Ein Kalkül für die Formale Schaltungssynthese

Dirk Eisenbiegler

Ein Kalkül für die Formale Schaltungssynthese

Zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften von der Fakultät für Informatik der Universität Karlsruhe (Technische Hochschule)

Genehmigte Dissertation von Dirk Eisenbiegler aus Karlsruhe

Tag der mündlichen Prüfung: 24. November 1998
Erster Gutachter: Prof. Dr.-Ing. D. Schmid
Zweiter Gutachter: Prof. Dr. rer. nat. P. H. Schmitt

Diese Arbeit ist in elektronischer Form verfügbar.
http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=1999/informatik/1

Vorwort

Die vorliegende Arbeit entstand in den Jahren 1993 - 1998 zunächst am Forschungszentrum Informatik und ab 1996 im Rahmen eines von der DFG finanzierten Forschungsprojekts am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe.

Ich möchte an dieser Stelle allen, die mich bei der Erstellung dieser Arbeit unterstützt haben, meinen Dank aussprechen. Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. D. Schmid für die konstruktive Betreuung meiner Arbeit und die Übernahme des Referats. Bei Herrn Prof. Dr.rer.nat. P. H. Schmitt möchte ich mich für die Übernahme des Korreferats und die sorgfältige Begutachtung meiner Arbeit bedanken.

Weiterhin möchte ich mich bei allen bedanken, die in den letzten Jahren fachlich mit mir zusammengearbeitet haben. Herr Ramayya Kumar stand mir mit seinem Rat Tag und Nacht zur Verfügung. Bedanken möchte ich mich ferner bei Christian Blumenröhr und Jörg Berdux für die gute und produktive Zusammenarbeit. Bei Michaela Huhn, Frank Mayer, Christian Blumenröhr und Jörn Eisenbiegler möchte ich mich für die orthographische Durchsicht der Arbeit bedanken.

Karlsruhe, im Oktober 1998 *Dirk Eisenbiegler*

Inhaltsverzeichnis

1 Motivation **1**

1.1 Ziel der Arbeit 2

1.2 Aufbau der Arbeit 3

2 Grundlagen und Stand der Technik **5**

2.1 Schaltungssynthese 5

2.2 Formale Schaltungsbeschreibungen 6

2.2.1 Flächenbedarf, Leistungsverbrauch, Taktfrequenz 6

2.2.2 Eindeutigkeit, Mehrdeutigkeit, Inkonsistenz 7

2.3 Korrektheit von Schaltungen 7

2.3.1 Horizontale und vertikale Korrektheit 7

2.3.2 Korrektheit, Konsistenz und Implementierbarkeit 10

2.4 Wege zu korrekten Schaltungen 13

2.4.1 Prä-Synthese-Verifikation 13

2.4.2 Post-Synthese-Verifikation 14

2.4.3 Formale Synthese 14

2.4.4 Vergleich 15

2.5 Stand der Technik 16

2.5.1 PBS — Proven Boolean Simplification 16

2.5.2 RLEXT — Register Level Exploration Tool 17

2.5.3 Ruby und T-Ruby 17

2.5.4 DDD — Digital Design Derivation 18

2.5.5 Dialog 19

2.5.6 Veritas 20

2.5.7 Forvertis 21

2.6 Diskussion 22

2.6.1 Allgemeine und hardwarespezifische Kalküle 22

2.6.2 Grad der Automatisierung 23

2.6.3 Einsatzgebiet, Erweiterbarkeit 23

3 Eine Methodik zur Formalen Synthese 25
3.1 Formale Synthese in einem Theorembeweiser 25
3.2 Beschränkung auf funktionale Schaltungsbeschreibungen 27
3.3 Aufteilung in Entwurfsraumuntersuchung und Schaltungstransformation 27
3.4 Automatisierung und Interaktion 29

4 Schaltungsbeschreibungen auf RT- und Gatterebene 31
4.1 Kombinatorische Schaltungsbeschreibungen 32
4.1.1 Grundkomponenten 32
4.1.2 Strukturen 32
4.1.3 Hierarchische Strukturen 38
4.1.4 Bausteinbibliotheken 39
4.2 Abstrakte kombinatorische Schaltungsbeschreibungen 41
4.2.1 Anmerkung zur Implementierbarkeit 41
4.2.2 Polymorphe Signalvariablen 43
4.2.3 Die Datentypen one, (α)option und $\alpha + \beta$ 45
4.2.4 Aufzählungsdatentypen 48
4.2.5 Felder und reguläre kombinatorische Strukturen 49
4.2.6 Abgeleitete abstrakte Schaltungsbeschreibungen 54
4.3 Sequentielle Schaltungsbeschreibungen 56
4.3.1 Beschreibung sequentieller Schaltungen durch Automaten . 59
4.3.2 Semantik von automaton 60
4.3.3 Kombinatorische Schaltungen als Spezialfälle sequentieller Schaltungen 61
4.4 Auswertung und Simulation 62

5 Vergleich mit anderen Formalisierungsansätzen 65
5.1 Relationale und funktionale Schaltungsbeschreibungen 66
5.2 Harmlose kombinatorische Zyklen 70
5.3 Strukturelles und funktionales Entwurfsprinzip 73
5.4 Typisierung und reguläre Signalbündel 76

6 Schaltungstransformationen auf RT- und Gatterebene 79
6.1 Klassifikation der Schaltungstransformationen 80
6.2 Kombinatorische Schaltungstransformationen 82
6.2.1 Auftrennung und Überlagerung 83
6.2.2 Expansion und Zusammenfassung 84
6.2.3 Eliminierung redundanter kombinatorischer Einheiten . . . 85
6.2.4 Boolesche Umformungen 85
6.2.5 Kodierungen 86
6.2.6 Strukturelle Umformungen auf regulären Strukturen 89
6.2.7 Arithmetisch motivierte Umformungen 92
6.2.8 Schaltnetzimplementierungen als ROM 95
6.3 Sequentielle Schaltungstransformationen 97

6.3.1 Allgemeine Definitionen 98
6.3.2 Zustandskodierung 99
6.3.3 Eliminierung unerreichbarer Zustände 100
6.3.4 Zustandsklassifikation 104
6.3.5 Retiming 106
6.3.6 Eliminierung redundanter Schaltungsteile 108
6.4 Einbindung der Schaltungstransformationen in einen Syntheseablauf 109

7 High-Level-Synthese mit Datenpfaden **115**
7.1 Vorgehensweise 116
7.2 Beschreibung von Schaltungen auf Algorithmischer Ebene 117
7.3 Implementierungstheoreme 121
7.3.1 Implementierungstheoreme für $n = 0$ 121
7.3.2 Implementierungstheoreme für $n > 0$ 122
7.4 Die Aufteilung des DFG-Terms und die Qualität der Implementierung 129
7.5 Entwurfsraumuntersuchung und logische Transformation 133

8 Experimentelle Ergebnisse **135**
8.1 Bedeutung von Retiming 135
8.2 Konzeptionelle Unterschiede der Ansätze 136
8.3 Beispielschaltungen 138
8.4 Ergebnisse 138
8.5 Zusammenfassung 142

Kapitel 1

Motivation

Moderne Synthesewerkzeuge erlauben heute die Synthese hochgradig komplexer Digitalschaltungen. Der Entwurfsablauf wird infolgedessen zunehmend automatisiert, und für den Schaltungsentwerfer sind der Syntheseablauf und die dabei produzierten Schaltungsimplementierungen nur noch schwer oder gar nicht mehr zu überblicken. Er muß deshalb darauf vertrauen können, daß die von den Syntheseprogrammen automatisch erzeugten Schaltungsimplementierungen bezüglich der vorgegebenen Eingabe-Schaltungsbeschreibung korrekt sind, denn fehlerhafte Synthesewerkzeuge führen im allgemeinen zu fehlerhaften Schaltungsimplementierungen.

Diese scheinbar minimale Forderung erfüllen moderne Synthesewerkzeuge jedoch oft nicht. Durch die wachsende Komplexität der Syntheseprogramme wird die Sicherstellung ihrer Korrektheit zu einem immer größeren Problem. Syntheseprogramme bestehen heute nicht selten aus mehreren 100.000 Zeilen Programmcode. Da die Verifikation von Programmen dieser Größenordnung jedoch am zeitlichen und finanziellen Aufwand scheitert, entstehen in der Praxis oftmals Syntheseprogramme, deren Korrektheit der Programmierer des Syntheseprogramms nicht garantieren und deren Grad an Vertrauenswürdigkeit der Anwender nur experimentell erfahren kann.

Ein Grund für die Zunahme der Komplexität der Programme ist in der Zunahme der Schaltungsgröße zu sehen. Um auch große Schaltungen schnell und möglichst optimal synthetisieren zu können, werden immer aufwendigere Syntheseverfahren notwendig.

Ein weiterer Grund für die Komplexität moderner Syntheseprogrammen ist die Unterstützung von zunehmend abstrakteren Schaltungsbeschreibungssprachen auf höheren Entwurfsebenen (VHDL, VERILOG). Im Gegensatz zu Schaltungsbeschreibungssprachen auf Gatterebene (BLIF, KISS etc.), mit denen nur einfache Netzstrukturen beschrieben werden können, erlauben Schaltungsbeschreibungssprachen wie VHDL und VERILOG eine Vielzahl an Konstrukten, mit denen Schaltungen abstrakt und weitgehend technologieunabhängig beschrieben werden können. Als besonders problematisch für die Erzeugung korrekter Synthesepro-

gramme erweist sich dabei, daß es bei abstrakteren Schaltungsbeschreibungssprachen bei weitem schwieriger ist, die Semantik der Sprache so klar und exakt zu definieren, daß sie für alle Anwender und auch für die Entwickler der Synthesewerkzeuge unmißverständlich ist. Einen Eindruck davon vermitteln beispielsweise die zahlreichen Diskussionen zur exakten Festlegung der Semantik von VHDL (siehe [KlBr95]). Eine exakte Semantik der Schaltungsbeschreibungssprache ist eine elementare Voraussetzung für eine korrekte Synthese, denn es macht erst dann Sinn, über die Korrektheit von Schaltungsbeschreibungen zu reden, wenn deren Bedeutung mathematisch exakt vorgegeben ist. Ein korrektes Syntheseprogramm zu schreiben, setzt voraus, daß der Programmierer die Bedeutung der Konstrukte der Schaltungsbeschreibungen genau versteht.

Daß die Sicherstellung der Korrektheit der Synthese ein Problem von großer praktischer Relevanz ist, ist heute vielen Schaltungsentwerfern bewußt. Der Anwender von Syntheseprogrammen kann heute nicht voraussetzen, daß die Syntheseprogramme formal verifiziert sind. Um das Ergebnis der Synthese zu validieren, können vergleichende Simulationen von Eingabeschaltungsbeschreibung und Ausgabeschaltungsbeschreibung durchgeführt werden. Da durch partielle Schaltungssimulation nie die vollständige Korrektheit bewiesen werden kann, werden heute auch bereits formale Beweistechniken eingesetzt, und dies obwohl die zugrundeliegende Problemstellung in der Regel NP-vollständig ist. Akzeptanz finden vor allem vollautomatisierte Model-Checking-Verfahren. Die diesen Verfahren zugrundeliegende Temporallogik eignet sich jedoch nur für die Beschreibung einfacher Systeme auf unteren Abstraktionsebenen. Aufgrund der Komplexität des Verifikationsproblems beschränkt sich ihr Einsatz zudem i. allg. auf kleine Schaltungen mit einer kleinen Zustandsmenge. Auch intensive Anstrengungen, diese Verfahren weiter zu optimieren, waren in den vergangenen Jahren nicht in der Lage, mit der Komplexitätssteigerung der in der Praxis vorkommenden Schaltungen Schritt zu halten — im Gegenteil, die Schere zwischen den technisch realisierbaren Schaltungen und den automatisch verifizierbaren Schaltungen geht immer weiter auseinander.

1.1 Ziel der Arbeit

In dieser Arbeit soll ein neuer Ansatz vorgestellt werden, bei dem die formale, logische Argumentation nicht erst nach der Synthese stattfindet (Post-Synthese-Verifikation), sondern bereits in den Syntheseablauf integriert wird (Formale Synthese). Jeder Syntheseschritt wird dabei durch eine logische Umformung innerhalb eines Theorembeweisers implementiert. Die Formale Synthese ist eine Folge von logischen Umformungen, die alle auf dem Kalkül des Theorembeweisers basieren. Als Resultat der Formalen Synthese ergibt sich nicht nur die Schaltungsimplementierung, sondern auch der Beweis der Korrektheit der Schaltungsimplementierung bezüglich der vorgegebenen Eingabe.

Der entscheidende Vorteil der Formalen Synthese besteht darin, daß der Beweis, anders als bei der Post-Synthese-Verifikation, im Verlauf der Synthese implizit entsteht. Die Post-Synthese-Verifikation hingegen ist von der Synthese entkoppelt.

Die Information darüber, wie die Implementierung aus der Spezifikation abgeleitet wurde, ist für eine effiziente Beweisführung oft zwingend erforderlich, steht bei der Post-Synthese-Verifikation jedoch nicht immer zur Verfügung. Der Beweisweg muß „erraten" werden. Im Gegensatz zur Post-Synthese-Verifikation — eine Problemstellung, die für einfache Logiken NP-vollständig und für mächtigere Logiken sogar unentscheidbar ist — ist die Formale Synthese nicht auf entscheidbare, ausdrucksschwache Logiken beschränkt.

Die Grundlage für den in dieser Arbeit entwickelten Kalkül zur Formalen Schaltungssynthese bildet eine funktionale Hardwarebeschreibungssprache, mit der Schaltungen von der algorithmischen Ebene bis hin zur Gatterebene beschrieben werden können. Die Sprache verwendet Konzepte aus der funktionalen Programmierung wie Polymorphie und die Parametrisierung mit Funktionen (hier: Schaltungen) und unterstützt so in systematischer Weise einen auf Wiederverwertung ausgelegten Entwurfsstil. Neben der Ausdrucksmächtigkeit, die über das in konventionellen Hardwarebeschreibungssprachen vorhandene hinausgeht, zeichnet sich die Hardwarebeschreibungssprache vor allem durch eine durchgängige Konsistenz und Implementierbarkeit der Schaltungsbeschreibungen und die einfache Simulierbarkeit auf verschiedenen Abstraktionsebenen aus. Ein weiterer Vorzug gegenüber konventionellen Hardwarebeschreibungssprachen besteht darin, daß die Konstrukte der Sprache mit allgemeinen logischen Mitteln definiert wurden. Die Semantik der Sprache ist, im Gegensatz zu zahlreichen konventionellen Schaltungsbeschreibungssprachen, transparent und eindeutig.

1.2 Aufbau der Arbeit

Die Arbeit beginnt mit einem kurzen Überblick über die Schaltungssynthese und die verschiedenen Verfahren zur Gewährleistung der Korrektheit von Schaltungsimplementierungen (Kapitel 2). Anschließend wird das Konzept des hier vorgestellten Ansatzes erläutert und bestehenden Ansätzen gegenübergestellt (Kapitel 3). Die weiteren Kapitel beschreiben die eigenen Arbeiten. In Kapitel 4 wird eine formale Schaltungsbeschreibungssprache für die RT- und Gatterebene definiert und in Kapitel 5 deren Vor- und Nachteile mit anderen Formen der formalen Schaltungsbeschreibung verglichen. Die in Kapitel 4 eingeführte Schaltungsbeschreibungssprache bildet die Grundlage für die Schaltungstransformationen im Bereich RT-Synthese und High-Level-Synthese, die in den Kapiteln 6 und 7 vorgestellt werden. Es folgen experimentelle Untersuchungen zur Laufzeiteffizienz von formalen Syntheseprogrammen (Kapitel 8).

Kapitel 2

Grundlagen und Stand der Technik

In diesem Kapitel wird zunächst ein Überblick über die Schaltungssynthese, die formale Repräsentation von Schaltungen und den Begriff der Korrektheit der Synthese gegeben. Anschließend werden verschiedene Ansätze miteinander verglichen, denen allen ein Ziel gemeinsam ist: die Konstruktion einer Schaltungsimplementierung sowie eines mathematischen Beweises dafür, daß die erzeugte Implementierung korrekt ist.

2.1 Schaltungssynthese

Die Synthese digitaler Schaltungen ist ein komplexer Vorgang, der sich aus mehreren Teilschritten auf verschiedenen Entwurfsebenen zusammensetzt. Auf den verschiedenen Entwurfsebenen werden die Schaltungen auf einem unterschiedlichen Abstraktionsniveau beschrieben. Jeder Syntheseschritt stellt eine Transformation dar, bei der eine Eingabeschaltungsbeschreibung auf eine Ausgabeschaltungsbeschreibung abgebildet wird. Dies gilt in gleicher Weise für zusammengesetzte Syntheseschritte und somit auch für den Syntheseablauf als Ganzes.

Die unterste Abstraktionsebene, die hier betrachtet werden soll, ist die Gatterebene. Hier werden Schaltungen durch Strukturen aus elementaren logischen Gattern und Speicherkomponenten beschrieben, die untereinander über boolesche Signale kommunizieren. Unterhalb der Gatterebene findet bei der Synthese die geometrische Anordnung der Objekte statt (Plazierung, Verdrahtung). In bezug auf Energieverbrauch, Taktfrequenz und Flächenbedarf möglichst optimale Plazierungen und Verdrahtungen zu finden, ist für große Schaltungen eine schwierige Aufgabenstellung und erfordert auch in algorithmischer Hinsicht sehr aufwendige Verfahren. Die Überprüfung, ob diese Schritte verhaltenserhaltend sind, d.h. ob durch das Layout auch tatsächlich die auf der Gatterebene beschriebene Struktur realisiert wird, kann hingegen mit recht einfachen und effizienten Prüfprogrammen

realisiert werden. Die Korrektheit in diesem Bereich wird deshalb i. allg. nicht mit formalen Methoden untersucht.

Ganz anders sieht es oberhalb der Gatterebene aus. Ohne die Kenntnis der internen Abläufe während der Synthese ist es i. allg. sehr aufwendig, die Korrektheit eines Syntheseschrittes im nachhinein zu verifizieren. Dies gilt insbesondere für große Schaltungen und komplexe Syntheseabläufe. Die Synthese verläuft oberhalb der Gatterebene zu großen Teilen automatisiert, und die zahlreichen Einzelschritte beim Ablauf eines Syntheseprogramms sind für den Anwender kaum nachvollziehbar und werden ihm i. allg. vom Syntheseprogramm auch nicht mitgeteilt.

Oberhalb der Gatterebene unterscheidet man zwischen drei Abstraktionsebenen: der Register-Transfer-Ebene (RT-Ebene), der algorithmischen Ebene und der Systemebene. Die in dieser Arbeit vorgestellten Verfahren reichen von der Gatterebene bis hin zur algorithmischen Ebene.

2.2 Formale Schaltungsbeschreibungen

Schaltungsbeschreibungen liegen im allgemeinen umgangssprachlich oder in der Notation einer Hardwarebeschreibungssprache vor. Weder eine umgangssprachliche Beschreibung noch die Darstellung in einer üblichen Hardwarebeschreibungssprache sind als Grundlage für eine mathematische Beweisführung geeignet.

Um im mathematischen Sinne über Korrektheit von Schaltungen sprechen zu können, benötigt man als formale Grundlage eine Logik, d. h. eine formale Sprache, zu deren syntaktischen Elementen, den Formeln, Wahrheitswerte (wahr oder falsch) eindeutig definiert sind. Der erste Schritt hin zu nachweislich korrekten Schaltungen besteht deshalb darin, für die Schaltungen geeignete formale Repräsentationen in der Logik zu finden.

2.2.1 Flächenbedarf, Leistungsverbrauch, Taktfrequenz

Schaltungsbeschreibungen setzen sich im allgemeinen aus mehreren Teilen zusammen. Dazu gehören das funktionale Verhalten, die zu erzielende Taktfrequenz, der maximale Bedarf an Gattern, die maximal benötigte Fläche auf dem Chip und die maximal zulässige Verlustleistung.

Die meisten dieser Kriterien können bei einer vorgegebenen Implementierung einfach und effizient überprüft werden. Relativ einfach zu überprüfen sind beispielsweise die maximal erzielbare Taktfrequenz, der maximale Leistungsbedarf und die benötigte Fläche. Sehr aufwendig ist hingegen die Überprüfung des funktionalen Verhaltens. Die Gewährleistung der funktionalen Korrektheit ist eine unabdingbare Forderung für alle Syntheseprogramme. Sicherzustellen, daß auch Anforderungen wie Flächenbedarf oder Taktfrequenz eingehalten werden, ist nur unter der Prämisse sinnvoll, daß die Schaltung die vorgegebene Funktion erfüllt. Die meisten Verifikationsverfahren beziehen sich deshalb ausschließlich auf das funktionale Verhalten.

Oft kann zu Anfang nur schwer abgeschätzt werden, ob es überhaupt eine Implementierung gibt, die die oben beschriebenen nichtfunktionalen Anforderungen erfüllt. Sie werden bei der Synthese deshalb nur als Randbedingungen betrachtet. Die Synthese verläuft so, daß die Schaltung durch das Syntheseprogramm in funktionserhaltender Weise transformiert wird, wobei gleichzeitig versucht wird, die einander oft widersprechenden Kosten bzgl. Flächenbedarf, Taktfrequenz etc. zu minimieren. Im frühen Stadium können diese Kosten oft nur grob abgeschätzt werden. Je weiter der Syntheseablauf voranschreitet, desto genauer können Aussagen über die Einhaltung der geforderten Randbedingungen gemacht werden.

2.2.2 Eindeutigkeit, Mehrdeutigkeit, Inkonsistenz

Real existierende Schaltungen stellen stets einen eindeutigen, funktionalen Zusammenhang zwischen Ein- und Ausgangssignalen her. Bei der Schaltungssynthese kann es jedoch auch nützlich sein, Freiheitsgrade offen zu lassen. Typische Beispiele hierfür sind „don't-cares" in Gatterebenen-Beschreibungen, nichtinstantiierte Variablenwerte in algorithmischen Schaltungsbeschreibungen sowie Freiheitsgrade bei der Schnittstellenbeschreibung bei der High-Level-Synthese.

Derartige Schaltungsbeschreibungen sind partiell. Sie beschreiben eine Schaltung nicht vollständig, sondern lassen Spielräume, so daß durch diese Beschreibungen nicht eine einzelne Schaltung eindeutig definiert, sondern vielmehr eine Menge von Schaltungen charakterisiert wird.

Durch Formeln beschriebene Schaltungen sind nicht notwendigerweise eindeutig. Es sind drei Fälle zu unterscheiden: Die Menge der durch eine Formel repräsentierten Schaltungen kann einelementig (Eindeutigkeit), mehrelementig (Mehrdeutigkeit) oder leer (Inkonsistenz) sein.

2.3 Korrektheit von Schaltungen

Eine Schaltung S soll als korrekt betrachtet werden, wenn sie die vorgegebenen Anforderungen A erfüllt. Aus mathematischer Sicht sind S und A Formeln, in denen die Ein- und Ausgangssignale frei vorkommen und die so jeweils eine Relation zwischen Ein- und Ausgangssignalen herstellen. Die umgangssprachliche Aussage, daß S bezüglich A korrekt ist, bedeutet aus logischer Sicht, daß die Menge der durch S beschriebenen Schaltungen eine Teilmenge der durch A beschriebenen Schaltungen ist. Dies ist gleichbedeutend mit der Forderung, daß die Formel $S \Rightarrow A$ allgemeingültig ist.

2.3.1 Horizontale und vertikale Korrektheit

In diesem Abschnitt sollen die Begriffe horizontale und vertikale Korrektheit definiert werden. Dazu wird zunächst der Begriff der synthesebezogenen Schaltungsbeschreibung eingeführt und erläutert.

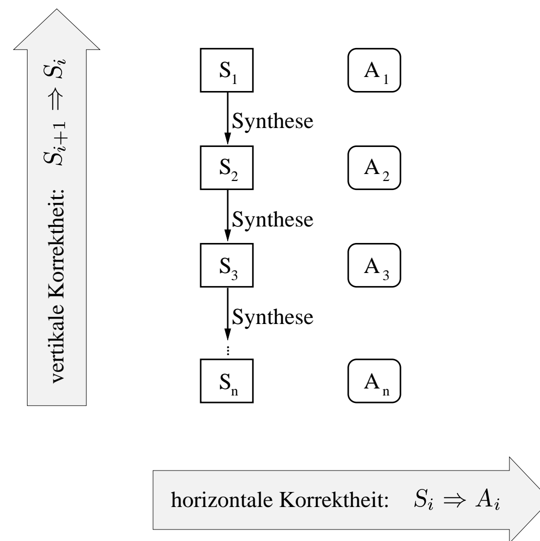


Abbildung 2.1: Horizontale und vertikale Korrektheit

Die Schaltungssynthese stellt eine Folge von Transformationen dar. Jede Schaltungstransformation bildet eine Schaltungsbeschreibung S_i auf eine Schaltungsbeschreibung S_{i+1} ab (Abbildung 2.1). Die Eingabeschaltungsbeschreibung S_i und die Ausgabeschaltungsbeschreibung S_{i+1} sind jedoch nicht willkürlich. Vielmehr müssen die einzelnen Schaltungstransformationen definierte Schnittstellen haben, d. h. die Menge der zulässigen Eingabeschaltungsbeschreibungen und die Menge der von der Schaltungstransformation produzierten Ausgabeschaltungsbeschreibungen muß genau festgelegt sein. Beide Mengen werden i. allg. durch Schaltungsbeschreibungssprachen bzw. durch Teilmengen von Schaltungsbeschreibungssprachen definiert. Eingabe- und Ausgabesprache können durchaus unterschiedlich sein. Beispielsweise ist es bei einer High-Level-Synthese vernünftig, für die Eingabe (algorithmische Schaltungsbeschreibungen) und die Ausgabe (Strukturen auf RT-Ebene) unterschiedliche Schaltungsbeschreibungssprachen zu wählen. Bei einer booleschen Optimierung des Schaltnetzes sind sowohl die Eingabe als auch die Ausgabe Schaltungsstrukturen auf Gatterebene und können in der gleichen Schaltungsbeschreibungssprache dargestellt werden.

Eine Schaltungsbeschreibung soll genau dann als *synthesebezogen* bezeichnet werden, wenn sie Ein- oder/und Ausgabe eines Syntheseschritts ist. Synthesebezogene Schaltungsbeschreibungen sind somit die Zustände des Syntheseablaufs. In Abbildung 2.1 sind S_1, S_2, ... S_n synthesebezogene Schaltungsbeschreibungen.

Es ist durchaus auch sinnvoll, Schaltungsbeschreibungen aufzustellen, die nicht synthesebezogen sind und die keine Entsprechung in einer Hardwarebeschreibungssprache haben. In Abbildung 2.1 wird angedeutet, daß jeder synthesebezogenen Schaltungsbeschreibung S_i eine beliebige Anforderung A_i gegenübergestellt wer-

den kann, die nicht synthesebezogen sein muß. Ein typisches Beispiel für eine nicht synthesebezogene Schaltungsbeschreibung ist:

Es sollen nie alle Ampeln gleichzeitig grün sein.

Erfahrene Schaltungsentwerfer wissen sofort, daß diese Schaltungsbeschreibung nur als eine Randbedingung verwendet werden kann, sie kommt weder als Eingabe noch als Ausgabe eines Syntheseprogramms in Frage, ist also nicht synthesebezogen. Es stellt sich nun die Frage, was synthesebezogene Schaltungsbeschreibungen auszeichnet und weshalb etwa obige Beschreibung nicht synthesebezogen ist.

Man mag versucht sein, von synthesebezogenen Schaltungen einfach zu fordern, daß sie eindeutig sind. Die obige Anforderung an die Ampelsteuerung ist nicht eindeutig. Vielmehr charakterisiert sie eine mehrelementige Menge von Schaltungen, nämlich die Menge aller Schaltungen, die diese Eigenschaft erfüllen.

Dieser Ansatz führt jedoch in die Irre. Auch synthesebezogene Schaltungsbeschreibungen können und sollen bisweilen mehrdeutig sein. Man denke nur an Gatterebenenbeschreibung mit „dont-cares", also booleschen Variablen, die in beliebiger Weise durch wahr oder falsch instantiiert werden dürfen. Eine solche Schaltungsbeschreibung steht nicht für eine einzelne Schaltung, sondern für die Menge aller Schaltungen, die entstehen, wenn man die „dont-cares" nacheinander mit allen möglichen Kombinationen boolescher Werte belegt.

Ein anderes Beispiel für eine Schaltungsbeschreibung, die nicht für eine einzelne Schaltung, sondern für eine Menge von Schaltungen steht, ist eine skalierbare Schaltungsstruktur, bei der sich mit einem natürlichzahligen Parameter die Bitbreite einstellen läßt. Als Beispiel betrachte man noch eine Schaltungsbeschreibung auf algorithmischer Ebene mit einem reinen Datenflußgraphen. Die Funktion der Schaltung kann in einer unterschiedlichen Anzahl von Takten realisiert werden. Dabei entstehen Schaltungen mit unterschiedlichem Ein-/Ausgabeverhalten. Die Anzahl der Takte gehört mit zur Schaltungsbeschreibung. Sie ist jedoch variabel und kann entweder vom Anwender oder durch das Syntheseprogramm festgelegt werden.

Bei diesen drei Beispielen für synthesebezogene, mehrdeutige Schaltungen fällt auf, daß die Mehrdeutigkeit immer auf die gleiche Art und Weise ausgedrückt wird: Es gibt variable Größen („dont-cares", Bitbreiten, Taktzahlen), die beliebig instantiiert werden dürfen. Die Menge der Werte, die diese variablen Größen annehmen dürfen, ist jeweils eindeutig definiert. Es gibt damit einen einfachen Weg, die einzelnen Schaltungen, für die eine solche Schaltungsbeschreibung steht, zu erzeugen, nämlich die Instantiierung der variablen Größen.

Anders ist es bei obiger Anforderung an die Ampelanlage. Sie enthält keine variablen Größen, die nur noch instantiiert werden müßten, um zu konkreten Schaltungen zu gelangen. Um diese Schaltungsbeschreibung zu synthetisieren, müßte das Syntheseprogramm unter der Menge aller Schaltungen, die diese Anforderung erfüllen, eine in einer definierten Weise möglichst gute Implementierung auswählen. Ein Syntheseprogramm zu schreiben, daß überhaupt eine Implementierung zu einer beliebigen Formel findet, wäre bereits sehr aufwendig (Erfüllbarkeitsproblem,

Suche nach dem Modell einer Formel). Es ist mir kein Syntheseprogramm bekannt, das in der Lage ist, mehrdeutige Schaltungsbeschreibungen zu handhaben, wenn diese nicht explizit durch variable Größen ausgedrückt wird. Zwar ist es prinzipiell nicht unmöglich, ein Syntheseprogramm zu schreiben, das diese Grenzen überschreitet und auch einige nichtsynthesebezogene Schaltungsbeschreibungen synthetisieren kann. Dies würde jedoch für den Programmierer der Syntheseprogramme erhebliche Probleme aufwerfen, und es darf bezweifelt werden, ob dies von den Anwendern der Syntheseprogramme überhaupt gewünscht wird.

Aufbauend auf dem Begriff der synthesebezogenen Schaltungsbeschreibung können nun zwei Formen der Schaltungskorrektheit unterschieden werden: die vertikale Korrektheit und die horizontale Korrektheit (siehe Abbildung 2.1).

Vertikale Korrektheit bezieht sich auf einen Syntheseschritt mit einer Eingabeschaltungsbeschreibung S_i und einer Ausgabeschaltungsbeschreibung S_{i+1}. Vertikale Korrektheit bedeutet, daß S_{i+1} die in S_i vorgegebenen Anforderungen erfüllt, d. h. daß $S_{i+1} \Rightarrow S_i$ gilt. Man bezeichnet einen solchen Schritt als eine Verfeinerung. Gegenstand dieser Arbeit ist ausschließlich die vertikale Korrektheit.

Die *horizontale Korrektheit* bezieht sich nicht auf einen Syntheseschritt, sondern auf eine synthesebezogene Schaltungsbeschreibung S_i und eine beliebige Anforderung A_i. Gefordert wird die Allgemeingültigkeit von $S_i \Rightarrow A_i$.

2.3.2 Korrektheit, Konsistenz und Implementierbarkeit

Ein Korrektheitsbeweis für eine Schaltung macht nur dann Sinn, wenn die Schaltungsbeschreibung konsistent ist. Bei inkonsistenten Schaltungsbeschreibungen ist die Menge der durch sie repräsentierten Schaltungen leer, sie sind nicht implementierbar. Inkonsistente Schaltungsbeschreibungen haben mit realen Schaltungen nichts zu tun. Es reicht nicht, zu gewährleisten, daß ein Schaltungsbeschreibung eine vorgegebene Anforderung erfüllt, es muß auch gewährleistet werden, daß die Schaltungsbeschreibung überhaupt eine real implementierbare Schaltung repräsentiert.

Inkonsistente Schaltungsbeschreibungen führen oft zu irreführenden Ergebnissen. Daß für alle Elemente einer durch eine inkonsistente Formel S charakterisierten leeren Schaltungsmenge jede beliebige Eigenschaft A gilt, ist trivial (ex falso quodlibet). Der Beweis $S \Rightarrow A$ ist in solchen Fällen in der Regel sehr einfach. Das erzielte Ergebnis beschreibt jedoch nur scheinbar die Korrektheit einer Schaltung, in Wirklichkeit ist es nutzlos.

Beispiele für inkonsistente Schaltungsbeschreibungen

Typische Beispiele für inkonsistente Schaltungsbeschreibungen sind synchrone Schaltungsbeschreibungen, bei denen die Strukturen Kurzschlüsse oder kombinatorische Zyklen enthalten. Abbildung 2.2 zeigt ein Beispiel für einen kombinatorischen Zyklus.

