dass jeweils die Theoreme (7.26) und (7.27) bzw. (7.28) und (7.29) in der Situation miteinander konkurrieren, in der beide Teile der Sequenz bzw. der Verzweigung über eine lokale Variable verfügen. Je nachdem, welches der beiden Theoreme dann angewandt würde, ergäbe sich eine andere ESF. Daher ist die Termersetzung mit diesen Theoremen nicht konfluent. Man könnte leicht eine konfluente Termersetzung garantieren. Dazu müssten aus jedem der vier oberen

Theoreme zwei Theoreme abgeleitet werden, in denen der Block B entweder ein Elementarrumpf oder eine Elementarschleife ist. Darüber hinaus bräuchte man dann noch ein Theorem mit zwei lokalen Variablen jeweils für die Sequenz und die Verzweigung. Statt der obigen vier Theoreme bräuchte man also deren zehn. Dem soll hier aber nicht nachgegangen werden, und zwar aus folgendem Grund: Die beiden Theoreme, die zwei lokale Variablen in den beiden Teilen der Sequenz bzw. der Verzweigung betrachten, müssten von zwei lokalen Variablen unterschiedlichen Typs ausgehen. Hätten die beiden Variablen einen gleichen Typ, würde dies nur einen Sonderfall darstellen. Genau das ist aber das Problem. Die Theoreme müssten so allgemein sein, dass letztlich beide lokale Variablen herausgezogen würden. Im Falle der Typgleichheit der beiden Variablen ist dies aber nicht nötig. Es würden daher unnötig viele lokale Variablen in der resultierenden ESF auftreten, gerade im Hinblick darauf, dass mit den Theoremen der ersten Gruppe immer boolesche Variablen eingeführt werden. Aus diesem Grund werden die beiden folgenden Theoreme (7.30) und (7.31) hinzugezogen, wenn die zwei lokalen Variablen den gleichen Typ haben:

$$\begin{aligned}
&\vdash (\mathsf{LOCVAR}\ (i_1 : \beta)\ A)\ \mathsf{THEN}\ (\mathsf{LOCVAR}\ (i_2 : \beta)\ B)\ = \\
&\qquad \mathsf{LOCVAR}\ i_1 \\
&\qquad\quad A\ \mathsf{THEN}\ (\mathsf{PARTIALIZE}\ (\lambda(x, y).\ (x, i_2)))\ \mathsf{THEN}\ B
\end{aligned} \tag{7.30}$$

$$\begin{aligned}
&\vdash \mathsf{IFTE}\ c\ (\mathsf{LOCVAR}\ (i_1 : \beta)\ A)\ (\mathsf{LOCVAR}\ (i_2 : \beta)\ B)\ = \\
&\qquad \mathsf{LOCVAR}\ i_1 \\
&\qquad\quad \mathsf{IFTE}\ (\lambda(x, h).\ c\ x)\ A \\
&\qquad\qquad\qquad (\mathsf{PARTIALIZE}\ (\lambda(x, y).\ (x, i_2)))\ \mathsf{THEN}\ B
\end{aligned} \tag{7.31}$$

Theorem (7.30) kann verwendet werden, wenn in beiden Teilen einer Sequenz jeweils eine lokale Variable eingeführt wird und diese denselben Typ haben. Auf diese Weise bleibt nur eine Variable übrig, deren Wert nach Ausführung des Blocks A auf den Initialwert i_2 des zweiten Teils der Sequenz geändert wird. Theorem (7.31) leistet entsprechendes für eine Verzweigung.

Auf eine konfluente Termersetzung wird also verzichtet, um die Anzahl der resultierenden Kontrollvariablen in der ESF möglichst gering zu halten. Wie kommt man nun aber zu einer eindeutigen ESF, wenn die Termersetzung mit den Theoremen nicht konfluent ist? Dies wird erreicht, indem eine bestimmte Reihenfolge der Theoreme bei der Termersetzung vorgegeben wird. Es wird dabei immer zuerst geprüft, ob das Theorem (7.30) bzw. (7.31) angewandt werden kann, bevor eines der Theoreme (7.26) und (7.27) bzw. (7.28) und (7.29) herangezogen wird. Unter dieser Voraussetzung führt die SPT immer zu einem eindeutigen Ergebnis.

Das Ergebnis der SPT kann im Hinblick auf die Anzahl der eingeführten Kontrollvariablen noch weiter verbessert werden. Theorem (7.30) bzw. (7.31) betrachtet nur den Fall, dass beide lokale Variablen denselben Typ haben. Es kann aber auch geschehen, dass in beiden Teilen der Sequenz bzw. der Verzweigung zusammengesetzte Variablen auftreten, deren Subtypen alle gleich sind, deren Gesamttyp sich aber unterscheidet. Dafür könnten die Theoreme (7.30) und (7.31) nicht angewandt

werden, obwohl im Endeffekt nur die lokale Variable mit der höheren Stelligkeit nötig wäre. Man betrachte dazu folgendes Beispiel. Gegeben sei der Term

$$(\mathsf{LOCVAR}\ ((\mathsf{F},\mathsf{F}),\mathsf{F})\ A)\ \mathsf{THEN}\ (\mathsf{LOCVAR}\ (\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F})\ B)$$

mit beliebigen Blöcken A und B. Theorem (7.30) könnte in diesem Fall nicht angewandt werden, da sich die Typen der beiden lokalen Variablen unterscheiden. Die Folge wäre, eines der Theoreme (7.26) oder (7.27) einzusetzen. Dies würde darauf hinauslaufen, dass sich schließlich eine siebenstellige (!) lokale Variable ergeben würde. Es ist aber offensichtlich, dass eine vierstellige ausreicht. Um diesem Problem zu begegnen, werden zu den bisher vorgestellten SPT-Theoremen noch die beiden folgenden hinzugefügt:

$$\begin{aligned}
&\vdash \mathsf{LOCVAR}\ (i_1,i_2)\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)) \ = \\
&\qquad \mathsf{LOCVAR}\ i_1 \\
&\qquad\quad \mathsf{LOCVAR}\ i_2 \\
&\qquad\qquad \mathsf{WHILE}(\lambda((x,y),z).c(x,y,z)) \\
&\qquad\qquad\quad \mathsf{PARTIALIZE}(\lambda((x,y),z).\mathsf{let}\ (x',y',z') \ = \ a(x,y,z)\ \mathsf{in}\ ((x',y'),z'))
\end{aligned} \tag{7.32}$$

$$\begin{aligned}
&\vdash \mathsf{LOCVAR}\ (i_1,i_2)\ (\mathsf{LOCVAR}\ i_3\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))) \ = \\
&\qquad \mathsf{LOCVAR}\ i_1 \\
&\qquad\quad \mathsf{LOCVAR}\ (i_2,i_3) \\
&\qquad\qquad \mathsf{WHILE}(\lambda((w,x),(y,z)).c((w,x,y),z)) \\
&\qquad\qquad\quad \mathsf{PARTIALIZE}(\lambda((w,x),(y,z)). \\
&\qquad\qquad\qquad\qquad \mathsf{let}\ ((w',x',y'),z') \ = \ a((w,x,y),z)\ \mathsf{in}\ ((w',x'),(y',z')))
\end{aligned} \tag{7.33}$$

Mit diesen beiden Theoremen können zusammengesetzte lokale Variablen wieder auseinandergenommen werden. Dabei wird davon ausgegangen, dass der innere Block entweder eine Elementarschleife ist oder eine Elementarschleife mit einer zusätzlichen lokalen Variablen. Mit den restlichen SPT-Theoremen kann jeder Teilterm in eine dieser beiden Formen gebracht werden, außer es ergäbe sich ein Elementarrumpf. Dieser könnte aber zusammen mit der äußeren Variablen mit Hilfe von (7.10) zu einem neuen Elementarrumpf umgeformt werden.

Die Anwendung der Theoreme (7.32) und (7.33) führt dazu, dass schließlich doch durch Anwenden von Theorem (7.30) im obigen Beispiel nur eine vierstellige Variable resultiert. (7.32) und (7.33) stellen die Umkehrung von Theorem (7.24) dar. Aus diesem Grunde besteht die Gefahr, dass die Termersetzung nicht terminiert, wenn die Theoreme in beliebiger Weise angewandt werden. Es muss also eine genaue Festlegung der Reihenfolge getroffen werden, in der die Termersetzung mit den Theoremen stattfindet.

Eine weitere Optimierung dient der Vereinfachung der Ausdrücke in den DFG-Termen, die sich während der SPT ergeben. Daher wird zusätzlich eine Termersetzung mit den in Tabelle 7.1 dargestellten einfachen Theoremen für Multiplexer

und Paare durchgeführt. Die elementaren Vereinfachungstheoreme für die booleschen Funktionen der Konjunktion $\wedge$, der Disjunktion $\vee$ und der Negation $\neg$ sind bereits Bestandteil des Theorembeweisers HOL. Sie können hinzugezogen werden, ohne gesondert bewiesen werden zu müssen.

$\vdash \mathsf{MUX}(\mathsf{T}, a, b) = a$	$\vdash \mathsf{MUX}(\mathsf{F}, a, b) = b$
$\vdash \mathsf{FST}(x, y) = x$	$\vdash \mathsf{SND}(x, y) = y$
$\vdash \mathsf{FST}(\mathsf{MUX}(c, (a1, a2), b1, b2)) = \mathsf{MUX}(c, a1, b1)$	
$\vdash \mathsf{SND}(\mathsf{MUX}(c, (a1, a2), b1, b2)) = \mathsf{MUX}(c, a2, b2)$	
$\vdash \mathsf{MUX}(c, a, \mathsf{T}) = \neg c \vee a$	$\vdash \mathsf{MUX}(c, \mathsf{T}, a) = c \vee a$
$\vdash \mathsf{MUX}(c, a, \mathsf{F}) = c \wedge a$	$\vdash \mathsf{MUX}(c, \mathsf{F}, a) = \neg c \wedge a$
$\vdash \mathsf{MUX}(b1, a, \mathsf{MUX}(b2, a, c)) = \mathsf{MUX}(b1 \vee b2, a, c)$	
$\vdash \mathsf{MUX}(b1, \mathsf{MUX}(b2, a, c), c) = \mathsf{MUX}(b1 \wedge b2, a, c)$	
$\vdash \mathsf{MUX}(a, (b, c), (d, e)) = (\mathsf{MUX}(a, b, d), \mathsf{MUX}(a, c, e))$	
$\vdash \mathsf{MUX}(c, \mathsf{MUX}(c, b, d), a) = \mathsf{MUX}(c, b, a)$	
$\vdash \mathsf{MUX}(c, a, \mathsf{MUX}(c, b, d)) = \mathsf{MUX}(c, a, d)$	
$\vdash \mathsf{MUX}(c, a, a) = a$	

Tabelle 7.1: Einige Vereinfachungstheoreme für DFG-Terme

In Abschnitt 5.1.6 wurden die zwei abgeleiteten Verdrahtungsstrukturen ASSOCVAR und SWAPVAR vorgestellt. Sie werden benötigt, um allgemeine Schablonen einer Schaltungsbeschreibung zur Verfügung zu stellen. Im weiteren Verlauf der Arbeit wird noch näher darauf eingegangen. Durch Termersetzung mit ihrer Definition ließen sich diese zwei Verdrahtungsstrukturen während der SPT beseitigen. Wenn man sich aber ihre Definitionen (5.24) und (5.25) auf Seite 65 genauer ansieht, erkennt man, dass dadurch zu viele Hilfsvariablen eingeführt würden. Es ist besser, für diese zwei wichtigen Strukturen eigene Theoreme zur Verfügung zu stellen. Dies soll hier nur exemplarisch für ASSOCVAR gezeigt werden.

$$\begin{aligned}&\vdash \mathsf{ASSOCVAR}\ (\mathsf{PARTIALIZE}\ a)\ (i_1, i_2)\ = \\ &\qquad \mathsf{PARTIALIZE}\ (\lambda((x, y), z).\ \mathsf{let}\ (x', y', z')\ =\ a\ (x, y, z)\ \mathsf{in}\ ((x', y'), z'))\end{aligned} \tag{7.34}$$

$$\begin{aligned}&\vdash \mathsf{ASSOCVAR}\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a))\ (i_1, i_2)\ = \\ &\qquad \mathsf{WHILE}\ (\lambda((x, y), z).\ c\ (x, y, z)) \\ &\qquad\quad \mathsf{PARTIALIZE}\ (\lambda((x, y), z). \\ &\qquad\qquad\qquad \mathsf{let}\ (x', y', z')\ =\ a\ (x, y, z)\ \mathsf{in}\ ((x', y'), z'))\end{aligned} \tag{7.35}$$

$$\begin{aligned}&\vdash \mathsf{ASSOCVAR}\ (\mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)))\ (i_1, i_2)\ = \\ &\qquad \mathsf{LOCVAR}\ i \\ &\qquad\quad \mathsf{WHILE}\ (\lambda(((x, y), z), h).\ c\ ((x, y, z), h)) \\ &\qquad\qquad \mathsf{PARTIALIZE}\ (\lambda(((x, y), z), h). \\ &\qquad\qquad\qquad\qquad \mathsf{let}\ ((x', y', z'), h')\ =\ a\ ((x, y, z), h)\ \mathsf{in}\ (((x', y'), z'), h'))\end{aligned} \tag{7.36}$$

Es sind – wie bei den Theoremen (7.18) bis (7.20) zur Beseitigung von LEFTVAR– nur

drei Theoreme erforderlich, da alle Blöcke, auf die ASSOCVAR angewandt werden kann, mit diesen und den bereits vorgestellten Theoremen in eine dieser drei Formen überführt werden können.

Ein Beispiel

In Abbildung 7.17 ist das Theorem dargestellt, das ausdrückt, dass das in Abschnitt 5.1.7 vorgestellte Programm fib äquivalent seiner durch die SPT erreichten ESF ist. Es wurden zwei boolesche Kontrollvariablen mit Initialwert F eingeführt. Dabei wurden nacheinander die Theoreme (7.12), (7.27), (7.18), (7.11) und zweimal (7.24) angewandt.

$$
\begin{array}{l}
\vdash\ \mathsf{fib} = \\
\quad \mathsf{PROGRAM}\ 1 \\
\quad\ \mathsf{LOCVAR}\ (1,1,0,0,\mathsf{F},\mathsf{F}) \\
\quad\quad \mathsf{WHILE}(\lambda((n,y1),a1,a2,y2,m,h1,h2).\neg h1 \vee \neg h2) \\
\quad\quad\ \mathsf{PARTIALIZE}(\lambda((n,y1),a1,a2,y2,m,h1,h2). \\
\quad\quad\quad \mathsf{let}\ x1' = \neg(m=0)\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ x1'' = \mathsf{MUX}(c,y1,a1)\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ x2 = \mathsf{MUX}(c,y2,a2)\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ x5 = x1'' * a1 + x2 * a2\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ x8 = x1'' * a2 + x2 * (a1 + a2)\ \mathsf{in} \\
\quad\quad\quad (\ (n, \mathsf{MUX}(h2, \mathsf{MUX}(x1', \mathsf{MUX}(c,x5,y1), \mathsf{MUX}(\mathsf{ODD}\ n, y2, y1)), y1)), \\
\quad\quad\quad\ \ \mathsf{MUX}(h2 \wedge x1', \mathsf{MUX}(c,a1,x5), a1), \\
\quad\quad\quad\ \ \mathsf{MUX}(h2 \wedge x1', \mathsf{MUX}(c,a2,x8), a2), \\
\quad\quad\quad\ \ \mathsf{MUX}(h2 \wedge x1', \mathsf{MUX}(c,x8,y2), y2), \\
\quad\quad\quad\ \ \mathsf{MUX}(h2, \mathsf{MUX}(x1', \mathsf{MUX}(c, m-1, m\ \mathsf{DIV}\ 2), m), (n\ \mathsf{DIV}\ 2)\ +\ 1), \\
\quad\quad\quad\ \ \mathsf{MUX}(h2,\ x1', h1), \mathsf{T}\ )\ )
\end{array}
$$

Abbildung 7.17: Einschleifenform (ESF) des Programms fib

In der ESF gibt die Variablenstruktur der DFG-Terme die benötigten Register an, die für die RT-Implementierung nötig werden. In Abbildung 7.17 besteht die Variablenstruktur $(n, y1, a1, a2, y2, m, h1, h2)$ aus acht Variablen, es wurden also acht Register bereitgestellt. Der Typ der bereitgestellten Variablen ist identisch mit dem Typ der entsprechenden Variablen. Wird ausgehend von einem Programm nur die SPT durchgeführt, ist die Bereitstellung und Zuordnung von Registern trivial. Zusätzlich zu den bereits in den Variablenstrukturen der Anfangsbeschreibung enthaltenen Variablen treten i.Allg. nur boolesche Kontrollvariablen hinzu. Im nächsten Abschnitt wird aber deutlich, dass während der Optimierungs-Programm-Transformation die Bereitstellung und Zuordnung von Registern nicht mehr trivial bleibt.

7.3.4 Optimierungs-Programm-Transformation (OPT)

Durch die SPT werden einem Programm in eindeutiger Weise Kosten zugewiesen. Dies geschieht durch eine bestimmte zeitliche Einordnung der Operationen sowie durch eine eindeutige Bereitstellung und Zuweisung von Registern. Um die durch die SPT festgelegten Kosten hinsichtlich einer gegebenen Kostenfunktion zu minimieren, wird vor der SPT eine Optimierungs-Programm-Transformation (OPT) durchgeführt. Während dieser Optimierung wird das Ausgangsprogramm in ein äquivalentes Programm überführt, das als Startpunkt für die SPT zu einer ESF führt, die einer RT-Implementierung mit geringeren Kosten entspricht.

Die OPT beruht wie die SPT auf vorbewiesenen Transformationstheoremen, die während der Synthese instantiiert werden. Im Gegensatz zur SPT muss aber eine Steuerinformation zur Verfügung gestellt werden, welches Theorem auf welchen Teilterm angewandt werden soll. Wie in Abbildung 7.15(b) dargestellt, können dazu nichtformale Techniken hinzugezogen werden. Es ist aber auch möglich, interaktiv die Theoreme auszuwählen und anzuwenden. Ein interessantes Verfahren, wie Optimierungs-Transformationen automatisch bestimmt werden können, wird in [GeRo98, GeKR99] vorgestellt.

Auf der algorithmischen Ebene liegt die Beschreibung des funktionalen Verhaltens einer Schaltung als reines Software-Programm vor. Es liegt also nahe, für die OPT solche Transformationen zu verwenden, die aus dem Compiler-Bau bekannt sind. Eine Reihe solcher Transformationen findet sich in [AhSU86]. Der Unterschied besteht darin, dass sich für die Auswahl und Parametrisierung der Transformationen die Kostenfunktion ändert, da für eine Hardware-Lösung andere Kosten relevant sind als für eine Software-Lösung. Es werden also für die OPT im Folgenden keine "neuen" Transformationen vorgestellt, sondern es werden bewiesene Theoreme für bekannte Transformationen präsentiert. Man muss sich dabei vor Augen halten, dass aufgrund der Tatsache, dass für die Transformationen Theoreme in HOL bewiesen wurden, die Korrektheit der Transformationen während der Synthese nachweislich garantiert wird. Dies ist im Compiler-Bau, in dem die Transformationen sonst verwendet werden, zumeist nicht der Fall.

Schleifenaufrollen

Die erste Transformation, die vorgestellt werden soll, ist das "Schleifenaufrollen", im Englischen loop-unrolling. Eine while-Schleife wird transformiert in eine äquivalente, n-fach aufgerollte Schleife. Im Rumpf der neuen Schleife wird der alte Schleifenrumpf n-mal wiederholt, wobei zwischen zwei alten Rümpfen die Schleifenbedingung überprüft wird, um zu garantieren, dass der zweite Rumpf nur dann ausgeführt ist, wenn die Bedingung noch wahr ist. Das in HOL bewiesene Theorem für diese Transformation ist in (7.37) dargestellt. FOR_1_NFOLD ist die in Abschnitt 5.1.6 auf Seite 67 unter (5.30) definierte spezielle for-Schleife, die eine n-malige Anwendung derselben Funktion realisiert. Theorem (7.37) ist parametrisierbar darin, wie oft der Schleifenrumpf wiederholt werden soll. Während der Synthese wird dieser Parame-

ter n instantiiert. Danach kann Theorem (7.38) verwendet werden, um die Funktion FOR_1_NFOLD zu beseitigen.

$$\begin{array}{l}\vdash \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a)\ = \\ \quad \mathsf{WHILE}\ c \\ \quad\quad (\mathsf{PARTIALIZE}\ a)\ \mathsf{THEN} \\ \quad\quad\quad \mathsf{FOR\_1\_NFOLD}\ n\ (\mathsf{PARTIALIZE}\ (\lambda x.\mathsf{MUX}(c\ x, a\ x, x)))\end{array} \tag{7.37}$$

$$\begin{array}{l}\vdash \mathsf{FOR\_1\_NFOLD}\ 0\ A\ =\ \mathsf{NOP}\quad \wedge \\ \quad \mathsf{FOR\_1\_NFOLD}\ (\mathsf{SUC}\ n)\ A\ =\ A\ \mathsf{THEN}\ (\mathsf{FOR\_1\_NFOLD}\ n\ A)\end{array} \tag{7.38}$$

Die Bedeutung des Schleifenaufrollens für die Schaltungssynthese liegt in einer erhöhten Ausführungsgeschwindigkeit, da die zur Abarbeitung notwendige Taktanzahl abnimmt. Durch die SPT werden alle Operationen eines Schleifenrumpfes einem Kontrollschritt zugeordnet. Durch das Schleifenaufrollen wird damit die Anzahl der während eines Kontrollschrittes ausgeführten Operationen erhöht. Wie viel schneller die Abarbeitung dadurch wird, lässt sich aber i.Allg. nicht vorhersagen, da dies von der Schleifenbedingung abhängt. Während bei einer while-Schleife der Zeitgewinn u.U. bescheiden ausfallen kann, lässt sich bei einer for-Schleife – die in Gropius als while-Schleife repräsentiert wird – die Ersparnis an Ausführungszeit durchaus genau abschätzen. Ein entscheidender Nachteil des Schleifenaufrollens ist aber der erhöhte Flächenbedarf, da alle Operationen $n+1$-mal realisiert werden müssen. Verbunden damit ist eine Erhöhung der kombinatorischen Tiefe in der korrespondierenden RT-Struktur und somit eine Erhöhung der Verzögerungszeit während eines Taktes. Die theoretisch mögliche Taktfrequenz wird damit verringert.

Schleifenauftrennen

Der zum Schleifenaufrollen umgekehrte Vorgang ist das "Schleifenauftrennen" (loop-cutting). Hier werden die Operationen eines Schleifenrumpfes nicht mehrmals in einem Kontrollschritt, sondern verteilt auf mehrere Kontrollschritte ausgeführt. Der Schleifenrumpf wird in mehrere kleinere Teile zerlegt. Jeder dieser Teile entspricht dann einem Kontrollschritt.

Das in HOL bewiesene Theorem für das Schleifenauftrennen ist in (7.39) dargestellt. Das Theorem geht davon aus, dass der Schleifenrumpf bereits in eine Komposition von Teilfunktionen eingeteilt wurde. Der Term list_o L bezeichnet eine solche Komposition von Funktionen, die in der Liste L aufgeführt sind. Indem vorgeschrieben wird, dass die Liste L das Aussehen (CONS k r) hat, wird garantiert, dass die Liste nicht leer ist.

$$\begin{array}{l}\vdash \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ \ (\mathsf{list\_o}\ (\mathsf{CONS}\ k\ r))) \\ \quad \mathsf{LOCVAR}\ (\mathsf{enum}\ (\mathsf{SUC}\ (\mathsf{LENGTH}\ r))\ 0) \\ \quad\quad \mathsf{WHILE}\ (\lambda(x,h).\ c\ x\ \vee\ \neg(h\ =\ 0)) \\ \quad\quad\quad \mathsf{PARTIALIZE}\ (\lambda(x,h).\ ((\mathsf{CASE}\ (k :: r)\ h)\ x, \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{next}\ (\mathsf{SUC}\ (\mathsf{LENGTH}\ r))\ h)\ )\end{array} \tag{7.39}$$

Die Funktionen list_o, CONS und LENGTH wurden bereits in Abbildung 7.12 auf Seite 126 definiert. Sie wurden zusammen mit den Funktionen CASE, enum und next in Abschnitt 7.2.2 eingeführt.

Das Theorem (7.39) sagt folgendes aus: Ist eine while-Schleife mit einem unterteilten Schleifenrumpf (list_o (CONS k r)) gegeben, wird sie umgewandelt in eine äquivalente while-Schleife, die (LENGTH r)-mal öfter ihren Rumpf ausführt als die gegebene Schleife. Während einer Ausführung des neuen Schleifenrumpfes wird genau eine der Funktionen der Liste (CONS k r) angewandt. Die Kontrollinformation, welche Funktion in welchem Durchlauf ausgeführt werden muss, steckt in einer neu eingeführten lokalen Variablen. Diese lokale Variable hat den Datentyp $\mathsf{enum}_{1+\,\mathsf{LENGTH}\ r}$ und den Initialwert (enum $(1 + \mathsf{LENGTH}\ r)$ 0), was gerade dem Wert 0 entspricht. Hat die lokale Variable den Wert 0, hat sie keinen Einfluss auf die Schleifenbedingung, und es hängt von der ursprünglichen Schleifenbedingung c ab, ob der Schleifenrumpf ausgeführt wird oder nicht. Ist der Wert der lokalen Variablen hingegen ungleich 0, wird der Schleifenrumpf unabhängig von c ausgeführt. Wird der Schleifenrumpf ausgeführt, wird einerseits die h-te Funktion der Liste (CONS k r) auf den ersten Teil des Eingangs (x, h) angewandt, andererseits wird h inkrementiert, bzw. auf 0 gesetzt, wenn ansonsten der Wertebereich des Aufzählungsdatentyps überschritten würde.

Das Schleifenauftrennen hat die umgekehrte Bedeutung des Schleifenaufrollens für den Schaltungsentwurf. Hier wird angestrebt, den Flächenbedarf möglichst zu verkleinern. Da die einzelnen Teilfunktionen verteilt auf mehrere Kontrollschritte angewandt werden, besteht die Möglichkeit, dass gleiche Operationen, die unterschiedlichen Teilfunktionen zugeordnet sind, auf der RT-Ebene durch gemeinsame Funktionseinheiten ausgeführt werden können. Das Schleifenauftrennen macht also nur dann Sinn, wenn der Schleifenrumpf so aufgeteilt ist, dass in unterschiedlichen Teilfunktionen gleiche Operationen vorhanden sind. Als Nachteil dieser Transformation ergibt sich auf jeden Fall eine höhere Ausführungszeit.

Die Transformation des Schleifenauftrennens soll nun exemplarisch am in Abschnitt 5.1.7 vorgestellten Programm fib präsentiert werden. Bevor Theorem (7.39) angewandt werden kann, muss der Schleifenrumpf von fib zuerst in eine Komposition von Teilfunktionen zerlegt werden. Das entspricht einer zeitlichen Einordnung der Operationen des Rumpfes. Dies wird in der in Abschnitt 7.2 vorgestellten Weise durchgeführt, indem der Syntheseschritt in Entwurfsraumuntersuchung und Transformation aufgeteilt wird. Der obere Teil von Tabelle 7.2 zeigt das Ergebnis des Einsatzes verschiedener Techniken für die zeitliche Einordnung, die bereits in Abschnitt 3.1.1 vorgestellt wurden. Für jede Teilfunktion werden dabei die lokalen Variablen des in Abbildung 5.8 vorgestellten Programms angegeben, die das Ergebnis der entsprechenden Operation speichern. Für das Verfahren des "list-based scheduling" wurde die Anzahl der Multiplikationen und Additionen pro Kontrollschritt auf jeweils zwei beschränkt. Daraus resultiert, dass sich im Vergleich zu den beiden anderen Techniken die Anzahl der Teilfunktionen um eins erhöht.

Es muss angemerkt werden, dass bei der Implementierung der Algorithmen zur Entwurfsraumuntersuchung kein Chaining berücksichtigt wurde. Dies stellt aber

Teilfunktion	ASAP	Force-directed	List-based
1	$c, m1, m2, x$	c	$c, m1, m2, x$
2	$m3, x1, x2$	$m1, m2x, x1, x2$	$m3, x1, x2$
3	$x3, x4, x6, x7$	$x3, x4, x6, x7$	$x3, x4$
4	$x5, x8$	$m3, x5, x8$	$x5, x6, x7$
5	$y1', a1', a2', y2'$	$y1', a1', a2', y2'$	$x8, y1', a1'$
6	$- - -$	$- - -$	$a2', y2'$
Bereitgestellte Register	Typ bool : 1 Typ num : 10	Typ bool : 1 Typ num : 10	Typ bool : 1 Typ num : 11

Tabelle 7.2: Steuerinformation verschiedener Techniken zur zeitlichen Einordnung

keine allgemeine Einschränkung dar.

Im Programm fib hat der Schleifenrumpf sechs Eingänge und sechs Ausgänge. Die Unterteilung in eine Komposition von Teilfunktionen wird aber i.Allg. ergeben, dass die Teilfunktionen eine unterschiedliche Anzahl von Ein- und Ausgängen haben. Um Theorem (7.39) anwenden zu können, müssen aber alle Teilfunktionen denselben Ein- und Ausgangstyp haben. Bevor also die logische Transformation zur zeitlichen Einordnung durchgeführt wird, müssen mehrere Variablen eingeführt werden. Da die Anzahl der Variablen in der Variablenstruktur der DFG-Terme in der ESF direkt der Anzahl der erforderlichen Register auf RT-Ebene entspricht, ist dies gleichbedeutend mit der Bereitstellung zusätzlicher Register. Im unteren Teil von Tabelle 7.2 ist für jedes Einordnungsverfahren die Anzahl und Art der Register angegeben, die benötigt werden, um alle Zwischenergebnisse zwischen den Teilfunktionen zu speichern. Die Anzahl der zusätzlich einzuführenden Variablen ergibt sich somit aus der Differenz zwischen benötigter Registerzahl und der Anzahl der bereits vorhandenen Eingangsvariablen des Schleifenrumpfes. Wendet man zur Unterteilung des Schleifenrumpfes das list-based Verfahren an, müssen sechs zusätzliche Variablen eingeführt werden. Das Einführen zusätzlicher Variablen vor einer Elementarschleife geschieht mit Hilfe von Theorem (7.40). Der Term (7.41) ergibt sich, indem dieses Theorem auf die Elementarschleife von fib angewandt wird, wobei i in geeigneter Weise instantiiert wurde.

$$\begin{aligned} &\vdash \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ \ a)\ = \\ &\qquad \mathsf{LOCVAR}\ i\ (\mathsf{WHILE}\ (\lambda(x,h).\ c\ x)\ (\mathsf{PARTIALIZE}\ (\lambda(x,h).(a\ x, i)))) \end{aligned} \tag{7.40}$$

$$\begin{aligned} &\mathsf{LOCVAR}(\mathsf{F},0,0,0,0,0) \\ &\quad \mathsf{WHILE}\ (\lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5,h6).\neg(m=0)) \\ &\qquad \mathsf{PARTIALIZE}(\lambda(((n,y1),a1,a2,y2,m),h1,h2,h3,h4,h5,h6). \\ &\qquad\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in}\quad \ldots\quad \mathsf{in} \\ &\qquad\quad (((n,y1'),a1',a2',y2',m3),\mathsf{F},0,0,0,0,0)\ ) \end{aligned} \tag{7.41}$$

Die zusätzlich eingeführten Variablen sind nur Platzhalter für die anschließen-

de zeitliche Einordnung und das Zuordnen der Register. Sie dürfen im Rumpf der λ-Abstraktion nicht verwendet werden, da sonst die Algorithmen zur zeitlichen Einordnung davon ausgehen, dass die Werte in diesen Variablen durchgereicht werden müssen. Aus diesem Grunde wird am Ausgang des Schleifenrumpfes an der Stelle der zusätzlichen Variablen deren Initialwert i ausgegeben. Da alle sechs bereits vorhandenen Eingangsvariablen vom Typ num sind und elf Register vom Typ num, aber eines vom Typ bool bereitgestellt wurde, wird neben fünf Variablen vom Typ num auch eine boolesche Variable eingeführt. Die Initialwerte für die jeweiligen Initialwerte sind beliebig, es sollen hier aber bestimmte Default-Werte für jeden Typ eingesetzt werden.

Erst jetzt kann die logische Transformation für die zeitliche Einordnung und das Zuordnen der Register durchgeführt werden. Dies geschieht auf demselben Wege, der bereits in Abschnitt 7.2.1 beschrieben wurde. Abbildung 7.18 zeigt das resultierende Theorem.

Die zeitliche Einordnung wurde aufgrund der Steuerinformation des "list-based scheduling"-Verfahrens durchgeführt, für die Registerzuordnung wurde eine Heuristik benutzt, die dafür sorgt, dass eine Variable solange wie möglich in einem Register bleibt, um unnötigen Registertransfer zu vermeiden. Die rechte Seite des Theorems in Abbildung 7.18 hat nun die Form (list_o L), wobei L eine Liste mit sechs DFG-Termen ist. Nun kann das Theorem (7.39) für das Schleifenauftrennen angewandt werden, worauf hier aber aus Platzgründen verzichtet werden soll.

Andere Schleifentransformationen

Zwei weitere Transformationen, die bei der Verwendung von Schleifen angewandt werden können, sind das Herausziehen einer Schleifeninvariante und das Herausziehen des Schleifenrumpfes. Die in HOL bewiesenen Theoreme, die dies leisten, lauten wie folgt:

$$\begin{aligned}
&\vdash \mathsf{WHILE}\ c\ \mathsf{LOCVAR}\ z_i\ (\ \mathsf{PARTIALIZE}\ (\lambda((x,y),z).((x,y),f\ y))\ \mathsf{THEN} \\
&\qquad\qquad\qquad\qquad \mathsf{ASSOCVAR(LEFT}\ A\ (i_2,i_3))\ (i_1,i_2,i_3)\ ) \\
&\quad = \\
&\quad \mathsf{LOCVAR}\ z_i \\
&\qquad \mathsf{PARTIALIZE}\ (\lambda((x,y),z).((x,y),f\ y))\ \mathsf{THEN} \\
&\qquad \mathsf{WHILE}\ (\lambda((x,y),z).c(x,y))\ (\mathsf{ASSOCVAR(LEFT}\ A\ (i_2,i_3))\ (i_1,i_2,i_3))
\end{aligned} \tag{7.42}$$

$$\vdash \mathsf{WHILE}\ c\ A\ =\ (\mathsf{IFT}\ c\ A)\ \mathsf{THEN}\ (\mathsf{WHILE}\ c\ A) \tag{7.43}$$

Theorem (7.42) geht von einer Schleife aus, in deren Rumpf eine lokale Variable eingeführt wird. Dieser lokalen Variablen wird in einem ersten Schritt ein Wert zugewiesen, der vom zweiten Teil des globalen Zustands (x, y) abhängt. Anschließend wird im Rumpf ein Block ausgeführt, durch den nur der erste Teil x des globalen Zustands verändert wird. Dies wird gewährleistet durch die zwei abgeleiteten Kontrollstrukturen ASSOCVAR und LEFT, die in Abschnitt 5.1.6 eingeführt wurden. Die Funktion A hat den Typ $(\alpha\times(\beta\times\gamma)) \to (\alpha\times(\beta\times\gamma))$partial. ASSOCVAR wandelt den

Typ $((\alpha \times \beta) \times \gamma)$ des Zustands $((x, y), z)$ in den Eingangstyp von A um, und LEFT sorgt dafür, dass nur die erste Komponente des Ergebnisses von A übernommen wird, sodass y und z durch diesen Block nicht verändert werden. Aus diesem Grund kann die lokale Variable samt dem Elementarblock des Schleifenrumpfes aus der Schleife herausgezogen werden. Der Vorteil dieser Transformation ist offensichtlich. Der DFG-Term f wird im transformierten Term nur einmal ausgeführt, anstatt in jedem Schleifenrumpfdurchlauf ausgeführt zu werden.

$$
\begin{array}{l}
\vdash\ \lambda(((n, y1), a1, a2, y2, m), h1, h2, h3, h4, h5, h6). \\
\quad \mathsf{let}\ c = \mathsf{ODD}\ m\ \mathsf{in}\quad \ldots\quad \mathsf{in} \\
\quad (((n, y1'), a1', a2', y2', m3), \mathsf{F}, 0, 0, 0, 0, 0) \\
= \\
\lambda(((r1, r2), r3, r4, r5, r6), r7, r8, r9, r10, r11, r12). \\
\quad \mathsf{let}\ a2' = \mathsf{MUX}(r7, r9, r2)\ \mathsf{in}\ \mathsf{let}\ y2' = \mathsf{MUX}(r7, r2, r10)\ \mathsf{in} \\
\quad \mathsf{let}\ n = r6\ \mathsf{in}\ \mathsf{let}\ y' = r3\ \mathsf{in}\ \mathsf{let}\ a1' = r4\ \mathsf{in}\ \mathsf{let}\ m3 = r5\ \mathsf{in} \\
\quad (((n, y'), a1', a2', y2', m3), \mathsf{F}, 0, 0, 0, 0, 0) \\
\circ \\
\lambda(((r1, r2), r3, r4, r5, r6), r7, r8, r9, r10, r11, r12). \\
\quad \mathsf{let}\ r2' = r4 + r2\ \mathsf{in}\ \mathsf{let}\ r3' = \mathsf{MUX}(r7, r3, r1)\ \mathsf{in} \\
\quad \mathsf{let}\ r4' = \mathsf{MUX}(r7, r8, r3)\ \mathsf{in}\ \mathsf{let}\ r7' = r7\ \mathsf{in}\ \mathsf{let}\ r5' = r5\ \mathsf{in} \\
\quad \mathsf{let}\ r6' = r6\ \mathsf{in}\ \mathsf{let}\ r9' = r9\ \mathsf{in}\ \mathsf{let}\ r10' = r10\ \mathsf{in} \\
\quad (((r1', r2'), r3', r4', r5', r6'), r7', r8', r9', r10', r11', r12') \\
\circ \\
\lambda(((r1, r2), r3, r4, r5, r6), r7, r8, r9, r10, r11, r12). \\
\quad \mathsf{let}\ r3' = r12 + r11\ \mathsf{in}\ \mathsf{let}\ r4' = r2 * r9\ \mathsf{in}\ \mathsf{let}\ r2' = r4 * r3\ \mathsf{in} \\
\quad \mathsf{let}\ r7' = r7\ \mathsf{in}\ \mathsf{let}\ r5' = r5\ \mathsf{in}\ \mathsf{let}\ r6' = r6\ \mathsf{in} \\
\quad \mathsf{let}\ r1' = r1\ \mathsf{in}\ \mathsf{let}\ r8' = r8\ \mathsf{in}\ \mathsf{let}\ r9' = r9\ \mathsf{in}\ \mathsf{let}\ r10' = r10\ \mathsf{in} \\
\quad (((r1', r2'), r3', r4', r5', r6'), r7', r8', r9', r10', r11', r12') \\
\circ \\
\lambda(((r1, r2), r3, r4, r5, r6), r7, r8, r9, r10, r11, r12). \\
\quad \mathsf{let}\ r12' = r2 * r8\ \mathsf{in}\ \mathsf{let}\ r11' = r4 * r9\ \mathsf{in}\ \mathsf{let}\ r2' = r2\ \mathsf{in} \\
\quad \mathsf{let}\ r4' = r4\ \mathsf{in}\ \mathsf{let}\ r3' = r3\ \mathsf{in}\ \mathsf{let}\ r7' = r7\ \mathsf{in} \\
\quad \mathsf{let}\ r5' = r5\ \mathsf{in}\ \mathsf{let}\ r6' = r6\ \mathsf{in}\ \mathsf{let}\ r1' = r1\ \mathsf{in} \\
\quad \mathsf{let}\ r8' = r8\ \mathsf{in}\ \mathsf{let}\ r9' = r9\ \mathsf{in}\ \mathsf{let}\ r10' = r10\ \mathsf{in} \\
\quad (((r1', r2'), r3', r4', r5', r6'), r7', r8', r9', r10', r11', r12') \\
\circ \\
\lambda(((r1, r2), r3, r4, r5, r6), r7, r8, r9, r10, r11, r12). \\
\quad \mathsf{let}\ r5' = \mathsf{MUX}(r7, r2, r4)\ \mathsf{in}\ \mathsf{let}\ r2' = \mathsf{MUX}(r7, r1, r8)\ \mathsf{in} \\
\quad \mathsf{let}\ r4' = \mathsf{MUX}(r7, r10, r9)\ \mathsf{in}\ \mathsf{let}\ r3' = r3\ \mathsf{in}\ \mathsf{let}\ r7' = r7\ \mathsf{in}\ \mathsf{let}\ r6' = r6\ \mathsf{in} \\
\quad \mathsf{let}\ r1' = r1\ \mathsf{in}\ \mathsf{let}\ r8' = r8\ \mathsf{in}\ \mathsf{let}\ r9' = r9\ \mathsf{in}\ \mathsf{let}\ r10' = r10\ \mathsf{in} \\
\quad (((r1', r2'), r3', r4', r5', r6'), r7', r8', r9', r10', r11', r12') \\
\circ \\
\lambda(((n, y1), a1, a2, y2, m), h1, h2, h3, h4, h5, h6). \\
\quad \mathsf{let}\ r7' = \mathsf{ODD}\ m\ \mathsf{in}\ \mathsf{let}\ r2' = m - 1\ \mathsf{in}\ \mathsf{let}\ r4' = m\ \mathsf{DIV}\ 2\ \mathsf{in} \\
\quad \mathsf{let}\ r3' = a1 + a2\ \mathsf{in}\ \mathsf{let}\ r6' = n\ \mathsf{in}\ \mathsf{let}\ r1' = y\ \mathsf{in}\ \mathsf{let}\ r8' = a1\ \mathsf{in} \\
\quad \mathsf{let}\ r9' = a2\ \mathsf{in}\ \mathsf{let}\ r10' = y2 \\
\quad (((r1', r2'), r3', r4', r5', r6'), r7', r8', r9', r10', r11', r12')
\end{array}
$$