Die Formalisierung der Gesamtstruktur setzt voraus, daß die Teilkomponenten bereits definiert sind. Nachfolgend sind die in Abbildung 2.2 verwendeten Grund-

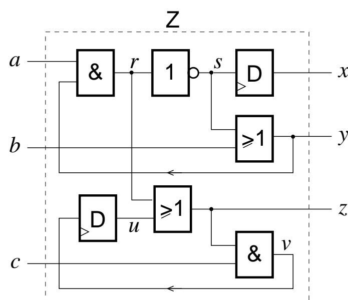


Abbildung 2.2: Kombinatorischer Zyklus

komponenten UND-Gatter (PAND), ODER-Gatter (POR), Inverter (PINV) und D-Flipflop (PDFF) im relationalen Stil in einem synchronen Zeitmodell definiert:

$$
\begin{aligned}
\mathsf{PAND}(a,b,c) &:= (\forall t.\, c(t) = a(t) \wedge b(t)) \\
\mathsf{POR}(a,b,c) &:= (\forall t.\, c(t) = a(t) \vee b(t)) \\
\mathsf{PINV}(a,b) &:= (\forall t.\, b(t) = \neg a(t)) \\
\mathsf{PDFF}(a,b) &:= (b(0) = \mathsf{F}) \wedge (\forall t.\, b(t+1) = a(t))
\end{aligned}
$$

Die Schaltungsstruktur Z kann man mit der üblichen Vorgehensweise sehr einfach in eine logische Formel übersetzen:

$$
\begin{aligned}
\mathsf{Z}(a,b,c,x,y,z) := \exists r,s,u,v.\; & \mathsf{PAND}(a,y,r) \wedge \\
& \mathsf{PINV}(r,s) \wedge \\
& \mathsf{PDFF}(s,x) \wedge \\
& \mathsf{POR}(s,b,y) \wedge \\
& \mathsf{PDFF}(v,u) \wedge \\
& \mathsf{POR}(r,u,z) \wedge \\
& \mathsf{PAND}(z,c,v)
\end{aligned}
$$

Die nachfolgend definierte Anforderung A besagt, daß unter der Voraussetzung, daß man zum Zeitpunkt 0 an die Eingangssignale a, b und c die Werte T, F bzw. F anlegt, zu den Zeitpunkten 17, 31 und 52 der Wert F an den Ausgängen x, y bzw. z anliegen wird.

$$
\begin{aligned}
\mathsf{A}(a,b,c,x,y,z) := \; & (a(0) = \mathsf{T}) \;\wedge\; (b(0) = \mathsf{F}) \;\wedge\; (c(0) = \mathsf{F}) \\
& \Rightarrow \;\; (x(17) = \mathsf{F}) \;\wedge\; (y(31) = \mathsf{F}) \;\wedge\; (z(52) = \mathsf{F})
\end{aligned}
$$

Routinierte Schaltungsentwerfer erkennen sofort, daß diese Anforderung für eine reale Schaltung mit der in Abbildung 2.2 beschriebenen Struktur nie und nimmer gefolgert werden kann. Um so mehr mag es überraschen, daß die Schaltungsbeschreibung Z die Anforderung A im mathematischen Sinne impliziert. Es gilt:

$$\vdash \forall a,b,c,x,y,z.\; \mathsf{Z}(a,b,c,x,y,z) \Rightarrow \mathsf{A}(a,b,c,x,y,z)$$

Zum Beweis betrachte man die Signale r, s und y zum Zeitpunkt 0. Die Schaltungsbeschreibung Z stellt unter anderem über den Inverter den Zusammenhang $\forall t.s(t) = \neg(r(t))$ her, was insbesondere auch $s(0) = \neg(r(0))$ impliziert. Aufgrund von $a(0) = \mathsf{T}$ und $b(0) = \mathsf{F}$ folgt $y(0) = s(0)$ und $r(0) = y(0)$, also auch $s(0) = r(0)$. Die Prämissen, unter denen die Konklusion

$$(x(17) = \mathsf{F}) \ \wedge \ (y(31) = \mathsf{F}) \ \wedge \ (z(52) = \mathsf{F})$$

bewiesen werden soll, implizieren sowohl $s(0) = r(0)$ als auch $s(0) = \neg(r(0))$. Die Prämissen sind also widersprüchlich. Der Beweis ergibt sich ex falso quodlibet.

Bei diesem Beweis fällt auf, daß die Konklusion austauschbar ist. Man hätte an der Stelle der obigen Konklusion jede beliebige andere Konklusion einsetzen können, und es wäre in gleicher Weise möglich gewesen, auch dies zu beweisen.

In diesem Beispiel ist ein „Korrektheitsbeweis" entstanden, der offenbar mit der Realität nichts zu tun hat. Dabei ist es nicht etwa deshalb zu einem Fehler gekommen, weil der Beweis fehlerhaft geführt worden wäre. Der Beweis ist mathematisch korrekt. Der Fehler ist auch nicht darauf zurückzuführen, daß die Teilkomponenten fehlerhaft formalisiert worden wären. Die Ursache für diesen Fehler ist vielmehr die fehlerhafte Formalisierung der Schaltungsstruktur Z. Das bei der Definition von Z angewandte Schema zur Formalisierung von Schaltungsstrukturen, bei der die Teilkomponten einfach konjunktiv verknüpft werden, die Variablennamen für Verbindungen stehen und die internen Variablen existenzquantifiziert werden, führt nicht für beliebige Schaltungsstrukturen zu einer stimmigen, der Realität entsprechenden Formalisierung. Unter anderem führen verzögerungsfreie Zyklen bei dieser Vorgehensweise i. allg. zu einer fehlerhaften Modellierung der Gesamtschaltung und zu fehlerhaften „Korrektheitsbeweisen".

Das so erzielte Ergebnis ist nicht nur nutzlos, sondern führt auch zu einer technisch problematischen Schaltung. Bei einer technischen Realisierung kommt es in bestimmten Fällen zu einem Schwingen der Schaltung. In dem betrachteten Beispiel passiert dies genau dann, wenn zu einem Zeitpunkt t für die Signale a und b gilt, daß $a(t) = \mathsf{T}$ und $b(t) = \mathsf{F}$. Bei anderen Schaltungen mit kombinatorischen Zyklen kann dieser Zusammenhang komplizierter sein, und insbesondere kann auch der aktuelle Zustand der Schaltung eine Rolle spielen. Das Verhalten der Schaltung kann in diesen Fällen nicht vorhergesagt werden, zumindest nicht durch die betrachtete synchrone Modellierung. Auf keinen Fall ist gewährleistet, daß sich die Beispielschaltung gemäß Anforderung A verhält.

Die Modellierung von Kurzschlüssen hat ähnliche Auswirkungen. Die Schaltung scheint nachweislich eine gewisse Anforderung zu erfüllen. Eine technische Realisierung führt dann jedoch zu einem erhöhten Energieverbrauch und eventuell sogar zur Zerstörung von Komponenten. Das scheinbar bewiesene Verhalten stellt sich nicht ein.

Eine weitere typische Ursache für Inkonsistenzen sind Signalbusse. Verwendet man Leitungen, auf die mehrere Teilnehmer schreibend zugreifen können, so ist zur Sicherstellung der Konsistenz der Schaltung immer auch nachzuweisen, daß es zu keinem Zeitpunkt dazu kommt, daß mehr als ein Teilnehmer schreibend auf diese Leitung zugreift. Derartige Fehler sind bei weitem schwieriger zu entdecken.

Es reicht nicht aus, allein die Struktur zu betrachten, sondern es muß auch für alle möglichen Fälle das Verhalten untersucht werden, was i. allg. sehr aufwendig ist.

Bisher wurde nur auf die Problematik inkonsistenter synchroner Schaltungsstrukturen auf Gatterebene eingegangen. Dabei ist deren logische Modellierung noch vergleichsweise einfach. Sicherzustellen, daß Schaltungsbeschreibungen konsistent sind, ist bei abstrakteren Beschreibungsformen auf RT-Ebene oder auf algorithmischer Ebene bei weitem aufwendiger.

Maßnahmen zur Sicherstellung der Konsistenz

Es liegt nahe, sich bei der Art und Weise, wie Schaltungen beschrieben werden, auf eine systematisch abgegrenzte Menge von Beschreibungsformen zu beschränken, die einerseits alle für einen bestimmten Anwendungsbereich erforderlichen Beschreibungsmöglichkeiten abdeckt und andererseits stets Konsistenz gewährleistet. Es gibt dazu verschiedene Ansätze. In dem formalen Synthesewerkzeug Dialog [AHL92] wird die Konsistenz der synchronen Schaltungsstrukturen dadurch gewährleistet, daß nach jedem Verfeinerungsschritt die Struktur auf Kurzschlüsse und kombinatorische Zyklen hin überprüft wird (siehe auch Abschnitt 2.5.5). Der Model-Checker SMV [McMi92b] gewährleistet die Konsistenz der Implementierung dadurch, daß bei der Übersetzung der gewählten Eingabesprache in eine Kripke-Struktur drei einfach zu entscheidende Kriterien überprüft werden: die Zuweisungen dürfen keine zyklischen Abhängigkeiten enthalten, es dürfen keine zwei Zuweisungen auf die gleiche Variable erfolgen, und die Zuweisungen müssen frei von Typfehlern sein.

2.4 Wege zu korrekten Schaltungen

Es gibt verschiedene Möglichkeiten, um ausgehend von einer Schaltungsbeschreibung zu einer nachweislich korrekten Schaltungsimplementierung, d. h. zu einer Schaltungsimplementierung und einem mathematischen Beweis, daß diese Schaltungsimplementierung die vorgegebene Schaltungsbeschreibung impliziert, zu gelangen. Prinzipiell können drei Ansätze unterschieden werden: Post-Synthese-Verifikation, Formale Synthese und Prä-Synthese-Verifikation [KBES96]. Sie unterscheiden sich im Zeitpunkt, zu dem die logische Beweisführung stattfindet. Bei der Prä-Synthese-Verifikation findet die Beweisführung *vor*, bei der Formalen Synthese *während* und bei der Post-Synthese-Verifikation *nach* dem Syntheseablauf statt.

2.4.1 Prä-Synthese-Verifikation

Es ist möglich, die Korrektheit der Synthese dadurch zu gewährleisten, daß die Korrektheit der einzusetzenden Syntheseprogramme durch Softwareverifikation abgesichert wird. Beweisziel der Softwareverifikation ist, daß das Programm zu allen erlaubten Eingabe-Schaltungsbeschreibungen funktional korrekte

Ausgabe-Schaltungsbeschreibungen erzeugt. Dieser Ansatz soll als Prä-Synthese-Verifikation bezeichnet werden, da die Korrektheit bereits vor der Durchführung der Synthese abgesichert wird. Aufgrund der Tatsache, daß es sich bei Syntheseprogrammen i. allg. um sehr große Programme handelt, ist diese Vorgehensweise mit einem erheblichem Aufwand verbunden und hat sich in der Praxis nicht etablieren können.

2.4.2 Post-Synthese-Verifikation

Bei der Post-Synthese-Verifikation wird der Syntheseschritt, durch den die Schaltungsbeschreibung S_i auf S_{i+1} abgebildet wird, auf konventionelle Weise durchgeführt. Eine logische Beweisführung findet bis zu diesem Zeitpunkt nicht statt. Der Syntheseschritt kann auf beliebige Weise durchgeführt werden. Er kann sowohl interaktiv als auch manuell erfolgen. Anschließend wird nach einer logischen Ableitung für das Beweisziel $S_{i+1} \Rightarrow S_i$ gesucht.

Prinzipiell ist die nachträgliche Post-Synthese-Verifikation sehr zeitaufwendig. Bereits bei booleschen Schaltnetzen ist diese Problemstellung NP-vollständig. Allgemeine Verifikationsverfahren erweisen sich deshalb für große Schaltungen als zu langsam. Verifikationsverfahren, die speziell für bestimmte Verifikationsschritte optimiert sind, nutzen die Information aus, daß in dem betrachteten Syntheseschritt nur bestimmte Umformungen stattfinden. Durch spezialisierte Verifikationsverfahren ist es möglich, die Effizienz der Beweisführung zu steigern. Spezialisierte Verifikationstechniken können jedoch nur für den betrachteten Syntheseschritt erfolgreich eingesetzt werden — für andere Syntheseschritte sind diese Verifikationsverfahren nicht hinreichend effizient oder sie scheitern.

Das Szenario aus Synthese und anschließender Verifikation hat den Vorteil, daß bestehende Syntheseprogramme unverändert übernommen werden können. Nachteilig ist, daß der interne Ablauf der Synthese bei der anschließenden Verifikation nicht bekannt ist. Das Syntheseprogramm bildet eine Schaltungsbeschreibung auf eine andere ab, und nur diese beiden Schaltungsbeschreibungen sind dem Post-Synthese-Verifikationsprogramm zugänglich. Welche Schritte jedoch im einzelnen wie abgelaufen sind, ist nach außen hin nicht sichtbar. Gerade diese Information wäre für eine schnelle Verifikation jedoch oft hilfreich. Beispielsweise wäre es nach einer Zustandskodierung für eine nachgeschaltete Verifikation nützlich zu wissen, wie die Zustände kodiert wurden.

2.4.3 Formale Synthese

Bei der Formalen Synthese wird die Schaltungsimplementierung durch logische Umformungen der Schaltungsbeschreibung abgeleitet. Damit erfolgt die logische Argumentation während der Synthese. Die einzelnen Schritte sind dabei Verfeinerungen, durch die S_i auf ein Theorem der Form $S_{i+1} \Rightarrow S_i$ abgebildet wird. S_{i+1} entsteht dabei während dieses Schritts.

2.4.4 Vergleich

Bei der Beurteilung der drei soeben vorgestellten Ansätze ist zwischen der Sicht des Programmierers des Entwurfswerkzeugs und der Sicht des Schaltungsentwerfers, dem Anwender des Entwurfswerkzeugs, zu unterscheiden. Für den Schaltungsentwerfer sind neben der Korrektheit vor allem die folgenden vier Punkte von Bedeutung:

- Das Syntheseergebnis muß eine möglichst hohe Qualität haben. Zu berücksichtigen sind dabei die Eigenschaften der Implementierung (Signallaufzeiten, Taktfrequenz) und deren Kosten (Leistungsaufnahme, Flächenverbrauch).
- Die Synthese muß hinreichend schnell erfolgen können, es muß also gewährleistet sein, daß auch größere Schaltungen in akzeptabler Zeit synthetisiert werden können.
- Die Synthese soll so weit wie möglich automatisiert ablaufen. Gleichzeitig soll es jedoch auch möglich sein, bestimmte Syntheseschritte interaktiv ausführen zu können.
- Schaltungsentwerfer möchten sich während der Synthese möglichst wenig mit logischer Argumentation auseinandersetzen. Auf wenig Akzeptanz stoßen deshalb Verfahren, bei denen speziell zur Sicherstellung der logischen Korrektheit interaktive Beweisschritte erforderlich sind.

Aus diesen Anforderungen ergibt sich, daß Syntheseprogramme, deren Korrektheit vorab durch Software-Verifikation bewiesen wurde (Prä-Synthese-Verifikation) vom Schaltungsentwerfer favorisiert werden. Industriell einsetzbare, formal vollständig verifizierte Syntheseprogramme gibt es bisher jedoch noch nicht. Das liegt vor allem an der Komplexität der Syntheseprogramme, und an den kurzen Zyklen, innerhalb derer die Programmierer die Syntheseprogramme weiterentwickeln müssen, um am Markt bestehen zu können.

Der Schaltungsentwerfer, der konventionelle Synthesewerkzeuge einsetzt, muß deshalb, wenn er zu einem Beweis der Korrektheit der Implementierung gelangen will, jedem Syntheselauf einen Verifikationsschritt nachschalten (Post-Synthese-Verifikation). Von den Verifikationswerkzeugen wird deshalb vom Schaltungsentwerfer ein hoher Grad an Automatisierung und eine hohe Beweisgeschwindigkeit gefordert. Für den Programmierer der Synthesewerkzeuge erweist sich bei der Post-Synthese-Verifikation als Vorteil, daß die Synthese und die logische Argumentation weitestgehend voneinander entkoppelt sind. Dies gilt insbesondere dann, wenn allgemeine Verifikationstechniken eingesetzt werden, da dann Syntheseprogramme und die Verifikationsprogramme unabhängig voneinander weiterentwickelt werden können.

Die Formale Synthese erzwingt dagegen vom Programmierer der Syntheseprogramme eine neue Vorgehensweise: alle elementaren Syntheseschritte müssen

durch logische Transformationen realisiert werden. Dies bedeutet für den Programmierer der Syntheseprogramme einen Mehraufwand bei der Implementierung und kann gleichzeitig dazu führen, daß die Syntheseprogramme eine geringere Laufzeiteffizienz haben. Ob Verfahren zur Formalen Synthese beim Schaltungsentwerfer Akzeptanz finden, hängt sehr davon ab, in welchem Grad die Werkzeuge einen automatisierten Syntheseablauf anbieten bzw. wie oft der Schaltungsentwerfer logische Schritte selbst interaktiv durchführen muß. Obwohl es aus theoretischer Sicht dazu keine Einschränkungen gibt, ist doch der Mehraufwand der Formalen Synthese im Vergleich zur konventionellen Synthese nicht unerheblich.

Zusammenfassend kann gesagt werden, daß alle drei Ansätze, Prä-Synthese-Verifikation, Post-Synthese-Verifikation und Formale Synthese, sinnvolle Wege darstellen, um zu nachweislich korrekten Schaltungsimplementierungen zu gelangen. Der geringste Aufwand zur Erreichung dieses Ziels entsteht dem Schaltungsentwerfer bei der Prä-Synthese-Verifikation und dem Programmierer der Synthesewerkzeuge bei der Post-Synthese-Verifikation. Die Formale Synthese stellt einen interessanten Mittelweg dar.

2.5 Stand der Technik

In diesem Abschnitt sollen mehrere Arbeiten vorgestellt werden, denen allen ein Ziel gemeinsam ist: die Sicherstellung der Korrektheit der Synthese. Die Ansätze unterscheiden sich zum Teil grundlegend in der Vorgehensweise. Ihre Vor- und Nachteile werden erörtert.

2.5.1 PBS — Proven Boolean Simplification

Bei PBS handelt es sich gemäß der vorgestellten Klassifikation um ein Prä-Synthese-Verifikationsverfahren. PBS wurde an der Cornell University von M. Aagaard und M. Leeser [AaLe91, AaLe94a, AaLe95] als ein Teil des High-Level Synthesesystems BEDROC [LeAL91, LCAL93] entwickelt. PBS ist die Implementierung des Schwache-Divisions-Algorithmus, eines Verfahrens zur Minimierung des kombinatorischen Anteils einer Schaltung auf Gatterebene [BrMc82]. Bei diesem Verfahren wird versucht, durch Wiederverwendung interner Signale, gleiche oder ähnliche Teilausdrücke einzusparen, um so die Hardwarekosten zu reduzieren.

Das Syntheseprogramm wurde in einer rein funktionalen Teilsprache von Standard ML implementiert. Die Verifikation der circa 1000 Zeilen langen Implementierung erfolgte in dem Beweissystem Nuprl [Lees92]. Dazu mußte zunächst die dabei verwendete Teilsprache von Standard ML in Nuprl eingebettet werden, um dann darauf aufbauend die Korrektheit des Syntheseprogramms verifizieren zu können. Die dazu benötigten Beweise wurden interaktiv geführt. Der Aufwand dafür war erheblich. Acht Mannmonate waren erforderlich, um die mehr als 500 dazu benötigten Teilziele manuell zu beweisen [AaLe95].

Obwohl es mit PBS gelungen ist, ein Stück Synthese-Software zu verifizieren, ist doch anzumerken, daß dabei erhebliche Einschränkungen gemacht werden

mußten. Zum einen handelt es sich bei dem Programm nur um ein sehr einfaches Syntheseverfahren, das nur einen kleinen Teil der Synthese, nämlich rein boolesche Optimierungen abdeckt, und das mit seiner Größe in keiner Weise mit kommerziellen Syntheseverfahren vergleichbar ist. Zum anderen hat man sich bei der verwendeten Programmiersprache auf eine von der Semantik einfach zu handhabende rein funktionale Programmiersprache und auf einfache Datentypen beschränkt.

2.5.2 RLEXT — Register Level Exploration Tool

Bei dem System RLEXT [KnWi89, KnWi92] handelt es sich um ein Werkzeug, mit dem Entwürfe durch Objekte im Sinne einer objektorientierten Programmierung repräsentiert und transformiert werden können. Schaltungen werden durch Quadrupel dargestellt. Die Quadrupel bestehen aus Datenfluß, Zeitverhalten, Struktur und einer Menge von Beziehungen, mit denen die Relationen zwischen Datenfluß, Zeitverhalten und Struktur beschrieben werden. Dazu steht eine Hierarchie aus Klassen und abgeleiteten Klassen zur Verfügung. Syntheseschritte sind in RLEXT Umformungen auf den Quadrupeln. Neben der reinen RT-Synthese unterstützt RLEXT auch Verfahren zur High-Level-Synthese, jedoch ohne Kontrollfluß.

Nicht jedes syntaktisch in RLEXT zulässige Konstrukt ist konsistent. Das System RLEXT berechnet aus den Quadrupeln Forderungen, die erfüllt sein müssen, damit diese wohlgeformte Schaltungsbeschreibungen darstellen. Zur Unterstützung einer korrekten Synthese bietet RLEXT zu Syntheseschritten spezifische Prüfprogramme an. Ein Syntheseschritt in RLEXT besteht aus einer Umformung auf einem Quadrupel und einem nachträglichen Aufruf eines Prüfprogramms, mit dem gewährleistet werden soll, daß der Schritt tatsächlich verhaltenserhaltend ist.

Mit RLEXT ist es auf einfache Weise möglich, für bestimmte standardisierte Syntheseschritte wie Registerallokierung (register allocation) und die Registerzuordnung (register binding) Fehler schnell aufzuspüren. Als problematisch erweist sich, daß die durch RLEXT berechneten Forderungen lediglich Plausibilitätskriterien darstellen, von denen selbst nicht formal erwiesen ist, daß sie zur Sicherstellung der Korrektheit hinreichend sind. Statt diese Forderungen durch logische Argumentation aus der Implementierung abzuleiten, werden die Forderungen durch in axiomatischer Weise konstruierte Prüfprogramme überprüft. Die Implementierung von RLEXT besteht aus 20.000 Zeilen Lisp-Code, und prinzipiell könnte jeder Programmierfehler dazu führen, daß das System falsche Aussagen über die Korrektheit der Schaltungen macht.

2.5.3 Ruby und T-Ruby

Ruby wurde zunächst lediglich als eine Sprache zur Repräsentation von Schaltungsstrukturen entwickelt; darauf aufbauend wurden dann jedoch auch Syntheseverfahren entwickelt [JoSh90, JoSh91a]. Statt, wie in klassischen Ansätzen, Schaltungsstrukturen in Form von Netzlisten zu beschreiben, werden in Ruby Schaltungsstrukturen durch Operationen auf Komponenten definiert. Einfache Beispie-

le hierfür sind Operatoren, durch die die Parallelschaltung oder die Serienschaltung zweier Komponenten ausgedrückt wird. Mit diesen Operatoren können einzelne Komponenten miteinander verknüpft und so größere Schaltungsstrukturen beschrieben werden. Insbesondere existieren auch Operatoren zur Beschreibung regulärer Strukturen wie etwa die n-fach-Parallelschaltung einer Komponente.

Zur Bündelung der Signale können Tupel und Listen verwendet werden. Listen sind insbesondere bei der Beschreibung regulärer Strukturen erforderlich. Mit speziellen Operatoren können Einzelsignale zu Signalbündeln zusammengefaßt und aus den Signalbündeln auch andererseits wieder Einzelsignale extrahiert werden. Ferner existieren Operatoren, um reguläre Verdrahtungsschemata darzustellen.

Ruby-Ausdrücke stehen für Schaltungsstrukturen, Äquivalenzumformungen auf Ruby-Ausdrücken entsprechen Schaltungstransformationen. Hierzu wurde eine Vielzahl von Gleichungen definiert, deren Anwendung das Verhalten der zugehörigen Schaltung nicht verändert. In Ruby ist es somit insbesondere auch möglich, reguläre Schaltungstransformationen effektiv zu repräsentieren und zu manipulieren, ohne dazu über von Modulgeneratoren erzeugte Netzlisten argumentieren zu müssen.

Die Semantik der Konstrukte von Ruby wurde zunächst umgangssprachlich und mit Hilfe von Schaubildern erläutert. Es entstanden jedoch auch verschiedene Ansätze, die Semantik von Ruby formal scharf zu fassen [Ross90a, Ross90b]. Das System T-Ruby [ShRa95] stellt schließlich die Implementierung eines formalen Kalküls dar, in dem Ruby-Ausdrücke repräsentiert und transformiert werden können.

2.5.4 DDD — Digital Design Derivation

Das System DDD wurde an der Indiana University entwickelt [John84, JoBB88]. In DDD werden Schaltungen durch funktionale, nichttypisierte λ-Ausdrücke dargestellt. Neben einfachen Transformationen im λ-Kalkül bietet DDD auch komplexere synthesetypische Transformationen an. Das System DDD wurde sowohl für Techniken der Schaltungsverifikation als auch zur Ableitung von Schaltungsimplementierungen durch Verfeinerung eingesetzt.

Die zum Teil sehr aufwendigen elementaren Schaltungstransformationen von DDD wurden weder auf einen elementaren Kalkül zurückgeführt, noch wurde ihre Korrektheit anderweitig verifiziert. Dies bedeutet eine Entscheidung zugunsten der Effizienz und zuungunsten der Sicherheit bzgl. der Korrektheit.

Als Vorteil dieses Ansatzes erweist sich die Simulierbarkeit der Schaltungsbeschreibungen. Die verwendete Notation für die Ausdrücke entspricht der Syntax von Lisp, so daß die Ausdrücke sehr leicht in einem Lisp-Interpreter simuliert werden können. Als problematisch erweist sich jedoch die fehlende Typisierung, die leicht zu unrealisierbaren Schaltungsbeschreibungen führen kann. Zwar stellen die in DDD beschriebenen Terme automatisch auch ausführbare Lisp-Programme dar, damit diese Beschreibungen jedoch auch realisierbare Hardwarekomponenten in adäquater Weise beschreiben, bedarf es gravierender Einschränkungen. So werden beispielsweise Listen zur Bündelung von Signalwerten eingesetzt. Sollen damit

Hardwarekomponenten modelliert werden, so ist darauf zu achten, daß sich die Anzahl der Listenelemente (Leitungen) nicht während der Laufzeit ändert. Funktionen mit wechselnder Listenlänge wären als reine Softwareprogramme betrachtet durchaus vernünftig, eine Implementierung als Hardware ist jedoch unmöglich. Leitet man in DDD aus einer Spezifikation eine Implementierung ab, so ist deshalb stets auf die Implementierbarkeit der aktuellen Schaltungsbeschreibung zu achten.

2.5.5 Dialog

Das Synthesesystem Dialog ist das bisher einzige kommerziell vertriebene Programm zur Formalen Synthese [FFFH89, FoMa89, FiFM91, MaFo91, HFFM92]. Dialog wurde bereits in der industriellen Anwendung erfolgreich eingesetzt [Busc92, BoCZ92, BBCC94]. Entwickelt wurde Dialog von AHL (Abstract Hardware Limited) in Uxbridge. Ebenfalls von AHL stammt der Theorembeweiser Lambda, der die Grundlage von Dialog darstellt. Sowohl Lambda als auch Dialog wurden in Poly ML, einem von AHL entwickelten Dialekt der funktionalen Programmiersprache ML, implementiert.

Lambda ist ein allgemeiner Theorembeweiser für Prädikatenlogik Höherer Ordnung mit polymorphen Typen, in dem Theoreme innerhalb eines eigenen Kalküls abgeleitet werden können. Als zentrale Funktion zur logischen Ableitung steht in Lambda ein Resolutionsmechanismus zur Verfügung. Dieser Resolutionsmechanismus wendet eine Regel auf ein Beweisziel an, wobei zunächst die Konklusion der Regel mit dem Beweisziel unifiziert wird. Im allgemeinen dient dieser Mechanismus dazu, Beweise zu führen, indem das Beweisziel sukzessive reduziert wird. Zur Formalen Synthese in Lambda wird ein „modifiziertes Beweisverfahren" verwendet. Bei klassischen Beweisen steht vor dem Beweis genau fest, wie das abzuleitende Theorem aussieht. Davon kann in Lambda abgewichen werden. Durch den Einsatz sogenannter flexibler Variablen kann bei der Anwendung der Regel das ursprüngliche Beweisziel an den Stellen der flexiblen Variablen verändert werden. Ausgangspunkt der Formalen Synthese in Lambda ist ein „Beweisziel" der Form $\{i\} \vdash S$. Dabei sind S die vorgegebene Spezifikation der Schaltung und i eine flexible Variable i, die für eine leere, noch nicht vorhandene Implementierung steht. Während der Synthese werden durch entsprechende Syntheseregeln Hardwarekomponenten in i eingefügt, wodurch gleichzeitig die Spezifikation S reduziert wird. Die Synthese ist abgeschlossen, sobald die Restspezifikation zu T geworden ist.

Lambda unterstützt ausschließlich synchrone Schaltungen auf Gatterebene. Ein besonderer Schwerpunkt wird in Lambda auf die Sicherstellung der Konsistenz gelegt, denn jeder Einfügeschritt könnte dazu führen, daß es in der Schaltungsstruktur zu einem Kurzschluß oder einem kombinatorischen Zyklus kommt und damit die Schaltung unrealisierbar würde. Deshalb wird in Dialog nach jeder Einfügeoperation die Schaltungsstruktur auf Kurzschlüsse und kombinatorischen Zyklen hin überprüft. Dazu muß dem System für alle Grundkomponenten mitgeteilt werden, welche Anschlüsse Ein- und welche Anschlüsse Ausgangssignale darstellen, und es müssen alle kombinatorischen Pfade zwischen Ein- und Aus-

gangssignalen angegeben werden. Dialog überprüft für Schaltungsstrukturen, die sich aus diesen Grundkomponenten zusammensetzen, daß sie weder Kurzschlüsse noch kombinatorische Zyklen enthalten. Außerdem bestimmt das System für zusammengesetzte Schaltungen die notwendigen Informationen (Zuordnung zu Ein- und Ausgabesignalen, kombinatorische Pfade), um auch diese Schaltungen selbst wieder als Komponenten in anderen Strukturen verwenden zu können.

Das System Dialog macht zunächst keine Einschränkungen bezüglich der zu synthetisierenden Spezifikation. Insbesondere können auch Spezifikationen angegeben werden, zu denen es keine Implementierung gibt. Dies stellt sich dann oft erst während des Syntheseablaufs heraus. Auch ist es i. allg. nicht möglich, eine Schaltungsspezifikation allein durch Anwendung von Einfügeregeln zu reduzieren, sondern es sind zusätzliche, interaktiv auszuführende logische Umformungsschritte erforderlich. Derartige Schritte erfordern vom Schaltungsentwerfer fundierte logische Kenntnisse und Erfahrungen im Umgang mit dem Theorembeweiser Lambda.

Dialog ist vor allem ein interaktives Entwurfswerkzeug, das mit seiner Oberfläche an graphische Schaltungsentwurfswerkzeuge erinnern soll. Neben den rein interaktiven, mit der Maus gesteuerten Einfügeoperationen gibt es zwar noch die Möglichkeit, eine Einfügeoperation beliebig oft und so häufig wie möglich automatisch auszuführen. Der so erzielbare Grad an Automatisierung ist jedoch gering und bei weitem nicht vergleichbar mit konventionellen Syntheseprogrammen.

2.5.6 Veritas

Das System Veritas wurde an der Kent State University entwickelt [HaLD89, HaDa92]. Es basiert auf einem Theorembeweiser für typisierte Prädikatenlogik Höherer Ordnung, der für diesen Zweck um sog. „Techniken” erweitert wurde. Mit Techniken werden Verfeinerungsschritte durchgeführt, durch die aus der vorgegebenen Schaltungsbeschreibung die Implementierung generiert werden kann. Eine Technik besteht aus zwei Anteilen: einer Teilzielfunktion und einer Validierungsfunktion. Die Teilzielfunktion zerlegt das Beweisziel in Teilziele und ist eine in beliebiger Weise implementierte Funktion. Die Korrektheit des Verfeinerungsschritts wird durch diesen Schritt noch nicht gewährleistet. Durch die Validierungsfunktion wird aus den bewiesenen Teilzielen das Gesamtziel abgeleitet. Dies ist erst dann möglich, wenn die Verfeinerung der Teilschritte bereits abgeschlossen ist.

Eine Synthese, die in Veritas mit Hilfe der Techniken durchgeführt wird, besteht deshalb aus zwei Phasen: eine Phase, in der die Zielbeschreibung mit Hilfe der Teilzielfunktionen sukzessive in mehrere Teile zerlegt wird, und eine zweite Phase, in der durch Anwendung der zugehörigen Validierungsfunktion aus den bewiesenen Teilzielen das Gesamtziel abgeleitet wird.

Das System Veritas bietet zur Formalen Synthese nur sehr allgemeine Hilfsmittel an. Als Einzelschritte stehen lediglich logische Elementarumformungen und sehr einfache Schaltungstransformationen (Zerlegung einer Komponente in zwei Teile etc.) zur Verfügung. Für den Schaltungsentwerfer bedeutet dies, daß er Schaltungstransformationen nicht direkt durch die in der Synthese üblicherweise

verwendeten Syntheseschritte ausdrücken kann, sondern diese auf der Ebene elementarer logischer Umformungen ausdrücken muß, was grundlegende logische Kenntnisse und Erfahrung im Umgang mit dem Theorembeweiser voraussetzt.

2.5.7 Forvertis

Das an der Universität Passau entwickelte System Forvertis [GrMT94, LoMu97] baut auf bereits bestehenden Syntheseprogrammen zur High-Level-Synthese auf und ergänzt diese um einen Verifikationsanteil, durch den sichergestellt werden soll, daß die durchgeführte Transformation korrekt ist. Forvertis ist eine Abkürzung für die „Formale Verifikation von Transformationsregeln in der Synthese digitaler Schaltungen". In dem betrachteten High-Level-Syntheseschritt wird eine Datenflußbeschreibung auf eine Schaltungsstruktur auf Register-Transfer-Ebene abgebildet. Sowohl Ein- als auch Ausgabe des Syntheseschritts liegen als VHDL-Beschreibung vor. Diese werden durch Parser eingelesen und in eine logische Repräsentation innerhalb des Theorembeweisers HOL umgewandelt.