Abbildung 7.18: Theorem nach zeitlicher Einordnung und Zuordnung der Register

Durch Theorem (7.43) wird der erste Schleifenrumpfdurchlauf aus der Schleife herausgezogen. Eine mögliche Anwendung dieses Theorem ist es, in Verbindung mit Theorem (7.11) eingesetzt zu werden. Liegt vor der Schleife ein Elementarrumpf, so kann der Schleifenrumpf zunächst herausgezogen werden und mit diesem Elementarrumpf zusammengefasst werden. Wendet man dann Theorem (7.11) an, wird im ersten Schleifendurchlauf der neuen Schleife nicht nur der Elementarrumpf, sondern auch der alte Schleifenrumpf ausgeführt, wenn die alte Schleifenbedingung erfüllt ist. Man gelangt so zu einer ESF, die einer RT-Implementierung entspricht, deren Ausführungszeit i.Allg. kürzer, deren Flächenbedarf aber auch höher ist.

Einfügen von Kontrollschritten

Durch die SPT werden in festgelegter Weise Kontrollschritte eingeführt und die Operationen auf diese Kontrollschritte verteilt. Eine einfache Maßnahme, um sicherzustellen, dass an einer bestimmten Stelle ein neuer Kontrollschritt eingeführt wird, stellt Theorem (7.45) dar:

$$\vdash \mathsf{BREAK} \;=\; \mathsf{WHILE}\ (\lambda x.\mathsf{F})\ \mathsf{NOP} \tag{7.44}$$

$$\vdash A\ \mathsf{THEN}\ B \;=\; A\ \mathsf{THEN\ BREAK\ THEN}\ B \tag{7.45}$$

Die abgeleitete Kontrollstruktur BREAK steht für eine Schleife, deren Rumpf niemals ausgeführt wird. Wird diese Pseudo-Schleife aber zwischen zwei Blöcken eingeführt, wird durch die SPT dafür gesorgt, dass diese beiden Blöcke in verschiedene Kontrollschritte eingeteilt werden. Die Konsequenz für die Hardware-Implementierung ist wieder ein evtl. geringerer Flächenbedarf (hängt von den Operationen in den Blöcken A und B ab) sowie auf der anderen Seite eine erhöhte Taktzahl.

Transformationen für bedingte Verzweigungen und Sequenzen

Im Folgenden sind einige Transformationstheoreme aufgelistet, die zur Vereinfachung von bedingten Verzweigungen und Sequenzen eingesetzt werden können. Sie lassen sich leicht durch einfache Fallunterscheidung beweisen.

$$\vdash \mathsf{IFTE}\ c\ A\ A \;=\; A$$

$$\vdash \mathsf{IFTE}\ c\ A\ B \;=\; \mathsf{IFTE}\ (\lambda x.\neg(c\ x))\ B\ A$$

$$\vdash \mathsf{IFTE}\ c_1\ (\mathsf{IFTE}\ c_2\ A\ B)\ D \;=$$
$$\qquad \mathsf{IFTE}\ (\lambda x.(c_1\ x) \wedge (c_2\ x))\ A\ (\mathsf{IFTE}\ (\lambda x.(c_1\ x) \wedge \neg(c_2\ x))\ B\ D)$$

$$\vdash \mathsf{IFTE}\ c_1\ A\ (\mathsf{IFTE}\ c_2\ B\ D) \;=\; \mathsf{IFTE}\ (\lambda x.(c_1\ x) \vee (c_2\ x))\ (\mathsf{IFTE}\ c_1\ A\ B)\ D$$

$$\vdash \mathsf{IFTE}\ c_1\ (\mathsf{IFTE}\ c_2\ A\ B)\ B \;=\; \mathsf{IFTE}\ (\lambda x.(c_1\ x) \wedge (c_2\ x))\ A\ B$$

$$\vdash \mathsf{IFTE}\ c_1\ A\ (\mathsf{IFTE}\ c_2\ A\ B) \;=\; \mathsf{IFTE}\ (\lambda x.(c_1\ x) \vee (c_2\ x))\ A\ B$$

$$\vdash \mathsf{IFTE}\ c_1\ (\mathsf{IFTE}\ c_1\ A\ D)\ B \;=\; \mathsf{IFTE}\ c_1\ A\ B$$

$$\vdash \mathsf{IFTE}\ c\ (B\ \mathsf{THEN}\ A)\ (D\ \mathsf{THEN}\ A) \;=\; (\mathsf{IFTE}\ c\ B\ D)\ \mathsf{THEN}\ A$$

$$\vdash \mathsf{IFT}\ c\ (B\ \mathsf{THEN}\ (\mathsf{IFT}\ c\ A)) \;=\; (\mathsf{IFT}\ c\ B)\ \mathsf{THEN}\ (\mathsf{IFT}\ c\ A)$$

$$\vdash (A \text{ THEN } B) \text{ THEN } D \;=\; A \text{ THEN } (B \text{ THEN } D)$$
$$\vdash A \text{ THEN NOP} \;=\; A \quad \wedge \quad \text{NOP THEN } A \;=\; A$$

Die in diesem Abschnitt vorgestellten Transformationen stellen nur einen Ausschnitt von möglichen Transformationen dar, mit deren Hilfe Programme optimiert werden können. Insbesondere aber Schleifentransformationen wie das Schleifenaufrollen und das Schleifenauftrennen sind mächtige Hilfsmittel hierfür. Im Rahmen des DFG-Projektes Schm-623/6-2 wird daran gearbeitet, weitere leistungsfähige Transformationstheoreme für die Synthese zur Verfügung zu stellen.

7.3.5 Schnittstellensynthese

In Abschnitt 5.2.1 wurden verschiedene Schnittstellenverhaltensmuster eingeführt, die mit einem beliebigen Programm zu einer algorithmischen P-Schaltungsbeschreibung kombiniert werden können. Für jedes dieser Schnittstellenverhaltensmuster wird zusätzlich ein korrektes Implementierungsmuster auf der RT-Ebene in Form eines vorbewiesenen Theorems zur Verfügung gestellt. Jedes dieser Implementierungstheoreme geht davon aus, dass das Programm in ESF vorliegt. Es sagt aus, dass das Implementierungsmuster auf der RT-Ebene das entsprechende Schnittstellenverhaltensmuster für jedes beliebige Programm in ESF erfüllt.

Abbildung 7.19(a) zeigt die Gropius-Beschreibung des Implementierungsmusters P_IMP_RESET_START, welches das Pendant zu dem Schnittstellenverhaltensmuster P_IFC_RESET_START darstellt. Diese RT-Struktur ist in Abbildung 7.19(b) schematisch dargestellt. Die Struktur lässt sich in zwei Teile gliedern: Der obere Teil stellt den Kontroller dar, der die Kommunikation mit der Umgebung regelt (Signale *reset*, *start* und *ready*). Der untere Teil dient der Berechnung (DFG-Terme a und c), der Kommunikation (Multiplexer) sowie der Datenspeicherung (Register). Diese Register werden im Laufe der OPT und SPT bereitgestellt. Die Variablen q_1, q_2 und q_3 sind beliebige Initialwerte in diesen Registern. Sie können im Verlauf der RT-Synthese in geeigneter Weise instantiiert werden.

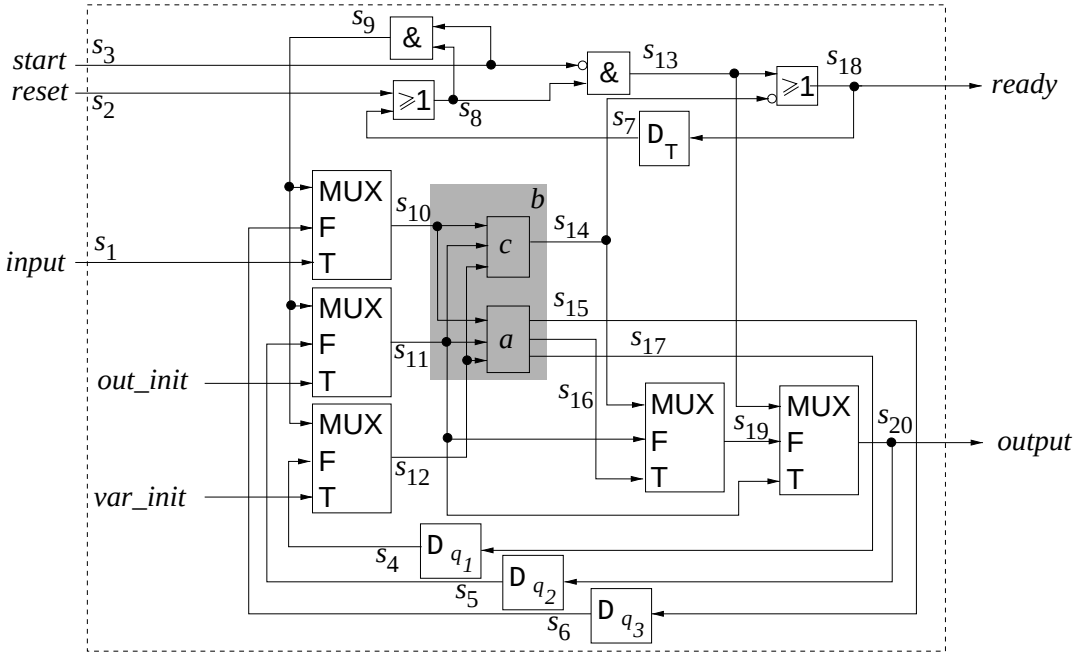
Theorem (7.46) drückt aus, dass für *jedes* Programm in ESF die Implementierung P_IMP_RESET_START die Spezifikation erfüllt, die durch das Schnittstellenverhaltensmuster P_IFC_RESET_START und eben dieses Programm beschrieben ist.

$$\begin{array}{l}\vdash\ \forall a\ c\ \mathit{out\_init}\ \mathit{var\_init}.\\ \quad \text{P\_IMP\_RESET\_START}\ (\mathit{input}, \mathit{reset}, \mathit{start}, \mathit{output}, \mathit{ready}, a, c, \mathit{out\_init}, \mathit{var\_init})\\ \quad \Rightarrow\\ \quad \text{P\_IFC\_RESET\_START}\\ \quad \text{PROGRAM}\ \mathit{out\_init}\ (\text{LOCVAR}\ \mathit{var\_init}\ (\text{WHILE}\ c\ (\text{PARTIALIZE}\ a)))\\ \quad (\mathit{input}, \mathit{reset}, \mathit{start}, \mathit{output}, \mathit{ready})\end{array} \tag{7.46}$$

Im Laufe der Schnittstellensynthese wird dieses Theorem mit einem konkreten Programm in ESF instantiiert und man erhält durch logische Verfeinerung eine Implementierung auf RT-Ebene.

P_IMP_RESET_START $(input, reset, start, output, ready, a, c, out\_init, var\_init) =$
$\exists q_1\ q_2\ q_3.$
$(\lambda t.\ (output\, t, ready\, t)) =$
automaton
$(\ (\lambda((s_1, s_2, s_3), (s_4, s_5, s_6, s_7)).$
let $s_8 = s_2 \vee s_7$ in
let $s_9 = s_3 \wedge s_8$ in
let $s_{10} =$ MUX (s_9, s_1, s_6) in
let $s_{11} =$ MUX (s_9, out_init, s_5) in
let $s_{12} =$ MUX (s_9, var_init, s_4) in
let $s_{13} = \neg s_3 \wedge s_8$ in
let $s_{14} = c\ ((s_{10}, s_{11}), s_{12})$ in
let $((s_{15}, s_{16}), s_{17}) = a\ ((s_{10}, s_{11}), s_{12})$ in
let $s_{18} = s_{13} \vee \neg s_{14}$ in
let $s_{19} =$ MUX (s_{14}, s_{16}, s_{11}) in
let $s_{20} =$ MUX (s_{13}, s_{11}, s_{19}) in
$((s_{20}, s_{18}), (s_{17}, s_{20}, s_{15}, s_{18})))$
$, (q_1, q_2, q_3, \mathsf{T})\ )$
$(\lambda t.\ (input\, t, reset\, t, start\, t))$

(a) Beschreibung in Gropius-1



(b) Schematische Darstellung

Abbildung 7.19: Implementierungsmuster P_IMP_RESET_START

Nachdem die Schnittstellensynthese durchgeführt wurde, können die Funktionseinheiten bereitgestellt und zugeordnet werden. Für das Implementierungsmuster P_IMP_RESET_START geschieht dies in dem DFG-Term b, der durch die graue Fläche in Abbildung 7.19(b) markiert ist. Er ergibt sich durch die parallele Ausführung der DFG-Terme a und c der ESF. Da dieser Vorgang im Wesentlichen bereits in Abschnitt 7.2.1 vorgestellt wurde, soll er hier nicht näher erläutert werden.

Für die vier Schnittstellenverhaltensmuster P_IFC_START, P_IFC_CYCLE_1, P_IFC_CYCLE_2 und P_IFC_NO_RESET, die alle aus P_IFC_RESET_START abgeleitet wurden, lassen sich in einfacher Weise die entsprechenden Implementierungsmuster aus P_IMP_RESET_START ableiten. Das Implementierungsmuster P_IMP_RESET für das Schnittstellenverhaltensmuster P_IFC_RESET unterscheidet sich von P_IMP_RESET_START lediglich durch einen anderen Kontroller. Dieser ist in Abbildung 7.20 dargestellt. Auch für dieses Implementierungsmuster wurde ein Implementierungstheorem bewiesen.

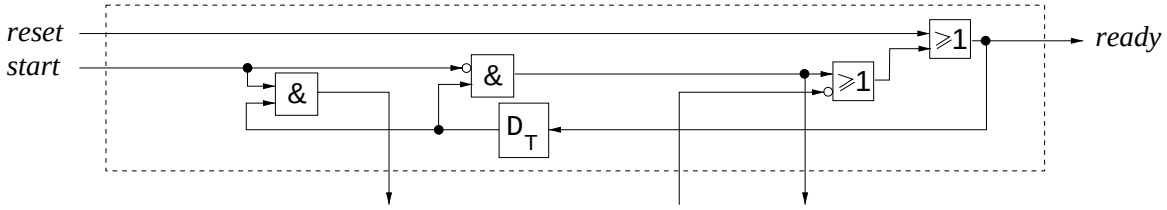


Abbildung 7.20: Kontroller von P_IMP_RESET

Fortsetzung des laufenden Beispiels

Das Programm fib sei mit dem Schnittstellenverhaltensmuster P_IFC_START zu einer algorithmischen Schaltungsbeschreibung zusammengesetzt. Das Implementierungsmuster P_IMP_START ergibt sich aus P_IMP_RESET_START, indem das Signal *reset* durch *start* ersetzt wird. Zusammen mit dem Theorem über die ESF, das in Abbildung 7.17 gezeigt wird, ergibt sich das Implementierungstheorem (7.47).

$$
\begin{aligned}
\vdash\ & \mathsf{P\_IMP\_START}\ (n, start, \mathsf{fib}(n), ready,\\
& \quad (\lambda((n, y1), a1, a2, y2, m, h1, h2).\mathsf{let}\ x1' = \neg(m = 0)\ \mathsf{in}\ \ldots),\\
& \quad (\lambda((n, y1), a1, a2, y2, m, h1, h2).\neg h1 \vee \neg h2),\\
& \quad 1, (1, 1, 0, 0, \mathsf{F}, \mathsf{F})\ )\\
& \Rightarrow\\
& \mathsf{P\_IFC\_START}\ \mathsf{fib}\ (n, start, \mathsf{fib}(n), ready)
\end{aligned}
\tag{7.47}
$$

Die entsprechende RT-Implementierung ist in Abbildung 7.21 gezeigt. Aus Platzgründen wurde dabei der DFG-Term a der in Abbildung 7.17 gezeigten ESF nicht eingesetzt. q bezeichnet die Initialwerte in den acht bereitgestellten Registern.

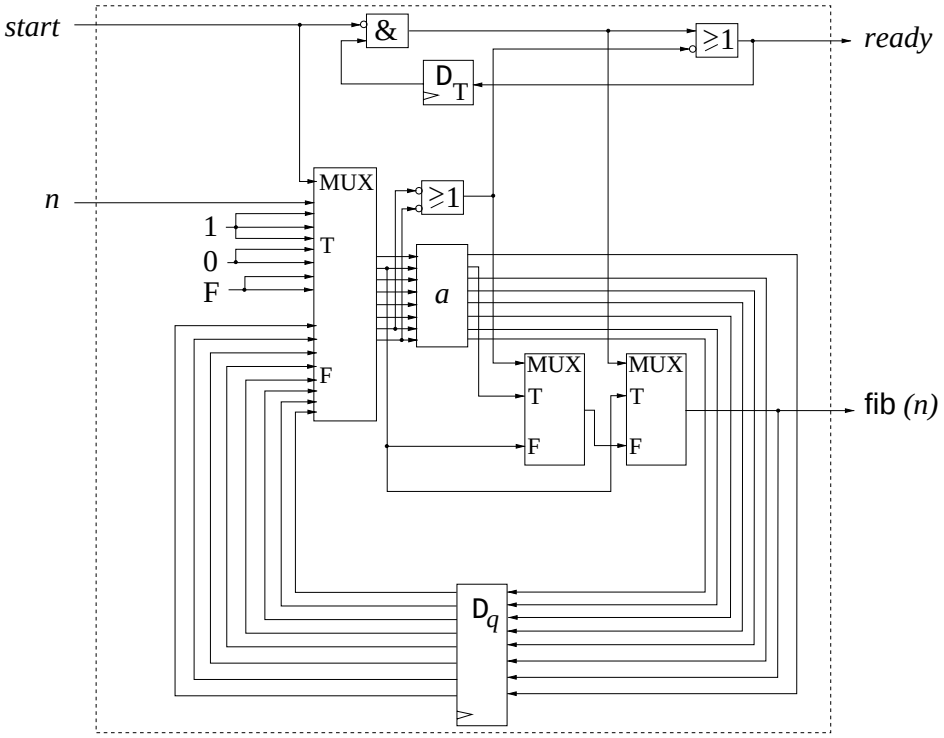


Abbildung 7.21: Implementierung des Programms fib für das Schnittstellenverhaltensmuster P_IFC_START

Kapitel 8

Schaltungssynthese auf der Systemebene

Das mit der Sprache Gropius verbundene Konzept sieht vor, Schaltungen auf der Systemebene durch miteinander kommunizierende Prozesse zu beschreiben. In diesem Ansatz entspricht jeder Prozess einer solchen Mehrprozessbeschreibung direkt einer Schaltungskomponente, und die Verbindungen zwischen zwei Prozessen, die zu einem Kanal zusammengefasst werden, entsprechen direkt Signalleitungen.

Diese Vorgehensweise ist nicht selbstverständlich. Bei anderen Ansätzen auf der Systemebene werden Systeme durch Programme beschrieben, die in direkter Weise via I/O-Operationen über Kommunikationskanäle miteinander kommunizieren. Die entsprechenden Kommunikationskanäle und Schnittstellen müssen dann erst synthetisiert werden, wozu i.Allg. auch aktive Komponenten erforderlich sind.

Im vorliegenden Ansatz können die einzelnen Prozesse direkt über eine High-Level Synthese unabhängig voneinander synthetisiert werden. Da die Kanäle direkt elektronischen Leitungen entsprechen, ist keine zusätzliche Synthese abstrakter Kommunikationskanäle erforderlich. Daher können die synthetisierten RT-Ebenen-Strukturen der einzelnen Prozesse ebenso wie auf der Systemebene in einfacher Weise "zusammengesteckt" werden.

8.1 Synthese von P- und DFG-Prozessen

Die Synthese von P- und DFG- Prozessen erfolgt prinzipiell in der gleichen Weise wie die von P- und DFG-Term basierten Schaltungsbeschreibungen auf der algorithmischen Ebene (siehe Abschnitte 7.3 und 7.2). Der einzige Unterschied besteht darin, dass während der Schnittstellensynthese ein anderes Implementierungstheorem angewandt werden muss. Während auf der algorithmischen Ebene aus verschiedenen Schnittstellenverhaltensmustern eines ausgewählt werden kann und damit auch eines der möglichen Implementierungstheoreme angewandt wird, gibt es auf der Systemebene für P- und DFG-Prozesse jeweils nur genau eine Schnittstellenverhaltensbeschreibung und damit auch jeweils nur genau ein Implementierungstheorem.

Das Implementierungstheorem für P-Prozesse (8.1) beschreibt, wie die Implementierungstheoreme in Abschnitt 7.3.5, eine logische Verfeinerung für alle Programme in ESF. Während der Synthese müssen dazu wieder die Komponenten a, c, *out_init* und *var_init* des Theorems durch die entsprechenden Komponenten des konkreten Programms in ESF instantiiert werden.

$$
\begin{array}{l}
\vdash\ \forall a\ c\ \mathit{out\_init}\ \mathit{var\_init}. \\
\quad \mathsf{P\_IMP\_SYSTEM} \\
\quad\quad (\ \mathit{reset}, \mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}, \mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2}, \\
\quad\quad\ a, c, \mathit{out\_init}, \mathit{var\_init}\ ) \\
\quad \Rightarrow \\
\quad \mathsf{P\_IFC\_SYSTEM} \\
\quad\quad \mathsf{PROGRAM}\ \mathit{out\_init} \\
\quad\quad\quad \mathsf{LOCVAR}\ \mathit{var\_init} \\
\quad\quad\quad\quad \mathsf{WHILE}\ c\ (\mathsf{PARTIALIZE}\ a) \\
\quad\quad (\mathit{reset}, (\mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}), (\mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2}))
\end{array}
\tag{8.1}
$$

In Abbildung 8.1 wird die schematische Darstellung des Implementierungsmusters P_IMP_SYSTEM gezeigt. Auf eine Beschreibung des entsprechende Terms in Gropius-1 wurde aus Platzgründen verzichtet. Wie in Abschnitt 5.2.1 erwähnt, stellt das Schnittstellenverhaltensmuster P_IFC_RESET die Grundlage für das Kommunikationsschema auf der Systemebene dar. Die Struktur lässt sich demnach in zwei Teile gliedern. Der grau unterlegte Teil entspricht gerade dem Implementierungsmuster P_IMP_RESET, das in den Abbildungen 7.19(b), bzw. 7.20 gezeigt wurde. Neben den Signalen *start* und *ready*, die den Ein- und Ausgang des Kontrollers von P_IMP_RESET bilden, werden hier zwei zusätzliche Ausgangssignale aus diesem Kontroller herausgeführt, die in Abbildung 8.1 mit *old_ready* und *idle* bezeichnet sind. *old_ready* gibt dabei an, ob die Schaltung im letzten Takt aufnahmebereit war, und *idle* ist dann auf wahr gesetzt, wenn die Schaltung momentan nicht mit einer Berechnung beschäftigt ist. Ist dies der Fall, ist auch *ready* wahr. *ready* kann aber auch wahr werden, wenn die Schaltung gerade rechnet und im selben Takt die Schleifenbedingung c nicht mehr erfüllt ist.

Der obere Teil von Abbildung 8.1 zeigt den zusätzlichen Kontroller, der benötigt wird, um das spezielle Kommunikationsschema der Systemebene zu realisieren. Im Gegensatz zum Kontroller von P_IMP_RESET, der den Zustand speichert, ob die Schaltung mit der Berechnung des Programms fertig ist (Signal *ready*), speichert dieser Kontroller den Zustand, ob die Schaltung zur Aufnahme einer neuen Marke bereit ist (Signal *ready_1*). Der Unterschied zwischen diesen beiden Zuständen besteht darin, dass die Schaltung nicht nur die Berechnung beendet, sondern das Ergebnis auch an ihren Nachfolger übergeben haben muss, bevor eine neue Marke aufgenommen werden kann.

Ähnlich verhält es sich mit dem Implementierungstheorem (8.2) für DFG-Prozesse. Der gegebene DFG-Term wird, wie in Abschnitt 7.2.2 gezeigt, in einen äquivalenten DFG-Term der Form $(g^n \circ \mathsf{FU} \circ h^n) \circ (\mathsf{list\_o}\ (\mathsf{MAP}\ (\lambda(p,q).p \circ \mathsf{FU} \circ q)L)) \circ$

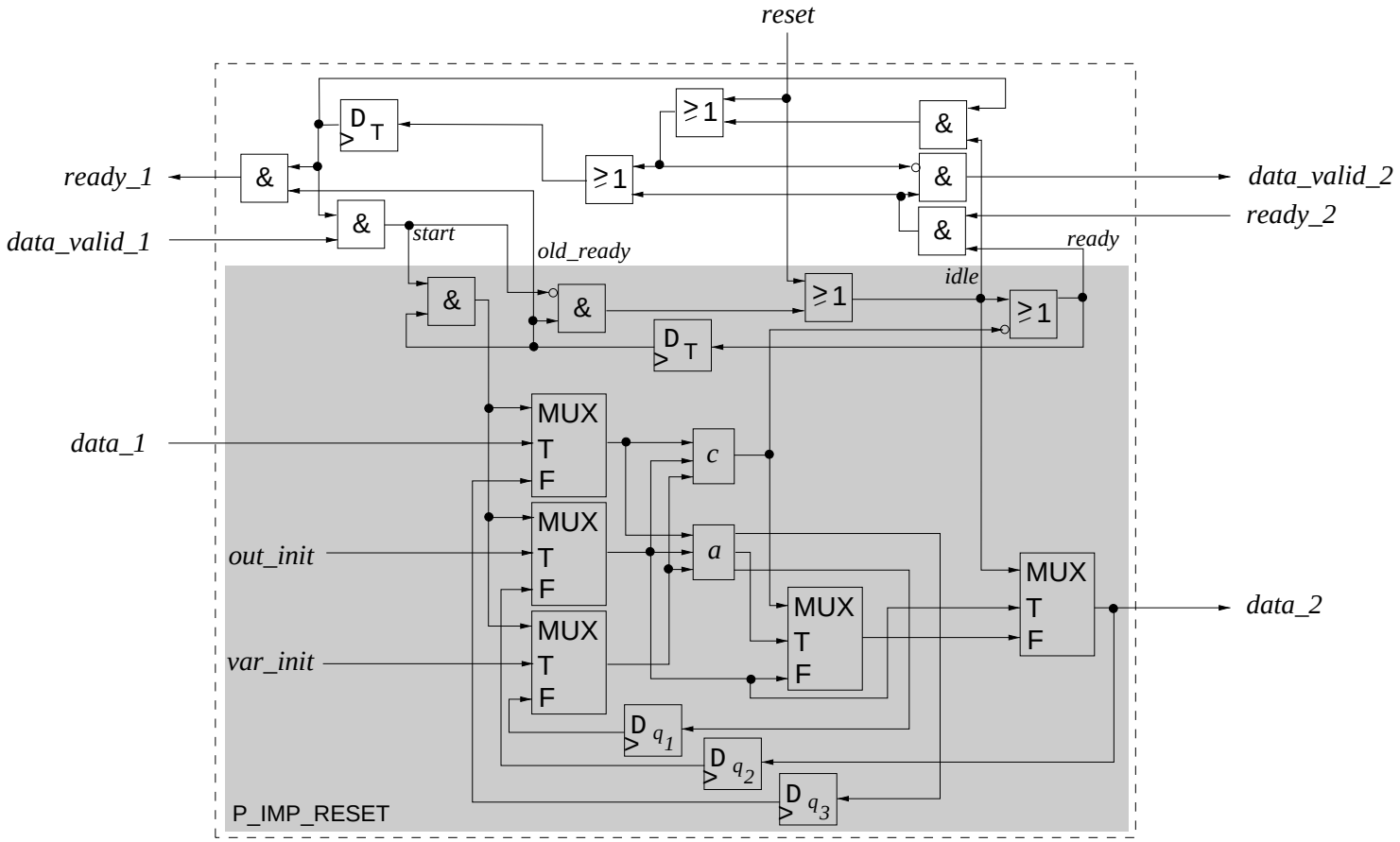


Abbildung 8.1: Implementierungsmuster P_IMP_SYSTEM für P-Prozesse auf der Systemebene

$(g^1 \circ \mathsf{FU} \circ h^1)$ umgewandelt. Anschließend kann Theorem (8.2) instantiiert werden.

$$
\begin{array}{l}
\vdash\ \forall\ \mathsf{FU}\ g^n\ h^n\ L\ g^1\ h^1. \\
\quad \mathsf{DFG\_IMP\_SYSTEM} \\
\qquad (\mathit{reset}, \mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}, \mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2}, \\
\qquad \mathsf{FU}, g^n, h^n, L, g^1, h^1) \\
\quad \Rightarrow \\
\quad \mathsf{DFG\_IFC\_SYSTEM} \\
\qquad ((g^n \circ \mathsf{FU} \circ h^n) \circ (\mathsf{concat}\ (\mathsf{MAP}\ (\lambda(p,q).p \circ \mathsf{FU} \circ q)L)) \circ (g^1 \circ \mathsf{FU} \circ h^1), \\
\qquad \mathsf{LENGTH}\ L + 2) \\
\qquad (\mathit{reset}, (\mathit{data\_1}, \mathit{data\_valid\_1}, \mathit{ready\_1}), (\mathit{data\_2}, \mathit{data\_valid\_2}, \mathit{ready\_2}))
\end{array}
\tag{8.2}
$$

Abbildung 8.2 zeigt das entsprechende Implementierungsmuster DFG_IMP_SYSTEM in schematischer Weise. Auch hier lässt sich die Struktur in zwei Teile gliedern. Der untere Teil entspricht dem Implementierungsmuster DFG_IMP_RESET (siehe Abbildung 7.13(b)), wobei wieder zwei zusätzliche Signale herausgeführt werden. Das Signal *idle* muss dabei durch einen zusätzlichen Baustein erst erzeugt werden. Der obere Teil ist identisch mit dem oberen Kontroller in Abbildung 8.1.

8.2 Synthese von K-Prozessen

Im Gegensatz zur Synthese von P- bzw. DFG-Prozessen, in deren Lauf verschiedene Aufgaben wie die zeitliche Einordnung von Operationen sowie die Bereitstellung und Zuordnung von Komponenten durchgeführt werden müssen, beschränkt sich die Synthese der K-Prozesse auf die Anwendung eines entsprechenden Implementierungstheorems. Für jeden der acht K-Prozesse wurde auf der RT-Ebene eine Implementierung gefunden und ein Theorem bewiesen, welches aussagt, dass diese Implementierung die Spezifikation des K-Prozesses erfüllt.

Diese Theoreme sollen hier nicht explizit aufgeführt werden. Stattdessen sind in den Abbildungen 8.3 bis 8.9 die Implementierungen der acht K-Prozesse schematisch dargestellt. Diese Implementierungen erfüllen die in den Abbildungen 6.7 bis 6.13 gezeigten Spezifikationen. Abbildung 8.6 zeigt die Implementierung für die zwei K-Prozesse Double und Split. Diese unterscheiden sich lediglich in dem unteren Datenteil. Während Double das gleiche Signal an beide Ausgänge *data_2* und *data_3* ausgibt, wird bei Split das Signal aufgeteilt, und jede Komponente wird in einem eigenen Register gespeichert und an einen der Ausgänge weitergegeben. Der Kontroller ist bei beiden Prozessen derselbe.

Die in Abbildung 8.9 gezeigte Implementierung des Prozesses Sink ist dergestalt, dass die beiden Eingangssignale *data_1* und *data_valid_1* nicht verwendet werden, und das Signal *ready_1* immer den Wert T aufweist. Dies hat zur Konsequenz, dass in den Prozessen, die über einen Kanal mit Sink verbunden sind, durch boolesche Umformungen einige Gatter und sogar der betreffende Kanal selbst eliminiert werden können. Abbildung 8.10 zeigt die restliche Implementierung eines Split-Prozesses, der über den ersten Ausgangskanal mit Sink verbunden war.

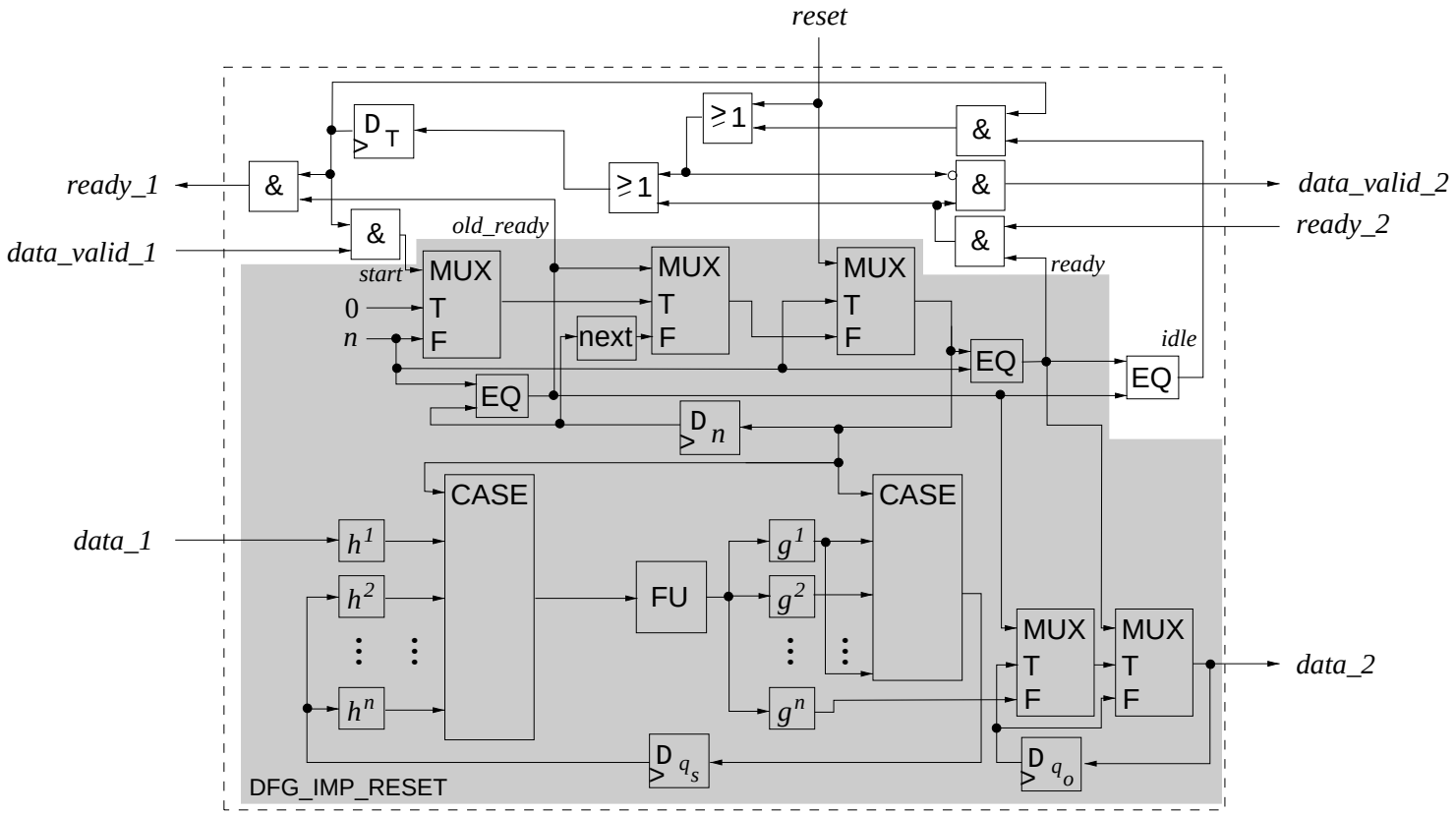


Abbildung 8.2: Implementierungsmuster DFG_IMP_SYSTEM für DFG-Prozesse auf der Systemebene