Das Syntheseprogramm liefert neben dem Syntheseergebnis zusätzliche Informationen, die während des Syntheseschritts intern bestimmt und die normalerweise nicht nach außen gegeben werden: eine Tabelle mit dem Ablaufplan (schedule) und eine Tabelle mit der Allokierung (allocation) und der Zuordnung (binding). Diese Information dient als Hilfsmittel für die anschließende Verifikation. Mit ihr ist es möglich, den Zusammenhang zwischen den Operationen und Zwischenwerten der Eingabeschaltungsbeschreibung einerseits und den Operationseinheiten und Registern der Strukturbeschreibung andererseits herzustellen. Auch diese Information wird, wie die Ein- und Ausgabeschaltungsbeschreibungen, in eine logische Repräsentation übersetzt. Anschließend wird damit und aus den logischen Beschreibungen für die Ein- und Ausgabe des Syntheseschritts ein Beweisziel aufgestellt, mit dem die funktionale Korrektheit des Syntheseschritts sichergestellt werden soll. Das Beweisziel wird dann mit Hilfe von vorab bewiesenen Hilfstheoremen in HOL abgeleitet. Dies geschieht großteils automatisiert.

Daß das während des Verifikationsschritts konstruierte Beweisziel tatsächlich hinreichend ist, um die Korrektheit des Syntheseschritts zu gewährleisten, wurde nicht explizit bewiesen. Auch wird bei der Verifikation lediglich der Datenflußanteil der Implementierung betrachtet, ein eventueller Fehler im Kontrollteil kann deshalb nicht erkannt werden.

Forvertis stellt kein allgemeines Verifikationssystem dar, sondern ein System, das ganz spezifisch auf die Sicherstellung der Korrektheit eines ganz bestimmten Syntheseschritts, nämlich der High-Level-Synthese von Datenpfadbeschreibungen, zugeschnitten ist. Für diese Problemstellung gibt es in der Literatur verschiedene Ansätze, die sich in bezug auf die Qualität der damit erzielten Implementierungen (Aufwand an Hardwareressourcen, benötigte Takte, erzielbare Taktfrequenz) und die Laufzeitkomplexität für den Syntheseschritt unterscheiden (siehe [GDWL94]).

Forvertis erlaubt es, beliebige Syntheseprogramme einzubinden. Voraussetzung ist jedoch, daß sie der vorgegebenen Schnittstelle entsprechen. Zu der Schnittstelle gehört in Forvertis nicht nur ein spezielles VHDL-Format für die

Eingabe- und Ausgabeschaltungsbeschreibung sondern auch eine in einer besonderen Notation anzugebenden Beschreibung von Ablaufplan, Allokierung und Zuordnung. Konventionelle Syntheseprogramme berechnen diese Information i. allg. nur intern und stellen sie nach außen nicht zur Verfügung. Bezeichnend für Ansätze zur Post-Synthese-Verifikation ist, daß die Information darüber, welche Schritte wie während der Synthese abgelaufen sind (hier: Ablaufplan-Tabelle und Allokierungs-/Zuordnungs-Tabelle), von elementarer Bedeutung für eine effiziente Beweisführung ist.

2.6 Diskussion

Gemäß der in Abschnitt 2.4 vorgestellten Klassifizierung handelt es sich bei PBS um Prä-Synthese-Verifikation, bei Forvertis um Post-Synthese-Verifikation und bei allen anderen Arbeiten um Formale Synthese. Auf die konzeptionellen Unterschiede zwischen diesen drei Klassen von Ansätzen wurde bereits in Abschnitt 2.4 eingegangen. Neben dieser sehr groben Klassifikation gibt es jedoch noch andere wesentliche Unterscheidungsmerkmale zwischen den Verfahren, auf die nun eingegangen werden soll. Die wichtigsten Aussagen sind in Tabelle 2.1 zusammengefaßt.

2.6.1 Allgemeine und hardwarespezifische Kalküle

Alle Systeme behaupten, die Korrektheit der Synthese garantieren zu können. Die Systeme unterscheiden sich in der formalen Basis, auf der diese Korrektheitsaussage fußt. Grob kann zwischen allgemeinen logischen Kalkülen und selbstdefinierten, hardwarespezifischen Kalkülen unterschieden werden. Bei den Systemen Dialog, Veritas und Forvertis werden alle Schaltungstransformationen in mathematischer Weise auf Regelanwendungen in einem allgemeinen Kalkül zurückgeführt. Bei diesen Systemen bilden Theorembeweiser für Prädikatenlogik höherer Ordnung die Grundlage. Dagegen werden bei RLEXT und Ruby eigene Sprachen mit eigenen Transformationen definiert. Das System DDD stellt einen Mittelweg dar: es basiert zwar auf dem λ-Kalkül, dieser wird jedoch um hardwarespezifische komplexe Regeln angereichert.

Bei der Abwägung zwischen dem Einsatz von logischen Kalkülen und hardwarespezifischen Kalkülen sind vor allem die Aspekte Effizienz und Sicherheit zu beachten. Die Verwendung eines allgemeinen Kalküls bedeutet, daß logische Ausdrücke verwendet werden müssen, um Schaltungen zu repräsentieren und daß alle elementaren Schaltungstransformationen auf logische Regelanwendungen zurückgeführt werden müssen. Bei hardwarespezifischen Kalkülen können hingegen beliebige Datenstrukturen zur Schaltungsrepräsentation eingesetzt werden, und für die Implementierung der elementaren Schaltungstransformationen gibt es keine Beschränkungen. Dies führt dazu, daß der Einsatz von hardwarespezifischen Kalkülen prinzipiell zu effizienteren Verfahren führt. Gleichzeitig ist jedoch zu beachten, daß die hardwarespezifischen Kalküle eine große Zahl komplexer Regeln enthalten können, deren Implementierung sicherheitskritisch in bezug

auf die Richtigkeit der Korrektheitsaussagen des Systems ist, da die fehlerhafte Implementierung der Regeln i. allg. dazu führt, daß fehlerhafte Schaltungsimplementierungen entstehen. Es gibt verschiedene Arbeiten mit dem Ziel, hardwarespezifische Kalküle nachträglich auf eine allgemeine logische Basis zu stellen [ShRa95, AnPP93].

2.6.2 Grad der Automatisierung

Für den Schaltungsentwerfer von elementarer Bedeutung ist der Grad der durch die Werkzeuge erzielbaren Automatisierung. Die Systeme PBS und Forvertis sind vollständig automatisiert und unterscheiden sich für den Schaltungsentwerfer nicht von konventionellen Verfahren. Das System RLEXT ist für eine Automatisierung ausgelegt. Bei den anderen Systemen ist der Umgang mit den Werkzeugen jedoch fast vollständig interaktiv. Einen gewissen Grad an Automatisierung bietet das System Dialog mit den Reduktionstransformationen und seinen Mehrfach-Einfüge-Operationen. Im allgemeinen lassen sich interaktive Beweisschritte jedoch nicht vermeiden. Die Tatsache, daß beliebige logische Terme als Eingabe zugelassen werden, also insbesondere auch Spezifikationen, zu denen es gar keine Implementierung gibt, schließt eine vollständige Automatisierung jedoch von vornherein aus.

2.6.3 Einsatzgebiet, Erweiterbarkeit

Die vorgestellten Arbeiten decken jeweils nicht das gesamte Spektrum der Schaltungssynthese ab, sondern beschränken sich jeweils auf einen definierten Bereich. Die Systeme PBS und Dialog beschränken sich auf die Gatterebene, die Systeme RLEXT, Ruby, DDD und Veritas erlauben zusätzlich Beschreibungen auf der RT-Ebene, und mit den Systemen RLEXT und Forvertis können auch einfache High-Level-Syntheseschritte gehandhabt werden.

Das System PBS ist auf eine ganz bestimmte Implementierung eines Syntheseschritts zugeschnitten. Änderungen an der Implementierung des Syntheseprogramms sind dabei nicht möglich. Flexibler sind die Systeme RLEXT und Forvertis. Zwar sind beide Systeme auf ganz spezifische Syntheseschritte festgelegt, es können jedoch innerhalb des durch die Schnittstellen vorgegebenen Rahmens beliebige Implementierungen angebunden werden. Allgemeiner einsetzbar sind die Verfahren Ruby, DDD und Dialog. Sie bieten zwar nur einen elementaren Satz von Schaltungsumformungen, diese können jedoch zu beliebigen komplexeren Syntheseschritten kombiniert werden.

Im allgemeinen gewährleisten die Werkzeuge nur die Einhaltung der funktionalen Korrektheit. Die Systeme RLEXT und Forvertis gehen hier einen Schritt weiter und prüfen auch, ob vorgegebene Beschränkungen bzgl. der Hardwareressourcen und Zeitbeschränkungen eingehalten wurden.

	PBS	RLEXT	Ruby	DDD	Dialog	Veritas	Forvertis
Vorgehensweise							
- Prä-Synthese-Verifikation	X						
- Formale Synthese		X	X	X	X	X	
- Post-Synthese-Verifikation							X
Kalkül							
- Hardware-spezifisch		X	X	X			
- allgemeiner Logikkalkül					X	X	X
Unterstützte Abstraktionsebenen							
- Gatterebene	X	X	X	X	X	X	
- RT-Ebene			X	X		X	X
- Algorithmische Ebene							X
Automatisierung der Synthese	X	X					X
Funktionale Schaltungsbeschreibung				X			
Typisierung	X		X		X	X	X
Konsistenzsicherung		X			X		
Berücksichtigung nichtfunktionaler Aspekte		X					X

Tabelle 2.1: Vergleich der verschiedenen Verfahren

Kapitel 3

Eine Methodik zur Formalen Synthese

In diesem Kapitel sollen die grundlegenden Konzepte und Strategien des mit dieser Arbeit vorgestellten Ansatzes zur Formalen Synthese beschrieben werden. Diese sind nicht zuletzt motiviert durch die in Kapitel 2 erörterten Probleme bestehender Ansätzen. Die einzelnen konzeptionellen Entscheidungen werden deshalb im Kontext der betreffenden Probleme erörtert.

3.1 Formale Synthese in einem Theorembeweiser

Eine strategische Entscheidung bei dieser Arbeit ist, daß die Schaltungstransformationen auf die elementaren Regelanwendungen in einem logischen Kalkül zurückgeführt werden sollen. Es sollen nicht nur, wie dies bisweilen geschieht, Plausibilitätskriterien für die Korrektheit überprüft, sondern es soll ein durchgängiger Beweis geführt werden. Diese Entscheidung bedeutet einen erheblich größeren Aufwand verglichen mit zahlreichen ad hoc-Lösungen, führt jedoch zu einem wesentlich höheren Grad an Vertrauenswürdigkeit bzgl. der Korrektheit.

Die hier vorgestellten Verfahren wurden in einem Theorembeweiser namens HOL [GoMe93] für Prädikatenlogik höherer Ordnung implementiert. Der Theorembeweiser stellt allgemeine logische Beschreibungsmittel und einen zugehörigen Kalkül bestehend aus 5 Axiomen und 8 Regeln zur Verfügung. Für Theoreme gibt es in HOL einen eigenen, geschützten Datentyp. Werte vom Typ Theorem können nur über Axiome und Regelanwendungen erzeugt werden. Dadurch ist gewährleistet, daß alle Ergebnisse vom Typ Theorem durch mathematische Ableitung entstanden sein müssen. Die Vertrauenswürdigkeit von Beweisen, die in HOL geführt werden unterscheiden sich damit grundlegend von den klassischen mit Papier und Bleistift hergeleiteten Beweisen aus der Mathematik, bei denen nur durch ein interpretierendes Lesen der Beweise versucht werden kann, deren Stimmigkeit zu validieren.

Aufbauend auf den allgemeinen logischen Beschreibungsmitteln, Axiomen und Regeln, wurde zunächst eine hardwarespezifische Teilsprache definiert und zu dieser wurden dann elementare Schaltungstransformationen abgeleitet. Diese elementaren Syntheseschritte können dann zu komplexeren Syntheseprogrammen zusammengefügt werden. Sowohl für die einfachen als auch für komplexere Synthesetransformationen gilt infolgedessen, daß sie auf den elementaren logischen Umformungen des Theorembeweisers aufbauen und damit von vornherein die logische Korrektheit der Synthese garantieren.

Theorembeweiser haben sich in der industriellen Praxis bisweilen den Ruf erworben, mathematische Spielzeuge zu sein, die nur von Logikexperten beherrscht werden können. Im Gegensatz zu Beweiswerkzeugen, die sich auf einfache entscheidbare Logiken beschränken und in denen vollständig automatisierte Entscheidungsverfahren für die gesamte betrachtete Logik zur Verfügung stehen (Model-Checking), bieten Theorembeweiser sehr mächtige, jedoch nichtentscheidbare Logiken und lediglich einen Satz von Axiomen und Regeln, mit denen der Anwender selbst nach dem Beweis suchen kann. Zwar bieten Theorembeweiser auch komplexe, auf den elementaren Regeln aufbauende logische Transformationen und Entscheidungsverfahren an, mit denen bestimmte Klassen von Beweisen automatisch oder teilautomatisch erbracht werden können, für allgemeine Verifikationsaufgaben sind jedoch vom Anwender fundierte Kenntnisse in Logik und in den Details des Theorembeweisers unabdingbar. Der hohe Aufwand für die Einarbeitung in Theorembeweiser ist mit ein Grund für deren geringe Akzeptanz.

Dem steht gegenüber, daß automatisierte Entscheidungsverfahren wie Model-Checking nur sehr einfache Beschreibungsmittel unterstützen, mit denen Schaltungsbeschreibungen auf höheren Abstraktionsebenen nur sehr unzureichend modelliert werden können. Dies führt dann oft dazu, daß Problemstellungen, die auf höheren Abstraktionsebenen formuliert wurden, in ein äquivalentes Problem in temporallogischen Formeln „übersetzt" werden. Dabei stellt sich natürlich sofort die Frage nach der Korrektheit dieser Übersetzung, zumal i. allg. auch große Teile des Beweises in eine solche Übersetzung verlagert werden können. Als zweites Problem erweist sich, daß aufgrund der Komplexität der Entscheidungsverfahren nur relativ kleine Schaltungsgrößen beherrscht werden können. Die Komplexität der Entscheidungsverfahren, die beispielsweise auf Zustandstraversierungen mit BDDs basieren, sind weit davon entfernt, Schaltungen in einer realen Größe beherrschbar zu machen.

Im Bereich der Formalen Synthese spielen, wie in Abschnitt 2.5 beschrieben, Theorembeweiser eine wesentliche Rolle. Das liegt daran, daß man bei einer Formalen Synthese vor einem grundlegend anderen Problem steht als bei einer Post-Synthese-Verifikation. Die Problemstellung der Post-Synthese-Verifikation lautet: „Gegeben die Eingabe und die Ausgabe eines Synthesedurchlaufs. Ist die Ausgabe der Synthese korrekt bezüglich der Eingabe?". Dieses Entscheidungsproblem ist für einfache Logiken wie die Aussagenlogik NP-vollständig und für ausdrucksstärkere Logiken wie die Prädikatenlogik Höherer Ordnung unentscheidbar. Die Problemstellung der Formalen Synthese lautet hingegen: „Es ist eine

Schaltung auf korrekte Weise durch Anwendung logischer Transformationen zu synthetisieren". Dieses Problem ist weder NP-vollständig noch unentscheidbar. Die Beschränkung auf möglichst effizient entscheidbare Logiken ist deshalb weder notwendig noch sinnvoll.

3.2 Beschränkung auf funktionale Schaltungsbeschreibungen

In dem in dieser Arbeit vorgestellten Kalkül werden Schaltungen stets durch Funktionen beschrieben, die Eingangssignale auf Ausgangssignale abbilden. Da real existierende Schaltungen stets deterministisch sind, erscheint diese Form der Modellierung als direkt einsichtig.

Wie bereits in Abschnitt 2.2.2 dargestellt, kann es für die Schaltungssynthese jedoch durchaus auch sinnvoll sein, Schaltungsbeschreibungen zu verwenden, die nicht eindeutig festgelegt sind, die also nicht für eine einzelne Schaltung stehen, sondern eine Menge von Schaltungsbeschreibungen charakterisieren. Aus diesem Grunde werden Schaltungen gern auch allgemeiner als Relationen beschrieben.

Die Modellierung von Schaltungen durch Relationen ist zugleich auch die im Bereich der Schaltungsverifikation am häufigsten angewandte Vorgehensweise [Melh93]. Wenn Schaltungsbeschreibungen in Zwischenstadien der Synthese auch mehrdeutig sein dürfen, so muß doch immer im Auge behalten werden, daß das Ergebnis einer Schaltungssynthese stets eine einzelne Schaltung sein muß. Bei der Formalen Synthese besteht bei jedem Verfeinerungsschritt die Gefahr, zu widersprüchlichen Schaltungsbeschreibungen zu kommen, also zu Schaltungsbeschreibungen, die eine leere Menge real konstruierbarer Schaltungen repräsentieren. Inkonsistenzen sind mit dem hier gewählten funktionalen Ansatz bereits syntaktisch ausgeschlossen. Der Aufwand, der bei anderen Systemen, die eine relationale Beschreibung verwenden, zur Überprüfung der Stimmigkeit einer Schaltungsbeschreibung entsteht, wird so vermieden.

Ein weiterer Vorteil der in dieser Arbeit gewählten funktionalen Schaltungsbeschreibung ist die einfache Simulierbarkeit. Es werden durchgängig λ-Ausdrücke verwendet, die innerhalb eines beliebigen Interpreters für funktionale Programmiersprachen evaluiert werden können.

3.3 Aufteilung in Entwurfsraumuntersuchung und Schaltungstransformation

In dieser Arbeit soll bei der Synthese zwischen Teilschritten zur Entwurfsraumuntersuchung und Teilschritten zur Schaltungstransformation unterschieden werden. Viele Syntheseschritte lassen sich, wie später noch genauer ausgeführt werden soll, klar in zwei Phasen aufteilen: in der ersten Phase findet die Entwurfsraumuntersuchung statt und in der zweiten Phase die Schaltungstransformation (siehe

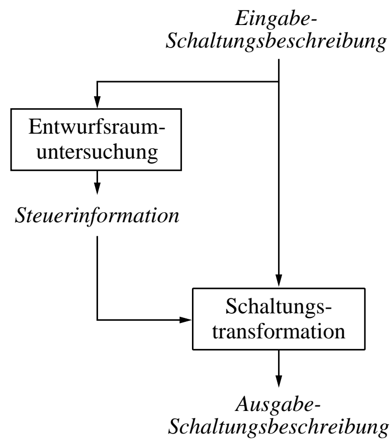


Abbildung 3.1: Entwurfsraumuntersuchung und Schaltungstransformation

Bild 3.1). Bei der Entwurfsraumuntersuchung wird mittels geeigneter Verfahren bestimmt, *wie* die eigentliche Schaltungstransformation durchgeführt werden soll. Die dabei berechneten Daten werden der eigentlichen Schaltungstransformation als Steuerinformation übergeben, die damit die vorgegebene Schaltungsbeschreibung auf die neue Schaltungsbeschreibung abbildet.

Bei dem vorgestellten Schema wird die Korrektheit eines jeden Syntheseschritts allein durch die zweite Phase, die Schaltungstransformation, bestimmt, wohingegen die erste Phase, die Entwurfsraumuntersuchung, lediglich Einfluß auf die Qualität des Ergebnisses hat. Während die Schaltungstransformation selbst oft sehr einfach zu implementieren, jedoch sicherheitskritisch in bezug auf die Korrektheit der Synthese ist, steckt das Know-how und die Raffinesse eines Syntheseprogramms in der Entwurfsraumuntersuchung.

Als Beispiel betrachte man die Zustandskodierung. Ausgehend von einer Schaltungsbeschreibung mit symbolischen Zustandswerten wird zunächst eine boolesche Kodierung der Zustände bestimmt — zum Beispiel in Form einer Tabelle. In dieser Phase, der Entwurfsraumuntersuchung, müssen verschiedene Kosten-/Qualitätsaspekte gegeneinander abgewogen werden. Die verwendete Kodierung hat u.a. Einfluß auf die Anzahl der benötigten Speicherelemente, die Häufigkeit der Wechsel der Werte in den Speicherelementen und damit den Energieverbrauch und schließlich auch auf die Anzahl der für den kombinatorischen Anteil benötigten Komponenten. Besonders im frühen Stadium der Synthese lassen sich derartige Entscheidungen und deren Konsequenzen nur schwer abschätzen. Dabei können ausgefeilte und aufwendige Verfahren eingesetzt werden. Nachdem man eine Kodierung bestimmt hat, wird durch eine Schaltungstransformation die ursprüngliche, symbolisch kodierte Schaltung auf eine äquivalente, boolesch kodierte Schaltung abgebildet. Dabei dient die Kodierungstabelle als Steuerinformation

für die Schaltungstransformation. Die eigentliche Schaltungstransformation ist vergleichsweise einfach zu implementieren. Kann man für diesen Schritt jedoch garantieren, daß er zu beliebigen Steuerinformationen Schaltungen auf äquivalente Schaltungen abbildet, so ergibt sich daraus auch die Korrektheit des gesamten Syntheseschritts. Gegenstand dieser Arbeit soll es deshalb sein, parametrisierbare Schaltungstransformationen zusammenzustellen, an die die Verfahren zur Entwurfsraumuntersuchung in flexibler Weise angebunden werden können.

In bestehenden Syntheseprogrammen findet man eine derartig strikte Aufteilung in Phasen zur Entwurfsraumuntersuchung und Phasen zur Schaltungstransformation oft nicht. Stattdessen werden diese beiden Vorgänge oft eng miteinander vermascht. Dies erschwert die Sicherstellung der Korrektheit der Synthese, denn große Teile der Syntheseprogramme sind in Bezug auf die Sicherstellung der Korrektheit nicht relevant. Eine Aufteilung nach dem vorgestellten Schema stellt somit bereits einen ersten Schritt in Richtung der Sicherstellung der Korrektheit der Synthese dar.

3.4 Automatisierung und Interaktion

Bei der Schaltungssynthese konkurriert die Anwendung von Syntheseprogrammen mit der Durchführung manueller Schritte. Zwar besteht aufgrund der Komplexität der Schaltungen der Wunsch, möglichst automatisierte Synthesewerkzeuge einzusetzen, im Einzelfall sind jedoch oft noch manuelle Nachbesserungen erforderlich. Die Gefahr, daß sich durch manuell durchgeführte Syntheseschritte Fehler einschleichen, ist bei weitem höher als bei gut getesteten Syntheseprogrammen.

Die in Kapitel 2 vorgestellten Arbeiten sind hauptsächlich für eine interaktive Synthese ausgelegt und unterstützen eine Automatisierung nur zum Teil. Die in dieser Arbeit vorgestellten Schaltungstransformationen können mit den in HOL bereits enthaltenen Mitteln in einfacher Weise kombiniert und zu größeren Syntheseschritten und ganzen Syntheseprogrammen zusammengefügt werden. Dabei ist das System weder auf eine rein interaktive noch auf eine vollständig automatisierte Vorgehensweise festgelegt. Einzelne Syntheseschritte können immer auch interaktiv angestoßen werden. Schaltungen können jedoch nicht in beliebiger Weise, sondern nur durch vorgegebenen Schaltungstransformationen umgeformt werden. Dies stellt zum einen eine Beschränkung für den Schaltungsentwerfer dar, der zur Erzielung eines bestimmten Ergebnisses erst nach den dafür geeigneten Schaltungstransformationen suchen muß, stellt aber andererseits sicher, daß nur logisch korrekte Schaltungstransformationen durchgeführt werden können.

Kapitel 4

Schaltungsbeschreibungen auf RT- und Gatterebene

In diesem Kapitel wird eine Hardwarebeschreibungssprache für Schaltungen auf RT- und Gatterebene vorgestellt. Die Sprache bildet eine Teilmenge der Logik des Theorembeweisers HOL. Im Gegensatz zu anderen Hardwarebeschreibungssprachen, deren Semantik oft nur umgangssprachlich festgelegt wird, verfügt diese Sprache somit über eine formal exakt definierte Semantik. Neben der formalen Fundierung der Sprache liegen die wesentlichen Vorteile der Sprache in der Konsistenz, der Simulierbarkeit und der Implementierbarkeit aller Sprachkonstrukte. Auch hierin unterscheidet sich diese Arbeit von zahlreichen anderen Ansätzen. In diesem Kapitel wird zunächst nur die Sprache vorgestellt, eine Diskussion und ein Vergleich mit anderen Ansätzen folgt im anschließenden Kapitel.

Grundsätzlich ist zu dem hier vorgestellten Ansatz anzumerken, daß er sich ausschließlich auf rein synchrone Schaltungen bezieht. Die verwendete Zeitabstraktion ist diskret: eine Zeiteinheit entspricht einer Taktperiode. Gatter werden verzögerungsfrei modelliert, elementare Speicherkomponenten verzögern die Signale um eine Zeiteinheit. In diesem Ansatz wird explizit zwischen kombinatorischen Schaltungsbeschreibungen (Schaltnetzen) und sequentiellen Schaltungsbeschreibungen (Schaltwerken) unterschieden. Dabei bauen sequentielle Schaltungsbeschreibungen auf kombinatorischen Schaltungsbeschreibungen auf. Es werden lediglich Strukturen zugelassen, bei denen alle Teilkomponenten kombinatorisch sind. Die Konsequenzen, die sich daraus ergeben, werden im fünften Kapitel erörtert.

Das Kapitel gliedert sich in vier Abschnitte. Zunächst werden einfache kombinatorische Schaltungsbeschreibungen vorgestellt, mit denen rein boolesche Schaltungen auf Gatterebene modelliert werden können. Im nächsten Abschnitt folgen dann abstraktere Mittel zur Beschreibung kombinatorischer Schaltungen, und es wird erläutert, wie diese auf rein boolesche kombinatorische Schaltungen zurückgeführt werden können. Der dritte Abschnitt beschreibt, wie auf der Ba-

sis der kombinatorischen Schaltungsbeschreibungen sequentielle Schaltungen beschrieben werden, und im letzten Abschnitt wird erläutert, wie Schaltungen durch Auswertung simuliert werden können.

4.1 Kombinatorische Schaltungsbeschreibungen

Kombinatorische Schaltungen werden durch Funktionen modelliert. Funktionen stehen für Schaltungen mit mindestens einem Eingang. Jede Funktion $\iota \rightarrow \omega$ beschreibt eine Abbildung von einem Signaltyp ι auf einen Signaltyp ω. Zeitliche Aspekte werden dabei nicht berücksichtigt. Die Typen ι und ω können beliebig sein. Es können insbesondere auch Signalbündel bestehend aus mehreren Einzelsignalen verwendet werden. Welche Funktionen im einzelnen zulässig sind und wie die zugehörigen Datentypen der Signale ι und ω aussehen, soll in diesem Abschnitt erläutert werden.

In kombinatorischen Schaltungsstrukturen kann es notwendig werden, konstante Werte an Eingängen von Teilkomponenten anzulegen. Konstante Werte haben einen beliebigen Signaltyp ω.

4.1.1 Grundkomponenten

Die Gatterebene ist die unterste Entwurfsebene, die hier betrachtet werden soll. Auf der Gatterebene werden alle elementaren Signale durch boolesche Werte repräsentiert. Signale $\alpha_1, \alpha_2, \ldots \alpha_n$ können durch Skalarproduktbildung zu Tupeln $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$ gebündelt werden. Dies darf in hierarchischer Weise erfolgen. Die Signale $\alpha_1, \alpha_2, \ldots \alpha_n$ dürfen also sowohl einfache boolesche Signale als auch selbst wieder zusammengesetzte Signale sein.

Auf der Gatterebene sind die drei booleschen Operationen AND, OR und INV sowie die Konstanten T und F definiert (siehe Tabelle 4.1). Diese Konstrukte können technisch sehr einfach durch logische Gatter bzw. durch die Versorgungsspannung und die Spannung auf der Masseleitung realisiert werden.

Daß diese Menge von Konstrukten eine Basis im Sinne der Aussagenlogik ist, daß mit ihr also beliebige boolesche Funktionen beschrieben werden können, ist offensichtlich. Die Festlegung auf genau diese Basis geschah willkürlich; andere Basen wären ebenfalls möglich gewesen.

4.1.2 Strukturen

Kombinatorische Schaltungsstrukturen stellen Datenflußgraphen (DFG) dar, deren Knoten den Teilkomponenten und deren gerichtete Kanten den Signalleitungen entsprechen. Datenflußgraphen sind azyklisch. In der Logik sollen kombinatorische Strukturen durch eine funktionale Schreibweise im Sinne des λ-Kalküls dargestellt werden. Dazu sollen zwei Notationsformen verwendet werden, die als DFG-Terme bzw. als FDFG-Terme (flache DFG-Terme) bezeichnet werden sollen. Sowohl mit DFG-Terme als auch mit FDFG-Termen lassen sich beliebige Datenflußgraphen

AND	Typ: $\mathsf{bool} \times \mathsf{bool} \rightarrow \mathsf{bool}$ boolesche Konjunktion
OR	Typ: $\mathsf{bool} \times \mathsf{bool} \rightarrow \mathsf{bool}$ boolesche Disjunktion
INV	Typ: $\mathsf{bool} \rightarrow \mathsf{bool}$ boolesche Negation
T	Typ: bool boolescher Wert "wahr"
F	Typ: bool boolescher Wert "falsch"

Tabelle 4.1: Grundkonstrukte auf Gatterebene

darstellen. DFG-Terme stellt eine Obermenge der FDFG-Terme dar, wobei es jedoch zu jedem DFG-Term einen äquivalenten FDFG-Term gibt. FDFG-Terme sind eine besondere Repräsentationsform, die eine sehr einfache Beziehung zu den üblichen Netzlistenstrukturen hat und auf deren Basis auch strukturelle Transformationen sehr einfach beschrieben werden können. Für bestimmte kombinatorische Transformationen eignet sich diese Darstellung jedoch nicht. In einem solchen Fall muß der FDFG-Term zunächst in einen geeigneten DFG-Term überführt werden, der kein FDFG-Term ist.

Zunächst einige Anmerkungen zur verwendeten Notation. Verwendet werden λ-Terme, also Ausdrücke zur Modellierung von Funktionen gemäß dem λ-Kalkül. Sei $p[x]$ ein beliebiger Term mit freien Vorkommen der Variablen x. Die Schreibweise $(\lambda x.\, p[x])$ steht für die Funktion, die x auf $p[x]$ abbildet. Dabei darf an der Stelle von x auch statt einer einzelnen Variablen ein Tupel von Variablen stehen. Der Ausdruck $\mathsf{let}\; x = f(a) \;\mathsf{in}\; p[x]$ ist eine Schreibweise, die für den Ausdruck $(\lambda x.\, p[x])\; (f(a))$ steht. Beide Ausdrücke sind äquivalent zu $p[f(a)/x]$, also dem Term, der entsteht, wenn in $p[x]$ alle Vorkommen von x durch $f(a)$ substituiert werden.

FDFG-Terme

FDFG-Terme sind funktionale Darstellungen von Schaltungsstrukturen. Die nachfolgende Definition beschreibt den syntaktischen Aufbau der FDFG-Terme durch Syntaxregeln in Backus-Naur-Form. Dabei stehen Variablen für Bezeichner, die beliebig gewählt werden dürfen. Die verwendeten Operatoren und Konstanten müssen bereits definiert sein, d. h. sie müssen entweder Grundfunktionen sein, oder sie müssen selbst wieder aus Grundfunktionen oder anderen definierten Operatoren abgeleitet worden sein.

Definition 1 (FDFG-Terme) *FDFG-Terme sind genau die wohltypisierten λ-Terme ohne freie Variablen, die gemäß den folgenden Syntaxregeln aufgebaut sind:*

Variablenblock ::=
  Variable |
  "`(`" *Variablenblock* { "`,`" *Variablenblock* } "`)`"

Einfachausdruck ::=
  Konstante |
  Operator "`(`" *Variablenblock* "`)`"

FDFG-Term ::=
  "λ" *Variablenblock* "`.`"
  { "`let`" *Variablenblock* "`=`" *Einfachausdruck* "`in`" }
  Variablenblock

Abbildung 4.1 zeigt ein Beispiel für eine kombinatorische Struktur und den zugehörigen DFG-Term. In der Struktur kommen lediglich die oben eingeführten Grundkomponenten vor. Die verwendeten Signale sind in diesem Beispiel alle boolesch.

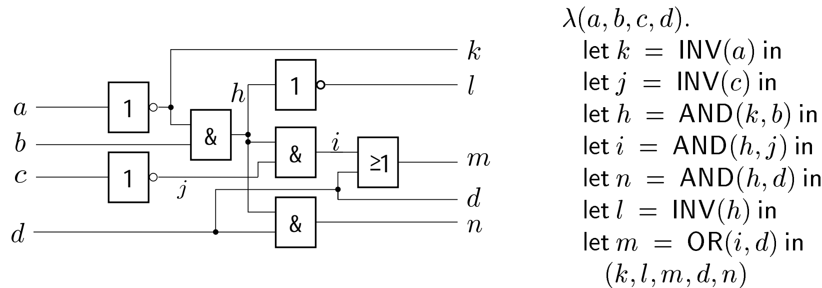


Abbildung 4.1: Kombinatorische Struktur

Für FDFG-Terme werden zwei wichtige Einschränkungen gemacht: sie sollen wohltypisiert sein, und sie sollen keine freien Variablen enthalten. Auf die Bedeutung dieser beiden Einschränkungen soll nun näher eingegangen werden.

Im Theorembeweiser HOL sind alle Terme wohltypisiert. Die Typüberprüfung findet bei der Erstellung der Terme statt. Dadurch, daß in den FDFG-Termen die Teilkomponenten bereits einen definierten Typ haben, ergibt sich eine Zuordnung der Typen für die angeschlossenen Signale. Für die FDFG-Terme bedeutet die Eigenschaft wohltypisiert zu sein, daß die Typzuordnung für die verwendeten Variablen, also die Signale, eindeutig ist.