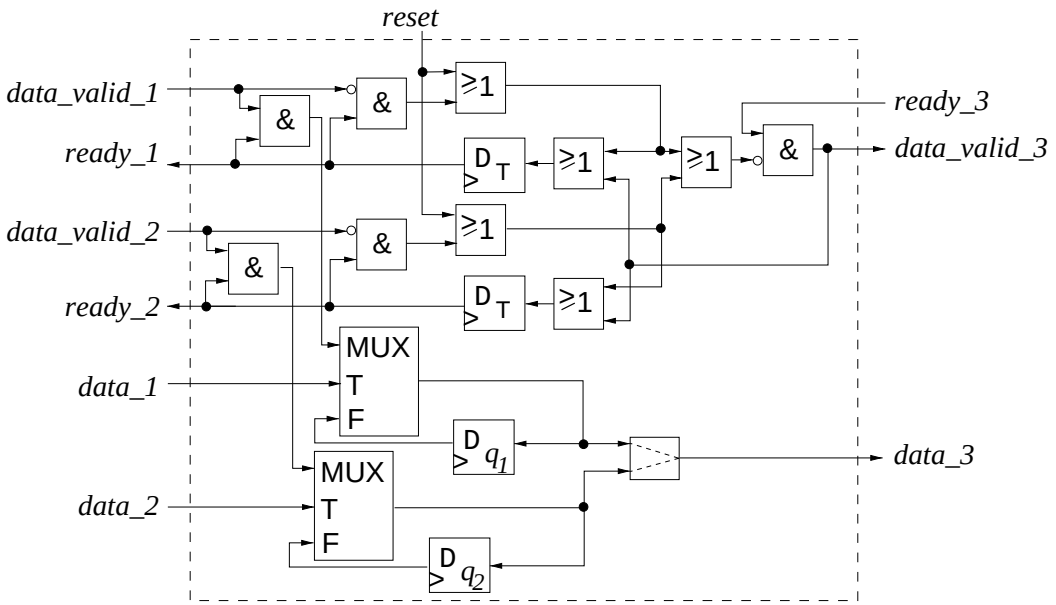


Abbildung 8.3: Implementierung für Join

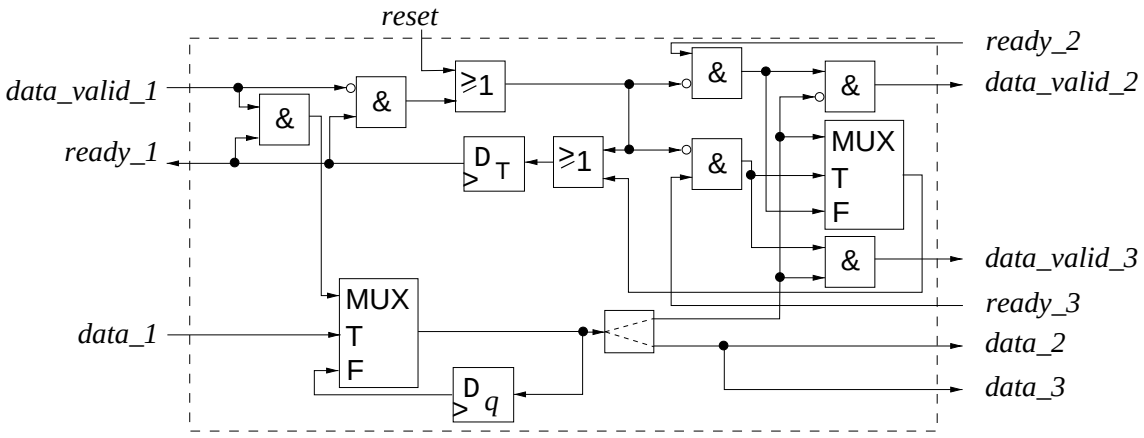


Abbildung 8.4: Implementierung für Fork

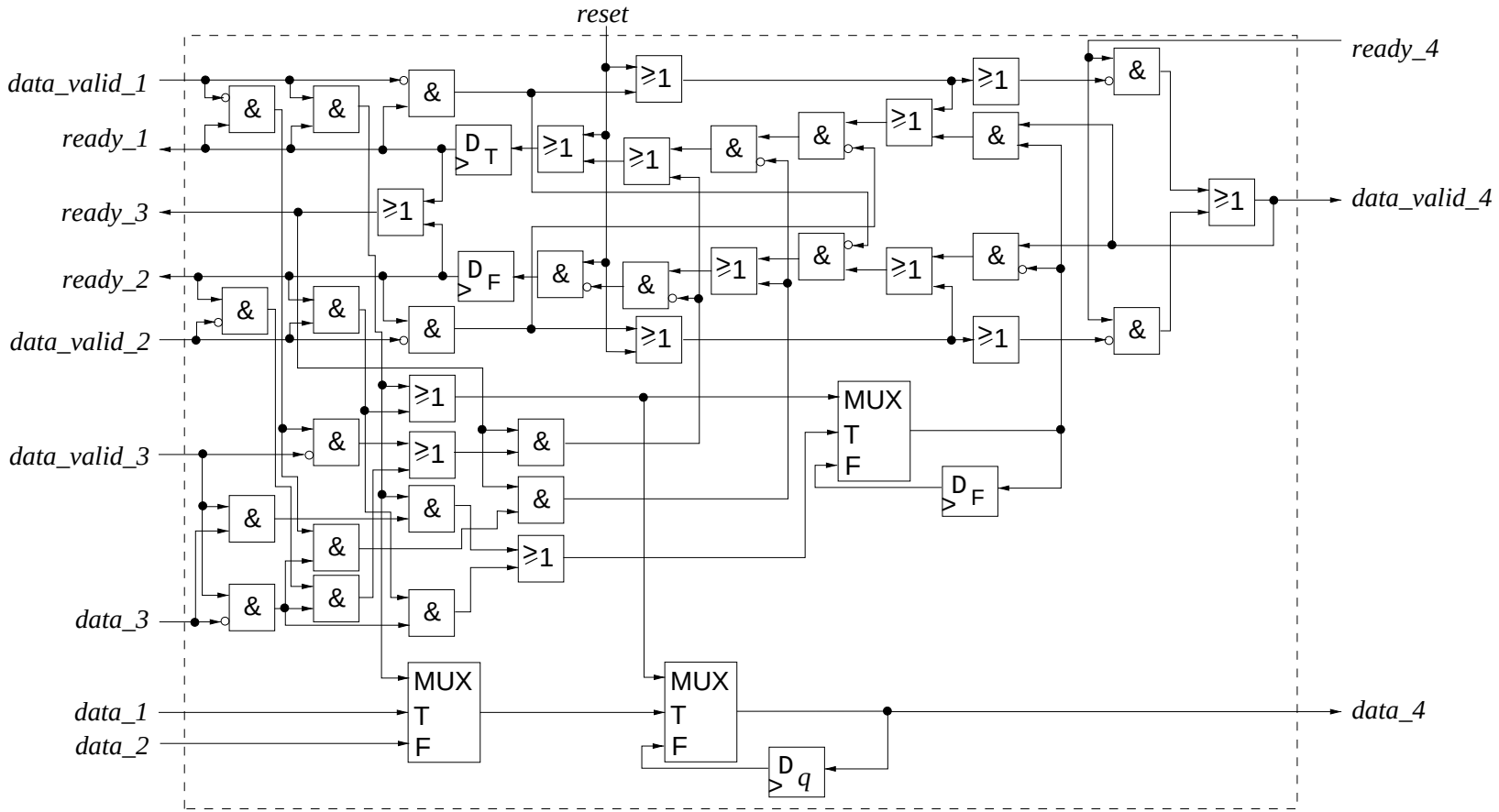


Abbildung 8.5: Implementierung für Choose

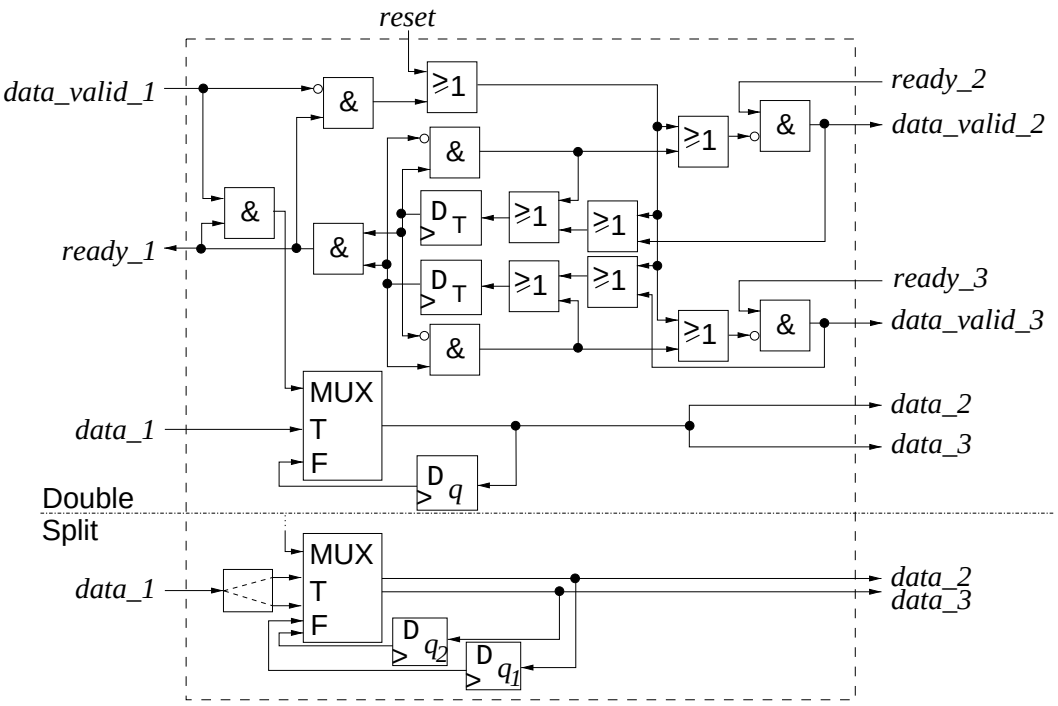


Abbildung 8.6: Implementierung für Double bzw. Split

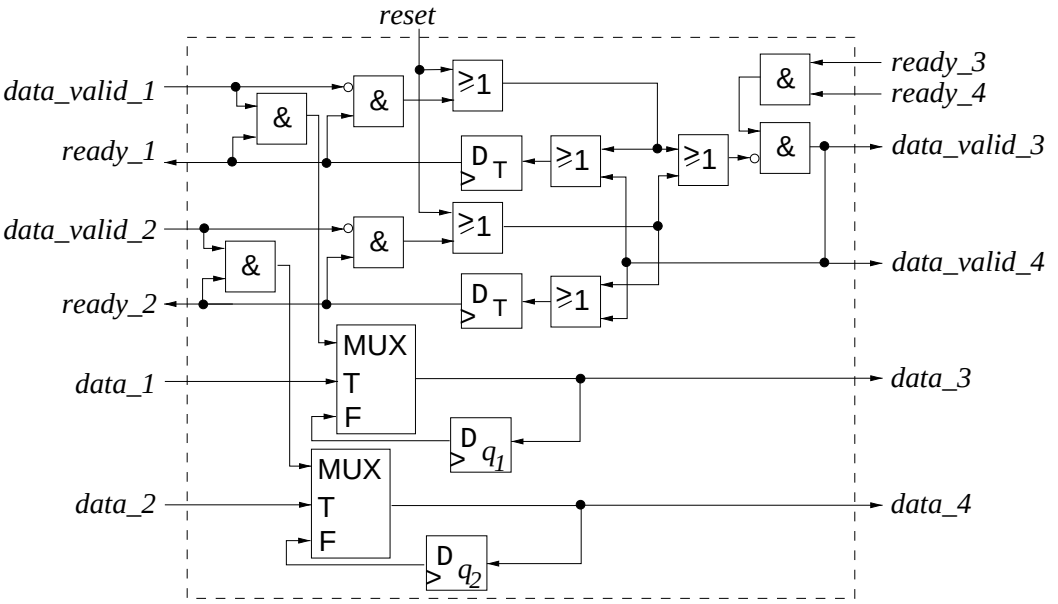


Abbildung 8.7: Implementierung für Synchronize

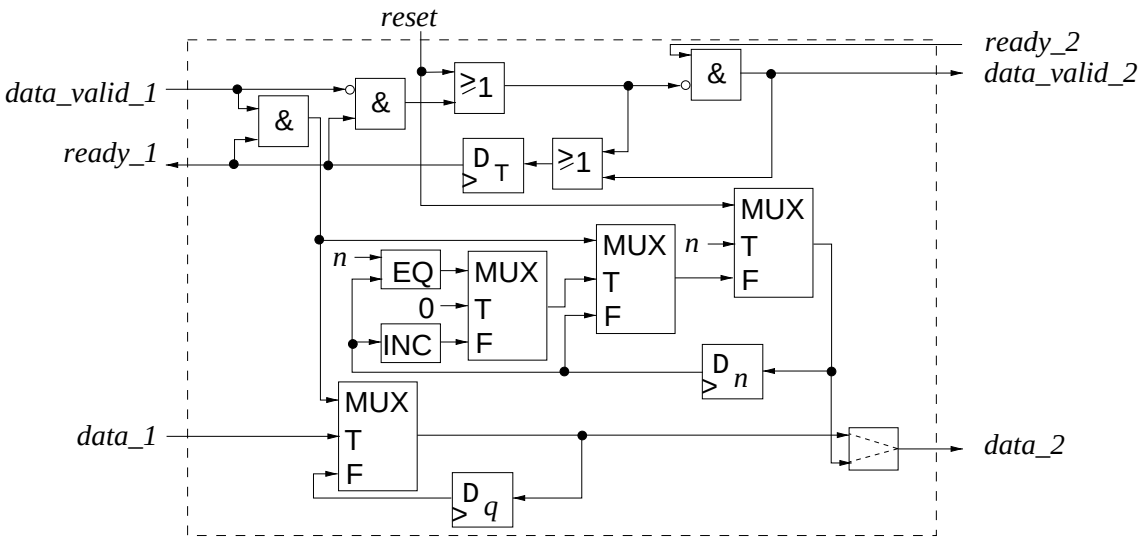


Abbildung 8.8: Implementierung für Counter

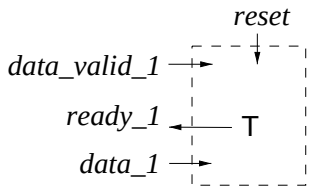


Abbildung 8.9: Implementierung für Sink

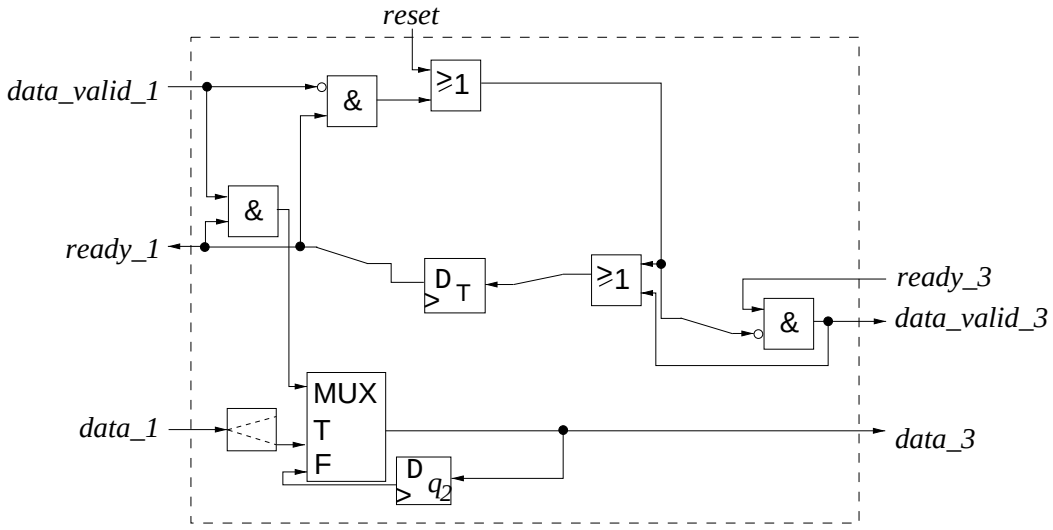


Abbildung 8.10: Restliche Implementierung von Split, wenn *Kanal_2* mit Sink verbunden war

8.3 Optimierungen

Durch die in den vorherigen Abschnitten vorgestellte Methode können alle Systeme auf die RT-Ebene überführt werden. Diese getrennte Synthese der einzelnen Prozesse hat den Nachteil, dass Optimierungen nur innerhalb der Prozesse durchgeführt werden können. Außerdem benötigt jeder Prozess auf der RT-Ebene seine eigenen Komponenten.

Ebenso wie auf der algorithmischen Ebene mit OPT und SPT kann auch auf der Systemebene eine zweistufige Synthese durchgeführt werden. Bevor die einzelnen Prozesse unabhängig voneinander synthetisiert werden, sind Optimierungen auf der Prozess-Struktur möglich. Im Weiteren sollen zwei Arten von Optimierungen betrachtet werden:

- Zusammenfassen von Prozessen
- Partitionieren von Prozessen

Die Vorteile dieser Optimierungen wurden in Abschnitt 3.1.1 erläutert. Wenn Prozesse zusammengefasst oder partitioniert werden sollen, ist zu beachten, dass sich damit i.Allg. das Zeitverhalten ändert. In Abschnitt 6.5 wurde dies bereits näher erläutert. Als Konsequenz ergibt sich, dass diese Transformationen auf der Basis der in Abschnitt 6.5 eingeführten funktionalen Semantik durchgeführt werden müssen. Eine Transformation auf Basis der Verhaltenssemantik ist nur möglich, wenn die betrachtete S-Struktur lediglich aus einem Zyklus besteht. Zur Beschreibung eines Zyklus' ist der K-Prozess Choose notwendig, der dafür sorgt, dass sich immer nur eine Marke im Inneren des Zyklus' befindet. Damit kann sich durch eine Transformation die Anzahl der umlaufenden Marken nicht verändern. Transformationen auf Basis der Verhaltenssemantik sind aber sehr aufwendig zu beweisen, was man schon an der Länge der Definitionen der Schnittstellenbeschreibungen erkennt. Sie sollen deshalb hier nicht weiter betrachtet werden.

Im Folgenden sind beispielhaft einige mögliche Theoreme für das Zusammenfassen und Partitionieren von P-Prozessen angegeben. Um diese Transformationstheoreme beschreiben zu können, werden zunächst einige abgeleitete Verdrahtungsstrukturen aufgeführt. In (5.24) und (5.25) auf Seite 65 wurden bereits die beiden Verdrahtungsstrukturen ASSOCVAR und SWAPVAR eingeführt, die die Grundlage für alle Verdrahtungsstrukturen darstellen. Die folgenden Strukturen lassen sich mit Hilfe von ASSOCVAR, SWAPVAR und den Kernkontrollstrukturen für P-Terme beschreiben.

$$\vdash \mathsf{REVASSOCVAR}\ A\ (h_1, h_2, h_3)\ =$$
$$\mathsf{SWAPVAR}\ (\mathsf{ASSOCVAR}\ (\mathsf{SWAPVAR}\ (\mathsf{ASSOCVAR}\ (\mathsf{SWAPVAR}\ A$$
$$((h_1, h_2), h_3))\ (h_3, h_1, h_2))\ ((h_3, h_1), h_2))\ (h_2, h_3, h_1))\ ((h_2, h_3), h_1)$$
$$\vdash \mathsf{RIGHTPAIR}\ A\ (h_1, h_2, h_3)\ =$$
$$\mathsf{ASSOCVAR}\ (\mathsf{RIGHTVAR}\ (\mathsf{SWAPVAR}\ A\ (h_3, h_2)))\ (h_1, h_2, h_3)$$

$\vdash$ OUTERVAR $A\ (h_1, h_2, h_3)\ =$
SWAPVAR (REVASSOCVAR (LEFTVAR (SWAPVAR $A\ (h_1, h_3)$)))
$(h_3, h_1, h_2))\ (h_3, (h_1, h_2))$
$\vdash$ FIRSTVAR $A\ (h_1, h_2, h_3, h_4)\ =$
REVASSOCVAR (LEFTVAR (OUTERVAR $A\ (h_1, h_2, h_3)$)) $((h_1, h_2), h_3, h_4)$
$\vdash$ SECONDVAR $A\ (h_1, h_2, h_3, h_4)\ =$
ASSOCVAR (RIGHTVAR (REVASSOCVAR (OUTERVAR $A\ (h_2, h_3, h_4)$)
$(h_2, h_3, h_4)))\ (h_1, h_2, (h_3, h_4))$

Die Funktion REVASSOCVAR wandelt einen Eingang mit Typ $(\alpha \times \beta \times \gamma)$ so um, dass auf ihn ein Block A mit Eingangstyp $((\alpha \times \beta) \times \gamma)$ angewandt werden kann. Diese Funktion stellt die Umkehrung zu ASSOCVAR dar. Die Funktion RIGHTPAIR passt den gleichen Eingang so an, dass ein Block A mit Eingangstyp $(\gamma \times \beta)$ angewandt werden kann, während OUTERVAR den Eingang zu $(\alpha \times \gamma)$ umwandelt. Die Funktionen FIRSTVAR und SECONDVAR gehen von einem Eingang mit Typ $((\alpha \times \beta) \times (\gamma \times \delta))$ aus. FIRSTVAR wandelt diesen um in $(\alpha \times \gamma)$, während SECONDVAR eine derartige Verdrahtung realisiert, dass ein Block mit Eingangstyp $(\beta \times \delta)$ angewandt werden kann. Natürlich ließen sich diese Verdrahtungsstrukturen auf die gleiche Weise definieren wie ASSOCVAR und SWAPVAR, nämlich durch Einführung lokaler Variablen. Der Vorteil, dass sie auf ASSOCVAR und SWAPVAR und die Kernkontrollstrukturen zurückgeführt werden, liegt darin, dass nur für diese Konstrukte Syntheseverfahren angeboten werden müssen. Die Theoreme (7.34) bis (7.36) auf Seite 141 beispielsweise zeigen, wie die Funktion ASSOCVAR bei der SPT beseitigt werden kann.

Mit Hilfe dieser Verdrahtungsstrukturen können nun Transformationstheoreme für Systeme abgeleitet werden. Die folgenden Theoreme beschreiben jeweils eine funktionale Äquivalenz $S_1 \approx S_2$ zwischen der Originalstruktur S_1 und der transformierten Struktur S_2. Die Information, welche Theoreme wann und wo in einer Systembeschreibung angewandt werden sollen, lässt sich auf Grund der Aufteilung zwischen Entwurfsraumuntersuchung und Transformation (siehe Abschnitt 7.1) durch konventionelle Verfahren [GVNG94, ELLS97] außerhalb der Logik bestimmen. Mit Hilfe der Theoreme (8.3) und (8.4) können zwei P-Prozesse zusammengefasst werden.

$$\begin{aligned}
&\vdash \exists r.\ \text{P_IFC_SYSTEM (PROGRAM } i_1\ A)\ (reset, a, r) \wedge \\
&\qquad \text{P_IFC_SYSTEM (PROGRAM } i_2\ B)\ (reset, r, b) \\
&\quad \approx \\
&\text{P_IFC_SYSTEM (PROGRAM } i_2\ (\text{LOCVAR } i_1 (\text{OUTERVAR } A\ (h_1, h_2, h_3) \\
&\qquad \text{THEN RIGHTPAIR } B\ (h_1, h_2, h_3)))))\ (reset, a, b)
\end{aligned} \tag{8.3}$$

$$
\begin{aligned}
\vdash\ & \exists r\ s.\ \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i_1\ A)\ (\mathit{reset}, a, r) \wedge \\
& \quad \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i_2\ B)\ (\mathit{reset}, b, s) \wedge \\
& \quad \mathsf{K\_IFC\_JOIN}\ (\mathit{reset}, r, s, c) \\
& \approx \\
& \exists r.\ \mathsf{K\_IFC\_JOIN}\ (\mathit{reset}, a, b, r) \wedge \\
& \quad \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ (i_1, i_2)\ ((\mathsf{FIRSTVAR}\ A\ (h_1, h_2, h_3, h_4))\ \mathsf{THEN} \\
& \qquad (\mathsf{SECONDVAR}\ B\ (h_1, h_2, h_3, h_4)))\ (\mathit{reset}, r, c)
\end{aligned}
\tag{8.4}
$$

In Theorem (8.3) werden zwei P-Prozesse, die hintereinander ausgeführt werden, zu einem einzigen P-Prozess zusammengefasst. Theorem (8.4) dagegen wandelt zwei parallel ausgeführte P-Prozesse, deren Ergebnisse über einen Join-Prozess kombiniert werden, in eine Struktur um, bei der zunächst die Eingänge über Join zusammengefasst und anschließend einem einzigen P-Prozess zugeführt werden.

Die umgekehrte Richtung, das Partitionieren von Prozessen, wird in den Theoremen (8.5) und (8.6) beschrieben.

$$
\begin{aligned}
\vdash\ & \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i\ (A\ \mathsf{THEN}\ B))\ (\mathit{reset}, a, b) \\
& \approx \\
& \exists r.\ \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ (i_x, i)\ ((\mathsf{PARTIALIZE}\ (\lambda(x, y, z).(x, x, z))) \\
& \qquad \mathsf{THEN}\ (\mathsf{RIGHTVAR}\ A)))\ (\mathit{reset}, a, r) \wedge \\
& \quad \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i\ ((\mathsf{LEFTVAR}\ B)\ \mathsf{THEN} \\
& \qquad (\mathsf{PARTIALIZE}\ (\lambda((x, y), z).((x, y), y)))))\ (\mathit{reset}, r, b) \\
\vdash\ & \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ (i_1, i_2)\ ((\mathsf{FIRSTVAR}\ A\ (h_1, h_2, h_3, h_4)) \\
& \qquad \mathsf{THEN}\ (\mathsf{SECONDVAR}\ B\ (h_1, h_2, h_3, h_4))))\ (\mathit{reset}, a, b) \\
& \approx \\
& \exists r\ s\ t\ u.\ \mathsf{K\_IFC\_SPLIT}\ (\mathit{reset}, a, r, s) \wedge \\
& \qquad \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i_1\ A)\ (\mathit{reset}, r, t) \wedge \\
& \qquad \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i_2\ B)\ (\mathit{reset}, s, u) \wedge \\
& \qquad \mathsf{K\_IFC\_JOIN}\ (\mathit{reset}, t, u, b)
\end{aligned}
\tag{8.5}
\tag{8.6}
$$

Mit Hilfe von Theorem (8.5) wird ein P-Prozess in eine Sequenz zweier P-Prozesse partitioniert. Theorem (8.6) schließlich beschreibt, wie ein P-Prozess in eine Struktur partitioniert werden kann, in der zwei P-Prozesse parallel ausgeführt werden.

Zum Schluss sollen noch zwei Transformationstheoreme für die K-Prozesse Choose und Counter vorgestellt werden. Obwohl in Abschnitt 6.5 deren funktionale Semantik nicht eindeutig beschrieben werden konnte, ist es dennoch möglich, Transformationen auf Basis der funktionalen Semantik durchzuführen.

$$\begin{array}{l} \vdash \exists r\ s.\ \mathsf{P\_IFC\_SYSTEM}\ A\ (reset, a, r) \wedge \\ \qquad\quad \mathsf{P\_IFC\_SYSTEM}\ A\ (reset, b, s) \wedge \\ \qquad\quad \mathsf{K\_IFC\_CHOOSE}\ (reset, r, s, c, d) \\ \quad \approx \\ \quad \exists r.\ \mathsf{K\_IFC\_CHOOSE}\ (reset, a, b, c, r) \wedge \\ \qquad \mathsf{P\_IFC\_SYSTEM}\ A\ (reset, r, d) \end{array} \tag{8.7}$$

Theorem (8.7) beschreibt, dass eine Struktur mit zwei gleichen P-Prozessen, die mit Choose über dessen ersten bzw. zweiten Eingangskanal verbunden sind, äquivalent ist einer Struktur mit jenem P-Prozess, der mit Choose über dessen Ausgangskanal verbunden ist. Gleiche Prozesse an den beiden ersten Eingängen von Choose können demnach über Choose hinweg geschoben werden, womit nur noch eine Instanz des Prozesses erforderlich ist.

$$\begin{array}{l} \vdash \exists r.\ \mathsf{P\_IFC\_SYSTEM}\ (\mathsf{PROGRAM}\ i\ A)\ (reset, a, r) \wedge \\ \qquad \mathsf{K\_IFC\_COUNTER}\ n\ (reset, r, b) \\ \quad \approx \\ \quad \exists r.\ \mathsf{K\_IFC\_COUNTER}\ n\ (reset, a, r) \wedge \\ \qquad \mathsf{P\_IFC\_SYSTEM} \\ \qquad\quad \mathsf{PROGRAM}\ (i, 0) \\ \qquad\qquad \mathsf{PARTIALIZE}\ (\lambda((x, m), (y, h)).((x, m), (y, m)) \\ \qquad\qquad \mathsf{THEN}\ (\mathsf{FIRSTVAR}\ A\ (h_1, h_2, h_3, h_4))\ (reset, r, b) \end{array} \tag{8.8}$$

Theorem (8.8) beschreibt eine Transformation, bei der ein P-Prozess über Counter hinweg geschoben wird. Der P-Prozess muss dabei aber leicht modifiziert werden. Diese Transformation erlaubt es, den P-Prozess anschließend mit einem weiteren Prozess zusammenzufassen, der bereits hinter Counter liegt.

Kapitel 9

Experimentelle Ergebnisse

Die Anwendbarkeit eines Verfahrens der Formalen Synthese hängt in starkem Maße davon ab, wie effizient der Syntheseprozess durchgeführt werden kann. Der Effizienzbegriff muss dabei von zwei Seiten betrachtet werden. Zum einen ist es erforderlich, effiziente Schaltungsimplementierungen hinsichtlich des Zeitverhaltens, Flächenbedarfs etc. zu erzielen. In den beiden letzten Kapiteln wurde gezeigt, dass dies nicht zuletzt durch die Trennung zwischen Entwurfsraumuntersuchung und logischer Transformation möglich wird. Indem Verfahren eingebunden werden können, die außerhalb der Logik den Entwurfsraum untersuchen und somit für die Qualität der Synthese sorgen, steht das Resultat einer Formalen Synthese i.Allg. dem einer konventionellen Synthese nicht nach.

Eine effiziente Synthese bedeutet aber auf der anderen Seite auch, dass die Dauer des Syntheseprozesses an sich möglichst kurz sein soll. Im Folgenden soll daher vor allem die Laufzeiteffizienz betrachtet werden. Im Abschnitt 9.2 werden dazu beispielhafte experimentelle Ergebnisse vorgestellt.

In Abschnitt 7.2.1 wurde bereits gezeigt, wie wichtig es ist, bei der logischen Transformation sich Gedanken über die Reihenfolge der Anwendung der elementaren Regeln zu machen. In Abbildung 7.6 auf Seite 121 wurde dies drastisch anhand des Syntheseschritts der zeitlichen Einordnung demonstriert.

Neben diesen Maßnahmen ist aber auch entscheidend, wie effizient der zugrunde liegende Theorembeweiser ist. Es soll daher zunächst im nächsten Abschnitt eine Modifikation des Theorembeweisers HOL vorgestellt werden, die aus einer kritischen Betrachtung hinsichtlich der Effizienz erwachsen ist.

9.1 Modifikation des Theorembeweisers

Alle in den vorangegangenen Abschnitten vorgestellten Transformationen basieren auf dem Theorembeweiser HOL. Deshalb wurde untersucht, wie dieser zugrunde liegende Theorembeweiser für die Formale Synthese optimiert werden kann, d.h. wie die benötigten Grundregeln effizienter implementiert werden können [BlEi97, BlES99]. Man könnte nun fragen, ob nicht ein anderer Theorembeweiser von vorneherein

besser geeignet wäre. Die Auswahl von HOL als Grundlage für den in dieser Arbeit vorgestellten Formalismus geschah aber nicht zufällig. HOL basiert auf einem streng-typisierten prädikatenlogischen Kalkül höherer Ordnung, der zur Formalisierung auf den höheren Abstraktionsebenen unumgänglich ist, und der sich für die Verifikation von Hardwarebeschreibungen besonders gut eignet [Gord86]. Darüber hinaus bietet er durch Beschränkung auf einen sehr kleinen Kern und durch eine ausgereifte Konsistenzsicherung ein großes Maß an Sicherheit, dass die entworfene Schaltung korrekt ist. Andere modernen Theorembeweiser wie etwa PVS [OwSR93] verfügen über solche Konzepte nicht.

Es wurde eine modifizierte Version von HOL implementiert, die im Folgenden HOL+ genannt werden soll. HOL+ unterscheidet sich in zwei Gesichtspunkten von HOL: Zum einen wurde die Termrepräsentation geändert, zum anderen wurden zwei neue Grundregeln hinzugefügt.

9.1.1 Veränderung der Termrepräsentation

Im Theorembeweiser HOL wird zur Implementierung von logischen Termen eine sogenannte *deBrujin*-Termrepräsentation gewählt [Bare92]. Dies bedeutet, dass freie und gebundene Variablen in unterschiedlicher Weise gespeichert werden. Während freie Variablen mit ihrem Namen und Typ repräsentiert werden, wird zur Repräsentation von gebundenen Variablen lediglich eine natürliche Zahl verwendet. Diese Zahl gibt an, um wie viele Schachtelungstiefen von λ-Abstraktionen die korrespondierende λ-Abstraktion von der gebundenen Variable entfernt ist. Betrachtet man beispielsweise den Term

$$\lambda x.(\lambda y.x + y + z),$$

dann werden in dem Teilterm $x + y + z$ die beiden gebundenen Variablen x und y intern mit den Zahlen 1 bzw. 0 gespeichert.

Der Vorteil dieser Art von Termrepräsentation liegt darin, dass die α-Äquivalenz zweier Terme in linearer Zeit überprüft werden kann. Zwei Terme sind α-äquivalent, wenn sie sich nur in den Namen ihrer gebundenen Variablen unterscheiden (siehe Abschnitt 2.1.2). Aus diesem Grunde müssen zur Überprüfung zweier Terme auf α-Äquivalenz nur die Typen der Parameter der λ-Abstraktionen und die Zahlen, die die gebundenen Variablen repräsentieren, auf Gleichheit überprüft werden.

Auf der anderen Seite hat diese Art von Termrepräsentation für Beschreibungen von (großen) Schaltungen einen erheblichen Nachteil. Schaltungsbeschreibungen zeichnen sich dadurch aus, dass gewöhnlich lokale Variablen mit einem großen Bindungsbereich vorkommen. Beispiele dafür sind die lokalen Variablen innerhalb von DFG-Termen, die zur Speicherung von Zwischenergebnissen verwendet werden. Aber auch in anderen Formalisierungsansätzen, die eine relationale Schaltungsbeschreibung verwenden, taucht dieses Problem auf. So werden dort etwa interne Signale existenzquantifiziert, was gerade im Hinblick auf große Schaltungsbeschreibungen zu erheblichen Bindungsbereichen führt. Das Problem der großen Bindungsbereiche liegt darin, dass während der Erzeugung und Zerlegung eines Terms zwi-

schen freien und gebundenen Variablen hin- und hergewechselt werden muss, und somit ein quadratischer Aufwand beim Erzeugen und Zerlegen und deshalb auch beim Transformieren von Termen entsteht.

Aus diesem Grunde wurde der Kern von HOL mit einer anderen Termrepräsentation reimplementiert. In diesem modifizierten Theorembeweiser HOL+ werden freie und gebundene Variablen in gleicher Weise mit ihrem Namen und dem Typ gespeichert. Diese Termrepräsentation wird im Fachjargon als "name-carrying" bezeichnet. Damit wird der quadratische Aufwand vermieden, und die Terme lassen sich in einer wesentlich effizienteren Weise transformieren. Andererseits wird dadurch die Überprüfung auf α-Äquivalenz etwas ineffizienter, was auf die Formale Synthese aber keinen gravierenden Einfluss hat. Als zweiter Nachteil ist ein erhöhter Speicherbedarf für Terme zu nennen, da für gebundene Variablen anstatt einer natürlichen Zahl nun ein Name und ein Typ gespeichert werden müssen.

9.1.2 Hinzufügen effizienterer Regeln

Eine wichtige Grundregel des λ-Kalküls, die auch für die Transformationen in dieser Arbeit eine entscheidende Rolle spielt, ist die β-Konversion (siehe Abschnitt 2.1.2). Für einen gegebenen Term erlaubt das HOL-System immer nur eine einzige β-Konversion gleichzeitig. Sollen mehrere β-Redizes aufgelöst werden, muss der Term für jede einzelne β-Konversion einmal durchlaufen werden.

Da bei der Transformation von Schaltungsbeschreibungen in Gropius in erheblicher Menge β-Konversionen anfallen, würde dies zu einem sehr ungünstigen Verhalten führen. Es wurde daher in HOL+ eine zusätzliche Regel aufgenommen, die in einem einzigen Traversierungsschritt mehrere β-Konversionen gleichzeitig durchführen kann. Darüber hinaus ist diese verbesserte β-Konversion mit einer Filterfunktion ausgestattet, die darüber entscheidet, welche β-Redizes eines Terms aufgelöst werden sollen und welche nicht.

Neben einfachen β-Konversionen fallen auch oft gepaarte β-Konversionen an. Ein gepaarter β-Redex $(\lambda(x,y).t)\ (a,b)$ wird dabei umgeformt in einen Term $t\,[a,b/x,y]$, in dem im Abstraktionsrumpf t die Variablen x bzw. y durch die Terme a bzw. b substituiert wurden. Die sogenannte gepaarte β-Konversion ist aber nicht auf Paare als Parameter für eine λ-Abstraktion beschränkt, sondern kann auf β-Redizes mit beliebig verschachtelten Tupeln angewandt werden. Beispielsweise wird der Term $(\lambda((a,b),(c,d)).a+b+c+d)\ ((1,2),(3,4))$ konvertiert zu dem Ausdruck $1+2+3+4$. Im Theorembeweiser HOL wird eine β-Konversion mit einem n-Tupel in n Traversierungsschritten durch den Term durchgeführt. Dies liegt daran, dass gepaarte β-Redizes mit Hilfe der in Abschnitt 2.6.1 vorgestellten Funktion UNCURRY repräsentiert werden. Um die Konversion von gepaarten β-Redizes zu beschleunigen, wurde eine weitere neue Funktion eingeführt, die einen gepaarten β-Redex in mehrere verschachtelte einfache β-Redizes umwandelt. So wird etwa der Term $(\lambda((a,b),(c,d)).a+b+c+d)\ ((1,2),(3,4))$ konvertiert zu dem Ausdruck $(\lambda a\ b\ c\ d.a+b+c+d)\ 1\ 2\ 3\ 4$ (siehe dazu auch die Konvention zur Klammerung auf Seite 7). Um diese Transformation durchzuführen, muss nicht der ge-

samte Term durchlaufen werden. Der Rumpf der gepaarten λ-Abstraktion bleibt unverändert. Anschließend können alle β-Redizes durch die oben beschriebene verbesserte β-Konversion in einem Durchlauf beseitigt werden. Je nachdem, welche Filterfunktion eingesetzt wird, ist es zudem auch möglich, die β-Redizes im Rumpf zu eliminieren. Ein besonderes Charakteristikum der Konversion von einem gepaarten β-Redex in eine ungepaarte Darstellung ist, dass diese Umwandlung auch stattfindet, wenn der Operand nicht in genügender Weise gepaart ist. Es werden dabei in geeigneter Weise die Konstanten FST und SND eingeführt. Der Term $(\lambda((a,b),(c,d)).a+b+c+d)\ ((m,n),p)$ etwa wird auf diese Weise in den Ausdruck $(\lambda a\ b\ c\ d.a+b+c+d)\ m\ n\ (\mathsf{FST}\,p)\ (\mathsf{SND}\,p)$ umgewandelt. Somit kann diese Konversion auf beliebige β-Redizes angewandt werden.

Diese hinzugefügten Regeln sind so allgemeiner Natur, dass sie auch für andere Benutzer des Theorembeweisers HOL von Interesse sein dürften. Allerdings hat die Erweiterung des Kerns einen erheblichen Einfluss auf die Sicherheit des Theorembeweisers. Wird der Kern erweitert, nimmt die Glaubwürdigkeit für seine Korrektheit um einiges ab. Bei der Implementierung dieser zusätzlichen Regeln muss daher mit größter Sorgfalt vorgegangen werden. Aus diesem Grunde sollte man bei der Erweiterung des Kerns sehr restriktiv sein. Ein Vorteil der hier beschriebenen Erweiterung ist jedoch die Tatsache, dass die erste der vorgestellten Regeln die Verallgemeinerung der bereits im HOL-Kern vorhandenen Regel der β-Konversion darstellt. Diese vorhandene Regel kann somit durch die allgemeinere ersetzt werden, wodurch sich die Anzahl der elementaren Regeln nur um eins erhöht.