Innerhalb einer Schaltungsstruktur hat jede Signalvariable eine Quelle und evtl. mehrere Senken. Die Anzahl der Senken kann auch null oder eins sein. Quellen sind die Eingänge der Gesamtschaltung und die Ausgänge der Teilkomponenten. Senken sind die Eingänge der Teilkomponenten und die Ausgänge der Gesamtschaltung. In Abbildung 4.1 hat beispielsweise das Signal a eine Quelle und eine

Senke. Die Quelle ist ein Eingang der Gesamtschaltung, und die Senke ist ein Eingang einer Teilkomponente. Das Signal k hat einen Ausgang einer Teilkomponente als Quelle. k hat zwei Senken: einen Ausgang der Gesamtschaltung und einen Eingang einer Teilkomponente.

Definition 2 (Signalquelle) *Die Ausgänge der Teilkomponenten einer Schaltungsstruktur sowie die Eingänge der Gesamtstruktur werden als Quellen von Signalen bezeichnet.*

Definition 3 (Signalsenke) *Die Eingänge der Teilkomponenten einer Schaltungsstruktur sowie die Ausgänge der Gesamtstruktur werden als Senken von Signalen bezeichnet.*

Freie Variablen in FDFG-Termen stünden für Leitungen mit mindestens einer Senke und keiner Quelle. Derartige Leitungen sind nicht zulässig, da ihr Wert nicht eindeutig ist. Er ergibt sich weder durch eine Belegung der Eingangssignale noch kann er durch Auswertung der Teilkomponenten aus den Eingangssignalen abgeleitet werden.

Kurzschlüsse sind Signale mit mehr als einer Quelle. Kurschlüsse lassen sich mit FDFG-Termen nicht darstellen. Gemäß der Semantik der λ-Terme ist bei einer Schachtelung von λ-Abstraktionen mit gleichen Variablennamen an jeder Stelle nur die letztgenannte λ-Abstraktion sichtbar. Zwei gleichbenannte Signalvariablen an Ausgängen von Operationen entsprächen also nicht einem, sondern zwei verschiedenen Signalen, die nicht miteinander verbunden sind. Der besseren Lesbarkeit halber sollen im weiteren jedoch stets die Signale einer Struktur durch unterschiedliche Variablennamen gekennzeichnet werden.

Definition 4 (Kurzschluß) *Eine Verbindung zwischen zwei Signalquellen wird als Kurzschluß bezeichnet.*

Kombinatorische Zyklen sind geschlossene Signalpfade, die ausschließlich über kombinatorische Komponenten führen. Rein kombinatorische Schaltungen mit Zyklen sind i. allg. nicht synchron, sondern können zu einem asynchronen Schwingverhalten führen. Es ist möglich, alleine durch kombinatorische Schaltungsstrukturen ohne Zyklen beliebige kombinatorische Funktionen zu realisieren. Schaltungen mit kombinatorischen Zyklen sind zur Schaltungsmodellierung nicht notwendig. Da in FDFG-Termen stets die Quelle einer Signalvariablen vor deren Senke erscheint, sind kombinatorische Zyklen in FDFG-Termen schon syntaktisch ausgeschlossen.

Definition 5 (Kombinatorischer Zyklus) *Ein kombinatorischer Zyklus ist ein geschlossener Signalpfad, der ausschließlich über kombinatorische Komponenten führt.*

Daß Signalwerte, die von den Teilkomponenten berechnet werden, nicht verwendet werden, ist zulässig. Es kann aus technischer Sicht durchaus gewollt sein, eine Teilkomponente einzusetzen, von der nicht alle Ausgangsleitungen verwendet werden. Dies ist zum Beispiel dann sinnvoll, wenn beim Entwurf bestimmte standardisierte Komponenten mehrfach verwendet werden sollen. Dabei kann es dann

passieren, daß bei den verschiedenen Instanzen jeweils einige der Ausgangssignale nicht benötigt werden. Das kann dann dazu führen, daß innerhalb der Komponente einige der Teilkomponenten ausschließlich nicht benötigte Signale produzieren. Komponenten, die ausschließlich nicht benötigte Signale produzieren, können einfach eliminiert werden. Bei den üblichen Syntheseprogrammen geschieht dies automatisch.

Soll eine Schaltungsstruktur durch einen FDFG-Term repräsentiert werden, so ist die Reihenfolge, in der die Teilkomponenten im FDFG-Term aufgelistet werden, i. allg. nicht eindeutig. Vertauschungen sind möglich. Aus den Datenabhängigkeiten ergibt sich jedoch eine partielle Ordnung: die Quelle muß stets vor der Senke eines Signals aufgeführt werden. In der Schaltungsbeschreibung aus Abbildung 4.1 bedeutet das beispielsweise, daß die beiden Zeilen

$\mathsf{let}\, k \;=\; \mathsf{INV}(a)\; \mathsf{in}$
$\mathsf{let}\, j \;=\; \mathsf{INV}(c)\; \mathsf{in}$

miteinander vertauscht werden dürfen. Zwischen diesen beiden Teilkomponenten besteht keine Datenabhängigkeit. Vertauscht man dagegen die Zeile

$\mathsf{let}\, k \;=\; \mathsf{INV}(a)\; \mathsf{in}$

mit der Zeile

$\mathsf{let}\, h \;=\; \mathsf{AND}(k, b)\; \mathsf{in}$

so verletzt man die partielle Ordnung, die durch die Datenabhängigkeit vorgegeben ist, denn der Ausgang des Inverters ist die Quelle von k und der erste Eingang des AND-Gatters deren Senke. Das Ergebnis dieser Vertauschung würde wie folgt aussehen:

$$
\begin{array}{l}
\lambda(a, b, c, d). \\
\quad \mathsf{let}\, h \;=\; \mathsf{AND}(k, b)\; \mathsf{in} \\
\quad \mathsf{let}\, j \;=\; \mathsf{INV}(c)\; \mathsf{in} \\
\quad \mathsf{let}\, k \;=\; \mathsf{INV}(a)\; \mathsf{in} \\
\quad \mathsf{let}\, i \;=\; \mathsf{AND}(h, j)\; \mathsf{in} \\
\quad \mathsf{let}\, n \;=\; \mathsf{AND}(h, d)\; \mathsf{in} \\
\quad \mathsf{let}\, l \;=\; \mathsf{INV}(h)\; \mathsf{in} \\
\quad \mathsf{let}\, m \;=\; \mathsf{OR}(i, d)\; \mathsf{in} \\
\quad\quad (k, l, m, d, n)
\end{array}
$$

Der Sichtbarkeitsbereich einer lokalen Variablen beginnt erst mit der Zeile in der die Quelle steht. Bei der Senke, im Ausdruck $\mathsf{AND}(k, b)$ ist somit die lokale Variable k, deren Sichtbarkeitsbereich erst mit der Zeile

$\mathsf{let}\, k \;=\; \mathsf{INV}(a)\; \mathsf{in}$

beginnt, noch nicht zugreifbar. Das Vorkommen der Variablen k in $\mathsf{AND}(k, b)$ ist deshalb frei und steht mit dem weiter unten aufgeführten k in keinem logischen Zusammenhang. Freie Variablen sind gemäß der Definition der FDFG-Terme nicht zulässig.

DFG-Terme

Die Menge der FDFG-Terme soll nun erweitert werden zur Menge der DFG-Terme. Im Gegensatz zu FDFG-Termen erlauben DFG-Terme auch Teilausdrücke, die sich aus mehreren Operatoren zusammensetzen. Die Syntax der DFG-Terme wird wieder mit Hilfe von Syntaxregeln in Backus-Naur-Beschreibung definiert.

Definition 6 (DFG-Terme) *FDFG-Terme sind genau die wohltypisierten λ-Terme ohne freie Variablen, die gemäß den folgenden Syntaxregeln aufgebaut sind:*

Variablenblock ::=
  Variable |
  "`(`" *Variablenblock* { "`,`" *Variablenblock* } "`)`"

Ausdruck ::=
  Variable |
  Konstante |
  "`(`" *Ausdruck* { "`,`" *Ausdruck* "`)`" } |
  Operator "`(`" *Ausdruck* "`)`"

DFG-Term ::=
  "λ" *Variablenblock* "`.`"
  { "`let`" *Variablenblock* "`=`" *Ausdruck)* "`in`" }
  Ausdruck

Das folgende Beispiel zeigt einen DFG-Term, der äquivalent zum FDFG-Beispiel in Abbildung 4.1 ist:

$$\begin{aligned}
&\lambda(a, b, c, d).\\
&\quad \mathsf{let}\ k\ =\ \mathsf{INV}(a)\ \mathsf{in}\\
&\quad \mathsf{let}\ j\ =\ \mathsf{INV}(c)\ \mathsf{in}\\
&\quad \mathsf{let}\ i\ =\ \mathsf{AND}(\mathsf{AND}(k, b), j)\ \mathsf{in}\\
&\quad \mathsf{let}\ n\ =\ \mathsf{AND}(\mathsf{AND}(k, b), d)\ \mathsf{in}\\
&\quad \mathsf{let}\ l\ =\ \mathsf{INV}(\mathsf{AND}(k, b))\ \mathsf{in}\\
&\qquad (k, l, \mathsf{OR}(i, d), d, n)
\end{aligned}$$

Man erkennt leicht, daß der Term mit weniger let-Ausdrücken und damit mit weniger Signalvariablen auskommt als die entsprechende Darstellung als DFG-Term. Die Signale h und m werden in diesem Beispiel nicht mehr explizit benannt, sie verlaufen „innerhalb" der Ausdrücke.

Dadurch, daß sich die Ausdrücke nun aus mehreren kombinatorischen Komponenten zusammensetzen dürfen, ist es möglich, auf diese zusammengesetzten Ausdrücke Äquivalenzumformungen anzuwenden. Beispielsweise läßt sich auf den Ausdruck $\mathsf{AND}(\mathsf{AND}(k, b), j)$ das Assoziativgesetz für die boolesche Konjunktion anwenden. Bei der entsprechenden FDFG-Darstellung ist dies nicht möglich.

An diesem Beispiel wird jedoch auch deutlich, welche Vorteile die Verwendung von let-Ausdrücken hat. Hat ein Signal mehrere Senken, so bedeutet der Verzicht auf das let-Konstrukt, daß der zugehörige Ausdruck mehrfach kopiert werden muß.

In der ursprünglichen Darstellung wurde das Ergebnis des Teilausdrucks $\mathsf{AND}(k, b)$ als h „abgekürzt" und mehrfach wiederverwendet. In der DFG-Darstellung, bei der auf das let-Konstrukt verzichtet wurde, muß der Ausdruck $\mathsf{AND}(k, b)$ an den drei Stellen, an denen ursprünglich die Signalvariable h stand, instantiiert werden. Dies ist unerläßlich, wenn weitere Umformungen wie etwa das oben beschriebene Assoziativgesetz angewandt werden sollen, auch wenn es zunächst einen Hardware-Mehraufwand bedeutet. Zusammengefaßt kann gesagt werden, daß die Verwendung eines let-Ausdrucks prinzipiell immer möglich und genau dann unumgänglich ist, wenn das betreffende Signal mehrere Senken hat und die Komponente, die das Signal erzeugt, nicht mehrfach instantiiert werden soll.

Durch Eliminierung aller let-Ausdrücke gelangt man zu einer normierten Darstellung, die für alle DFG-Terme, die die gleiche Struktur repräsentieren, gleich ist. Nachteil dieser normierten Darstellung ist, daß die Größe des Terms bei mehrfachverwendeten Signalen exponentiell mit der kombinatorischen Tiefe der Schaltung anwächst. Zu obigem Beispiel sieht die normierte Darstellung wie folgt aus:

$$
\begin{array}{l}
\lambda((a,b),(c,d)). \\
\quad (\\
\qquad (\\
\qquad\quad \mathsf{INV}(a), \\
\qquad\quad (\\
\qquad\qquad \mathsf{INV}(\mathsf{AND}(\mathsf{INV}(a),b)), \\
\qquad\qquad \mathsf{OR}(\mathsf{AND}(\mathsf{AND}(\mathsf{INV}(a),b),\mathsf{INV}(c)),d) \\
\qquad\quad) \\
\qquad), \\
\qquad d, \\
\qquad \mathsf{AND}(\mathsf{AND}(\mathsf{INV}(a),b),d) \\
\quad)
\end{array}
$$

4.1.3 Hierarchische Strukturen

Schaltungsstrukturen setzen sich aus Teilkomponenten zusammen, sie beschreiben jedoch auch selbst wieder Komponenten. Läßt man zu, daß in den Schaltungsstrukturen nicht nur elementare, sondern auch zusammengesetzte Teilkomponenten verwendet werden dürfen, so gelangt man zu hierarchischen Schaltungsstrukturen. Dies ist insbesondere dann sinnvoll, wenn bestimmte Teilstrukturen mehrfach wiederverwendet werden können. Zur Beschreibung hierarchischer kombinatorischer Strukturen sollen zusammengesetzte kombinatorische Strukturen mit einem Namen bezeichnet werden. Unter diesem Namen, darf die zusammengesetzte Komponente dann in weiteren Strukturen als Teilkomponente verwendet werden.

In der nachfolgenden Definition wird eine kombinatorische Schaltung in Form eines DFG-Terms mit dem Namen OR3 bezeichnet:

$$\mathsf{OR3} \quad :\hat{=} \quad \lambda(a,b,c).\ \mathsf{OR}(a, \mathsf{OR}(b,c))$$

Dazu äquivalent ist die folgende Definition mit freien Eingangsvariablen a, b und

c. Die freien Variablen sind implizit als allquantifiziert zu betrachten.

$$\mathsf{OR3}(a,b,c) \quad :\hat{=} \quad \mathsf{OR}(a,\mathsf{OR}(b,c))$$

Im folgenden soll insbesondere auch diese Form der Definition von DFG-Termen verwendet werden. Sie ist jedoch nur für den Fall relevant, daß der Komponente ein Name zugeordnet wird. Mit der bisherigen Darstellungsform ist es hingegen auch möglich, mit Schaltungen und Teilschaltungen zu arbeiten, denen kein Name zugeordnet wird.

Das Symbol $:\hat{=}$ soll in dieser Arbeit verwendet werden, um Definitionen von Schaltungen in der hier vorgestellten Schaltungsbeschreibungssprache zu kennzeichnen. Bei diesen Definitionen wird eine neue Komponente in Form eines Terms mit Grundkomponenten und bereits vorher definierten Komponenten beschrieben. Zur Definition der Semantik der Grundkonstrukte der Hardwarebeschreibungssprache mit Hilfe von Konstrukten in HOL soll dagegen zur Kennzeichnung das Symbol $:=$ verwendet werden.

Die folgenden Definitionen beschreiben eine hierarchische Schaltungsstruktur von aufeinander aufbauenden kombinatorischen Komponenten.

$$\begin{array}{rcl}
\mathsf{AND3}(a,b,c) & :\hat{=} & \mathsf{AND}(a,\mathsf{AND}(b,c)) \\
\mathsf{EQ1}(a,b) & :\hat{=} & \mathsf{OR}(\mathsf{AND}(a,b),\mathsf{AND}(\mathsf{INV}(b),\mathsf{INV}(c))) \\
\mathsf{EQ3}((a_1,a_2,a_3),(b_1,b_2,b_3)) & :\hat{=} & \mathsf{AND3}(\mathsf{EQ1}(a_1,b_1),\mathsf{EQ1}(a_2,b_2),\mathsf{EQ1}(a_3,b_3)) \\
\mathsf{XOR}(a,b) & :\hat{=} & \mathsf{INV}(\mathsf{EQ1}(a,b)) \\
\mathsf{MUX1}(s,a,b) & :\hat{=} & \mathsf{OR}(\mathsf{AND}(\mathsf{INV}(s),a),\mathsf{AND}(s,b)) \\
\mathsf{HADD}(a,b) & :\hat{=} & (\mathsf{XOR}(a,b),\mathsf{AND}(a,b)) \\
\mathsf{FADD}(cin,a,b) & :\hat{=} & \mathsf{let}\ (x,y) = \mathsf{HADD}(a,b)\ \mathsf{in} \\
 & & \mathsf{let}\ (sum,z) = \mathsf{HADD}(cin,x)\ \mathsf{in} \\
 & & \mathsf{let}\ cout = \mathsf{OR}(y,z)\ \mathsf{in} \\
 & & \quad (sum, cout)
\end{array}$$

Unter diesen Bausteinen und den Grundbausteinen besteht eine Benutzt-Relation. Eine Komponente A benutzt eine Komponente B, wenn A eine zusammengesetzte Komponente ist, in deren Strukturbeschreibung die Komponente B als Teilkomponente vorkommt. Die Benutzt-Relation ist eine partielle Ordnungsrelation. Für die oben definierten Komponenten ist diese Relation in Abbildung 4.2 graphisch dargestellt.

4.1.4 Bausteinbibliotheken

Alle bisher beschriebenen Bausteine wurden in direkter oder indirekter Weise aus den Grundkonstrukten AND, OR, INV, T und F abgeleitet. Bei der Synthese wird i. allg. eine Bibliothek von Bausteinen festgelegt, aus denen sich die bei der Synthese erzeugte Schaltungsstruktur zusammensetzen soll. Der hier vorgestellte Ansatz ist nicht darauf beschränkt, daß diese Bibliothek genau aus den Grundkonstrukten

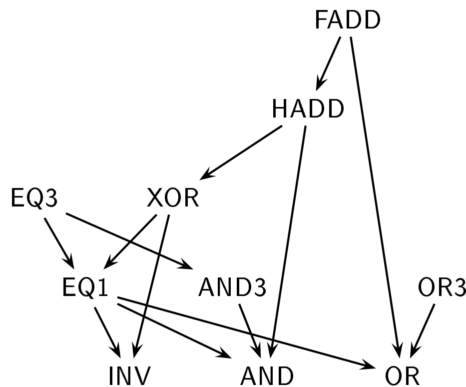


Abbildung 4.2: Hierarchie von Bausteinbeschreibungen

AND, OR, INV, T und F besteht. Es ist möglich, eine beliebige andere Bibliothek zu verwenden. Dazu müssen zunächst die Komponenten der Bibliothek definiert werden. Dies geschieht aufbauend auf AND, OR, INV, T und F. Anschließend muß ein Verfahren gefunden werden, um alle Schaltungen auf Strukturen mit Komponenten aus dieser Bibliothek abzubilden.

Dazu ein kleines Beispiel. Es könnte gefordert werden, daß das Ergebnis der Synthese eine Schaltungsstruktur sein soll, die sich nur aus NAND-Bausteinen und der Konstanten F zusammensetzt. Die NAND-Komponente ist noch nicht definiert und wird deshalb wie folgt aus AND und INV abgeleitet:

$$\mathsf{NAND}(a,b) \quad :\hat{=} \quad \mathsf{INV}(\mathsf{AND}(a,b))$$

Um nun Schaltungen auf die Bibliothek aus NAND und F abzubilden, sollen die folgenden Gleichungen verwendet werden, die leicht durch Fallunterscheidung bewiesen werden können:

$\vdash \mathsf{INV}(a) = \mathsf{NAND}(a,a)$
$\vdash \mathsf{AND}(a,b) = \mathsf{INV}(\mathsf{NAND}(a,b))$
$\vdash \mathsf{OR}(a,b) = \mathsf{NAND}(\mathsf{INV}(a),\mathsf{INV}(b))$
$\vdash \mathsf{T} = \mathsf{INV}(\mathsf{F})$

Diese Gleichungen beschreiben, wie Schaltungsbeschreibungen, die auf AND, OR, INV, T und F aufbauen, in Schaltungsbeschreibungen überführt werden, die sich nur noch aus den Konstrukten NAND und F zusammensetzt. Solche Gleichungssysteme beschreiben also den Wechsel von einer Bibliothek zu einer anderen. Eine notwendige und hinreichende Voraussetzung dafür, daß ein solches Gleichungssystem gefunden werden kann, ist stets, daß die neue Bibliothek eine Basis im aussagenlogischen Sinn darstellt, daß also aus den Grundfunktionen alle booleschen Funktionen abgeleitet werden können.

4.2 Abstrakte kombinatorische Schaltungsbeschreibungen

Es wurden Beschreibungsmittel vorgestellt, mit denen es möglich ist, beliebige Schaltungen auf Gatterebene zu beschreiben. In diesem Abschnitt sollen nun sprachliche Erweiterungen vorgestellt werden. Diese Erweiterungen bedeuten nicht, daß dadurch Schaltungen beschrieben werden können, die vorher nicht beschreibbar gewesen wären. Sie erlauben lediglich, Schaltungen auf einem abstrakteren Niveau zu beschreiben. Mit den Erweiterungen gelangt man zu Ausdrücken, die jeweils für Mengen von Schaltungen stehen. Durch Typinstantiierung entstehen konkrete Instanzen.

4.2.1 Anmerkung zur Implementierbarkeit

Eine unabdingbare Forderung an jede Schaltungsbeschreibungssprache, die bei der Synthese verwendet werden soll, ist die Implementierbarkeit der Beschreibungen. Schaltungsbeschreibungssprachen auf unteren Abstraktionsebenen wie etwa BLIF (Berkeley Logic Interchange Format) haben eine sehr einfache Semantik. BLIF ist eine kleine Schaltungsbeschreibungssprache, die, wie der Name bereits andeutet, weitgehend auf Schaltungsstrukturen auf Gatterebene beschränkt ist — das einzige, was über reine Gatterebenenbeschreibungen hinausgeht, sind Komponenten mit symbolischen Zuständen. Jedes Konstrukt von BLIF besitzt eine einfach verständliche, eindeutige Semantik und kann direkt einer realen Schaltung zugeordnet werden. Durch die geringe Anzahl an Konstrukten und durch die einfache Semantik ist es recht einfach, für Synthesewerkzeuge eine BLIF-Schnittstelle anzubieten. Der Austausch von Syntheseergebnissen über das BLIF-Format gilt als sehr robust.

Ganz anders sieht es bei Schaltungsbeschreibungssprachen aus, die auch abstraktere Formen der Schaltungsbeschreibung unterstützen. Oft enthalten solche Hardwarebeschreibungssprachen Konstrukte, die dazu führen, daß auch Beschreibungen entstehen können, die nicht implementierbar sind. Um diese Sprachen dann in der Praxis einsetzen zu können, müssen Einschränkungen gemacht werden, um die Sprache auf eine implementierbare Teilsprache zurückzustutzen. Ein typisches Beispiel für eine abstrakte, ausdrucksstarke Schaltungsbeschreibungssprache, die eindeutig über die Gatterebene hinausgeht, ist VHDL [VHDL96]. Zahlreiche Beschreibungen in VHDL sind nicht als Hardware implementierbar. Man betrachte dazu die VHDL-Beschreibung in Abbildung 4.3, durch die eine Schaltung namens `seltsam` beschrieben wird.*

Die Semantik, die sich hinter den Sprachkonstrukten verbirgt, wird in VHDL durch Simulationsläufe beschrieben. Simuliert man die Schaltung in Abbildung

*Hinweis zum VHDL-Code: Vereinfacht gesagt definiert in VHDL eine „entity"-Deklaration (hier: `seltsam_entity`) die Schnittstelle einer Schaltung, und die „architecture"-Deklaration die Implementierung (hier: `seltsam`). Zu jeder „architecture" muß die zugehörige „entity" explizit angegeben werden. Diese Trennung zwischen Schnittstelle und Implementierung wird gemacht, da es durchaus möglich und sinnvoll ist, daß es mehrere „architectures" zu einem „entity" gibt.

```
entity seltsam_entity is
  port (a:in Bit; y:out Bit);
end;

architecture seltsam of seltsam_entity is
  signal x : Bit;
begin
  process
    variable s, u : Bit := '0';
  begin
    s := not(s);
    y <= s;
    u := x;
  end;
  process
  begin
    x <= not(y);
  end;
end;
```

Abbildung 4.3: Nichtrealisierbare VHDL-Beschreibung

4.3, so stellt sich am Ausgang ein sonderbares Verhalten ein: die Schaltung legt alternierend `'0'`- und `'1'`-Werte am Ausgang an und zwar ohne (!) zeitliche Verzögerungen.

Es soll erläutert werden, wie es dazu kommt. Die VHDL-Beschreibung in Abbildung 4.3 besteht aus zwei Teilprozessen, die sich wechselseitig über die Signale `x` und `y` Werte zuschicken. Immer, wenn der erste Prozeß einen neuen Wert `x` erhält, produziert er einen neuen Wert `y` für den zweiten Prozeß. Der zweite Prozess produziert wiederum für jeden empfangenen `y`-Wert einen `x`-Wert und schickt diesen zum ersten Prozeß. Da diese Vorgänge bei beiden Prozessen ohne (!) Zeitverzögerung durchgeführt werden, kommt die Simulation über die Simulationszeit $0s$ nicht hinaus. Man betrachte nun den Wert `y`, der durch den ersten Prozeß unendlich oft hintereinander zum immer gleichen Zeitpunkt $0s$ erzeugt wird. Der Wert der Variablen `s`, der dabei jedesmal auf den Ausgang `y` gelegt wird alterniert zwischen `'0'` und `'1'`, denn in jedem Durchlauf wird `s` einmal invertiert.

Damit macht diese Schaltung keine vernünftige Aussage darüber, wie sich die „Schaltung" `seltsam` verhalten soll. Es geht aus der Beschreibung weder hervor, welchen Wert der Ausgang zum Zeitpunkt $0s$ haben soll, noch macht die Schaltung eine vernünftige Aussage über spätere Zeitpunkte. Dieser Schaltung eine konkrete Realisierung zuzuordnen, wäre reine Spekulation. Syntheseprogramme, die von sich in Anspruch nehmen, eine korrekte Synthese zu betreiben, dürfen deshalb Schaltungsbeschreibungen wie diese nicht zulassen.

Aus diesen Grunde verwendet jedes VHDL-Syntheseprogramm eine spezifi-

sche Synthese-Teilsprache von VHDL. Diese Teilsprachen werden gern als Syntheseteilsprachen (sog. „synthesis subsets") bezeichnet. Es gibt verschiedene Möglichkeiten, VHDL auf eine synthetisierbare Teilsprache zurückzuschneiden, und es sind mir auch noch keine zwei VHDL-Synthesesprachen unterschiedlicher Syntheseprogramme bekannt, die vollkommen gleich und kompatibel wären. Die Synthese-Teilsprachen werden in der Regel dadurch aus der Gesamtsprache VHDL abgeleitet, daß zahlreiche Einschränkungen gemacht werden. Oft sind diese Einschränkungen nur unscharf und beispielhaft formuliert und für den Entwerfer schwer durchschaubar.

Es sei noch darauf hingewiesen, daß es zahlreiche VHDL-Konstrukte gibt, deren Semantik nicht hinreichend geklärt ist und deshalb von unterschiedlichen Werkzeugen mit VHDL-Schnittstelle auch unterschiedlich interpretiert werden. VHDL ist als Austauschformat zwischen Syntheseprogrammen nur mit Vorsicht einsetzbar. Eine ausführliche Diskussion zur Semantik von VHDL sowie mehrere konkurrierende Vorschläge zur mathematisch exakten Festlegung der Semantik von VHDL findet man in [KlBr95].

Mit der bisher definierten Schaltungsbeschreibungssprache können nur sehr einfache Gatternetzstrukturen beschrieben werden, denen jeweils in eindeutiger Weise eine reale Schaltung zugeordnet werden kann. Die in dieser Arbeit vorgestellte Schaltungsbeschreibungssprache soll es jedoch erlauben, Schaltungen auch abstrakter als durch reine Gatterebenen-Netzlisten zu beschreiben. Trotz der Tatsache, daß auch abstrakte Schaltungsbeschreibungen zugelassen werden, soll doch gewährleistet bleiben, daß alle Konstrukte eine mathematisch eindeutige Semantik haben und daß alle Schaltungsbeschreibungen auch implementierbar sind.

Es soll vermieden werden, daß der Entwerfer Schaltungsbeschreibungen konstruiert und mit ihnen arbeitet, bei denen sich erst im Laufe der Synthese herausstellt, daß sie nicht als Hardware realisiert werden können oder daß sie gar widersprüchlich sind. Um die Implementierbarkeit der Schaltungsbeschreibungen zu belegen, soll zu den neuen Sprachkonstrukten auch erläutert werden, wie sie mit den bisherigen Konstrukten implementiert werden können. Indem eine bestimmte Implementierung angegeben wird, soll nicht ausgesagt werden, daß diese die einzig mögliche oder gar die kostengünstigste Implementierung darstellt. Es soll lediglich die Implementierbarkeit nachgewiesen werden. Gibt es eine Implementierung, so gibt es in der Regel auch verschiedene Alternativen.

4.2.2 Polymorphe Signalvariablen

Zusätzlich zu den bisherigen Formen von Schaltungsbeschreibungen sollen Schaltungen mit Signalleitungen von variablem Typ (Polymorphie) zugelassen werden, und es sollen zwei Grundoperationen MUX und EQ für Signalleitungen mit variablem Typ vorgestellt werden.

Ein variabler Typ bedeutet, daß er durch jeden beliebigen Typ ersetzt werden darf. Während die bisherigen Operationen nur an den Stellen eingesetzt werden konnten, wo der Typ der Komponente konform mit den anliegenden Signalen war, können die polymorphen Operationen überall eingesetzt werden, wo es ei-

ne Instantiierung der Typvariablen gibt, so daß die anliegenden Signale mit dem resultierenden Typ konform sind.

Bereits ohne die beiden Grundoperationen lassen sich einfache Verdrahtungskomponenten als Strukturen mit variablen Signaltypen definieren:

$$\begin{aligned} \mathsf{FST}\,(x,y) &\mathrel{:\hat{=}} x \\ \mathsf{SND}\,(x,y) &\mathrel{:\hat{=}} y \\ \mathsf{swap}\,(x,y) &\mathrel{:\hat{=}} (y,x) \end{aligned}$$

Die beiden Grundoperationen MUX und EQ sind in Tabelle 4.2 mit ihren Typen aufgelistet. Dabei sind, wie auch im weiteren, Typvariablen durch griechische Buchstaben gekennzeichnet. Die Funktion MUX wählt in Abhängigkeit des ersten Eingangssignals eines der beiden anderen Eingangssignale aus. Hat das erste Eingangssignal den Wert T so nimmt der Ausgang den Wert des zweiten Eingangssignals an, ansonsten den des dritten. Durch die Funktion EQ wird die Äquivalenz zweier Signale ausgedrückt. Der Funktionswert nimmt genau dann den Wert T an, wenn die beiden Signale den gleichen Wert haben.

MUX	Typ: $\mathsf{bool} \times \alpha \times \alpha \;\rightarrow\; \alpha$ Multiplexer
EQ	Typ: $\alpha \times \alpha \;\rightarrow\; \mathsf{bool}$ Äquivalenz

Tabelle 4.2: Polymorphe Operationen

Werden diese Komponenten so instantiiert, daß alle Signalvariablen durch zulässige Signalvariablentypen ohne Typvariablen substituiert werden, und werden für alle Signaltypen boolesche Kodierungen angegeben, so kann dieser Schaltungsbeschreibung eine Schaltungsstruktur aus elementaren booleschen Gatterelementen zugeordnet werden. Wie eine derartige Instantiierung aussehen kann, soll am Beispiel des EQ-Bausteins erläutert werden. Die Komponente EQ, die allgemein den Typ $\alpha \times \alpha \rightarrow \mathsf{bool}$ hat, werde in folgendem Kontext verwendet:

$$\ldots \mathsf{let}\; z \;=\; \mathsf{EQ}((a,b,c,d),(e,f,g,h))\;\mathsf{in}\ldots$$

Die Typen von $a, b, \ldots h$ seien alle boolesch. Der Typ α wurde also in diesem Kontext zu $\mathsf{bool} \times \mathsf{bool} \times \mathsf{bool} \times \mathsf{bool}$. Eine mögliche Implementierung für diese Instanz von EQ ist die nachfolgend definierte Schaltung EQ4. Analog zu EQ4 lassen sich Implementierungen von EQ für beliebige andere Tupel boolescher Werte erstellen.

$$\begin{aligned} \mathsf{EQ1}(a,b) &\mathrel{:\hat{=}} \mathsf{OR}(\mathsf{AND}(a,b),\mathsf{AND}(\mathsf{INV}(b),\mathsf{INV}(c))) \\ \mathsf{AND4}(a_1,a_2,a_3,a_4) &\mathrel{:\hat{=}} \mathsf{AND}(a_1,\mathsf{AND}(a_2,\mathsf{AND}(a_3,a_4))) \end{aligned}$$

$$\mathsf{EQ4}((a_1,a_2,a_3,a_4),(b_1,b_2,b_3,b_4)) \;:\hat{=}\; \mathsf{AND4}(\mathsf{EQ1}(a_1,b_1),\ \mathsf{EQ1}(a_2,b_2),\ \mathsf{EQ1}(a_3,b_3),\ \mathsf{EQ1}(a_4,b_4))$$

4.2.3 Die Datentypen one, (α)option und $\alpha + \beta$

In diesem Abschnitt sollen drei nichtboolesche Datentypen und zugehörige Operationen eingeführt werden. Die Operationen sind in Tabelle 4.3 aufgelistet. Sie basieren auf den drei Datentypen one, (α)option und $\alpha+\beta$. Um aus Schaltungsbeschreibungen mit diesen Operationen Schaltungsbeschreibungen auf Gatterebene abzuleiten, müssen sie Signaltypen one, (α)option und $\alpha + \beta$ zunächst durch boolesche Werte kodiert werden.

Operation	Beschreibung
one	Typ: one das einzige Element vom Typ one
none	Typ: (α)option Konstruktor vom Typ (α)option
any	Typ: $\alpha \rightarrow (\alpha)$option Konstruktor vom Typ (α)option
CASE_option	Typ: $(\alpha)\mathsf{option} \times \gamma \times (\alpha \rightarrow \gamma) \rightarrow \gamma$ Fallunterscheidung über (α)option
INL	Typ: $\alpha \rightarrow \alpha + \beta$ Konstruktor vom Typ $\alpha + \beta$
INR	Typ: $\beta \rightarrow \alpha + \beta$ Konstruktor vom Typ $\alpha + \beta$
CASE_sum	Typ: $(\alpha + \beta) \times (\alpha \rightarrow \gamma) \times (\beta \rightarrow \gamma) \rightarrow \gamma$ Fallunterscheidung über $\alpha + \beta$

Tabelle 4.3: Endliche, nichtboolesche Operationen

Der Datentyp one repräsentiert eine einelementige Menge. Die Konstante, die ebenfalls den Namen one hat, steht für dieses eine Element vom Typ one. Eine Signalleitung, die Werte vom Typ one überträgt, ist redundant, denn über sie können keine Informationen übertragen werden. Soll über eine Signalleitung von einer Quelle zu einer Senke eine Information übertragen werden, so muß es möglich sein, unterschiedliche Werte auf die Signalleitung legen zu können. Hat die Signalleitung dagegen den Typ one, so kann die Signalleitung immer nur den gleichen Wert one annehmen. Die Quelle kann zu jedem Zeitpunkt nur den Wert

one an die Leitung anlegen, und die Senke empfängt auch immer den gleichen Wert one.

Der Datentyp one soll zur Modellierung redundanter Leitungen verwendet werden, also von Leitungen, die weggelassen werden können. Weglassen bedeutet dabei, daß bei den Senken des Signals die Verbindung abgekappt wird und an deren Stelle die Konstante one angelegt wird. Abbildung 4.4 zeigt dazu ein Beispiel. In dieser Abbildung sowie in allen weiteren Abbildungen sind Leitungen vom Typ one durch gestrichelte Linien dargestellt. In der linken Struktur verbindet die beiden Teilkomponenten A und B ein Signal vom Typ one. In der rechten Struktur, die äquivalent zur linken ist, ist die Verbindung aufgetrennt. In der rechten Schaltung ist die Quelle des Signals, der zweite Ausgang von A, mit keiner Senke mehr verbunden. Dafür wird an der ursprünglichen Senke des Signals, dem ersten Eingang von B die Konstante one angelegt.

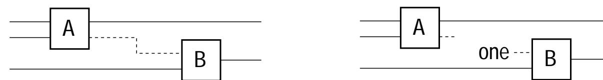


Abbildung 4.4: Signalleitung vom Typ one

Man mag sich nun die Frage stelle, weshalb Leitungen modelliert werden sollen, die ohnehin redundant sind. Wie später noch an mehreren Beispielen (siehe Abschnitt 4.2.6 und Abschnitt 4.3.3) ausgeführt werden wird, dienen sie dazu, allgemeinere Schaltungsstrukturen mit polymorphen Leitungen auf speziellere Fälle anzuwenden, bei denen an der Stelle, an der die allgemeinere Schaltung eine polymorphe Leitung hat, bei der spezielleren Schaltung keine Leitung ist. Dazu wird die Typvariable der polymorphen Leitung einfach mit one instantiiert, und anschließend wird die Leitung dann nach dem Schema in Abbildung 4.4 aufgebrochen.

Die Datentypen (α)option und $\alpha + \beta$ sind Datentypen, die selbst wieder von Typen abhängen. Dabei sind option und + Typoperatoren, die in Postfix- bzw. Infixschreibweise verwendet werden. Unter der Voraussetzung, daß α und β zulässige Signaldatentypen sind, seien auch (α)option und $\alpha + \beta$ zulässige Signaltypen.

Die Menge der durch (α)option charakterisierten Werte wird durch die Konstruktoren none und any beschrieben: Es sind die Werte none sowie für alle x vom Typ α die Werte any(x). Diese Werte sind alle disjunkt, und es gibt in der durch (α)option charakterisierten Menge sonst keine Elemente. Die durch (α)option charakterisierte Menge enthält also genau ein Element mehr als die Menge α. Der Datentyp (α)option kann dazu verwendet werden, einen Datentypen um einen einzelnen Wert anzureichern. Ein typisches Beispiel ist ein Datentyp, der das Ergebnis einer Division beschreibt. Zu dem zur Repräsentation der Zahlen vorgesehenen Datentyp wird noch ein einzelner Wert hinzugefügt werden, der besagt, daß das Ergebnis aufgrund einer Division durch 0 undefiniert ist. Bei einer Division, deren Wert 5 annimmt, wäre das Ergebnis durch any(5) dargestellt, eine Division durch 0 würde durch none repräsentiert.

Der Datentyp $\alpha + \beta$ basiert auf beliebigen, zulässigen Signaldatentypen α und

β. Die Menge, die durch $\alpha+\beta$ charakterisiert wird, kann durch die Konstruktoren INL und INR beschrieben werden: Es sind die Werte $\mathsf{INL}(x)$ für alle x vom Typ α sowie die Werte $\mathsf{INR}(y)$ für alle y aus β. Diese Werte sind alle disjunkt und es gibt sonst keine Elemente vom Typ $\alpha+\beta$. Für endliche Mengen ist die Kardinalität der Menge $\alpha+\beta$ also gerade die Summe der Kardinalitäten der Teilmengen.

Der Datentyp $\alpha+\beta$ dient der Modellierung von Varianten (auch variante Records genannt). Bei Varianten ist eine Steuervariable explizit ausgewiesen. Ihr Wert, der sich während der Laufzeit ändern kann, bestimmt, auf welche der anderen Teile zugegriffen werden kann. Da nie auf alle Teilkomponenten gleichzeitig zugegriffen werden kann, erlauben Varianten im Gegensatz zu Verbunden (Records) eine effizientere Speicherung. Der Datentyp $\alpha+\beta$ entspricht einer Varianten mit einer booleschen Steuervariablen. Der Datentyp $\alpha_1+\alpha_2+\ldots+\alpha_n$ steht für eine Variante, deren Steuervariante die Kardinalität n hat.

Für die beiden Datentypen $(\alpha)\mathsf{option}$ und $\alpha+\beta$ wird je ein Destruktor namens $\mathsf{CASE_option}$ bzw. $\mathsf{CASE_sum}$ angeboten, mit dem eine Fallunterscheidung durchgeführt werden kann. Ihre Semantik ist wie folgt definiert:

$$\begin{aligned}\mathsf{CASE_option}(\mathsf{none},a,f) &:= a\\ \mathsf{CASE_option}(\mathsf{any}(p),a,f) &:= f(p)\end{aligned}$$

$$\begin{aligned}\mathsf{CASE_sum}(\mathsf{INL}(q),f,g) &:= f(q)\\ \mathsf{CASE_sum}(\mathsf{INR}(r),f,g) &:= g(r)\end{aligned}$$

Im Gegensatz zu den bisher beschriebenen Operatoren haben die Destruktoren $\mathsf{CASE_option}$ und $\mathsf{CASE_sum}$ auch Funktionen als Parameter. Hierbei soll die Einschränkung gemacht werden, daß die Funktionen stets die Form $\lambda r.p[r]$ haben sollen, wobei in $p[r]$ eine zusätzliche Signalvariable, nämlich r, vorkommen darf.

Es soll jetzt erläutert werden, wie die Konstrukte des Datentyps $\alpha+\beta$ implementiert werden können. Als Kodierung für den Datentyp $\alpha+\beta$ kann $\mathsf{bool}\times\alpha\times\beta$, verwendet werden. Die Werte $\mathsf{INL}(x)$ werden mit $(\mathsf{F},x,\mathsf{b})$ und die Werte $\mathsf{INR}(y)$ werden mit $(\mathsf{T},\mathsf{a},y)$ kodiert, wobei a und b beliebige aber feste Konstanten vom Typ α bzw. β sind. Bei einer Kodierung von $\alpha+\beta$ durch ein Tupel (s,x,y) vom Typ $\mathsf{bool}\times\alpha\times\beta$ kann $\mathsf{CASE_sum}$ durch einen Multiplexer realisiert werden: die Steuervariable wird s, der erste Dateneingang bekommt den Wert $f(x)$, der zweite den Wert $g(y)$. Es sei darauf hingewiesen, daß diese Kodierung nicht die einzig mögliche und insbesondere auch nicht die effizienteste Kodierung darstellt. Wählt man diese Kodierung, so verzichtet man ja gerade wieder auf den spezifischen Vorteil von Varianten gegenüber Verbunden.

Der Datentyp $(\alpha)\mathsf{option}$ kann durch eine Kodierung auf $\mathsf{one}+\alpha$ zurückgeführt werden, bei der none auf $\mathsf{INL}(\mathsf{one})$ und $\mathsf{any}(x)$ auf $\mathsf{INR}(x)$ abgebildet wird. Der Funktion $\mathsf{CASE_option}(x,a,f)$ entspricht dann $\mathsf{CASE_sum}(x',(\lambda r.\,a),f)$. Der Typ $\mathsf{one}+\alpha$ kann wiederum, wie oben beschrieben, z.B. durch $\mathsf{bool}\times\mathsf{one}\times\alpha$ kodiert werden. Durch eine einfache Umkodierung gelangt man zu $\mathsf{bool}\times\alpha$.

4.2.4 Aufzählungsdatentypen

Sieht man einmal von Typvariablen ab, so sind alle bisher vorgestellten Datentypen zur Modellierung von Signalen endlich. Jedem Signaltyp kann fest eine Kardinalität zugeordnet werden. In diesem und dem folgenden Teilabschnitt sollen Aufzählungsdatentypen bzw. Felder vorgestellt werden. Hier ist die Kardinalität wenn auch fest, so doch beliebig.

Aufzählungsdatentypen und Felder können eigentlich auch bereits mit den bisher vorgestellten Konstrukten dargestellt werden. Mit Datentypen der Form $(\ldots((\mathsf{one})\mathsf{option})\mathsf{option}\ldots)\mathsf{option}$ können Aufzählungsdatentypen beliebiger Kardinalität und mit Datentypen der Form $\alpha \times \alpha \times \ldots \times \alpha$ können Felder beliebiger Länge beschrieben werden. Für jede Kardinalität von Aufzählungsdatentypen und für jede Feldlänge muß so jedoch ein eigener Datentyp verwendet werden und damit auch jeweils eine eigene Schaltungsbeschreibung. Die Operationen in diesem und dem nachfolgenden Abschnitt gehen darüber hinaus. Zur Modellierung von Aufzählungsdatentypen und Feldern werden in der Logik zwei rekursiv definierte, unendliche Datentypen verwendet. Mit ihnen ist es möglich, Aufzählungsdatentypen bzw. Felder von beliebiger Länge jeweils durch einen einzigen Datentyp zu repräsentieren. Dadurch wird es möglich, Schaltungen in generischer Weise zu modellieren.

Bisher konnten nur Schaltungen mit einer für die Schaltung spezifischen Bitbreite modelliert werden. Zur Modellierung eines 16-Bit-Addierers und eines 32-Bit-Addierers war es erforderlich, jeweils eine eigene Strukturbeschreibung anzugeben. Mit diesem Formalisierungsschema wird es möglich, den Addierer zunächst generisch in Form eines n-Bit Addierers zu modellieren und daraus dann durch Instantiierung der Variablen n beliebige konkrete Implementierungen abzuleiten. Der n-Bit Addierer ist, ähnlich wie die oben beschriebenen polymorphen Bausteine, lediglich ein Hilfskonstrukt, durch das eine Menge von Schaltungen beschrieben wird. Erst durch die Instantiierung des Parameters n gelangt man zu einem Modell einer realen Schaltung.

Ein Aufzählungsdatentyp der Kardinalität n soll mit enum_n bezeichnet werden, die einzelnen Elemente mit den Konstanten $\mathsf{enum}\,n\,m$ mit $m = 0, 1, \ldots n-1$. Der Nachfolgeoperator $\mathsf{next}\,n\,e$ bildet ein Element e vom Typ enum_n auf ein anderes Element vom Typ enum_n ab (Tabelle 4.4).

$\mathsf{enum}\,n\,m$	Typ: enum_n Elemente des Aufzählungsdatentyps mit Kardinalität n
$\mathsf{next}\,n$	Typ: $\mathsf{enum}_n \rightarrow \mathsf{enum}_n$ Nachfolgeoperation für Aufzählungsdatentypen

Tabelle 4.4: Operationen für Aufzählungsdatentypen

Üblicherweise werden Aufzählungsdatentypen durch eine endliche Menge von

benutzerdefinierten Bezeichnern charakterisiert. In diesem Ansatz werden die einzelnen Elemente eines Aufzählungsdatentypen mit Kardinalität n zunächst einfach mit Hilfe der Zahlworte $0, 1, 2, \ldots (n-1)$ beschrieben. Dem Anwender ist es jederzeit gestattet, die Zahlworte durch beliebige Bezeichner abzukürzen.

Alle Aufzählungsdatentypen enum_n sind in der Logik von HOL durch einen einzigen Typ, nämlich num formalisiert worden. Die Semantik der Konstanten $\mathsf{enum}\, n\, i$ und die Funktion $\mathsf{next}\, n$ wurden wie folgt definiert.

$$\mathsf{enum}\, n\, m \quad := \quad \begin{cases} m & \text{für} \quad m < n \\ 0 & \text{sonst} \end{cases}$$

$$\mathsf{next}\, n\, e \quad := \quad \begin{cases} e+1 & \text{für} \quad e+1 < n \\ 0 & \text{sonst} \end{cases}$$

Obwohl der zugrundeliegende Datentyp in HOL eine unendliche Menge repräsentiert, sind die beiden Konstrukte für beliebige aber feste n und m implementierbar. Es ist zu beachten, daß m und n keine Signalwerte, sondern Parameter von enum und next sind.

Ein Nachteil, der bei diesem Vorgehen sofort ins Auge springt, ist der, daß zwischen den einzelnen Aufzählungsdatentypen in HOL nicht unterschieden wird. Der Theorembeweiser HOL gewährleistet durch seine elementaren Termkonstruktoren, daß nur syntaktisch richtige und wohltypisierte Terme entstehen können. Bei Aufzählungsdatentypen ist nun zu beachten, daß in HOL zwischen verschiedenen Aufzählungsdatentypen unterschiedlicher Kardinalität nicht unterschieden wird und daß beispielsweise auch der Term $\mathsf{enum}\ 5\ 3 = \mathsf{enum}\ 6\ 4$ von HOL als syntaktisch richtig eingestuft wird, obwohl die linke Seite vom Typ enum_5 und die rechte vom Typ enum_6 ist. Das bedeutet jedoch nicht, daß diese Terme nicht implementierbar wären. Für beliebige, aber feste Parameter m und n sind die beiden Konstrukte $\mathsf{enum}\, m\, n$ und $\mathsf{next}\, m$ immer implementierbar.

4.2.5 Felder und reguläre kombinatorische Strukturen

Reguläre Schaltungsstrukturen führen zu regulären Signalbündeln. Die reguläre Signalbündelung ist somit eine Voraussetzung zur Modellierung regulärer Schaltungsstrukturen. In diesem Abschnitt sollen Felder und zugehörige Operationen auf Feldern vorgestellt werden. Sie bilden die Grundlage für Konstrukte zur Formalisierung regulärer Schaltungsstrukturen.

Felder der Länge n, deren Elemente alle den Typ α haben, sollen mit $(\alpha)\mathsf{array}_n$ bezeichnet werden. Die auf Feldern definierten Operationen sind in Tabelle 4.5 aufgelistet. Die Funktion ripple ist ein Grundkonstrukt zur Beschreibung regulärer Strukturen. Alle anderen Funktionen dienen der Manipulation von Feldern.

Die Funktion $\mathsf{mkarray}$ wandelt einzelne Signale in ein Signalbündel vom Typ $(\alpha)\mathsf{array}_n$ um. Dabei müssen die zu bündelnden Signale als eine endlichen Liste in der Form $[x_0, x_1, x_2, \ldots, x_{n-1}]$ gegeben sein. Die Länge n des Feldes ergibt sich aus der Anzahl der Listenelemente. Die Funktion $\mathsf{spread}\, n$ erzeugt ein Feld der Länge n, deren Elemente alle den gleichen Wert x haben. Mit den Funktionen

mkarray	Typ: $(\alpha)\mathsf{list} \rightarrow (\alpha)\mathsf{array}_n$ Feld aus Einzelsignalen
spread n	Typ: $\alpha \rightarrow (\alpha)\mathsf{array}_n$ Feld aus n gleichen Werten
pick n	Typ: $\mathsf{enum}_n \times (\alpha)\mathsf{array}_n \rightarrow \alpha$ Auswahl eines Elements aus dem Feld
modify n	Typ: $\mathsf{enum}_n \times \alpha \times (\alpha)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_n$ Verändern eines Elements eines Feldes
cut n	Typ: $(\alpha)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_n$ Trimmen auf Länge n
append m n	Typ: $(\alpha)\mathsf{array}_m \times (\alpha)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_{m+n}$ Hintereinanderreihen zweier Felder gleichen Typs
shift m n	Typ: $(\alpha)\mathsf{array}_{m+n} \rightarrow (\alpha)\mathsf{array}_n$ Verschieben der Elemente nach links
rev n	Typ: $(\alpha)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_n$ Umkehrung der Reihenfolge der Elemente
comb n	Typ: $(\alpha)\mathsf{array}_n \times (\beta)\mathsf{array}_n \rightarrow (\alpha \times \beta)\mathsf{array}_n$ Kombination zweier Felder
split n	Typ: $(\alpha \times \beta)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_n \times (\beta)\mathsf{array}_n$ Inverse Funktion zu comb_n
shrink n	Typ: $(\alpha)\mathsf{array}_{2n} \rightarrow (\alpha \times \alpha)\mathsf{array}_n$ Zusammenfassen benachbarter Elemente
unshrink n	Typ: $(\alpha \times \alpha)\mathsf{array}_n \rightarrow (\alpha)\mathsf{array}_{2n}$ inverse Funktion zu shrink
ripple n f	Typ: $\gamma \times (\alpha)\mathsf{array}_n \rightarrow (\beta)\mathsf{array}_n \times (\gamma)\mathsf{array}_n$ reguläre Struktur aus $(f\,1), (f\,2), \ldots (f\,(n-1))$

Tabelle 4.5: Operationen auf Feldern

pick n und modify n kann ein Element eines Feldes herausgegriffen bzw. verändert werden. In beiden Fällen werden zur Indizierung Aufzählungsdatentypen vom Typ enum_n verwendet. Die Funktion cut n kann verwendet werden, um die Länge eines Feldes auf die Länge n zu trimmen. Mit der Funktion append n können zwei Felder mit Elementen gleichen Typs aneinandergehängt werden. Die Funktion shift $m\,n$ verschiebt die Elemente eines Feldes um n Positionen nach links, so daß das Element $(n+i)$ an der Position i zu liegen kommt. Die ersten m Elemente des ursprünglichen Feldes kommen im Ergebnisfeld nicht vor. Das Eingangsfeld hat die Länge $m+n$, das Resultat die Länge n. Die Funktion rev n dreht die Reihenfolge der Elemente um. Mit der Funktion comb n können zwei Felder gleicher Länge zu einem einzigen Feld von Paaren zusammengefaßt werden. Die Funktion split n ist die Inverse zu comb n. Die Funktion shrink n bildet ein Feld der Länge $2 *$ n auf ein Feld der Länge n ab, wobei immer benachbarte Elemente zu Paaren zusammengefaßt werden. Die Funktion unshrink n ist die Inverse zu shrink n.

Die Operation ripple $n\,f$ beschreibt eine reguläre Struktur. Als Parameter hat sie neben einer natürlichen Zahl n auch eine Funktion f vom Typ $\mathsf{num} \to \gamma \times \alpha \to \beta \times \gamma$. Die Funktion f stellt selbst eine abstrakte kombinatorische Schaltung dar. Die Schaltung f ist eine Schaltung mit einem natürlichzahligen Parameter, mit Eingangswerten vom Typ $\gamma \times \alpha$ und Ausgangswerten vom Typ $\beta \times \gamma$. Durch den Ausdruck ripple $n\,f$ wird eine Schaltung beschrieben, die sich gemäß Abbildung 4.5 aus den Teilkomponenten $f(0)$, $f(1)$, $\ldots$ $f(n-1)$ zusammensetzt. Die Gesamtschaltung ripple $n\,f$ bildet ein Paar (a,b) bestehend aus einem Einzelsignal vom Typ γ und einem Signalbündel vom Typ $(\alpha)\mathsf{array}_n$ auf ein Paar (c,d) bestehend aus zwei Signalbündeln vom Typ $(\beta)\mathsf{array}_n$ bzw. $(\gamma)\mathsf{array}_n$ ab.

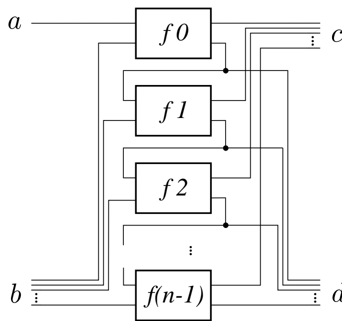


Abbildung 4.5: Reguläre Grundstruktur $(c,d) = \mathsf{ripple}\; n\; f\; (a,b)$

Die Operationen auf den Feldern stellen zunächst nur abstrakte Hilfsmittel zur Schaltungsmodellierung dar. Indem Felder fester Länge durch Tupel kodiert werden, können aus ihnen jedoch direkt Schaltungsbeschreibungen auf Gatterebene abgeleitet werden. Das Schema zur Erzeugung einer Implementierung von ripple $n\;f$ für ein konkretes n kann Abbildung 4.5 entnommen werden. Zu einem vorgegebenen n und zu einer vorgegebenen Kodierung des Aufzählungsdatentypen

kann die Implementierung von $\mathsf{pick}\ n$ und $\mathsf{modify}\ n$ durch polymorphe Multiplexer erfolgen, die dann für konkrete Instantiierungen der Typvariablen α selbst wieder auf elementare Gatterstrukturen zurückgeführt werden müssen. Alle anderen Operationen aus Tabelle 4.5 stellen lediglich Verdrahtungsstrukturen dar, zu deren Implementierung keine aktiven Gatterelemente benötigt werden.

Die hier vorgestellten Operationen auf Feldern sind absichtlich sehr sparsam ausgewählt worden, und insbesondere wurde ganz bewußt nur ein elementares Konstrukt zur Modellierung regulärer Strukturen definiert. Es lassen sich jedoch, wie in Abschnitt 4.2.6 noch ausgeführt werden soll, durch abgeleitete abstrakte Schaltungsbeschreibungen sehr viele verschiedene reguläre Strukturen daraus ableiten.

Die Konstante E

Bevor auf die formale Definition der Feld-Operationen in HOL eingegangen wird, soll zunächst eine Konstante E eingeführt werden. E bezeichnet einen Standardwert mit einem variablem Typ α. Dieser Standardwert kann mit jedem beliebigen Typen instantiiert werden. Bei der nachfolgenden Spezifikation sind nur die Instantiierungen mit den betrachteten Signaldatentypen von Interesse, und auch nur für diese Typinstantiierungen ist für E ein Wert spezifiziert. Dies sind: bool, $\alpha\times\beta$, one, $(\alpha)\mathsf{option}$, $\alpha+\beta$, num, $\alpha\times\beta$ und $\mathsf{num}\rightarrow\alpha$. Es sei darauf hingewiesen, daß E in den einzelnen Gleichungen jeweils mit unterschiedlichem Typ instantiiert ist.

$$\begin{aligned}
\mathsf{E} &:= \mathsf{F}\\
\mathsf{E} &:= \mathsf{one}\\
\mathsf{E} &:= \mathsf{none}\\
\mathsf{E} &:= \mathsf{INL}(\mathsf{E})\\
\mathsf{E} &:= 0\\
\mathsf{E} &:= (\mathsf{E},\mathsf{E})\\
\mathsf{E} &:= (\lambda x.\,\mathsf{E})
\end{aligned}$$

Die Konstante E ist nur partiell spezifiziert. Für alle anderen Typinstantiierungen von E macht die obige Spezifikation keine Aussage. Die Spezifikation von E ist widerspruchsfrei, da für jede Instantiierung von E nur maximal eine der obigen Gleichungen einen äquivalenten Wert definiert.

Definition der Feld-Operatoren in HOL

Zur Modellierung der Felder $(\alpha)\mathsf{array}_n$ in der Logik werden Folgen x vom Typ $\mathsf{num}\rightarrow\alpha$ verwendet, die die Eigenschaft $\forall i \geq n.\, x(i) = \mathsf{E}$ haben. Die Folgen, mit denen Felder einer bestimmten Länge n modelliert werden sollen, unterscheiden sich nur in den ersten n Werten, den Einträgen des Feldes. Für einen beliebigen endlichen Datentyp α ist auch ein Feld der Länge der n endlich. Die Kardinalität des Feldes ergibt sich zu $|\alpha|^n$, wobei $|\alpha|$ die Kardinalität von α ist. Da die Felder in der Logik durch einen unendlichen Datentyp (Folgen) repräsentiert werden,

wird dieser Datentyp durch die Einschränkung $\forall i \geq n. x(i) = \mathsf{E}$ auf eine endliche Teilmenge begrenzt. Dies ist von elementarer Bedeutung, um die Implementierbarkeit zu gewährleisten, denn unendliche Datentypen ließen sich nicht durch eine endliche Anzahl boolescher Werte kodieren.

Die Grundoperationen in Tabelle 4.5 wurden so definiert, daß die Endlichkeit und damit auch die Implementierbarkeit aller aus ihnen abgeleiteten Schaltungen gewährleistet wird. Die Grundoperationen aus Tabelle 4.5 haben zwei wesentliche Eigenschaften:

1. Alle Felder, die am Ausgang einer Funktion produziert werden, haben eine feste Länge n, die nur von den Parametern der jeweiligen Operationen abhängt. Für alle Folgenelemente mit $i \geq n$ wird gewährleistet, daß $x(i) = \mathsf{E}$ gilt.

2. Bei jedem Feld, das am Eingang einer Funktion liegt, hängt der Funktionswert nur von den ersten n Elementen ab. Auf alle anderen Folgenelemente greift die Funktion nicht zu. Dabei ist n ein Wert, der nur von den Parametern der Funktion abhängt.

Felder unterschiedlicher Länge unterscheiden sich in HOL nicht in ihrem Typ. Werden in Strukturen Felder zur Signalbündelung eingesetzt, so wird deren Länge stets durch die Quelloperation festgelegt. Das Signal darf jedoch mit Eingängen verbunden werden, die eine beliebige andere Länge vorsehen. Wenn in Tabelle 4.5 angegeben wird, daß ein Feld am Eingang einer Funktion den Typ $(\alpha)\mathsf{array}_n$ hat, so bedeutet dies nicht, daß dieser Eingang nur mit einem Ausgang verbunden werden kann, der Felder der Länge n erzeugt, sondern es bedeutet lediglich, daß nur die ersten n Werte berücksichtigt werden. Gegeben sei ein Signal x, dessen Quelloperation eine Länge von m erzwingt. Wird dieses Signal an den Eingang einer Operation gelegt, bei dem die Länge n beträgt, so wird die Länge automatisch „angepaßt". Dabei sind drei Fälle zu unterscheiden: $m = n$, $m < n$ und $m > n$. Vereinfacht gesagt geschieht am Eingang folgendes: Ist $m = n$, so werden alle Werte des Feldes eins zu eins weitergeleitet, ist $m < n$, so wird das Feld mit E-Werten aufgefüllt und ist $m > n$ so wird das Feld auf die Länge n abgeschnitten.

Wie dieses Prinzip nun bei der formalen Umsetzung der Felder mit Folgen aussieht, soll im folgenden an Beispielen erläutert werden. Eine elementare Rolle spielt dabei die Funktion $\mathsf{cut}\, n$. Mit ihr werden Felder auf die Länge n gestutzt. Sie ist wie folgt definiert:

$$\mathsf{cut}\, n\, x\, i \quad := \quad \begin{cases} x(i) & \text{für} \quad 0 \leq i < n \\ \mathsf{E} & \text{sonst} \end{cases}$$

Die anderen Feldoperationen in Tabelle 4.5 sind nach dem gleichen Prinzip definiert. Exemplarisch sei die Definition der Funktion append aufgeführt:

$$\mathsf{append}\, m\, n\, (x, y)\, i \quad := \quad \begin{cases} x(i) & \text{für} \quad 0 \leq i < m \\ y(i - m) & \text{für} \quad m \leq i < m + n \\ \mathsf{E} & \text{sonst} \end{cases}$$

Die beiden oben postulierten Eigenschaften für elementare Feldoperationen setzen sich bei Strukturbeschreibungen auf zusammengesetzte Komponenten fort. Dazu ein Beispiel:

$$\mathsf{h}(a,b) \quad :\hat{=} \quad \mathsf{append}\ 6\ 5\ (\mathsf{cut}\ 4\ a, b)$$

Das intern erzeugte Signal $\mathsf{cut}\ 4\ a$ hat die Länge 4. Daß die Funktion $\mathsf{cut}\ 4$ das Signal auf die Länge 4 trimmt, bedeutet, daß als Ergebnis eine Folge entsteht, die zwar unendlich ist, deren Elemente jedoch ab der Position 4 alle den gleichen Werte, nämlich E, haben. Die Folgenelemente von $\mathsf{cut}\ 4\ a$ haben die folgenden Werte:

$$a(0), a(1), a(2), a(3), \mathsf{E}, \mathsf{E}, \mathsf{E}, \ldots$$

Die Senke des Signals $\mathsf{cut}\,4\,a$ hat in dem Ausdruck $\mathsf{append}\,6\,5\,(\mathsf{cut}\,4\,a, b)$ die Länge 6. Damit wird das Signal am Eingang der Funktion $\mathsf{append}\ 6\ 5$ um zwei E-Werte ergänzt. Die Folgenwerte von $\mathsf{h}(a,b)$ sind somit:

$$a(0), a(1), a(2), a(3), \mathsf{E}, \mathsf{E}, b(0), b(1), b(2), b(3), b(4), \mathsf{E}, \mathsf{E}, \mathsf{E}, \ldots$$

Die Funktion h erzwingt, daß das Ausgangssignal eine Länge von 11 hat, d. h. daß alle Folgenwerte ab der Position 11 den Wert E haben. Der Funktionswert $\mathsf{h}(a,b)$ hängt nur von den ersten 4 Elementen des Signals a und von den ersten 5 Elementen des Signals b ab.

Bei parametrisierten, zusammengesetzten Operationen auf Feldern hängen die Längen der Ein- und Ausgangsfelder in funktionaler Weise von den Parameterwerten ab. Die Funktion $\mathsf{h'}$ bildet beispielsweise zwei Felder mit jeweils der Länge n auf ein Feld der Länge $2 * n + 2$ ab.

$$\mathsf{h'}\ n\ (a,b) \quad :\hat{=} \quad \mathsf{append}\ (n+2)\ n\ (\mathsf{cut}\ n\ a, b)$$

4.2.6 Abgeleitete abstrakte Schaltungsbeschreibungen

In diesem Abschnitt wurden Grundoperationen vorgestellt, mit denen Schaltungen in abstrakter Weise beschrieben werden können. Diese lassen sich nun wiederum in Schaltungsstrukturen zu komplexeren abstrakten Schaltungsbeschreibungen kombinieren.

Im Vergleich zu den rein booleschen Grundoperationen, sind einige der abstrakten Grundoperationen parameterbehaftet. Dabei sind die Parameter entweder natürliche Zahlen oder kombinatorische Schaltungen. Auch bei zusammengesetzten abstrakten Schaltungsbeschreibungen sind für die Gesamtschaltung natürliche Zahlen und kombinatorische Schaltungen als Parameter zulässig.

Parameter, die selbst Schaltungen sind, können entweder direkt an eine Teilkomponente weitergereicht werden oder aber an beliebiger anderer Stelle als Teilkomponente verwendet werden. Natürlichzahlige Parameter der Gesamtschaltung dürfen ebenfalls als Parameter an die Teilkomponenten weitergereicht werden. Statt sie direkt weiterzureichen, ist es auch erlaubt, einen einfachen arithmetischen Ausdruck aus den Parametern weiterzureichen. Darunter soll ein Ausdruck

bestehend aus den Parametern, aus Konstanten und den arithmetischen Operationen Addition, Subtraktion, Multiplikation und Potenz verstanden werden. Es ist anzumerken, daß diese Funktionen in HOL alle total sind. Bei der Subtraktion gilt $a - b = 0$ für $a < b$. Die Liste der zulässigen Operationen ließe sich um beliebige weitere totale arithmetische Funktionen auf natürlichen Zahlen erweitern.

Welche Möglichkeiten sich daraus ergeben und wie damit insbesondere verschiedenartige reguläre Strukturen beschrieben werden können, soll an einem Beispiel erläutert werden. Mit der folgenden Beschreibung wird eine n-fach Parallelschaltung definiert (Hinweis: die Funktion FSTbestimmt das erste Element eines Paares, siehe Abschnitt 4.2.2).

$$\mathsf{par}\, n\ f\ x \quad :\hat{=} \quad \mathsf{FST}(\mathsf{ripple}\, n\ (\lambda i.\ \lambda(a,b).\ (f\ i\ b, \mathsf{one}))\ (\mathsf{one}, x))$$

Bei der Definition abstrakter zusammengesetzter Schaltungen ist auf eine strikte Trennung von Schaltungsparametern (hier: n und f) und Eingangssignalen (hier: x) zu achten. Schaltungsparameter dürfen nicht an Eingänge von Schaltungen gelegt und Signalvariablen dürfen nicht als Parameter von Teilkomponenten verwendet werden. Die Schaltung, die dem ripple-Konstrukt als Parameter übergeben wird, ist selbst eine Strukturbeschreibung in Form eines parametrisierten DFG-Terms: $(\lambda i.\ \lambda(a,b).\ (\mathsf{one}, f\ i\ b))$.

Abbildung 4.6 stellt die oben definierte reguläre Struktur graphisch dar. Die Signalleitungen vom Typ one wurden mit gestrichelten Linien dargestellt. Bei der Definition von par wird die Bedeutung des Datentyps one deutlich. Die n-fach Parallelschaltung par konnte aus der regulären Struktur ripple abgeleitet werden, obwohl einige der dabei vorkommenden Leitungen nicht benötigt werden. Für nicht benötigte Leitungen wurden Leitungen vom Typ one verwendet. In Strukturbeschreibungen mit Teilkomponenten, die Anschlüsse vom Typ one haben, wurden alle Eingänge mit der Konstanten one belegt, die Ausgänge werden nicht weitergeleitet.