9.2 Experimente für OPT, SPT und Schnittstellensynthese

In Kapitel 7.3 wurde die Synthese von algorithmischen P-Schaltungsbeschreibungen vorgestellt. Es wurde gezeigt, wie mit Programmtransformationen eine abstrakte Schaltungsbeschreibung auf die RT-Ebene abgebildet werden kann. Die Synthese gliedert sich dabei in mehrere Schritte. Während der OPT wird ein gegebenes Programm in ein äquivalentes, aber hinsichtlich einer bestimmten Kostenfunktion günstigeres Programm transformiert. Es gibt zahlreiche Möglichkeiten für eine solche Optimierung. In Abschnitt 7.3.4 wurden verschiedene Programmtransformationstheoreme dazu vorgestellt. Um zu einem kostenoptimalen Programm zu gelangen, muss dazu in geeigneter Weise der Entwurfsraum untersucht werden, wie es in Abschnitt 7.1 angedeutet wurde. Ziel dieser Arbeit war es jedoch, eine Methode zu entwickeln, um eine Formale Synthese innerhalb eines Theorembeweisers durchführen zu können, die zu garantiert korrekten Ergebnissen führt. Die Entwicklung von Heuristiken zur Entwurfsraumuntersuchung geht über diesen Rahmen weit hinaus. Zwar ist es möglich, bereits vorhandene Syntheseverfahren einzubinden; da aber in dieser Arbeit ein Synthesekonzept entwickelt wurde, das von den gängigen Verfahren nur partiell unterstützt wird, ist diese Vorgehensweise nur eingeschränkt möglich. Im Rahmen des DFG-Projektes Schm 623/6-2 wird aber bereits

an Entwurfsraumuntersuchungsmethoden gearbeitet, die auf das in dieser Arbeit vorgestellte Vorgehen abgestimmt sind.

Aus diesen Gründen ist es nicht sinnvoll, für beliebige Programme eine Formale Synthese durchzuführen und deren Ergebnis mit dem von herkömmlichen Syntheseverfahren zu vergleichen. Anhand des laufenden Beispiels in Abschnitt 7.3 kann man erkennen, dass unterschiedlichste Implementierungen mit dem vorgestellten Verfahren erzielt werden können. Wie in Kapitel 3.4.4 erläutert, beschäftigen sich die meisten anderen Ansätze für eine Formale Synthese mit den unteren Abstraktionsebenen. Die wenigen Autoren, die sich mit einer Formalen Synthese auf den höheren Abstraktionsebenen beschäftigt haben, geben keine Auskünfte über Experimente bezüglich der Laufzeiteffizienz ihrer Ansätze. Aus diesem Grunde konnte auch kein experimenteller Vergleich mit solchen Ansätzen durchgeführt werden.

Es sollen daher zur Verdeutlichung der Leistungsfähigkeit des vorgestellten Ansatzes verschiedene Experimente vorgestellt werden. (Alle Experimente wurden auf einer SUN UltraCreator mit Solaris 5.5.1 und 196 MB Hauptspeicher durchgeführt.)

Das erste Experiment bezieht sich auf die OPT und betrachtet die Transformation des Schleifenauftrennens. Bevor das auf Seite 144 eingeführte Theorem (7.39) zum Schleifenauftrennen angewandt werden kann, muss der Rumpf der entsprechenden Schleife in eine Kombination mehrerer DFG-Terme unterteilt worden sein. Dies entspricht dem Problem der zeitlichen Einordnung von Operationen in Datenflussgraphen. Es sollen daher im folgenden zwei skalierbare DFG-Terme betrachtet werden, die mit Hilfe verschiedener Entwurfsraumuntersuchungsmethoden unterteilt werden.

Als erstes Beispiel dient der bereits in Abschnitt 7.2 vorgestellte DFG-Term zur Berechnung einer Polynomdivision. Der DFG-Term ist skalierbar in den Parametern p und q, wobei p den Grad des Zählerpolynoms und $p+q$ den Grad des Nennerpolynoms bezeichnen. Der DFG-Term besteht aus $p+q$ Subtraktionsoperationen, $p(q+1)$ Multiplikationen und $q(p-1)$ Additionen. Somit gibt es eine Gesamtzahl von $2p(q+1)$ Operationen. Der kritische Pfad hat eine Länge von $3q+2$ Operationen.

Als zweites Beispiel soll ein DFG-Term zur Berechnung der diskreten Cosinustransformation (DCT) betrachtet werden. Die DCT wird gewöhnlich zur Kompression von Bildern eingesetzt. Die DCT eines Bildes mit den Pixeln $x(n,m)$ ist definiert durch:

$$X(u,v) = \frac{2}{\sqrt{N \cdot M}} \cdot c(u) \cdot c(v) \cdot \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x(n,m) \cdot \mathsf{cos}[\frac{\pi \cdot u}{2N} \cdot (2n+1)] \cdot \mathsf{cos}[\frac{\pi \cdot v}{2M} \cdot (2m+1)]$$

$$\text{mit} \qquad c(0) = \frac{1}{\sqrt{2}} \qquad \text{und} \qquad c(u), c(v) = 1 \text{ für } u, v > 0$$

Eine ausführliche Beschreibung dieses Datenflussgraphen findet sich in [BlEK96]. Im Folgenden sei angenommen, dass nur Bilder quadratischen Umfangs vorliegen und somit $N = M$ gilt. Unter dieser Annahme verfügt der DFG-Term über $2N^3 -$

$N^2 - N$ Additionen und $2N^3 - N + 2$ Multiplikationen, also über $4N^3 - N^2 - 2N + 2$ Operationen. Der kritische Pfad hat bei diesem DFG-Term eine Länge von $2N + 1$.

In Abbildung 9.1 sind die Laufzeiten angegeben, die erforderlich sind, um einen DFG-Term für die Polynomdivision in eine Kombination von DFG-Termen zu unterteilen.

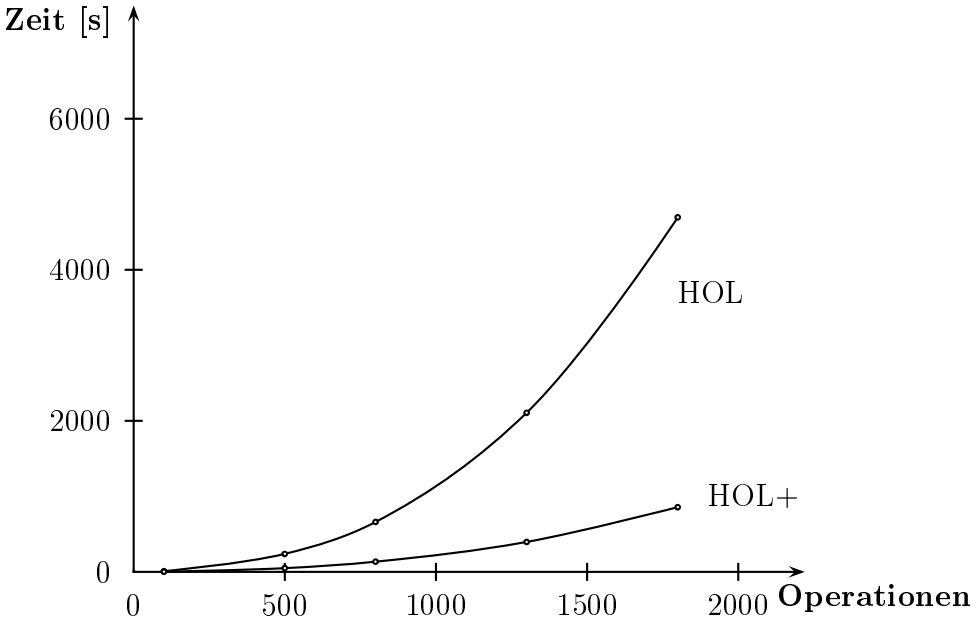


Abbildung 9.1: Laufzeitvergleich für die logische Transformation bei der Polynomdivision

Der Parameter p wurde dabei zu 25 gesetzt und der Parameter q stufenweise erhöht. Die obere Kurve zeigt dabei die Laufzeiten für eine Transformation im HOL-System, während die untere Kurve die Laufzeiten im veränderten Theorembeweiser HOL+ angibt. Wie man sieht, lässt sich der zeitliche Aufwand durch die Verwendung von HOL+ erheblich reduzieren.

Das gilt auch für die Laufzeiten, um DFG-Terme der DCT zu transformieren (siehe Abbildung 9.2). Hier wurde der Parameter N von 2 bis 7 erhöht. Wie man erkennt, ergibt sich bei der Transformation von DFG-Termen der DCT bei gleicher Anzahl der Operationen eine bei weitem geringere Laufzeit. Dies liegt an der verschiedenen Struktur der jeweiligen Datenflussgraphen. Sie unterscheiden sich sowohl hinsichtlich der Tiefe (Länge des kritischen Pfades) als auch in der Anzahl von mehrfach verwendeten Zwischenergebnissen. Beide Kriterien sind bei der Polynomdivision stärker ausgeprägt und führen damit zu einem höheren Zeitverbrauch.

In den Experimenten in den Abbildungen 9.1 und 9.2 wurde jeweils die Transformation ausgehend von einer bestimmten Steuerinformation der Entwurfsraumuntersuchung durchgeführt. Interessant ist aber, wie die Transformation im Theorembeweiser im zeitlichen Verhältnis steht zur Entwurfsraumuntersuchung außerhalb

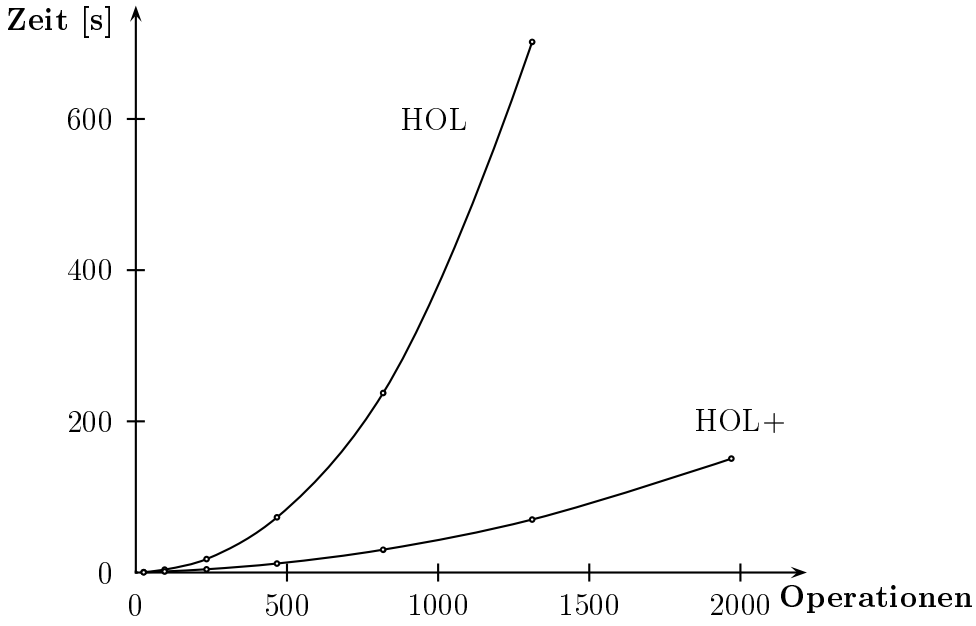


Abbildung 9.2: Laufzeitvergleich für die logische Transformation bei der DCT

der Logik. Damit verbunden ist die Frage nach dem Mehraufwand für eine Formale Synthese gegenüber einer konventionellen Synthese ohne formale Absicherung.

In Abbildung 9.3 ist dies gezeigt für die Polynomdivision und in Abbildung 9.4 für die DCT. Es wurden zur Entwurfsraumuntersuchung zwei verschiedene Verfahren angewandt. Ein sehr schneller Algorithmus, der allerdings oft auch nicht die besten Ergebnisse erzielt, ist der ALAP (as-late-as-possible), der die Operationen gemäß ihrer Datenabhängigkeiten so spät wie möglich einem Kontrollschritt zuteilt. Eine andere Methode, die aufwendiger ist, aber in der Regel bessere Ergebnisse liefert, ist das "force-directed scheduling". Hier wird versucht, in Analogie zu den Federkräften eines mechanischen Modells die Operationen möglichst gleichmäßig auf die Kontrollschritte zu verteilen.

Wie man an den Abbildungen erkennen kann, steht das Verhältnis zwischen dem Aufwand für die Entwurfsraumuntersuchung und dem für die Transformation in einem annehmbaren Rahmen (für die logische Transformation wurde der Theorembeweiser HOL+ verwendet). Die Laufzeiten für die Transformation sind unabhängig von der eingesetzten Entwurfsraumuntersuchungsmethode. Bei der Verwendung aufwendiger Entwurfsraumuntersuchungsmethoden kann der Mehraufwand für die logische Transformation sogar vernachlässigbar gering sein, wie man in Abbildung 9.3 bei der Verwendung des force-directed scheduling erkennt. Aber auch bei der Transformation der DCT erkennt man, dass die Komplexität des force-directed scheduling eine höhere ist als die der Transformation, sodass bei noch größeren DFG-Termen ein noch günstigeres Verhältnis zu erwarten ist. Man erkennt weiterhin, dass selbst

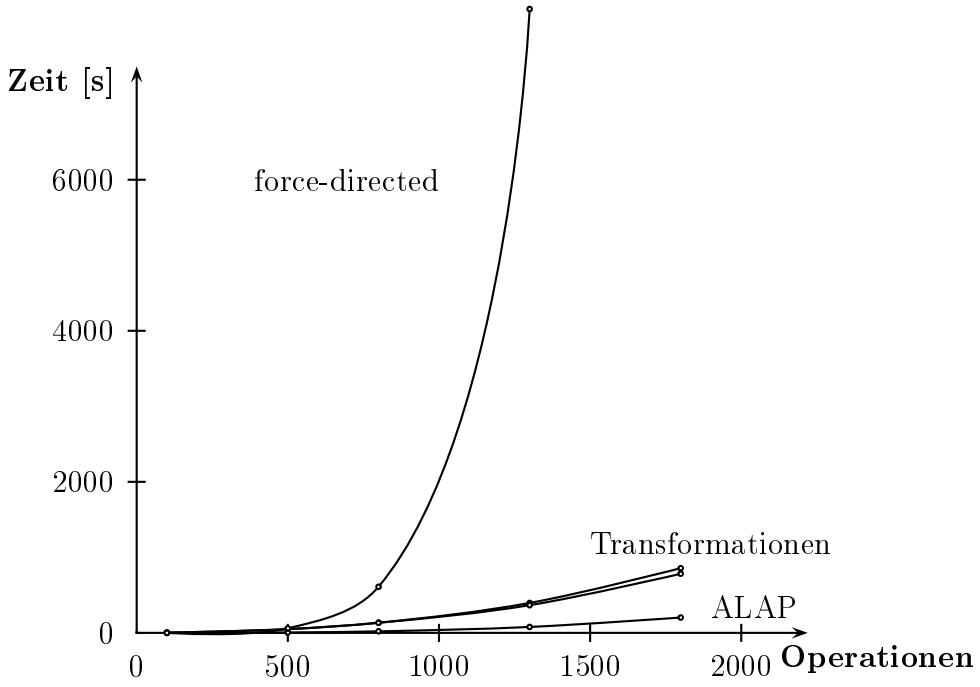


Abbildung 9.3: Laufzeit für Heuristik und Transformation bei der Polynomdivision

bei der Verwendung sehr einfacher Verfahren zur Entwurfsraumuntersuchung (wie dem ALAP-Algorithmus) der Mehraufwand für die Formale Synthese vertretbar ist. Man muss sich dabei vor Augen halten, dass der Mehraufwand für eine vollständige Simulation oder für eine Postsynthese-Verifikation mit interaktivem Theorembeweisen erheblich höher ausfallen würde.

Das zweite Experiment bezieht sich auf die SPT und die Schnittstellensynthese. Während dieser Transformationen wird eine Termersetzung mit vorgegebenen Theoremen durchgeführt. Aus diesem Grunde lassen sich die SPT und die Schnittstellensynthese vollständig automatisieren. In Tabelle 9.1 werden für beide Transformationen die Laufzeiten für verschiedene Programme gezeigt, angefangen beim Programm zur Berechnung des größten gemeinsamen Teilers (gcd), der sehr einfach ist und nur über sehr wenige Kontrollstrukturen verfügt, bis hin zum Programm für den Kalman-Filter, der aus vielen, verschachtelten Kontrollstrukturen besteht. Die Codes dieser Programme in der Programmiersprache C finden sich in [cdes].

Die Kosten für die SPT wachsen hauptsächlich mit der Anzahl der Kontrollstrukturen, aber auch mit der Größe der DFG-Terme des Programms. Die Anzahl der Kontrollstrukturen ist dafür verantwortlich, mit wie vielen der in Abschnitt 7.3.3 vorgestellten Theoremen eine Termersetzung durchgeführt werden muss. Da aber nach der Anwendung eines jeden Theorems eine β-Konversion in den DFG-Termen durchgeführt werden muss, um diese anzupassen, spielt auch die Größe der DFG-Terme eine entscheidende Rolle. Die Laufzeiten in Tabelle 9.1 wurden mit dem Theorembeweiser HOL+ erzielt. Die Verwendung von HOL würde aufgrund der

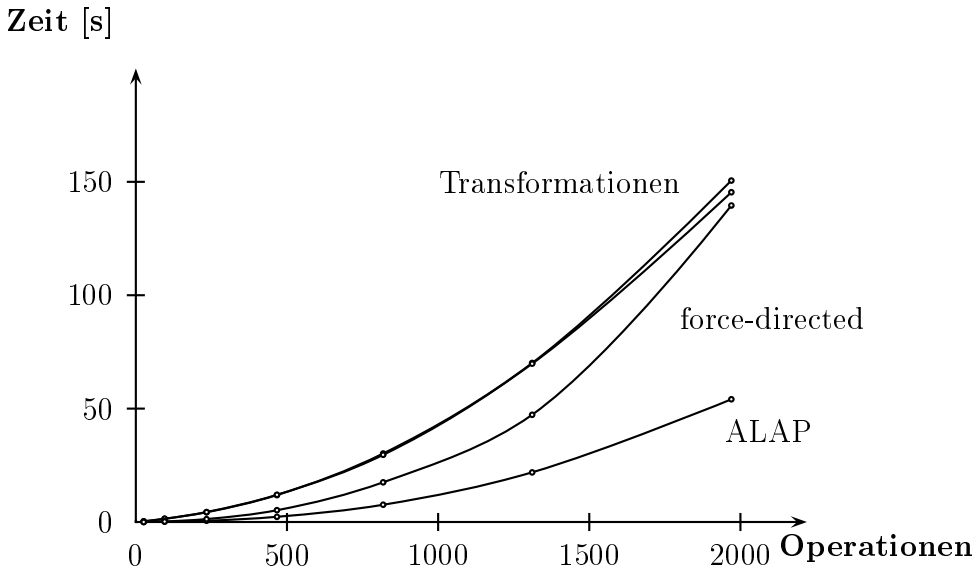


Abbildung 9.4: Laufzeit für Heuristik und Transformation bei der DCT

Programm	SPT	Schnittstellen-synthese
gcd	1.5	0.2
atoi	1.9	0.2
fibonacci	2.1	0.1
diffeq	3.4	0.3
bubble	4.4	0.5
fidelity	5.1	0.6
fuzzy	19.0	1.2
dct	50.0	5.2
kalman	103.9	2.8

Tabelle 9.1: Zeit in Sekunden für die Synthese

vielen erforderlichen β-Konversionen schlechtere Zeiten ergeben. Die Kosten für die Schnittstellensynthese hängen in wesentlichem Maße von der Größe des Schleifenrumpfes in der ESF ab, der sich durch die SPT ergibt. Nach der Anwendung eines der Implementierungstheoreme müssen auch hier β-Konversionen durchgeführt werden, deren Aufwand von den DFG-Termen der ESF abhängt.