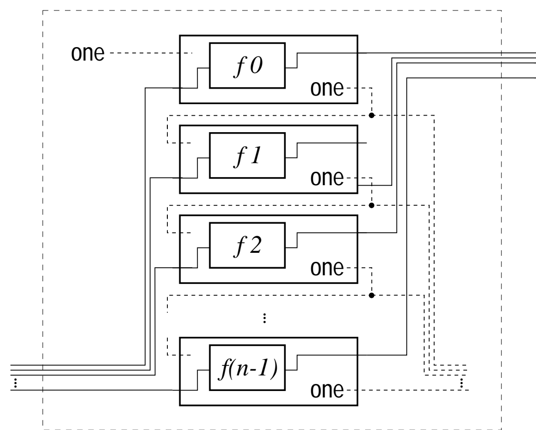


Abbildung 4.6: n-fach Parallelschaltung

In Abbildung 4.7 sind weitere reguläre Strukturen definiert. Einige dieser Strukturen sind in Abbildung 4.8 graphisch dargestellt. Anhand der Beispiele wird deutlich, daß mit den Konzepten der polymorphen Signalleitungen und der Parametrisierung durch Funktionen ein Entwurfsstil möglich wird, der von üblichen Hardwarebeschreibungssprachen wie VHDL, Verilog, ELLA etc. nicht unterstützt wird: es ist hier möglich, zunächst allgemeine Verdrahtungsstrukturen (ripple, rect, par, ser) mit Platzhaltern als Teilkomponenten zu beschreiben. Bei den Verdrahtungsstrukturen sind die Typen der Leitungen noch offen. Durch die Instantiierung der Platzhalter durch geeignete konkrete Komponenten, entstehen in einfacher Weise verschiedenartige Schaltungen (ANDN, MUXN, EQN).

Ein wichtiger Spezialfall für reguläre Strukturen, die auf ripple aufbauen, sind jene, bei denen alle Teilkomponenten gleich sind. Hierzu wurde die Funktion constantly definiert. Die Funktion (constantly f) beschreibt eine Folge von Komponenten, wobei die Elemente der Folge alle gleich f sind.

4.3 Sequentielle Schaltungsbeschreibungen

Mit den bisher vorgestellten Mitteln ist es lediglich möglich, kombinatorische Schaltungen zu modellieren. Zur Modellierung sequentieller Schaltungen sollen Mealy-Automaten verwendet werden.[†] Mealy-Automaten sind deterministische, endliche Automaten mit Ausgabe und ohne Terminierungszustände. Deterministisch bedeutet, daß sich Nachfolgezustand und Ausgabewert stets in eindeutiger Weise aus der Eingabe und dem aktuellen Zustand ergeben. Daß der Automat endlich ist, steht dafür, daß die Anzahl der Zustände endlich ist. Bei jedem Zustandsübergang produziert der Automat einen Ausgabewert. Automaten mit Ausgabe werden auch als übersetzende Automaten oder Transduktoren bezeichnet [EnCS98]. Terminierungszustände kennen Mealy-Automaten nicht.

Gewöhnlich werden Mealy-Automaten als 6-Tupel dargestellt, die sich aus Eingangsalphabet, Ausgangsalphabet, Zustandsmenge, Ausgangsfunktion, Übergangsfunktion und Initialzustand zusammensetzen. Das hier vorgestellte Formalisierungsschema unterscheidet sich davon in zweierlei Hinsicht: Zum einen werden Eingangsalphabet, Ausgangsalphabet und Zustandsmenge nicht explizit angegeben, zum anderen wird statt zweier getrennter Funktionen für das Aus- und

[†]Beim Schaltungsentwurf werden die Begriffe Mealy-Schaltwerk und Moore-Schaltwerk oft fälschlicherweise so verwendet, daß sie sich gegenseitig ausschließen. Als Moore-Schaltwerke werden jene Schaltungen bezeichnet bei denen der Nachfolgezustand nur vom aktuellen Zustand und nicht von der aktuellen Eingabe abhängt, oder anders ausgedrückt: es besteht zwischen den Eingängen und den Ausgängen kein kombinatorischer Signalpfad. Dies entspricht genau dem formalen Modell des Moore-Automaten. Oft werden fälschlicherweise alle synchronen Schaltungen, die nicht Moore-Schaltwerke sind, als Mealy-Schaltwerke bezeichnet. Nach dieser Definitionen sind nur jene Schaltungen Mealy-Schaltwerke, bei denen es mindestens einen kombinatorischen Pfad zwischen Ein- und Ausgabe gibt. Dies entspricht jedoch nicht dem Begriff des Mealy-Automaten, der vielmehr eine Obermenge von Automaten definiert, die insbesondere auch alle Moore-Automaten einschließt. Dies sei angemerkt, um darauf hinzuweisen, daß sich mit dem Modell des Mealy-Automaten alle synchronen Schaltungen formal darstellen lassen — insbesondere auch Moore-Schaltwerke.

$$
\begin{array}{rcl}
\mathsf{EQP}\;m\;n & :\hat{=} & \mathsf{EQ}(\mathsf{enum}\;(m+n+1)\;m, \mathsf{enum}\;(m+n+1)\;n) \\
\mathsf{first}\;n\;x & :\hat{=} & \mathsf{pick}\;n(\mathsf{enum}\;n\;0, x) \\
\mathsf{maxenum}\;n & :\hat{=} & \mathsf{enum}\;n\;(n-1) \\
\mathsf{last}\;n\;x & :\hat{=} & \mathsf{pick}\;n(\mathsf{maxenum}\;n, x) \\
\mathsf{last'}\;n\;(x,y) & :\hat{=} & \mathsf{MUX}(\mathsf{EQP}\;n\;0, y, \mathsf{last}\;n\;x) \\
\mathsf{constantly}\;f\;i & :\hat{=} & f \\
\\
\mathsf{ser}\;n\;f\;x & :\hat{=} & \mathsf{last'}\;n\;(\\
 & & \quad \mathsf{SND}(\mathsf{ripple}\;n \\
 & & \quad\quad (\lambda i.\,\lambda(a,b).\,(\mathsf{one}, f\;i\;a))\;(x, \mathsf{spread}\;n\;\mathsf{one})), \\
 & & \quad x\;) \\
\mathsf{par}\;n\;f\;x & :\hat{=} & \mathsf{FST}(\mathsf{ripple}\;n\;(\lambda i.\,\lambda(a,b).\,(f\;i\;b, \mathsf{one}))\;(\mathsf{one}, x)) \\
\mathsf{for}\;n\;f & :\hat{=} & \mathsf{par}\;n\;(\lambda i.\,\lambda x.\,f\;i)\;(\mathsf{spread}\;n\;\mathsf{one}) \\
\mathsf{rippleb}\;n\;f\;(a,b) & :\hat{=} & \mathsf{let}\;(c,d) = \mathsf{ripple}\;n\;f\;(a,b)\;\mathsf{in}\;(c, \mathsf{last'}\;n\;(d,a)) \\
\mathsf{ripplec}\;n\;f\;(a,b) & :\hat{=} & \mathsf{last'}\;n \\
 & & \quad (\mathsf{SND}(\mathsf{ripple}\;n\;(\lambda i.\,\lambda x.\,(\mathsf{one}, f\;i\;x)))\;(a,b),\;a) \\
\mathsf{rippled}\;n\;f\;(x,d) & :\hat{=} & \mathsf{MUX}(\mathsf{EQP}\;n\;0,\;d, \\
 & & \quad \mathsf{MUX}(\mathsf{EQP}\;n\;1,\;\mathsf{first}\;n\;x, \\
 & & \quad\quad \mathsf{ripplec}\;(n-1)\;f \\
 & & \quad\quad\quad (\mathsf{first}\;n\;x, \mathsf{shift}\;1\;(n-1)\;x)\;)\;) \\
\mathsf{rect}\;m\;n\;f\;(a,b) & :\hat{=} & \mathsf{ser}\;m \\
 & & \quad (\lambda i.\,\lambda(x,y).\,\mathsf{swap}(\mathsf{rippleb}\;n\;(f\;i)\;(x,y))\;(a,b)) \\
\mathsf{MUXN}\;n\;(c,a,b) & :\hat{=} & \mathsf{par}\;n\;(\mathsf{constantly}(\lambda(s,(a,b)).\,\mathsf{MUX}(s,a,b))) \\
 & & \quad (\mathsf{comb}\;n\;(\mathsf{spread}\;n\;c, \mathsf{comb}\;n\;(a,b))) \\
\mathsf{ANDN}\;n\;x & :\hat{=} & \mathsf{rippled}\;n\;(\mathsf{constantly}\;\mathsf{AND})\;(\mathsf{T}, x) \\
\mathsf{ADDERN}\;n\;(cin,a,b) & :\hat{=} & \mathsf{rippleb}\;n\;(\mathsf{constantly}(\lambda(c,(x,y)).\mathsf{FADD}(c,x,y)) \\
 & & \quad (cin, \mathsf{comb}\;n\;(a,b))) \\
\mathsf{EQN}\;n\;(a,b) & :\hat{=} & \mathsf{ANDN}\;n\;(\mathsf{par}\;n\;(\mathsf{constantly}\;\mathsf{EQ})\;(\mathsf{comb}\;n\;(a,b))) \\
\mathsf{MUXT}\;n\;(s,a) & :\hat{=} & \mathsf{first}\;1\;(\\
 & & \quad \mathsf{SND}(\mathsf{ser}\;n\;(\lambda i.\,\lambda(s',a'). \\
 & & \quad\quad \mathsf{let}\;(x,y) = \mathsf{split}\;2^{n-(i+1)}\;(\mathsf{shrink}\;2^{n-(i+1)}\;a') \\
 & & \quad\quad\quad \mathsf{in}\;(s', \mathsf{MUX}(\mathsf{pick}\;n\;(\mathsf{enum}\;n\;i, s'), y, x)) \\
 & & \quad)(s,a))\;)
\end{array}
$$

Abbildung 4.7: Abgeleitete reguläre Strukturen

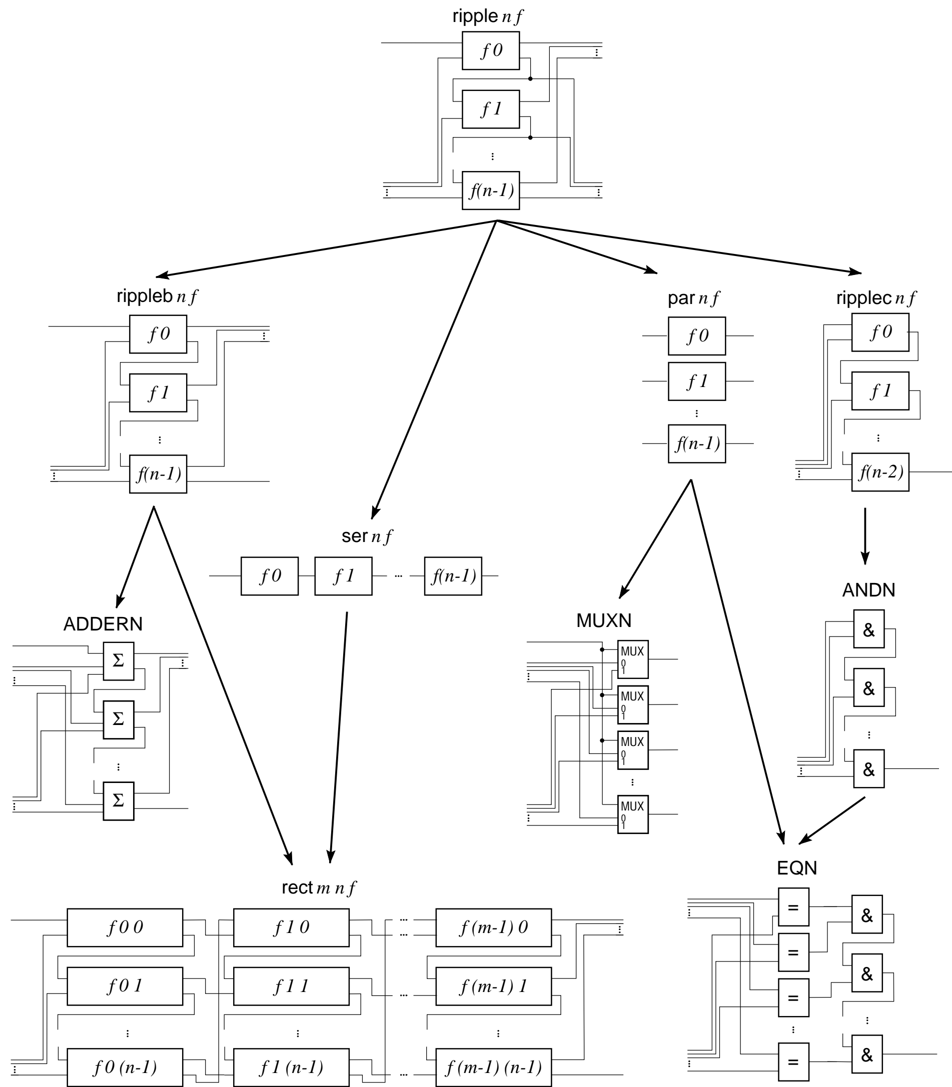


Abbildung 4.8: Graphische Darstellung von regulären Strukturen aus Abbildung 4.7

Übergangsverhalten eine einzige Aus- und Übergangsfunktion verwendet.

4.3.1 Beschreibung sequentieller Schaltungen durch Automaten

Jeder Automat wird durch ein Paar (f, q) bestehend aus einer Aus- und Übergangsfunktion f und einem Initialzustand q repräsentiert. Der Typ von f sei $\iota \times \sigma \to \omega \times \sigma$ und der Typ von q sei σ. Dabei stehen ι, ω und σ für die Typen der Eingangssignale, Ausgangssignale bzw. Zustandswerte. Da dem Term (f, q) in HOL diese Typen bereits fest zugeordnet sind, müssen Eingangsalphabet, Ausgangsalphabet und Zustandsmenge nicht explizit angegeben werden.

Die Aus- und Übergangsfunktion f sei gemäß Abschnitt 4.1 durch einen DFG-Term beschrieben und q durch einen Ausdruck. Es ist zu beachten, daß (f, q) nur dann als Automatenbeschreibung zulässig ist, wenn der Typ von (f, q) eine beliebige Spezialisierung des Typs $(\iota \times \sigma \to \omega \times \sigma) \times \sigma$ ist.

Definition 7 (Automat/sequentielle Schaltung) *Ein Automaten (eine sequentielle Schaltungen) ist ein Pärchen (f, q), wobei f ein DFG-Term mit dem Typ $\iota \times \sigma \to \omega \times \sigma$ und q ein Wert vom Typ σ ist.*

Die Entscheidung, die Ausgangsfunktion und die Übergangsfunktion zu einer Funktion zusammenzufassen, rührt daher, daß diese in der Praxis durch eine Struktur elementarer Gatterelemente realisiert werden soll. Im allgemeinen ist es dabei so, daß innerhalb der Schaltungsstruktur Signalwerte berechnet werden, die sowohl zur Bestimmung der Ausgangswerte als auch zur Bestimmung des Nachfolgezustands verwendet werden. Die Komponenten, mit denen diese Werte berechnet werden, lassen sich weder eindeutig nur der Ausgangs- noch nur der Übergangsfunktion zuordnen. Werden Ausgangsfunktion und Übergangsfunktion getrennt formalisiert, so müßten diese Komponenten doppelt instantiiert werden.

Die bisher betrachteten kombinatorischen Schaltungen wurden zeitlos modelliert. Durch eine Automatenbeschreibung (f, q) wird eine Abbildung zwischen einem zeitabhängigen Eingangssignal auf ein zeitabhängiges Ausgangssignal definiert. Zeit wird dabei durch natürliche Zahlen vom Typ num repräsentiert. Die zeitabhängigen Ein- und Ausgangssignale werden durch Folgen vom Typ $\mathsf{num} \to \iota$ bzw. $\mathsf{num} \to \omega$ modelliert. Durch den Automaten (f, q) wird eindeutig eine Funktion g beschrieben, die ein zeitabhängiges Eingangssignal auf ein zeitabhängiges Ausgangssignal abbildet. g hat also folgenden Typ:

$$(\mathsf{num} \to \iota) \to (\mathsf{num} \to \omega)$$

Durch die Funktion $\mathsf{automaton}$, deren Semantik nachfolgend noch formal definiert werden soll, wird der Zusammenhang zwischen einem Automaten (f, q) und der Ein- und Ausgabefunktion $g = \mathsf{automaton}(f, q)$ hergestellt. Ihr Typ ist:

$$((\iota \times \sigma \to \omega \times \sigma) \times \sigma) \;\to\; (\mathsf{num} \to \iota) \to (\mathsf{num} \to \omega)$$

Sequentielle Schaltungen sollen durch Ausdrücke der Form $\mathsf{automaton}(f,q)$ beschrieben werden. Abbildung 4.9 zeigt die technische Umsetzung von $\mathsf{automaton}(f,q)$ in Form einer Schaltungsstruktur bestehend aus einer kombinatorischen Einheit f und einer Speicherkomponente $\mathcal{D}[q]$ mit Anfangswert q.

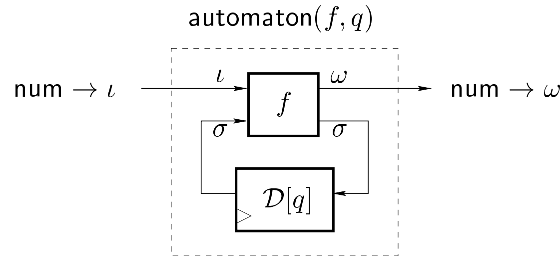


Abbildung 4.9: Implementierung von $\mathsf{automaton}(f,q)$

Äquivalenzumformungen auf Automaten stellen Schaltungstransformationen dar, die das synchrone Verhalten der Schaltung nicht verändern. Zwei Automaten (f_1,q_1) und (f_2,q_2) sind äquivalent genau dann, wenn $\mathsf{automaton}(f_1,q_1) = \mathsf{automaton}(f_2,q_2)$ gilt. Trivialerweise sind zwei Automaten äquivalent, wenn $(f_1,q_1) = (f_2,q_2)$ gilt. Dies ist eine hinreichende, jedoch keine notwendige Bedingung für die Äquivalenz. Es sollen im folgenden insbesondere auch Schaltungstransformationen betrachtet werden, die zwar die Äquivalenz erhalten, bei denen jedoch diese Bedingung nicht erfüllt ist.

4.3.2 Semantik von automaton

Zur Definition der Semantik der Funktion automaton wird zunächst eine Hilfsfunktion namens state definiert werden. Die Funktion state bildet einen Automaten (f,q) auf eine Funktion ab, die eine Eingangsfolge vom Typ $\mathsf{num} \to \iota$ auf eine Zustandsfolge $s = \mathsf{state}(f,q)$ mit dem Typ $\mathsf{num} \to \sigma$ abbildet.

Zu einem vorgegebenen, zeitabhängigen Eingangssignal *input* beschreibt der Ausdruck $\mathsf{state}\ (f,q)\ \mathit{input}$ eine Zustandsfolge und $\mathsf{state}\ (f,q)\ \mathit{input}\ t$ den Wert der Zustandsfolge zu einem Zeitpunkt t. Die folgende Definition von state erfolgt mittels primitiver Rekursion über der Zeit t.

$$\begin{aligned} \mathsf{state}\ (f,q)\ \mathit{input}\ 0 &:= q \\ \mathsf{state}\ (f,q)\ \mathit{input}\ (\mathsf{SUC}\ t) &:= \mathsf{SND}(\ f(\mathit{input}(t), \mathsf{state}\ (f,q)\ \mathit{input}\ t)\ ) \end{aligned}$$

Auch zu $\mathsf{state}(f,q)$ kann eine entsprechende technische Realisierung in Form einer Schaltungsstruktur angegeben werden (siehe Abbildung 4.10).

Auf der Definition von state aufbauend kann jetzt die Semantik von automaton definiert werden. Der Ausdruck $\mathsf{automaton}\ (f,q)\ \mathit{input}$ steht für die Ausgangsfolge. Der Ausdruck $\mathsf{automaton}\ (f,q)\ \mathit{input}\ t$ bezeichnet den Ausgangswert, den ein

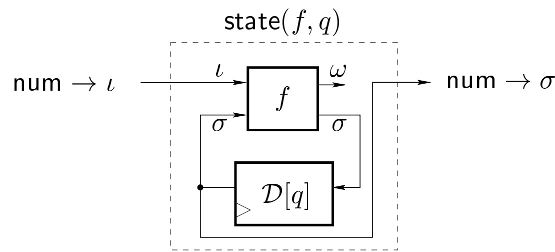


Abbildung 4.10: Implementierung von state(f, q)

Automat (f, q) mit Eingangssignal i zum Zeitpunkt t erzeugt.

$$\mathsf{automaton}\,(f,q)\;input\,t \quad := \quad \mathsf{FST}(f(input(t), \mathsf{state}\;(f,q)\;input\,t))$$

Zur Definition von automaton waren drei Folgen von Bedeutung: das Eingangssignal, das Ausgangssignal und die Zustandsfolge. Abbildung 4.11 illustriert den Zusammenhang, der durch obige Definition zwischen diesen Folgen hergestellt wurde. Gegeben sind die mit gefüllten Kreisen dargestellten Werte: der Initialzustand q und die Eingangsfolge *input*. Ausgehend von diesen Werten werden mit Hilfe der Aus- und Übergangsfunktion die weiteren Zustandswerte und die Ausgangswerte bestimmt. Bei automaton ist die Zustandsfolge nach außen hin nicht sichtbar. Bei der Abbildung der Eingangssignalfolge auf die Ausgangssignalfolge ist sie lediglich ein internes Zwischenergebnis.

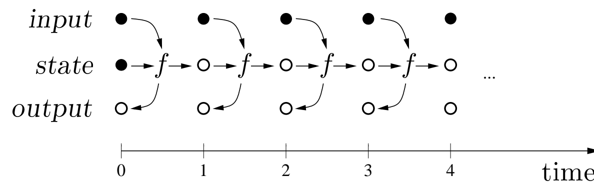


Abbildung 4.11: Signalfolgen bei automaton(f, q)

4.3.3 Kombinatorische Schaltungen als Spezialfälle sequentieller Schaltungen

Mit Hilfe des Konstrukts automaton ist es möglich, beliebige sequentielle Schaltungen zu beschreiben, insbesondere aber auch rein kombinatorische Schaltungen. Dazu wird für den Typ des Speichers σ der Typ one instantiiert. In der Aus- und Übergangsfunktion wird durch eine kombinatorische Funktion der Eingangswert auf den Ausgangswert abgebildet. Der alte Speicherinhalt wird ignoriert und als Nachfolgewert des Speichers wird konstant der Wert one zugewiesen (Abbildung

4.12).

$$\mathsf{combinatorial_block}\ g \quad :\hat{=} \quad \mathsf{automaton}\ ((\lambda(x, y_{\mathsf{one}}).\ (g(x), \mathsf{one})),\ \mathsf{one})$$

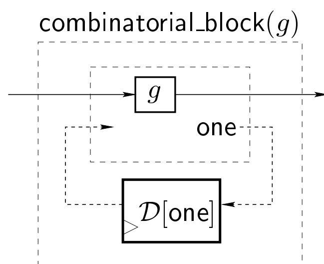


Abbildung 4.12: Kombinatorische Schaltung als Sonderfall des Automaten

4.4 Auswertung und Simulation

Alle Schaltungen werden in diesem Ansatz als Funktionen beschrieben, die Eingänge auf Ausgänge abbilden. Schaltungsbeschreibungen zu simulieren, heißt nichts anderes, als zu gegebenen Eingangswerten die Funktion auszuwerten. Dies kann für den Schaltungsentwerfer nützlich sein, wenn er für bestimmte Fälle überprüfen will, ob sich die Schaltung wie gewünscht verhält. Ein derartiger Schritt kann jedoch auch während der Schaltungssynthese erforderlich werden. So muß beispielsweise beim Retiming (Abschnitt 6.3.5) ein neuer Initialzustand $f_1(q)$ für die neue Speicherkomponente berechnet werden. Dabei ist f_1 eine kombinatorische Schaltung und q der alte Initialzustand. Die Auswertung dieses Ausdrucks ist nichts anderes als ein Simulationslauf der Schaltung f_1 mit q als Eingangswert. Der Begriff Auswertung soll ab sofort für die Berechnung eines Funktionswerts $f(x)$ verwendet werden, wobei f eine kombinatorische Schaltung und x ein konstanter Wert ist. Will man bei der Formalen Synthese Schaltungstransformationen wie den Retiming-Schritt automatisiert ausführen können, so ist die automatisierte Auswertung solcher Ausdrücke dafür eine notwendige Voraussetzung.

Voraussetzung für die Auswertung zusammengesetzter kombinatorischer Schaltungen ist die Auswertung der Grundoperationen. Dazu können sehr einfache Gleichungen verwendet werden, die für verschiedene Eingangswerte die Funktionswerte explizit angeben. Die nachfolgenden Gleichungen beschreiben die Auswertung der Grundoperation AND:

$$\vdash \mathsf{AND}(\mathsf{F},\mathsf{F}) = \mathsf{F}$$
$$\vdash \mathsf{AND}(\mathsf{F},\mathsf{T}) = \mathsf{F}$$
$$\vdash \mathsf{AND}(\mathsf{T},\mathsf{F}) = \mathsf{F}$$
$$\vdash \mathsf{AND}(\mathsf{T},\mathsf{T}) = \mathsf{T}$$

Zusammengesetzte Schaltungen werden in dieser Arbeit durchgängig durch Funktionen beschrieben, die, wie alle Terme in HOL, typisierte λ-Ausdrücke sind. Für diese Funktionen gibt es ein einfaches Verfahren, um Ausdrücke auszuwerten:

- Expansion zusammengesetzter Ausdrücke
- Durchführen von β-Reduktionen
- Anwendung der Auswertungsgleichungen für die Grundoperationen

In welcher Reihenfolge man diese Schritte durchführt, ist für das Ergebnis unerheblich, denn das Verfahren ist konfluent, führt also immer zu dem gleichen Ergebnis. Oft muß bei einer Auswertung eine sehr große Anzahl solcher Schritte durchgeführt werden, und es kann auch passieren, daß sehr große Zwischenergebnisse entstehen. Die Reihenfolge, in der Schritte durchgeführt werden, hat einen wesentlichen Einfluß darauf, wie viele Schritte angewandt werden müssen und wie groß die Zwischenergebnisse geraten. Die Reihenfolge bestimmt also maßgeblich Zeit- und Speichereffizienz des Auswertungsvorgangs.

Außer der Optimierung der Reihenfolge, kann es auch interessant sein, zusätzliche Theoreme für die Auswertung bereitzustellen. Das Theorem

$$\vdash \mathsf{AND}(\mathsf{F}, a) = \mathsf{F}$$

erlaubt es zum Beispiel, den Funktionswert bereits dann zu bestimmen, wenn erst einer der Eingangswerte ermittelt ist. Dies kann den Aufwand für die Auswertung erheblich reduzieren. Man bedenke dabei, daß a für einen beliebigen, eventuell sehr großen booleschen Term stehen kann. Verwendet man dieses Theorem, so muß man nicht zuerst a auswerten, um $\mathsf{AND}(\mathsf{F}, a)$ auszuwerten zu können. Beschränkt man sich dagegen auf die oben aufgeführten Auswertungsgleichungen von AND, so müssen stets alle Eingangswerte bestimmt werden, bevor der Funktionswert bestimmt werden kann.

Um die Auswertungsgeschwindigkeit weiter zu erhöhen, ist es auch möglich, Auswertungstheoreme für zusammengesetzte Komponenten abzuleiten. Mit Hilfe der Theoreme

$$\vdash \mathsf{EQ1}(x, x) = \mathsf{T}$$
$$\vdash \mathsf{EQ1}(\mathsf{F}, \mathsf{T}) = \mathsf{F}$$
$$\vdash \mathsf{EQ1}(\mathsf{T}, \mathsf{F}) = \mathsf{F}$$

kann man beispielsweise den Baustein EQ1 direkt auswerten und muß ihn nicht auf erst auf seine Grundkomponenten AND, OR und INV zurückführen.

Kapitel 5

Vergleich mit anderen Formalisierungsansätzen

Es gibt viele verschiedene Möglichkeiten, synchrone Schaltungen in der Logik zu formalisieren. Die formale Modellierung einer Schaltung bildet die Grundlage für deren Verifikation. Eine fehlerhafte Formalisierung einer Schaltungen macht jede Anstrengung zur Verifikation sinnlos. Bei zahlreichen gebräuchlichen Ansätzen ist oft nicht a priori erkennbar, ob die Formalisierung fehlerhaft ist. In diesem Kapitel werden zwei Problemfelder angesprochen, die bei zahlreichen bestehenden Ansätzen auf RT- und Gatterebene zu falschen Ergebnissen führen können: die relationale Schaltungsmodellierung und die unsachgemäße Verwendung nichtboolescher Datentypen und Operationen.

Aufbauend auf den Erfahrungen mit bisherigen logischen Modellen für Schaltungsbeschreibungen, sind es vor allem drei Forderungen, durch die dieser Ansatz motiviert wurde:

- Gewährleistung der Konsistenz
- Gewährleistung der Implementierbarkeit
- Simulierbarkeit der Schaltungsbeschreibung

Die in Kapitel 4 vorgestellte Sprache wurde so konstruiert, daß diese Eigenschaften für alle Schaltungsbeschreibungen in der Sprache erfüllt sind. Im Gegensatz zu den meisten Formalisierungsansätzen werden in dieser Sprache Schaltungen nicht relational [Melh93], sondern funktional modelliert. In bezug auf nichtboolesche Datentypen und Operationen ist die Sprache konservativ. Jeder Operator ist für sich als Hardware implementierbar. Oft wird in diesem Bereich der Fehler gemacht, einfach Operationen aus den Softwarprogrammiersprachen zu übernehmen. Nicht jede Operation aus Softwareprogrammiersprachen ist jedoch eine geeignete Modellierung einer Hardwarekomponente.

5.1 Relationale und funktionale Schaltungsbeschreibungen

Beim relationalen Ansatz werden Schaltungen als Relationen zwischen Signalen beschrieben. Zwischen Ein- und Ausgangssignalen wird beim relationalen Formalisierungsschema nicht unterschieden. Der funktionale Charakter, der allen real existierenden synchronen Schaltungen zu eigen ist, geht durch die relationale Schaltungsbeschreibung verloren. Mit Hilfe der Relationen kann lediglich eine Aussage getroffen werden, ob eine Menge von Signalen in der Relation zueinander stehen, wie dies von der Schaltung erzwungen wird. Die Relationen eignen sich jedoch nicht dazu, konstruktiv aus den Eingangssignalen die Ausgangssignale zu bestimmen.

Auf eine bestimmte Zeitabstraktion ist man beim relationalen Formalisierungsansatz nicht festgelegt. Zur Modellierung synchroner Schaltungen werden Signale durch Funktionen vom Typ $\mathsf{num} \rightarrow \alpha$ repräsentiert. Nachfolgend vier Beispiele zur Formalisierung synchroner Grundkomponenten im relationalen Stil: ein UND-Gatter (PAND), ein ODER-Gatter (POR), ein Inverter (PINV) und ein D-Flipflop (PDFF).

$$
\begin{aligned}
\mathsf{PAND}(a,b,c) &:= (\forall t.\, c(t) = a(t) \wedge b(t)) \\
\mathsf{POR}(a,b,c) &:= (\forall t.\, c(t) = a(t) \vee b(t)) \\
\mathsf{PINV}(a,b) &:= (\forall t.\, b(t) = \neg a(t)) \\
\mathsf{PDFF}(a,b) &:= (b(0) = \mathsf{F}) \wedge (\forall t.\, b(t+1) = a(t))
\end{aligned}
$$

Synchrone Schaltungen stellen stets einen funktionalen Zusammenhang zwischen Ein- und Ausgabe her. Relationale Schaltungsbeschreibungen sind allgemeiner: zu gegebenen Eingangssignalen gibt es i. allg. mehrere Ausgangssignale, die die Relation erfüllen. Eine relationale Schaltungsbeschreibung steht somit i. allg. nicht für eine einzelne Schaltung, sondern für eine Menge von Schaltungen. Beispiel für eine mehrdeutige Schaltungsbeschreibung ist die folgender Term $\mathsf{XY}(a,b,c)$:

$$
\mathsf{XY}(a,b,c) := (\forall t.\, c(t) = a(t) \wedge b(t)) \;\vee\; (c(5) = \mathsf{T})
$$

Beim relationalen Ansatz lassen sich Schaltungsstrukturen sehr einfach darstellen. Abbildung 5.1 zeigt ein Beispiel für die relationale Formalisierung einer Schaltungsstruktur R, die sich aus den Komponenten A, B und C zusammensetzt. Die Signalvariablen werden an die Komponenten angelegt, diese werden dann konjunktiv verknüpft und die internen Signalvariablen werden existenzquantifiziert.

Mit relationalen Schaltungsstrukturen ist es scheinbar möglich, beliebige Verdrahtungen zwischen beliebigen synchronen Komponenten darzustellen. Dies ist jedoch nur bedingt richtig. Auch dann, wenn die Grundkomponenten einer synchronen Schaltung richtig modelliert wurden, so stellt doch nicht jede mögliche Schaltungsstruktur die Modellierung einer realisierbaren Schaltung dar.

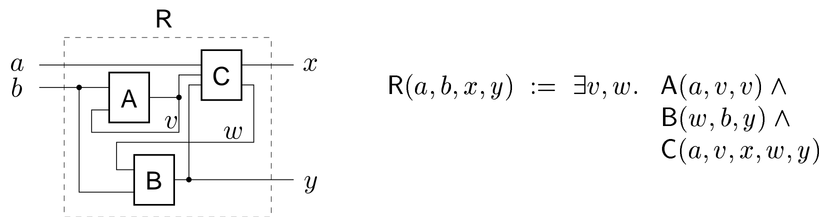


Abbildung 5.1: Relationale Modellierung einer Schaltungsstruktur

Relationale Schaltungsstrukturen sind z. B. nur dann zutreffend formalisiert, wenn gewährleistet werden kann, daß zu keinem Zeitpunkt zwei Komponenten gleichzeitig mit unterschiedlichen Werten schreibend auf eine Leitung zugreifen. In [Melh93] wird vorgeschlagen, geeignete Konsistenzbedingungen aufzustellen, die dann explizit bewiesen werden müssen. Abbildung 5.2 zeigt das Problem in seiner elementaren Form: eine Signalleitung x verbindet zwei Komponenten U und V. Im allgemeinen können sowohl U als auch V schreibend auf x zugreifen. Es kann sein, daß sowohl durch die Relation U als auch durch die Relation V zu einem bestimmten Zeitpunkt eine eindeutige Aussage über den Signalwert gemacht wird, die der jeweils anderen widerspricht. Diese Schaltungsbeschreibung ist nur dann implementierbar, wenn es ein Signal x mit $\mathsf{U}(x)$ und $\mathsf{V}(x)$ gibt. Dazu muß das Verhalten von U und V in allen mögliche Situationen und für alle Zeitpunkte untersucht werden. Beweise dieser Art sind bereits für reine Schaltnetze NP-vollständig.

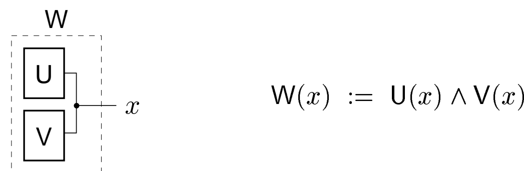
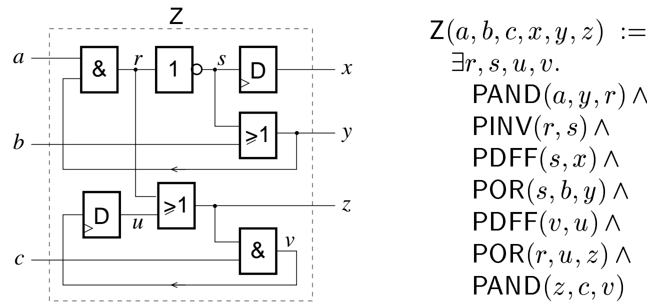


Abbildung 5.2: Relational beschriebene Verbindung

Allgemein kann gesagt werden, daß relationale Strukturbeschreibungen mit beliebigen Verdrahtungen i. allg. nicht gewährleisten können, daß die Gesamtschaltung richtig modelliert wird, und zwar auch dann nicht, wenn alle Teilkomponenten richtig modelliert wurden. Es muß stets explizit untersucht werden, ob die gegebene Beschreibung tatsächlich eine implementierbare Schaltung darstellt.

Die Konsequenzen und Irrtümer, die sich aus derartigen Fehlern bei der Formalisierung für die Verifikation ergeben, sollen anhand eines Beispiels erläutert werden. Abbildung 5.3 zeigt eine Schaltungsstruktur Z, die durch eine relationale Schaltungsbeschreibung modelliert wird. Die Schaltungsstruktur von Z enthält einen kombinatorischen Zyklus, und der klassische relationale Formalisierungsansatz führt hier zu einer fehlerhaften Modellierung. Als Konsequenz entsteht eine Schaltungsbeschreibung Z, die nicht implementierbar ist. Es gibt keine Funk-

tion, die beliebige Eingangssignale a, b und c auf x, y und z abbildet, so daß $\mathsf{Z}(a, b, c, x, y, z)$ gilt. Dazu betrachte man drei Signale a, b und c mit der Eigenschaft $a(0) = \mathsf{T}$, $b(0) = \mathsf{F}$ und $c(0) = \mathsf{F}$. Für diese Eingangssignale a, b und c gibt es keine Ausgangssignale x, y und z mit der Eigenschaft $\mathsf{Z}(a, b, c, x, y, z)$. Ein solcher Widerspruch kann technisch nicht interpretiert werden. In einer realen Schaltung gibt es den Fall nicht, daß einem Signal zu einem bestimmten Zeitpunkt kein Wert zugeordnet werden kann. Schaltungsbeschreibungen wie Z, zu denen es kein reales Gegenstück gibt, sollen als *Scheinschaltungen* bezeichnet werden.



$$
\begin{aligned}
&\mathsf{Z}(a, b, c, x, y, z) \;:= \\
&\quad \exists r, s, u, v. \\
&\qquad \mathsf{PAND}(a, y, r) \wedge \\
&\qquad \mathsf{PINV}(r, s) \wedge \\
&\qquad \mathsf{PDFF}(s, x) \wedge \\
&\qquad \mathsf{POR}(s, b, y) \wedge \\
&\qquad \mathsf{PDFF}(v, u) \wedge \\
&\qquad \mathsf{POR}(r, u, z) \wedge \\
&\qquad \mathsf{PAND}(z, c, v)
\end{aligned}
$$

Abbildung 5.3: Kombinatorischer Zyklus

Bezüglich der Konsequenzen, die sich aus derartigen Formalisierungsfehlern ergeben, betrachte man folgende Schaltungsbeschreibung A:

$$
\begin{aligned}
\mathsf{A}(a, b, c, x, y, z) \;:=\; & (a(0) = \mathsf{T}) \;\wedge\; (b(0) = \mathsf{F}) \;\wedge\; (c(0) = \mathsf{F}) \\
& \Rightarrow \;\; (x(17) = \mathsf{F}) \;\wedge\; (y(31) = \mathsf{F}) \;\wedge\; (z(52) = \mathsf{F})
\end{aligned}
$$

Der Beweis, dafür daß Z eine Verfeinerung von A ist, daß also

$$\vdash \forall a, b, c, x, y, z.\, \mathsf{Z}(a, b, c, x, y, z) \Rightarrow \mathsf{A}(a, b, c, x, y, z)$$

gilt, kann logisch korrekt erbracht werden. Der Beweis legt die Vermutung nahe, eine Schaltung Z gefunden zu haben, die die Eigenschaft A erfüllt. Dabei ist der einzige Grund dafür, daß dies zutrifft, die fehlerhafte Formalisierung der Schaltung Z.

In der Praxis haben auch kombinatorische Komponenten eine Verzögerungszeit. Die Vereinfachung, kombinatorische Schaltungen bei einer synchronen Modellierung als verzögerungsfrei zu betrachten, ist nicht zulässig, wenn die Schaltungsstruktur kombinatorische Zyklen enthält. Tatsächlich führt die Realisierung zu einem asynchron schwingenden Verhalten. Die aus dem fehlerhaften synchronen Modell abgeleiteten Behauptungen werden jedenfalls nicht erfüllt. Verifikationen mit Scheinschaltungen sollen als *Scheinverifikationen* bezeichnet werden.

Oft wird bei Beweisen mit relationalen Schaltungsbeschreibungen der Fehler gemacht, nicht abzusichern, daß die Schaltung frei von Inkonsistenzen ist. Die Tatsache, daß beim relationalen Formalisierungsansatz korrekt formalisierte Teilkomponenten manchmal nicht zu einer korrekt formalisierten Gesamtschaltung

führen, wird ignoriert. Die darauf aufbauenden Beweise sind dann stets mit Vorbehalt zu betrachten. Die Aussage, die aus diesen Beweisen tatsächlich gewonnen werden kann, lautet lediglich: Die Schaltung ist korrekt, wenn die Schaltungsstruktur keine Kurzschlüsse, kombinatorischen Zyklen, etc. enthält. Enthält die Schaltung z. B. einen Kurzschluß, so hat man die Spezifikation ex falso quodlibet bewiesen — man hätte aber auch jede andere Spezifikation beweisen können. Verifikationstechniken, bei denen man sich nicht sicher sein kann, ob das erzielte Ergebnis nicht einfach damit zusammenhängt, daß man sich bei der Formalisierung der Schaltung in Widersprüche verwickelt hat und nur dies der Grund für den scheinbaren Erfolg war, sind irreführend und für die Praxis des Schaltungsentwurfs nicht geeignet.

Bei einer Formalen Synthese hat die Sicherstellung der Implementierbarkeit eine besondere Bedeutung. Bei jedem Verfeinerungsschritt kann es, wie etwa bei dem Übergang von A nach Z, zu einer Scheinschaltung kommen. Das Formale-Synthese-System Dialog [AHL92] weist ein klares Konzept zur Sicherstellung von Konsistenz und Implementierbarkeit auf. In Dialog werden Schaltungsstrukturen relational beschrieben. Dadurch daß jedoch strikt zwischen Ein- und Ausgängen unterschieden wird, vereinfacht sich die Sicherstellung der Konsistenz. In Dialog wird nach jedem Verfeinerungsschritt überprüft, ob die Schaltung kombinatorische Zyklen oder Kurzschlüsse enthält (siehe Abschnitt 2.5.5). Synchrone Schaltungsstrukturen ohne kombinatorische Zyklen und Kurzschlüsse werden durch die relationale Modellierung korrekt wiedergegeben. Für diese Überprüfung wird in Dialog für jede Schaltung eindeutig festgelegt, welche Anschlüsse Ein- und welche Anschlüsse Ausgänge sind. Ferner wird für jeden Ausgang bestimmt, zu welchen Eingängen er einen kombinatorischen Pfad hat. Diese Informationen sind für elementare Schaltungen vorgegeben und werden für zusammengesetzte Schaltungen aus der Struktur und aus den entsprechenden Informationen der Teilkomponenten abgeleitet. Vorteil dieser Methode ist, daß zur Sicherstellung der Konsistenz lediglich der Graph der Struktur untersucht werden muß und nicht das Verhalten der Teilkomponenten.

Der in Kapitel 4 vorgestellte Ansatz zur funktionalen Schaltungsbeschreibung unterscheidet sich wesentlich von der relationalen Schaltungsbeschreibung. Er ist bei weitem restriktiver. Durch die funktionalen Beschreibungen können ausschließlich implementierbare kombinatorische Schaltungen beschrieben werden. Zwischen Ein- und Ausgangssignalen wird implizit unterschieden, eine Verwechslung ist syntaktisch ausgeschlossen. Die kombinatorischen Schaltungsstrukturen sind zyklenfrei und können keine Kurzschlüsse enthalten. Zyklen entstehen lediglich bei sequentiellen Schaltungen, und diese Zyklen führen immer über Speicherbausteine. Kurzschlüsse und kombinatorische Zyklen sind damit syntaktisch ausgeschlossen. Es gilt das Prinzip, daß alle syntaktisch ausdrückbaren Beschreibungen auch real produzierbare synchrone Schaltungen sind. Eine gesonderte Überprüfung entfällt damit. Dies ist ein wesentlicher Vorteil gegenüber Ansätzen wie Dialog, bei denen nach jedem Verfeinerungsschritt mit recht aufwendigen Verfahren die Konsistenz der Beschreibung überprüft werden muß.

5.2 Harmlose kombinatorische Zyklen

Es gibt vor allem zwei wichtige Gründe, die dagegen sprechen, bei synchronen Schaltungen Zyklen im kombinatorischen Teil zuzulassen: ein mögliches asynchrones Schwingverhalten und Speichereffekte. Kombinatorische Zyklen können zu einem asynchronen Schwingverhalten führen, d. h. trotz stabil anliegender Eingangssignale und Ausgangssignale der Speicherbausteine konvergieren die Werte an den Ausgängen nicht. Ist gewährleistet, daß das Verhalten des kombinatorischen Anteils der Schaltung stets konvergiert, so ist noch zu beachten, daß die Schaltung einen speichernden Effekt haben kann, was dazu führt, daß das Verhalten der kombinatorischen Schaltung nicht nur von der aktuellen Eingangsbelegung, sondern auch von der Vorgeschichte abhängt. In diesen Fällen wäre es zweckmäßig, die zyklische kombinatorische Schaltungsstruktur in Speicheranteile und zyklenfreie kombinatorische Teile aufzutrennen.

Interessant sind somit nur noch jene zyklenbehafteten Schaltungsstrukturen aus kombinatorischen Schaltungen, die stets konvergieren und die kein speicherndes Verhalten haben. Diese Zyklen sollen als *harmlose* kombinatorische Zyklen bezeichnet werden. Eine Schaltung mit harmlosen kombinatorische Zyklen stellt zu jedem Zeitpunkt einen direkten funktionalen Zusammenhang zwischen anliegenden Eingangs- und Ausgangswerten dar. Die Bestimmung der Ausgangswerte aus den Eingangswerten ist bei Schaltungen mit harmlosen kombinatorischen Zyklen ungleich aufwendiger als bei zyklenfreien kombinatorischen Schaltungen. Die Ausgangswerte ergeben sich aus den Eingangswerten als der Grenzwert der Ausgangsignale nach einem Einschwingvorgang. Dieser Grenzwert muß für alle Eingangsbelegungen existieren und jeweils eindeutig sein. Dabei sind insbesondere auch die konkreten zeitlichen Verzögerungen der kombinatorischen Einheiten zu berücksichtigen.

Definition 8 (Harmloser kombinatorischer Zyklus) *Ein kombinatorischer Zyklus in einer kombinatorischen Schaltungsstruktur soll als harmlos bezeichnet werden, wenn sich die Schaltung trotzdem als eine Funktion zwischen aktuellen Eingangs- und Ausgangswerten beschreiben läßt.*

Da mit zyklenfreien kombinatorischen Schaltungen bereits beliebige Funktionen realisiert werden können, also auch all jene, die mit harmlosen kombinatorischen Zyklen beschreibbar sind, mag es erscheinen, als ob derartige Schaltungsstrukturen nicht notwendig sind und man prinzipiell auf kombinatorische Zyklen verzichten könnte. Im allgemeinen ist dies richtig. Es gibt jedoch, wie noch gezeigt werden soll, bei der Synthese Fälle, in denen eine Schaltung mit kombinatorischem Zyklus allen äquivalenten Schaltungen ohne kombinatorischen Zyklus überlegen ist.

Die Handhabung kombinatorischer Schaltungen mit harmlosen Zyklen ist bei weitem aufwendiger als die Handhabung zyklenfreier kombinatorischer Schaltungen (siehe auch [Sent97, ShBT96]). Dafür gibt es drei Gründe:

1. Die Untersuchung, ob kombinatorische Zyklen überhaupt harmlos sind, ist NP-vollständig.

2. Es ist aufwendig, zu vorgegebenen Eingangswerten die Funktionswerte der Schaltung zu bestimmen, die sich im allgemeinen erst nach einer Einschwingphase als stabile Werte einstellen.

3. Es ist aufwendig, die Verzögerungszeit bzw. die erzielbare Taktfrequenz zu bestimmen.

Im allgemeinen müssen die genauen Verzögerungszeiten der kombinatorischen Komponenten untersucht werden, um entscheiden zu können, ob kombinatorische Zyklen harmlos sind. Abbildung 5.4 zeigt eine Schaltung, deren Zyklus i. allg. nicht harmlos ist. Kommt es an der Leitung c zu einem T-Signal, so fängt der Ausgang y an zu schwingen. Ist hingegen die Signalleitung c konstant auf F, so wird durch diese Schaltung eine Funktion realisiert, die ein Eingangssignal a auf ein Ausgangssignal y abbildet, das konstant den Wert F hat. Die Leitung b hat normalerweise den Wert F. Nur nach einer Flanke von F nach T am Eingang a kommt es an b zu einem kurzen T-Impuls, dessen Länge gerade der Verzögerungszeit des Inverters entspricht. Dieser Impuls wird dann nach c weitergeleitet, wenn der Betrag der Differenz aus den Verzögerungszeiten von NAND- und NOR-Gatter größer als die Verzögerungszeit des Inverters ist. In Abhängigkeit der Verzögerungszeiten der Gatter ist der Zyklus der Schaltung also harmlos oder nicht.

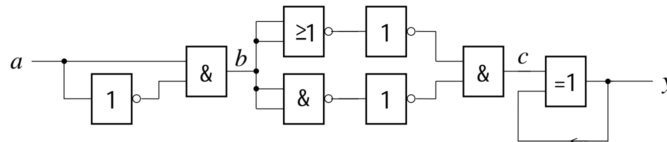


Abbildung 5.4: Kombinatorischer Zyklus

Schaltungsbeschreibungen, bei denen die Konvergenz der Funktionswerte nur dann gewährleistet werden kann, wenn bestimmt Zusammenhänge zwischen den Signallaufzeiten eingehalten werden können, sind bei der Synthese nicht praktikabel. Da die konkreten Verzögerungszeiten sehr stark vom Layout der Schaltung abhängen, müßten diese Anforderungen an Plazierungs- und Verdrahtungsverfahren als Nebenbedingung weitergereicht werden.

Es gibt jedoch auch Schaltungen mit kombinatorischen Zyklen, die unabhängig von den Verzögerungszeiten der Teilkomponenten garantieren können, daß die Zyklen harmlos sind. Abbildung 5.5 zeigt eine solche Schaltung. Daß der Zyklus in dieser Schaltung tatsächlich harmlos ist, läßt sich einfach durch Fallunterscheidung über dem Eingangssignal s zeigen. Durch die Schaltung wird für $s = \mathsf{T}$ die Serienschaltung $y = \mathsf{A}(\mathsf{B}(x))$ und für $s = \mathsf{F}$ die Serienschaltung $y = \mathsf{B}(\mathsf{A}(x))$ realisiert. Für die Funktion, die diese Schaltung realisiert, gibt es keine äquivalente zyklenfreie kombinatorische Schaltung, in der A und B jeweils nur einmal vorkommen. Sind die Kosten für A und B höher als die für die beiden zusätzlichen Multiplexer und stuft man diesen Vorteil bezüglich des Hardwareaufwands höher ein als die durch die beiden Multiplexer verursachte zusätzliche Signalverzögerung, so gibt es

in diesem Fall keine effizientere Implementierung als diese mit einem kombinatorischen Zyklus.

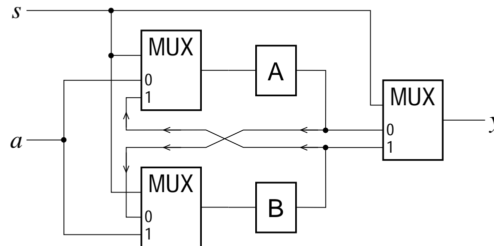


Abbildung 5.5: Harmloser kombinatorischer Zyklus

Schaltungen wie die in Abbildung 5.5 dargestellte können bei ganz bestimmten Verfahren zur High-Level-Synthese sinnvoll sein. Voraussetzung ist, daß die Verfahren „Chaining", also die Verkettung von Operationseinheiten innerhalb eines Taktzyklus unterstützen. Dies ist bei den wenigsten Verfahren der Fall, und selbst dann werden kombinatorische Zyklen in der Regel ausgeschlossen [PaKn89, KJBC93, RaJe95]. Der wichtigste Grund zur Vermeidung kombinatorischer Zyklen dürfte die fehlende Unterstützung durch Werkzeuge zur Weiterverarbeitung auf RT- und Gatterebene sein. Gängige Syntheseprogramme auf RT- und Gatterebene wie ESPRESSO, SIS, etc. weisen Schaltungen mit kombinatorischen Zyklen von vornherein ab. Der Verzicht auf harmlose kombinatorische Zyklen bedeutet nicht, daß damit die High-Level-Synthese mit Chaining nicht möglich wäre, es bedeutet lediglich, daß nicht alle Optimierungspotentiale ausgeschöpft werden, da sich daraus Restriktionen für die Anordnung der Operatoren innerhalb der einzelnen Takte ergeben.

Es gibt jedoch auch Gründe, auf Chaining prinzipiell zu verzichten. Chaining bedeutet einen Mehraufwand bei der Kommunikationssynthese. Ohne Chaining gestaltet sich die Kommunikationssynthese vergleichsweise einfach: Man benötigt Kommunikationseinheiten, die die Registerwerte und Eingangswerte den Operationseinheiten zuführen, und man benötigt Kommunikationseinheiten, die die Ergebnisse an den Ausgängen der Operationseinheiten abholen, um sie in den Registern abzuspeichern bzw. sie den Ausgängen zuzuführen. Mit Chaining müssen zusätzlich Kommunikationswege von den Ausgängen zu den Eingängen der Operationseinheiten zur Verfügung gestellt werden. Diese führen dann unter Umständen zu kombinatorischen Zyklen, von denen dann abgesichert werden muß daß sie harmlos sind.

5.3 Strukturelles und funktionales Entwurfsprinzip

Bei der Synthese synchroner Schaltungen gibt es zwei Entwurfsprinzipien, die sich gegenseitig ausschließen: das strukturelle Entwurfsprinzip und das funktionale Entwurfsprinzip. Beim strukturellen Entwurfsprinzip wird die Gesamtschaltung als Struktur aus den Grundkomponenten abgeleitet. Aus logischen Gattern und elementaren Speicherbausteinen wie D-Flipflops werden mit Hilfe von Schaltungsstrukturen sukzessive komplexere synchrone Schaltungen abgeleitet. Dieses Entwurfsprinzip führt stets zu Schaltungen mit einem eindeutigen, festen zeitlichen Verhalten. Soll das zeitliche Verhalten der Bausteine verändert werden, so führt dies zu einem Neuentwurf.

Dieser klassischen Vorgehensweise stehen heute Verfahren im Bereich der High-Level Synthese gegenüber. Ausgangspunkt ist hier eine rein funktionale Beschreibung, die lediglich den Zusammenhang zwischen Ein- und Ausgangswerten, nicht jedoch das zeitliche Verhalten beschreibt. Bei der High-Level-Synthese wird aus der rein funktionalen Beschreibung eine synchrone Implementierung mit konkretem zeitlichem Verhalten abgeleitet. Dabei hat der Entwerfer die Möglichkeit, Ziele bezüglich der Anzahl benötigter Takte, des maximalen Hardwareaufwands oder der Taktfrequenz vorzugeben und so die Schaltung in flexibler Weise an die jeweilige Aufgabenstellung anzupassen. Dieser Entwurfsstil hat vor allem deshalb so sehr an Bedeutung gewonnen, da er in besonderer Weise die Wiederverwertung von Entwürfen unterstützt. Eine algorithmische Beschreibung bildet den gemeinsamen Ausgangspunkt für verschiedenartig an konkrete zeitliche Abläufe und Kommunikationsprotokolle angepaßte Implementierungen. Ein hohes Maß an Flexibilität bezüglich zeitlichem Verhalten und Hardwareaufwand ist dann von Bedeutung, wenn aufgrund der Größe des Entwurfs und der Vielzahl der Randbedingungen (Hardwareaufwand, Low-Power-Aspekte, etc.) nicht a priori abgeschätzt werden kann, welche konkrete Implementierung angestrebt werden soll. Der Entwerfer hält sich so wichtige Freiheitsgrade offen. Gleichzeitig sind Simulationen auf einer funktionalen, zeitlosen Ebene bei weitem einfacher und effizienter durchführbar als konventionelle Schaltungssimulationen.

Das strukturelle und das funktionale Entwurfsprinzip sind miteinander unvereinbar. Der Schaltungsentwerfer muß sich entscheiden, ob er das zeitliche Verhalten explizit durch seinen Entwurf festlegen oder ob er diese Spielräume beim Entwurf zunächst offen lassen will.

Der in dieser Arbeit vorgestellte Ansatz zielt auf ein funktionales Entwurfsprinzip ab. Schaltungen werden prinzipiell rein funktional als Strukturen kombinatorischer Bausteine entworfen. Die funktionale Beschreibung kann dann entweder direkt als Aus- und Übergangsfunktion einer sequentiellen Schaltung verwendet werden, oder sie kann, wie in Kapitel 7 noch näher ausgeführt werden soll, als Grundlage für eine High-Level-Synthese verwendet werden, wobei dann die zeitlichen Abläufe und der Hardwareaufwand flexibel abgestimmt werden können.

Daß dieser Ansatz auf ein funktionales Entwurfsprinzip abzielt, bedeutet jedoch

nicht, daß es nicht auch möglich wäre, zusammengesetzte sequentielle Strukturen zu modellieren. Es können beliebige synchrone Schaltungen ohne Kurzschlüsse und kombinatorische Zyklen modelliert werden. Es ist ferner, wie beschrieben, möglich, Schaltungen hierarchisch zu beschreiben. Bei der hierarchischen Gliederung einer Schaltung wird jedoch die Einschränkung gemacht, daß nur Strukturen aus kombinatorischen Einheiten und keine Strukturen aus sequentiellen Teilkomponenten beschrieben werden können.

Prinzipiell wird die Schaltung zunächst ohne Speicherbausteine entworfen. Strukturen aus synchronen Schaltungen können in diesem Ansatz jedoch dadurch dargestellt werden, daß die jeweiligen Aus- und Übergangsfunktionen zusammengefaßt werden. Die Einschränkung, die für die hierarchischen Beschreibung einer Schaltung bei dieser Vorgehensweise jedoch gemacht werden muß, ist die, daß die Schaltung keine Scheinzyklen enthalten darf. Als ein Scheinzyklus soll eine Zyklus kombinatorischer Schaltungen verstanden werden, bei denen die nachfolgende Schaltung von den Ergebnissen der Vorgängerschaltung abhängt, ohne daß es dabei zu einem kombinatorischen Zyklus bezüglich der Grundkomponenten kommt.

Definition 9 (Scheinzyklus) *Ein zyklischer Signalpfad in einer Struktur aus mehreren sequentiellen Teilkomponenten wird genau dann als Scheinzyklus bezeichnet, wenn dies nicht zu einem kombinatorischen Zyklus führt.*

Abbildung 5.6 zeigt drei Schaltungen COUNTER2a, COUNTER2b und COUNTER2c, die alle die gleiche Struktur von Grundkomponenten darstellen, bei denen jedoch die Bausteine in unterschiedlicher Weise hierarchisch zusammengefaßt wurden und die deshalb in Abbildung 5.6 auch unterschiedlich angeordnet sind. Die Schaltungsstruktur enthält weder kombinatorische Zyklen noch Kurzschlüsse. Sie läßt sich somit mit dem in dieser Arbeit vorgestellten Formalisierungsschema darstellen, wenngleich, wie gleich gezeigt werden soll, die Komponenten nicht in beliebiger Weise zusammengefaßt werden können. Die Schaltung COUNTER2a besteht aus zwei Teilkomponenten COUNTER. Da die beiden Teilkomponenten nicht rein kombinatorisch sind, läßt sich diese Struktur mit dem vorgestellten Schema zunächst nicht darstellen. In diesem Fall können jedoch, wie in Schaltung COUNTER2b, einfach statt der speicherbehafteten Teilschaltungen vom Typ COUNTER deren Aus- und Übergangsfunktionen count zu einer Funktion count2 zusammengefaßt werden.

Dies ist aber nicht bei jeder Entwurfshierarchie möglich. Die Schaltung COUNTER2c, die sich aus den Teilkomponenten P und Q zusammensetzt, enthält einen Zyklus, denn interne Leitungen führen von ihren kombinatorischen Anteilen p und q hin und zurück. Dabei handelt es sich nur um einen Scheinzyklus, denn der Zyklus führt nicht zu einem kombinatorischen Zyklus. Dieser Scheinzyklus führt dazu, daß es dem in dieser Arbeit vorgestellten Ansatz nicht möglich ist, die beiden kombinatorischen Schaltungen p und q zu einer gemeinsamen kombinatorischen Schaltung zusammenzufassen, und somit ist es nicht möglich die in COUNTER2c vorgegebene Entwurfshierarchie nachzubilden.

Am Beispiel der Schaltung COUNTER2c wird deutlich, daß es bei einem strukturellen Entwurfsprinzip i. allg. notwendig ist, den internen Aufbau von Teilkom-

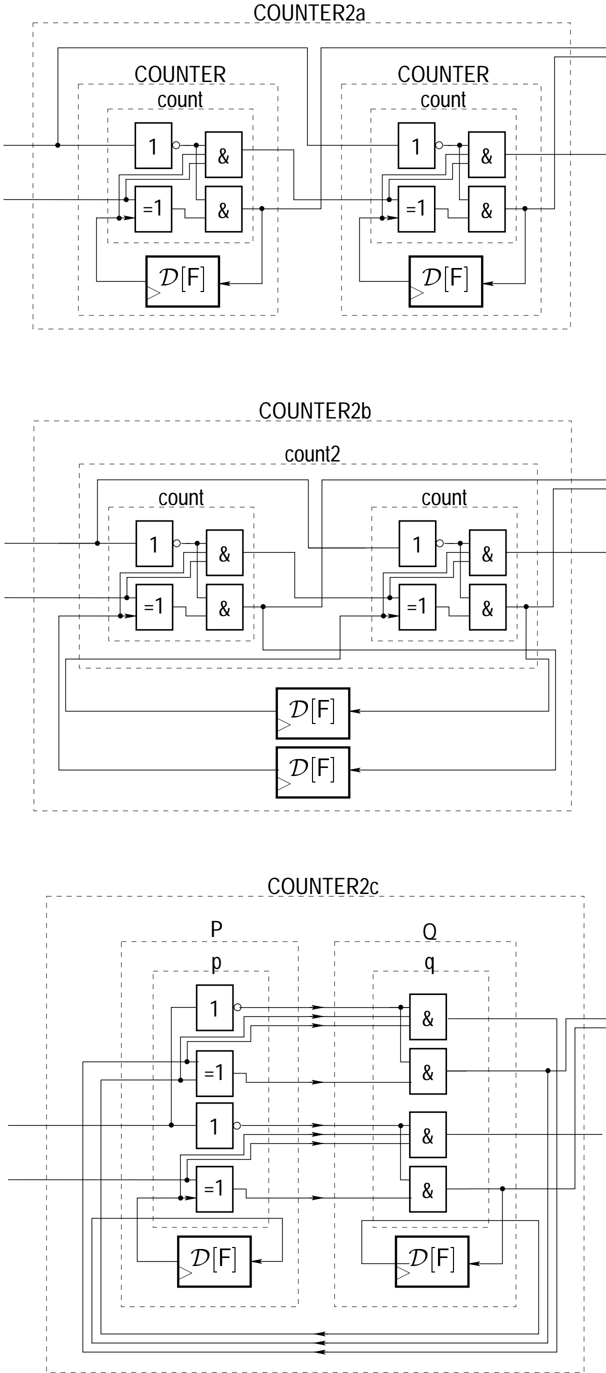


Abbildung 5.6: Schaltungsstruktur mit unterschiedlichen Entwurfshierarchien

ponenten zu kennen, will man beim Zusammenfügen kombinatorische Zyklen vermeiden. Um kombinatorische Zyklen in einer Schaltung zu finden, kann man einfach den gesamten Entwurf ausflachen, um dann in der i. allg. großen Struktur aus Grundkomponenten zu überprüfen, ob sie kombinatorische Zyklen enthält. Eine bessere und effizientere Methode besteht darin, zu jeder Komponente, die man neu konstruiert, die Menge der kombinatorischen Abhängigkeiten zwischen Ein- und Ausgängen zu berechnen [AHL92]. Zu jedem Ausgang wird bestimmt, von welchen Eingängen er in direkter, kombinatorischer Weise abhängt. Verwendet man dann diese Komponente als Teilkomponente in einer Struktur, so ist es zur Vermeidung kombinatorischer Zyklen nicht mehr notwendig, den internen Aufbau der Komponente zu betrachten. Die kombinatorischen Abhängigkeiten zwischen Ein- und Ausgangssignalen sind dazu hinreichend.

Zusammenfassend kann gesagt werden, daß der in dieser Arbeit vorgestellte Ansatz ein funktionales Entwurfsprinzip unterstützt. Dies bedeutet zwar nicht, daß dadurch eine Vorgehensweise gemäß dem strukturellen Entwurfsprinzip prinzipiell ausgeschlossen wäre, es ist jedoch nur eingeschränkt anwendbar. Die Einschränkungen ergeben sich aus der getrennte Formalisierung des kombinatorischen Anteils. Andererseits ist aber in dieser Arbeit jede syntaktisch zulässige Schaltung ein zutreffendes Abbild einer realen Schaltung, und zwar ohne, daß es zur Sicherstellung der Konsistenz einer gesonderten Untersuchung bedarf.

5.4 Typisierung und reguläre Signalbündel

Sollen oberhalb der Gatterebene in Schaltungen nicht nur boolesche Signale und Operatoren verwendet werden, so müssen dafür geeignete Sprachkonstrukte zur Verfügung gestellt werden. Für diese sprachlichen Erweiterungen muß gewährleistet sein, daß sie unter einer geeigneten Kodierung auf eine Schaltungsbeschreibung auf Gatterebene zurückgeführt werden können. Diese Problemstellung soll hier am Beispiel regulärer Schaltungen und regulärer Signalbündel diskutiert werden.

Ähnlich wie bei der Softwareprogrammierung ist die Verwendung einer streng typisierten Sprache zur systematischen Modellierung komplexer Hardwarestrukturen unabdingbar. Es gibt jedoch auch Ansätze, die darauf verzichten. Das System DDD [John84, JoBB88] baut beispielsweise auf nichttypisierten λ-Termen, einer Teilsprache von Lisp, auf. Während die so beschriebenen Terme als Software stets ausführbar sind, stellt sich bei der Verwendung als Hardwarebeschreibungssprache die Frage, ob die so beschriebenen Konstrukte tatsächlich Hardware sein könnten. Um beispielsweise in einer DDD-Schaltungsbeschreibung eine Signalleitung durch einen boolesche Wert zu kodieren, muß gewährleistet sein, daß diese Signalleitung tatsächlich nur zwei Werte annehmen kann. Den Wertebereich einer Signalleitung, die durch einen LISP-Term erzeugt wird, kann man man i. allg. nur durch Simulationsläufe bestimmen. Für die Synthese sind derartige Beschreibungen daher kaum geeignet.

Zur Formalisierung regulärer Schaltungen und damit regulärer Signalbündel

gibt es verschiedene Ansätze. Sollen reguläre Schaltungen generisch beschrieben werden, so muß prinzipiell ein Typ verwendet werden, um Signalbündel beliebiger Länge n zu modellieren. Eine Möglichkeit besteht darin, „dependent types" zu verwenden, also Typen, die selbst von einem Term abhängen. Dependent types sind z. B. in dem Theorembeweiser ISABELLE [Paul94] realisiert. Die Verwendung von dependent types bedeutet jedoch einen erhebliche Mehraufwand, da der Nachweis, daß ein Term wohltypisiert ist, explizit erbracht werden muß und nicht wie in HOL mit seinen „simply typed terms" statisch anhand des Termaufbaus untersucht werden kann. Schaltungsentwerfern eine derartige Aufgabe bei der Beschreibung von Schaltungen zuzumuten, ist nicht praktikabel.