Bei der Beurteilung der Synthesezeiten in Tabelle 9.1 ist zu beachten, dass für die betrachteten Syntheseschritte eine automatische Postsynthese-Verifikation mit Hilfe von Modellprüfungsprogrammen aufgrund der hohen Abstraktionsebene nicht möglich ist. Auch eine Postsynthese-Verifikation durch interaktives Theorembewei-

sen wäre dagegen extrem aufwendig und bei weitem zeitintensiver. Ebenso stellt eine Simulation keine Alternative dar, um besser zu einer garantiert korrekten Implementierung zu gelangen, da die Dauer für die Formale Synthese unabhängig von einer bestimmten Bitbreite ist, die Simulation dagegen nicht.

Kapitel 10

Zusammenfassung

In dieser Arbeit wurde eine neue Methode vorgestellt, um Schaltungen auf hohen Abstraktionsebenen formal zu beschreiben und daraus durch eine Formale Synthese Schaltungsbeschreibungen auf der RT-Ebene zu erzeugen. Aufgrund der Tatsache, dass die Synthese komplett in einem Theorembeweiser realisiert wurde, ergibt sich eine garantierte Korrektheit des Syntheseergebnisses. Während in den bisherigen Arbeiten zur Formalen Synthese auf höheren Abstraktionsebenen nur eingeschränkte Schaltungsbeschreibungen synthetisiert werden konnten, erlaubt der vorgestellte Ansatz die Synthese beliebiger berechenbarer, also μ-rekursiver Funktionen. Der Ansatz basiert auf einer neuentwickelten Hardwarebeschreibungssprache, die einerseits über eine präzise, formal definierte Semantik verfügt und andererseits sowohl ausdrucksstark ist als auch systematisch die Wiederverwertbarkeit von Schaltungsbeschreibungen ermöglicht.

Darüber hinaus wurde sowohl für die algorithmische wie für die Systemebene ein neuer Entwurfsstil erarbeitet. Er unterscheidet sich von bisherigen Ansätzen dadurch, dass auf der Systemebene ein exakt definiertes Kommunikationsschema zugrunde gelegt wird, das formale Schaltungsbeschreibungen in flexibler Weise zulässt und darüber hinaus eine separate Synthese abstrakter Kommunikationskanäle vermeidet. Auf der algorithmischen Ebene wird in vielen konventionellen Verfahren die Synthese kontrollflussbehafteter Schaltungsbeschreibungen oft auf die Synthese reiner Datenflussbeschreibungen zurückgeführt. In diesem Ansatz dagegen werden Schaltungen durch Anwendung von Programmtransformationen synthetisiert. Die Programmtransformationen basieren dabei auf vorab im Theorembeweiser abgeleiteten Theoremen, die während der Synthese in geeigneter Weise instantiiert werden.

Die Entscheidung, welche Transformationstheoreme wann und an welcher Stelle eingesetzt werden sollen, ist wesentlich für die Implementierungskosten des Syntheseergebnisses. Sie kann einerseits interaktiv getroffen werden, andererseits ist es aber auch möglich, eine automatische Entwurfsraumuntersuchung außerhalb des Theorembeweisers durchzuführen. Aufgrund der durch die Entwurfsraumuntersuchung erzeugten Steuerinformation wird anschließend die Transformation im Theorembeweiser automatisch ausgeführt. Als Methoden für eine Entwurfsraumuntersuchung können dabei Verfahren der konventionellen Hardware-Synthese eingesetzt werden.

Aufgrund dieser strikten Trennung zwischen Entwurfsraumuntersuchung und Transformation wird es möglich, nicht nur zu garantiert korrekten, sondern auch zu kostengünstigen Hardwareimplementierungen zu gelangen, die den durch konventionelle Schaltungssynthese erzeugten nicht nachstehen. Da die Formale Schaltungssynthese die Korrektheit konstruktiv ableitet und nicht wie die üblichen Verifikationsverfahren im Nachhinein überprüft, steht der zeitliche Mehraufwand für eine Formale Synthese in einem angemessenen Verhältnis zur konventionellen Synthese. Experimentelle Ergebnisse zeigen, dass insbesondere bei Anwendung komplexerer Entwurfsraumuntersuchungsmethoden der zusätzliche Aufwand für die formale Ableitung im Theorembeweiser keinen entscheidenden Nachteil darstellt.

Literaturverzeichnis

[AhSU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[Alle69] F. Allen. Program Optimization. *Annual Review in Automatic Programming*, 5:239–308, Elsevier, 1969.

[Amic96] TIMA, Grenoble, France. *Amical – Architectural Synthesis from VHDL*, 1996.

[Bare92] H.P. Barendregt. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 7: Functional Programming and Lambda Calculus, pages 321–364, Elsevier, 1992.

[BaWr90] R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, Springer-Verlag, 1990.

[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design*, Nijmegen, The Netherlands, June 1992, volume A-10, pages 129–156, North-Holland.

[BGHT91] R. Boulton, M. Gordon, J. Herbert, and J. van Tassel. The HOL Verification of ELLA Designs. In *International Workshop on Formal Methods in VLSI Design*, Miami, USA, January 1991, Springer-Verlag.

[BlEi97] C. Blumenröhr and D. Eisenbiegler. An efficient representation for formal synthesis. In *10th International Symposium on System Synthesis*, Antwerp, Belgium, September 1997, pages 9–15, IEEE Computer Society Press.

[BlEi98] C. Blumenröhr and D. Eisenbiegler. Deriving structural RT-implementations from algorithmic descriptions by means of logical transformations. In [GI98], pages 38–49.

[BlEi98b] C. Blumenröhr and D. Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In [Euro98], pages 34–37.

[BlEK96] C. Blumenröhr, D. Eisenbiegler, and R. Kumar. Applicability of formal synthesis illustrated via scheduling. In *Workshop on Logic and Architecture Synthesis*, Grenoble, France, December 1996, pages 345–352, Institut National Polytechnique de Grenoble.

[BlES99] C. Blumenröhr, D. Eisenbiegler, and D. Schmid. On the Efficiency of Formal Synthesis — Experimental Results. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):25–32, January 1999.

[BlSa99] C. Blumenröhr and V. Sabelfeld. Formal synthesis at the algorithmic level. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, Bad Herrenalb, Germany, September 1999, Lecture Notes in Computer Science (1703), pages 187–201, Springer-Verlag.

[Blum99] C. Blumenröhr. A formal approach to specify and synthesize at the system level. In [GI99], pages 11–20.

[BoSa95] D. Borrione and A. Salem. Denotational Semantics of a Synchronous VHDL Subset. *Formal Methods in System Design*, 7(1/2):53–72, Kluwer Academic Publishers, 1995.

[BrFK95] P. Breuer, L. Fernandez, and C. Kloos. A Functional Semantics for Unit-Delay VHDL. In C. Kloos and P. Breuer, editor, *Formal Semantics for VHDL*, volume 307 of *The Kluwer International series in engineering and computer science*, Kluwer Academic Publishers, 1995.

[BRSM97] H.J. Brand, S. Riedel, T. Schleinig, and D. Müller. Reuse of components for the redesign of an MPEG-2-HDTV video decoder. In *1. GI Reuse Workshop*, Karlsruhe, Germany, September 1997, pages 79–87, FZI-Report 4/97.

[Cade92] Cadence Design Systems. *Synergy: Verilog Design Guide*, 1992.

[cdes] *http://goethe.ira.uka.de/fsynth/Charme/<name>.c.*

[ClGL96] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. *Nato ASI Series F*, 152, 1996.

[DeBo95] D. Deharbe and D. Borrione. Semantics of a verification-oriented subset of VHDL. In P.E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, Frankfurt/Main, Germany, October 1995, Lecture Notes in Computer Science (987), pages 293–310, Springer-Verlag.

[EiBK96] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In [HOL96], pages 157–172.

[EiBl98] D. Eisenbiegler and C. Blumenröhr. Gropius – advanced reuse concepts in a new hardware desription language. In *2. GI Reuse Workshop*, Karlsruhe, Germany, September 1998, pages 93–104, FZI-Report 3-13-9/98.

[EiBl99] D. Eisenbiegler and C. Blumenröhr. Reuse Concepts in Gropius. In R. Seepold and A. Kunzmann, editors, *Reuse Techniques for VLSI Design*, chapter 11, pages 125–137, Kluwer Academic Publishers, 1999.

[Eise99] D. Eisenbiegler. *Ein Kalkül für die Formale Schaltungssynthese.* Dissertation, Universität Karlsruhe, Deutschland, 1999. Logos-Verlag, auch verfügbar unter http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=1999/informatik/1.

[EiSK93] D. Eisenbiegler, K. Schneider, and R. Kumar. A functional approach for formalizing regular hardware structures. In [HOL93], pages 99–112.

[ELLS97] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[Euro98] *Digital System Design Workshop at the 24th EUROMICRO 98 Conference*, Västerås, Sweden, August 1998, IEEE Computer Society Press.

[EvHR99] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proceedings Design Automation and Test in Europe 1999*, München, Germany, March 1999, IEEE Computer Society Press.

[FMCAD96] M. Srivas and A. Camilleri, editors. *Formal Methods in Computer-Aided Design. First International Conference*, Palo Alto, USA, November 1996, Lecture Notes in Computer Science (1166), Springer-Verlag.

[FMCAD98] G. Gopalakrishnan and P. Windley, editors. *Formal Methods in Computer-Aided Design. Second International Conference*, Palo Alto, USA, November 1998, Lecture Notes in Computer Science (1522), Springer-Verlag.

[FuMe95] M. Fuchs and M. Mendler. A functional semantics for delta-delay VHDL based on focus. In C.D. Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*, Madrid, Spain, March 1995, volume 307 of *The Kluwer international series in engineering and computer science*, chapter 1, Kluwer Academic Publishers.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[GeKR99] J. Gerlach, T. Klöpfer und W. Rosenstiel. Algorithmischer Ansatz zur automatisierten Entwurfsraum-Exploration auf hoher Abstraktionsebene. In [GI99], Seiten 110–120.

[GeRo98] J. Gerlach and W. Rosenstiel. A Scalable Methodology for Cost Estimation in a Transformational High-Level Design Space Exploration Environment. In *Design, Automation and Test in Europe*, Paris, France, February 1998, pages 226–231, IEEE Computer Society Press.

[GI98] F.J. Rammig und W. Müller, Hrsg., *GI/ITG/GME Workshop Methoden des Entwurfs und der Verifikation digitaler Systeme*, Paderborn, Deutschland, März 1998, HNI-Verlagsschriftenreihe.

[GI99] M. Mutz und N. Lange, Hrsg., *GI/ITG/GME Workshop Modellierung und Verifikation von Schaltungen und Systemen*, Braunschweig, Deutschland, Februar 1999, Shaker-Verlag.

[GoMe93] M. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Goos97a] G. Goos. *Vorlesungen über Informatik, Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag, 1997.

[Goos97c] G. Goos. *Vorlesungen über Informatik, Band 3: Berechenbarkeit, formale Sprachen, Spezifikationen*. Springer-Verlag, 1997.

[Gord85] M. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, Computer Laboratory, University of Cambridge, England, 1985.

[Gord86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 57–77, Elsevier Science Publishers, 1986.

[GrLS98] W. Grass, S. Lenk, and C. Sontheim. Design of control dominated hardware based on formal methods. In [Euro98], pages 357–364.

[Gupt92] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1(2/3):151–238, Kluwer Academic Publishers, 1992.

[GVNG94] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and design of embedded systems*. Prentice Hall, 1994.

[HaPe95] J. Hallberg and Z. Peng. Synthesis under local timing constraints in the CAMAD hig-level synthesis system. In *21th EUROMICRO Conference*, Como, Italy, September 1995, IEEE Computer Society Press.

[HoJS93] C. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, Springer-Verlag, 1993.

[HOL93] J.J. Joyce and C.-H. Seger, editors. *Higher Order Logic Theorem Proving and its Applications*, Vancouver, Canada, August 1993, Lecture Notes in Computer Science (780), Springer-Verlag.

[HOL96] J. von Wright, J. Grundy, and J. Harrison, editors. *9th International Conference Theorem Proving in Higher Order Logics*, Turku, Finland, August 1996, Lecture Notes in Computer Science (1125), Springer-Verlag.

[Huij98] C. Huijs. Design correctness of digital systems. In [Euro98], pages 30–33.

[Jens92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*, Springer-Verlag, 1992.

[JePO93] A. Jerraya, I. Park, and K. O'Brien. AMICAL: An Interactive High-Level Synthesis Environment. In *Proceedings European Design Automation Conference*, Paris, France, February 1993, IEEE Computer Society Press.

[JiPB93] H. Jifeng, I. Page, and J. Bowen. Towards a provably correct hardware implementation of occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, Arles, France, May 1993, Lecture Notes in Computer Science (683), pages 214–225, Springer-Verlag.

[KBES96] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid . Formal synthesis in circuit design-A classification and survey. In [FMCAD96], pages 294–309.

[Keut96] K. Keutzer. The need for formal methods for integrated circuit design. In [FMCAD96], pages 1–18.

[KlBr95] C.D. Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, Madrid, Spain, March 1995, volume 307 of *The Kluwer international series in engineering and computer science*, Kluwer Academic Publishers.

[Kond83] N. Kondakow. *Wörterbuch der Logik.* VEB Bibliographisches Institut Leipzig, 1983.

[Lars94] M. Larsson. An engineering approach to formal system design. In T.F. Melham and J. Camileri, editors, *Higher Order Logic Theorem Proving and its Applications*, Valletta, Malta, September 1994, Lecture Notes in Computer Science (859), pages 300–315, Springer-Verlag.

[Lars95] M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, Oxford University Press, 1995.

[Lars96] M. Larsson. Improving the result of high-level synthesis using interactive transformational design. In [HOL96], pages 299–314.

[LCAL93] M. Leeser, R. Chapman, M. Aagaard, M. Linderman, and S. Meier. High level synthesis and generating FPGAs with the BEDROC system. *Journal of VLSI Signal Processing*, 6(2):191–214, Kluwer Academic Publishers, 1993.

[LeAL91] M. Leeser, M. Aagaard, and M. Linderman. The BEDROC high level synthesis system. In *IEEE ASIC Conference*, IEEE Computer Society Press, 1991.

[Lips91] R. Lipsett. *VHDL: Hardware Description and Design.* Kluwer Academic Publisher, 1991.

[LoMe98] T. Lock und M. Mendler. Formale Modellierung von kontrollfluß-dominierten High-Level-Synthese-Eingabebeschreibungen zur Verifikation von Ergebnissen kontrollflußgesteuerter Einplanungsverfahren. In [GI98], Seiten 75–84.

[LoMe99] T. Lock und M. Mendler. Äquivalenz von annotierten Kontrollflussgraphen zur Darstellung von HLS-Ein- und Ausgaben bei Verwendung pfadbasierter Einplanungsverfahren. In [GI99], Seiten 51–60.

[LoMM98] T. Lock, M. Mendler, and M. Mutz. Combined formal post- and presynthesis verification in high level synthesis. In [FMCAD98], pages 222–236.

[LoMu97] T. Lock und M. Mutz. Automatische formale Post-Synthese-Verifikation von High-level Syntheseergebnissen. In R. Hagelauer, M. Pfaff, Hrsg., *Methoden des Entwurfs und der Verifikation digitaler Systeme*, Seiten 7–16, Linz, Österreich, April 1997, Universitätsverlag Rudolf Trauner.

[MaFo91] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991, pages 59–70, North-Holland.

[MaVe98] N. Mansouri and R. Vemuri. A Methodology for Automated Verification of Synthesized RTL Designs and Its Integration with a High-Level Synthesis Tool. In [FMCAD98], pages 204–221.

[McFa93] M. McFarland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2(3):231–257, Kluwer Academic Publishers, 1993.

[Melh89] T.F. Melham. Automating Recursive Type Definitions in Higher Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386, Springer-Verlag, 1989.

[Melh92] T.F. Melham. The HOL logic extendet with quantification over type variables. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992, volume A-20, pages 3–18, North-Holland.

[Melh93] T.F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[MHMP96] P. Middelhoek, C. Huijs, G. Mekenkamp, E. Prangsma, E. Engels, J. Hofstede, and T. Krol. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE International High Level Design Validation and Test Workshop*, Oakland, USA, November 1996.

[Mich94] G. DeMicheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[Mich96] G. DeMicheli, editor. *Hardware Software Co-Design.* Kluwer Academic Publishers, 1996.

[Midd97] P. Middelhoek. *Transformational Design: An Architecture Independent Interactive Design Methodology for the Synthesis of Correct and Efficient Digital Systems.* PhD thesis, Universiteit Twente, The Netherlands, 1997.

[MiRa96] P. Middelhoek and S. Rajan. From VHDL to efficient and first-time-right designs: A formal approach. *ACM Transactions on Design Automation of Electronic Systems*, 1(2), April 1996.

[MMEK96] G. Mekenkamp, P. Middelhoek, E. Engels, and T. Krol. Transformational design of a six constant multiplier. In *Proceedings ProRISC/IEEE Workshop on CSSP*, Mierlo, The Netherlands, November 1996, pages 233–238.

[Mutz97] M. Mutz. Automatic post-synthesis verification support for a high level synthesis step by using the HOL theorem prover. In H. Li and D. Probst, editors, *Advances in Hardware Design and Verification*, Montreal, Canada, October 1997, pages 291–308, Chapman&Hall.

[NaVe98] N. Narasimhan and R. Vemuri. On the effectiveness of theorem proving guided discovery of formal assertions for a register allocator in a high-level synthesis system. In *11th International Conference Theorem Proving in Higher Order Logics*, Canberra, Australia, September 1998, Lecture Notes in Computer Science (1479), Springer-Verlag.

[NTRG98] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. In *International Conference on Computer Design*, Austin, USA, October 1998, IEEE Computer Society Press.

[OwSR93] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language (Beta Release)*. SRI International, 1993.

[Paig97] R. Paige. Future directions in program transformations. *ACM Sigplan*, 32(1):94–98, January 1997.

[PaKn89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer Aided Design*, 8(6):661–679, June 1989.

[Paul91] P.G. Paulin. Global Scheduling and Allocation Algorithms in the HAL System. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*, pages 255–281, Kluwer Academic Publishers, 1991.

[PeKu94] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):150–166, February 1994.

[PePr97] A. Pettorossi and M. Proietti. Future directions in program transformations. *ACM Sigplan*, 32(1):99–102, January 1997.

[Pier97] L. Pierre. VHDL Description, Boyer-Moore Specification and Formal Verification of a Parallel System for Finding a Maximum. In *Proceedings Formal Methods for Parallel Programming : Theory and Applications*, Geneva, Switzerland, April 1997.

[Raja95a] S. Rajan. Formal verification of transformations on dependency graphs in optimizing compilers. In *Proceedings of California Software Engineering Symposium*, Irvine, USA, March 1995.

[Raja95b] S. Rajan. Correctness of transformations in high-level synthesis. In *Conference on Computer Desription Languages and their Applications*, Chiba, Japan, August 1995, IEEE Computer Society Press.

[Reet97] R. Reetz. *Eine Formalisierung der Hardwarebeschreibungssprache VHDL für die Hardware-Verifikation.* Dissertation, Universität Karlsruhe, Deutschland, 1997.

[Schö89] U. Schöning. *Logik für Informatiker.* Informatik (Nummer 56), B.I. Wissenschaftsverlag, 1989.

[VHDL96] IEEE. *IEEE Standard VHDL Language Reference Manual Std 1076.3*, 1996.

[ViBT98] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings Int. Conference on Functional Programming*, Baltimore, USA, September 1998, ACM.

[Wrig94a] J. von Wright. Program refinement by theorem prover. In *Proceedings 6th Refinement Workshop*, London, England, January 1994, Springer-Verlag.

[WrSe91] J. von Wright and K. Sere. Program Transformations in HOL. In M. Archer, J. Joyce, K. Levitt, and P. Windley, editors, *HOL Theorem Proving System and its Applications*, Davis, USA, August 1991, pages 231–239, IEEE Computer Society Press.