Werden reguläre Signalbündel durch rekursiv definierte Datentypen wie Folgen oder Listen repräsentiert, so steht ein Datentyp für Signalbündel beliebiger Länge. Funktionen, bei denen sich diese Länge mit der Zeit verändert, sind als Schaltungsbeschreibungen ungeeignet.

Für kombinatorische Schaltungen muß gewährleistet werden, daß die Anzahl der Ausgangsleitungen höchstens von der Anzahl der Leitungen des Eingangs, nicht aber von den Werten des Eingangs abhängt. Reale Signalleitungsbündel haben eine feste Anzahl von Leitungen. Diese Anzahl hängt nicht von den an den Einzelsignalen anliegenden Werten ab. Die folgende Funktion, die eine Liste auf eine andere Liste abbildet, ist deshalb nicht nicht als Schaltnetz synthetisierbar, es handelt sich um eine Scheinschaltung.

$$\mathsf{f}([x_1, x_2, x_3]) \quad := \quad \begin{cases} [x_3] & \text{für } x_1 = \mathsf{T} \\ [x_1, x_2, x_3] & \text{sonst} \end{cases}$$

Diese Forderung allein ist bei sequentiellen Schaltungen nicht ausreichend. Bei sequentiellen Schaltungen muß ferner gefordert werden, daß sich die Anzahl der Signalleitungen am Ein- und Ausgang eines Speicherbausteins nicht unterscheiden. Außerdem soll auch der Initialwert des Speicherbausteins diese Breite haben. In Abbildung 5.7 werden zur Signalbündelung Listen verwendet. Der Listenoperator APPEND fügt zwei Listen aneinander. Der kombinatorische Anteil in Abbildung 5.7 erfüllt obige Anforderung. Die Länge der Liste am Ausgang von APPEND hängt nur von der Länge der Eingänge ab, nicht jedoch von den Werten der Listenelemente. Im Ganzen ist die Schaltung jedoch nicht synthetisierbar, da sich bei der Speicherkomponente die Anzahl der Eingänge von der Anzahl der Ausgänge unterscheidet.

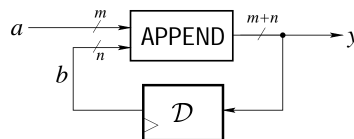


Abbildung 5.7: Unterschiedliche Bitbreiten am Speicherbaustein

Es gibt zwei Möglichkeiten, mit rekursiv definierten Datentypen reguläre Schaltungsstrukturen zu beschreiben. Sie unterscheiden sich darin, wie die Schaltun-

gen skaliert werden: entweder wird die Größe durch die Breiten der Eingänge bestimmt oder, wie in dem in dieser Arbeit vorgestellten Ansatz, durch einen natürlichzahligen Schaltungsparameter. Auf diese beiden Möglichkeiten soll nun näher eingegangen werden.

Steuert man, wie etwa bei APPEND, über die Breite der Eingänge die Dimensionierung der Schaltung und damit dann auch die Breite der Ausgänge, dann ergibt sich für eine Schaltungsstruktur mit diesen Bausteinen eine Menge von Abhängigkeiten zwischen den Breiten aller in der Schaltungsstruktur vorkommenden Signale. So könnte beispielsweise der Wert eines Speicherbausteins der Breite m zunächst durch eine Multiplikation auf ein internes Signal der Länge $2 * m$ überführt worden sein. Anschließend wird das Signal um n boolesche Werte am rechten Ende erweitert, was zur Länge $2 * m + n$ führt, um dann um $(m + n)$ Bit nach links verschoben zu werden, was schließlich zu einem Signal der Länge $(2 * m + n) - (m + n)$ führt. Dieses Signal soll schließlich wieder zurück in den Speicherbaustein gespeichert werden. Dazu muß sichergestellt werden, daß nun die Bitbreite wieder mit der ursprünglichen Bitbreite m übereinstimmt. Es müssen zwei symbolische Ausdrücke, nämlich m und $(2 * m + n) - (m + n)$ miteinander verglichen werden. Der Vergleich symbolischer arithmetischer Ausdrücke ist i. allg. nicht leicht zu automatisieren. Bei komplexeren sequentiellen Schaltungen kann die Sicherstellung der Konsistenz dieser Abhängigkeiten deshalb sehr aufwendig werden.

In dem in Kapitel 4 vorgestellten Formalisierungsschema wurde ein prinzipiell anderer Weg eingeschlagen. Die Skalierung der Schaltung erfolgt ausschließlich über natürlichzahlige Parameter. Durch diese natürlichzahligen Parameter wird nicht nur die Größe der Schaltung festgelegt, sondern sie bestimmt auch die Bitbreiten der Ein- und Ausgänge. Die Signale werden in diesem Ansatz durch Folgen dargestellt. Wenn eine Schaltung an einem Eingang eine Breite von n hat, so heißt dies lediglich, daß die Schaltung nur auf die ersten n Elemente der Folge zugreift, und wenn von einem Ausgang mit der Breite m die Rede ist, so heißt dies lediglich, daß dieser Baustein ein Signal produziert, bei dem alle Folgenwerte ab dem Index m den Wert E haben.

Der wesentliche Vorteile des hier beschriebenen Formalisierungsschemas besteht darin, daß alle Schaltungsbeschreibungen konsistent sind. Es ist nicht erforderlich, zur Konsistenzsicherung die Abhängigkeiten zwischen den Breiten der Signale zu untersuchen. Ein weiterer Vorteil besteht darin, daß alle Teilkomponenten unabhängig voneinander synthetisierbar sind. Werden die Schaltungsparameter einer Komponente instantiiert, so kann dieser Schaltung eine konkrete Gatternetzstruktur zugeordnet werden. Aufbau und Größe einer Komponente hängen nur von den Schaltungsparametern ab, nicht jedoch, wie bei obigem Ansatz, von anderen, vorgeschalteten Komponenten.

Kapitel 6

Schaltungstransformationen auf RT- und Gatterebene

Zu den vorgestellten sprachlichen Konstrukten zur Schaltungsbeschreibung werden in diesem Kapitel Schaltungstransformationen vorgestellt. Die Schaltungstransformationen basieren auf dem Kalkül des Theorembeweisers HOL. Sie bilden die Grundlage für eine Formale Synthese. Ganz bewußt lehnen sich die Transformationen an in der Synthese gebräuchliche Syntheseschritte an, um so eine möglichst einfache Umsetzung bestehender Syntheseverfahren in dieses Konzept zu ermöglichen. Aufgrund der Vielzahl möglicher Schaltungstransformationen, kann nur ein systematischer Überblick mit Beispielen für die jeweiligen Konzepte gegeben werden.

Die Transformationen stellen insofern keine Neuerungen dar, als sie weder neue theoretische Erkenntnisse sind, noch neue Syntheseverfahren. Man halte sich jedoch vor Augen, daß weder konventionell implementierte Syntheseprogramme noch theoretische Erkenntnisse aus Mathematikbüchern in der Lage sind, die Korrektheit heutiger Syntheseprogramme zu garantieren.

Die hier vorgestellten Transformationen schlagen die Brücke zwischen diesen beiden Welten und ermöglichen es so, mathematische Theorie direkt in der Synthese einsetzen zu können. Der Fortschritt besteht darin, daß die hier vorgestellten Transformationen — im Gegensatz zu konventionellen Syntheseverfahren — die Ergebnisse in einer mathematisch exakten und objektiv nachvollziehbaren Weise in einem Theorembeweiser ableiten. Die Qualität dieser Beweise ist durch den Einsatz eines Theorembeweisers eine ganz andere als bei klassischen Beweisen mit Papier und Bleistift (siehe Abschnitt 3.1). Auch ein von Experten durchgeführter Beweis mit Papier und Bleistift kann fehlerhaft sein und sollte gegengelesen werden. Bei Beweisen mit mehreren hundert Gattern und einer in bezug auf die Größe der Gatter exponentiell anwachsenden Anzahl von Fällen sind menschliche Fehler bei der Beweisführung nicht auszuschließen. Bei einem hinreichend langen Beweisen mit mehreren tausend oder Millionen Fallunterscheidungen —

diese Größenordnungen werden im Schaltungsentwurf oft schon bei mittelgroßen Schaltungen erreicht — ist es sogar eher unwahrscheinlich, daß ein klassischer, mit Papier und Bleistift geführter Beweis absolut fehlerfrei ist. Es bedarf eines sicheren Konzeptes, um auch sehr große Beweise objektiv nachvollziehbar und sicher durchzuführen. Hierzu eignen sich, wie bereits in Abschnitt 3.1 erörtert, Theorembeweiser in besonderer Weise.

Bei konventionell implementierten Syntheseverfahren kann angenommen werden, daß sich die Programmierer alle Mühe gegeben haben, die Syntheseprogramme korrekt zu implementieren. Allein die Komplexität der Syntheseprogramme und der hohe Aufwand für die Softwareverifikation machen es in der Praxis unmöglich, durchgängige mathematische Beweise für deren Korrektheit zu führen. Man mag nun argumentieren, daß die Implementierung von Syntheseprogrammen stets Umsetzungen theoretisch fundierter Erkenntnisse seien. Das theoretische Verständnis für eine Schaltungstransformation ist in der Tat eine notwendige Voraussetzung dafür, sie korrekt zu implementieren. Zum Nachweis dafür, daß die i. allg. sehr komplexe Implementierung konventioneller Syntheseprogramme auch tatsächlich korrekt ist, ist es jedoch noch ein langer Weg.

6.1 Klassifikation der Schaltungstransformationen

Die hier angebotenen elementaren Schaltungstransformationen können in zwei Klassen eingeteilt werden: kombinatorische Umformungen und sequentielle Umformungen. Kombinatorische Schaltungstransformationen haben die Eigenschaft, daß sich durch sie die Zustandsrepräsentation nicht ändert. Dieser Gruppe sind Syntheseschritte wie die boolesche Optimierung, die Signalkodierung oder die Technologieabbildung zuzuordnen. Neben diesen rein kombinatorischen Schaltungstransformationen sollen auch sequentielle Schaltungstransformationen vorgestellt werden. Sequentielle Schaltungstransformationen ändern die Zustandsrepräsentation. Beispiele für sequentielle Transformationen sind Zustandsminimierung, Retiming und Reencoding.

Alle in diesem Kapitel vorgestellten elementaren Schaltungstransformationen erhalten die Funktionsweise der Schaltung. Ein Syntheseschritt bildet eine Schaltungsbeschreibung auf eine äquivalente Schaltungsbeschreibung ab. Die Äquivalenz zweier Schaltungen (f_1, q_1) und (f_2, q_2) bedeutet, daß

$$\mathsf{automaton}(f_1, q_1) = \mathsf{automaton}(f_2, q_2)$$

gilt. Diese Äquivalenz von Automaten besagt, daß die beiden Schaltungen nach außen hin das gleiche Verhalten haben, d. h. die gleiche Funktion zwischen der Eingangssignalfolge und der Ausgangssignalfolge herstellen.*

*In der Begriffswelt der Automatentheorie bezeichnet $\mathsf{automaton}(f, q)$ die Leistung des Mealy-Automaten bezüglich des Zustands q. Die Äquivalenz wird, wie auch in dieser Arbeit, als die Gleichheit der Leistungen bezüglich des Initialzustands definiert.

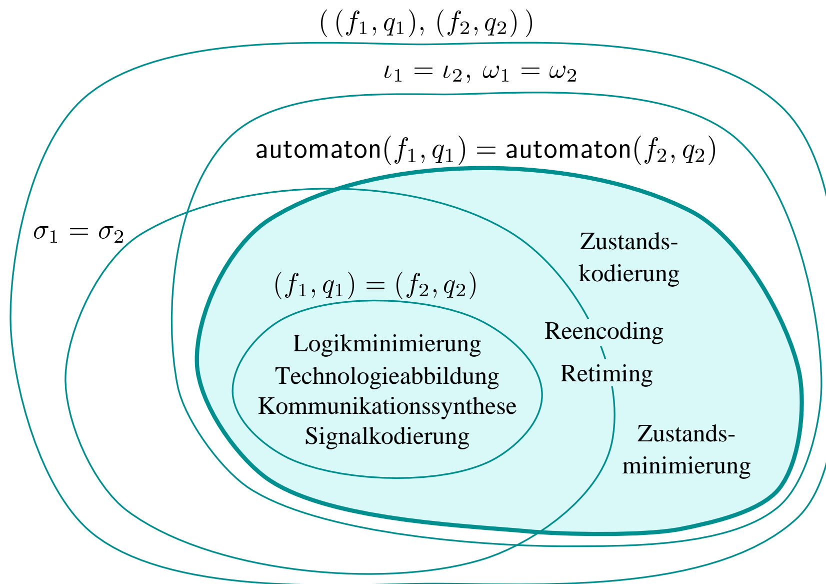


Abbildung 6.1: Äquivalenz von Automaten

In Abbildung 6.1 soll die Äquivalenz von Automaten illustriert werden. Die in Abbildung 6.1 dargestellten Mengen sind Mengen von Paaren von Automaten. Der äußerste Kreis umfaßt die Menge aller Paare von Automaten. Jedes Paar steht für einen Syntheseschritt, wobei der erste Automat den Zustand vor und der zweite Automat den Zustand nach dem Syntheseschritt repräsentiert. Der schattierte Bereich repräsentiert eine Teilmenge, nämlich jene Automatenpaare, die zueinander äquivalent sind ($\mathsf{automaton}(f_1, q_1) = \mathsf{automaton}(f_2, q_2)$). Diese Menge steht für die Menge der korrekten Schaltungstransformationen. Schaltungstransformationen dürfen Schaltungen nur auf äquivalente Schaltungen abbilden. Das Pärchen aus Eingabeschaltung und Ausgabeschaltung muß sich also im schattierten Bereich befinden.

Die Typen ι_1, ω_1 und σ_1 bzw. ι_2, ω_2 und σ_2 stehen für den Eingangstyp, den Ausgangstyp und den Zustandstyp der beiden Schaltungen. Eine notwendige Voraussetzung für die Äquivalenz zweier Schaltungen ist, daß die Typen von Ein- und Ausgangssignal übereinstimmen. Gefordert wird also $\iota_1 = \iota_2$ und $\omega_1 = \omega_2$. Da diese beiden Forderungen unabdingbar für die Äquivalenz von Schaltungen ist, schließt diese Menge die Menge der äquivalenten Schaltungen vollständig ein (siehe Abbildung 6.1).

Daß bei äquivalenten Automaten ι_1 mit ι_2 und ω_1 mit ω_2 übereinstimmen, wird in HOL bereits syntaktisch sichergestellt, denn der Term $\mathsf{automaton}(f_1, q_1) = \mathsf{automaton}(f_2, q_2)$ wäre ansonsten nicht wohltypisiert. Dazu betrachte man die beiden Teilterme $\mathsf{automaton}(f_1, q_1)$ und $\mathsf{automaton}(f_2, q_2)$, die die Typen

$$(\mathsf{num} \rightarrow \iota_1) \rightarrow (\mathsf{num} \rightarrow \omega_1)$$

bzw.

$$(\mathsf{num} \rightarrow \iota_2) \rightarrow (\mathsf{num} \rightarrow \omega_2)$$

haben. Die Äquivalenz = ist in HOL ein Infixoperator vom Typ

$$\alpha \rightarrow \alpha \rightarrow \mathsf{bool}.$$

Aus diesem Grunde ist der Term

$$\mathsf{automaton}(f_1, q_1) = \mathsf{automaton}(f_2, q_2)$$

nur dann syntaktisch zulässig, wenn die Typen ι_1 und ι_2 bzw. ω_1 und ω_2 übereinstimmen. Zwei Schaltungen, die diese Bedingung nicht erfüllen, sind nicht vergleichbar. Eine derartige Äquivalenz kann nicht bewiesen werden, denn sie ist nicht einmal syntaktisch ausdrückbar.

Die Eigenschaft, daß die Zustandstypen σ_1 und σ_2 übereinstimmen, ist keine notwendige Bedingung für die Äquivalenz von Schaltungen. In Abbildung 6.1 erkennt man, daß die Menge der Automatenpaare mit gleichem Zustandstyp ($\sigma_1 = \sigma_2$) den schattierten Bereich mit den Automatenpaaren, die zueinander äquivalent sind, nicht vollständig umfaßt. Man denke hier an Schaltungstransformationen wie eine Zustandskodierung, bei der symbolische Zustände in boolesche Tupel überführt werden. Trotz eines Wechsels des Zustandstyps — in der Eingabeschaltung sind es symbolische Zustände, in der Ausgabeschaltung boolesche Tupel — unterscheidet sich das Verhalten der Schaltungen nach außen nicht. Die Schaltungen sind äquivalent. Verschiedene sequentielle Schaltungstransformationen wie Zustandskodierung, Retiming, Zustandsminimierung etc. verändern i. allg. den Zustandstyp und befinden sich deshalb außerhalb der mit ($\sigma_1 = \sigma_2$) bezeichneten Menge.

Eine interessante Teilmenge unter den äquivalenten Schaltungspaaren ist die Menge, bei der sowohl Aus- und Übergangsfunktion als auch Anfangszustand äquivalent sind. Diese Menge ist in Abbildung 6.1 mit $(f_1, q_1) = (f_2, q_2)$ gekennzeichnet. Die Eigenschaft $(f_1, q_1) = (f_2, q_2)$ impliziert unmittelbar die Äquivalenz von Automaten ($\mathsf{automaton}(f_1, q_1) = \mathsf{automaton}(f_2, q_2)$), weshalb diese Menge in Abbildung 6.1 vollständig innerhalb des schattierten Bereichs liegt. Die Forderung $(f_1, q_1) = (f_2, q_2)$ impliziert ferner, daß die Typen σ_1 und σ_2 übereinstimmen, ansonsten wäre der Ausdruck $(f_1, q_1) = (f_2, q_2)$ syntaktisch nicht zulässig. Aus diesem Grunde liegt diese Menge in Abbildung 6.1 auch vollständig in der Menge $\sigma_1 = \sigma_2$. Beispiele für Äquivalenzumformungen, bei denen $(f_1, q_1) = (f_2, q_2)$ erfüllt ist, sind Syntheseschritte wie Logikminimierung und Technologieabbildung.

6.2 Kombinatorische Schaltungstransformationen

Dieser Abschnitt widmet sich ausschließlich kombinatorischen Schaltungstransformationen, d. h. Äquivalenztransformationen auf DFG-Termen. Zum Teil wurden

für die Schaltungstransformationen vorab in HOL Theoreme bewiesen, die dann zur Durchführung der Transformation während der Synthese nur noch auf den gegebenen Term angewandt werden müssen.

6.2.1 Auftrennung und Überlagerung

Signale mit mehreren Senken werden in diesem Ansatz durch let-Ausdrücke dargestellt. Äquivalent zu einer Schaltung mit einem mehrfach verwendeten Ausgang ist eine Schaltung, bei der die Quellkomponente des Signals dupliziert wird. Dieser Vorgang soll als *Auftrennung* bezeichnet werden. Notwendig wird eine Auftrennung, wenn nachfolgend eine Transformation auf einem Ausdruck bestehend aus der Quellkomponente eines Signals und einer bestimmten Komponente an der Senke dieses Signals durchgeführt werden soll. Abbildung 6.2 zeigt ein Beispiel für eine Auftrennung.

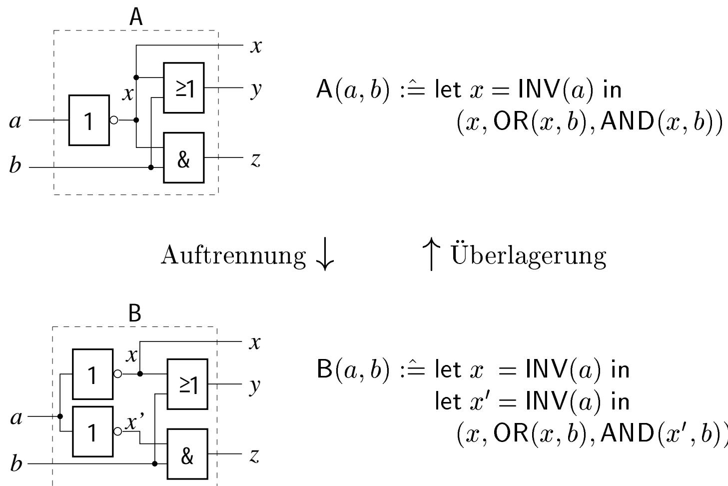


Abbildung 6.2: Auftrennung und Überlagerung

In der Schaltung A von Abbildung 6.2 hat das durch den Inverter produzierte Signal x drei Senken: den Ausgang der Gesamtschaltung und je einen Eingang der Komponenten OR und AND. Es sei beabsichtigt, den Inverter und die Komponente an der dritten Senke, den AND-Baustein, mit der Gleichung

$$\vdash \mathsf{AND}(\mathsf{INV}(a), b) = \mathsf{NOR}(a, \mathsf{INV}(b))$$

umzuformen. Dazu muß zunächst der Inverter dupliziert werden, wobei der erste Inverter sein Ergebnis an die ersten beiden Senken und der zweite Inverter sein Ergebnis an die dritte Senke weiterleitet (Abbildung 6.2, Komponente B). Eine anschließende β-Reduktion wird selektiv auf den zweiten Inverter angewandt. Als Zwischenergebnis erhält man:

$\mathsf{let}\ x = \mathsf{INV}(a)\ \mathsf{in}$
$\quad (x, \mathsf{OR}(x, b), \mathsf{AND}(\mathsf{INV}(a), b))$

Dieses Zwischenergebnis enthält nun endlich den Teilterm $\mathsf{AND}(\mathsf{INV}(a), b)$, sodaß obige boolesche Umformungsgleichung angewandt werden kann. Als Ergebnis der booleschen Umformung erhält man eine Schaltung C gemäß Abbildung 6.3.

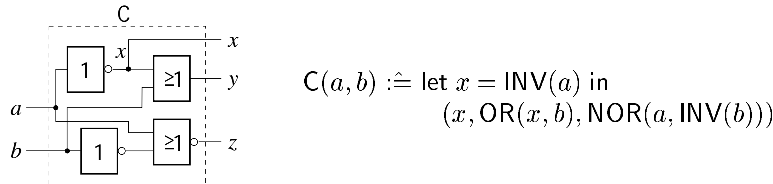


Abbildung 6.3: Ergebnis nach einer booleschen Umformung der Schaltung B aus Abbildung 6.2

Die Auftrennung erfolgt logisch durch β-Reduktion. Im Theorembeweiser HOL ist die β-Reduktion so implementiert, daß stets alle Vorkommen der lokalen Variablen substituiert werden. Der Übergang von Schaltung A zu Schaltung B muß deshalb wie folgt realisiert werden: zunächst wird gemäß den gewünschten Anforderungen der neue Term B konstruiert. Anschließend wird die Äquivalenz bewiesen, indem die beiden Terme durch β-Reduktionen auf den gleichen Term abgebildet werden. Im betrachteten Beispiel müssen dazu in Schaltung A der β-Redex mit der Variablen x und in Schaltung B die β-Redizes mit den Variablen x und x' durch β-Reduktion umgeformt werden. Man kommt so jeweils zu dem gleichen Term, nämlich:

$(\mathsf{INV}(a), \mathsf{OR}(\mathsf{INV}(a), b), \mathsf{AND}(\mathsf{INV}(a), b))$

Der zur Auftrennung entgegengesetzte Vorgang soll als *Überlagerung* bezeichnet werden. Trifft man während der Synthese auf eine Schaltungsstruktur, bei der, wie in Schaltung B, durch mehrere Komponenten das gleiche Ergebnis berechnet wird, so können diese zusammengefaßt werden. Das vordergründige Ziel eines solchen Schrittes ist es, Komponenten einzusparen. Die Umsetzung der Überlagerung durch eine Transformation in HOL erfolgt in der gleichen Weise wie bei der Auftrennung: zunächst wird der Zielterm konstruiert, und anschließend wird das Äquivalenztheorem zwischen altem und neuem Term durch β-Reduktionen abgeleitet.

6.2.2 Expansion und Zusammenfassung

Durch den Entwerfer werden Schaltungen in der Regel hierarchisch entworfen. Viele Synthesewerkzeuge flachen einen so vorgegebenen Entwurf sofort aus. Es kann jedoch durchaus sinnvoll sein, auch während der Synthese mit hierarchischen Schaltungsstrukturen zu arbeiten. Im allgemeinen sind hierarchische Schaltungsbeschreibungen bei weitem kompakter.

Um mit hierarchischen Schaltungsbeschreibungen arbeiten zu können, sind vor allem zwei Schritte notwendig: die Expansion und die Zusammenfassung. Bei der *Expansion* wird eine zusammengesetzte Komponente, die innerhalb einer Struktur verwendet wird, durch ihre Teilkomponenten substituiert. In HOL geschieht dies durch Termersetzung mit der Definitionsgleichung. Der inverse Vorgang dazu soll als *Zusammenfassung* bezeichnet werden. Findet man in einer Schaltung mehrere äquivalente Teilstrukturen, so können diese durch Zusammenfassung jeweils mit dem gleichen Namen abgekürzt werden.

6.2.3 Eliminierung redundanter kombinatorischer Einheiten

Wie bereits in Abschnitt 4.1 erläutert, ist es durchaus sinnvoll, auch kombinatorische Strukturen zuzulassen, innerhalb derer Signale zwar eine Quelle aber keine Senke haben. Produziert eine kombinatorische Schaltung nur Signale, die keine Senken haben, so kann sie eliminiert werden. In dem beschriebenen Formalisierungsansatz können Signale ohne Senken nur durch let-Ausdrücke beschrieben werden. Daß die darin erzeugten Signale, d. h. gebundene Variablen, keine Senke haben, bedeutet, daß sie im Rumpf des Ausdrucks nicht frei vorkommen. In dieser Situation kann die Komponente und mit ihr der betreffende let-Ausdruck durch β-Reduktion zum Verschwinden gebracht werden. Es verbleibt der Rumpf.

Beispiel: In dem Term $\mathsf{let}\ x = \mathsf{INV}(y)\ \mathsf{in}\ \mathsf{AND}(a, b)$ wird das von der Komponente $\mathsf{INV}(y)$ erzeugte Signal x nicht verwendet. Durch β-Reduktion werden alle Vorkommen von x im Rumpf, und davon gibt es keine, durch $\mathsf{INV}(y)$ substituiert. Man gelangt zu $\mathsf{AND}(a, b)$.

6.2.4 Boolesche Umformungen

Der Theorembeweiser HOL enthält bereits zahlreiche Theoreme mit booleschen Gleichungen. Diese können durch Termersetzung in beliebiger Weise auf DFG-Terme angewandt werden. Zusätzlich können für selbstdefinierte, zusammengesetzte boolesche Komponenten in einfacher Weise neue Gleichungen abgeleitet werden.

Eine konzeptionell andere Möglichkeit zur Durchführung einer booleschen Umformung besteht darin, erst das Ergebnis der Umformung zu bestimmen und es anschließend zu verifizieren. Zur Bestimmung des Ergebnisses können unterschiedliche konventionelle Logikminimierungsverfahren eingesetzt werden. Die Verifikation der Äquivalenz zweier boolescher Schaltnetze ist zwar NP-vollständig, es existieren jedoch einigermaßen effiziente Verfahren, die es erlauben, diesen Schritt für hinreichend kleine Schaltungen in akzeptabler Zeit zu verifizieren.

An dieser Stelle ist anzumerken, daß das Konzept der Post-Synthese-Verifikation im allgemeinen nicht angewandt werden soll. Wenn nicht nur zwei boolesche Schaltnetze, sondern zwei Schaltwerke mit unterschiedlicher Zustandsrepräsentation miteinander verglichen werden sollen, gestaltet sich ein automatisierter Verifikationsschritt nämlich sehr viel aufwendiger. Boolesche Schaltnetze

sind die einfachste hier betrachtete Teilmenge von Schaltungen, und nur für sie soll überhaupt eine Post-Synthese-Verifikation in Erwägung gezogen werden.

6.2.5 Kodierungen

Es wurden verschiedene nichtboolesche Datentypen und Operationen eingeführt. Um diese auf boolesche Schaltungen zurückzuführen, müssen die Werte kodiert und die Operationen auf boolesche Verknüpfungen abgebildet werden. Um Schaltungen vor und nach der Kodierung überhaupt einander zuordnen beziehungsweise miteinander vergleichen zu können, muß für die Außenanschlüsse eine Kodierung fest vorgegeben werden.

Im allgemeinen sollen Kodierungen injektiv sein. Für die Formale Synthese ist es, wie sich zeigen wird, nützlich, zu einer Kodierungsfunktion stets auch eine „inverse" Funktion zur Verfügung zu haben. An jeder Stelle der Schaltung kann so eine Serienschaltung aus Kodierungs- und Dekodierungsfunktion eingefügt werden. Da sich diese beiden Funktionen aufheben, ist dieser Schritt verhaltenserhaltend.

Es soll deshalb stets mit Paaren von Funktionen (g, h) bestehend aus einer Kodierungsfunktion g und einer Dekodierungsfunktion h gearbeitet werden, wobei die Eigenschaft $\forall x.\, h(g(x)) = x$ gefordert wird (siehe Abbildung 6.4). Diese Eigenschaft, die sich auf ein Paar von Funktionen bezieht, wird durch das Prädikat $\mathsf{is_encoding}$ abgekürzt:

$$\mathsf{is_encoding}(g, h) \quad := \quad \forall x.\, h(g(x)) = x$$

Es ist anzumerken, daß $\mathsf{is_encoding}(g, h)$ bereits impliziert, daß g injektiv ist. Daß g auch bijektiv ist, wird nicht gefordert. So können beispielsweise die Ampelfarben rot, gelb und grün durch die booleschen Paare (F, F), (F, T) und (T, F) kodiert werden. Diese Funktion ist injektiv. Die Kodierung verwendet (T, T) nicht als Funktionswert, ist also nicht surjektiv und damit auch nicht bijektiv. Trotzdem kann eine Dekodierungsfunktion h angegeben werden, die (F, F), (F, T) und (T, F) zurück auf rot, gelb und grün abbildet und deren Funktionswert für (T, T) eine beliebige Ampelfarbe ist. In diesem Beispiel ist h keine echte Inverse zu g, denn $g(h(\mathsf{T}, \mathsf{T})) = (\mathsf{T}, \mathsf{T})$ gilt nicht. Dies ist für die Kodierung jedoch auch nicht erforderlich — gefordert werden muß lediglich $\forall x.\, h(g(x)) = x$.

Abbildung 6.5 illustriert das prinzipielle Vorgehen bei der Kodierung einer Schaltung. Die dick dargestellten Signalleitungen stehen für die ursprünglichen Signaltypen, die dünn dargestellten Signalleitungen für die kodierten Signaltypen. Kodierungsfunktionen sind als gefüllte Kreise und Dekodierungsfunktionen als ungefüllte Kreise dargestellt. Die Kodierungsfunktionen für die Ausgänge und die Dekodierungsfunktionen für die Eingänge seien vorgegeben. Auf den internen Signalen werden dann Serienschaltungen aus Kodierungs- und Dekodierungsfunktionen eingefügt. Aufgrund der Eigenschaft $\forall x.\, h(g(x)) = x$ verändern diese Einfügeschritte das Verhalten der Schaltung nicht. Indem die Komponenten mit den Kodierungsfunktionen am Ausgang und den Dekodierungsfunktionen am Eingang zusammengefaßt werden, entsteht eine Struktur mit kodierten Signalleitungen. Die Teilkomponenten haben dann nur noch Außenanschlüsse mit kodierten

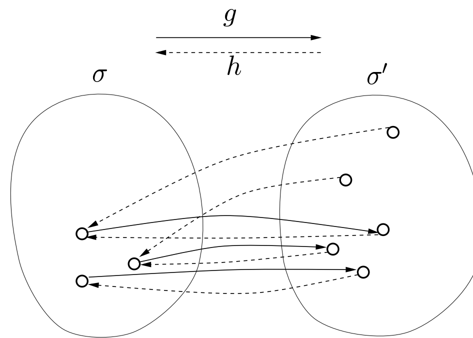


Abbildung 6.4: Kodierung

Signaltypen. Dieser Kodierungsschritt kann bei hierarchischen Schaltungsbeschreibungen in rekursiver Weise auf die Teilkomponenten angewandt werden.

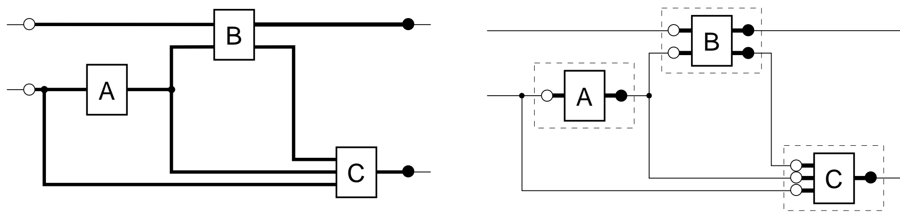


Abbildung 6.5: Kodierung einer kombinatorischen Schaltungstruktur

Aus den eingeführten abstrakten Schaltungsbeschreibungen sollen stets Schaltungsbeschreibungen auf der Gatterebene abgeleitet werden. Besondere Bedeutung bei der Kodierung hat deshalb die Abbildung auf Tupel boolescher Werte. Alle nichtpolymorphen Schaltungen ohne Parameter haben endliche Singaltypen und lassen sich deshalb durch boolesche Tupel kodieren. Voraussetzung für die Ableitung einer Gatterebenenbeschreibung ist, daß alle Typvariablen und alle Parameter instantiiert sind.

Prinzipiell gibt es zwei interessante Möglichkeiten, Paare aus Kodierungs- und Dekodierungsfunktionen zu erzeugen. Zum einen können sie erst aufgestellt und nachträglich durch Fallunterscheidung verifiziert werden. Dieser Ansatz eignet sich nur für Datentypen mit einer kleinen Kardinalität. Ansonsten empfiehlt es sich, die Kodierungen systematisch abzuleiten.

Zu diesem Zweck wurden für die eingeführten Signaldatentypen Kodierungstheoreme beweisen. Die folgenden Definitionen beschreiben Kodierungen, durch die beliebige Typausdrücke bestehend aus one, (α)option und $\alpha + \beta$ durch Tupel boolescher Werte kodiert werden können.

$$\mathsf{one_enc} \quad := \quad ((\lambda x.\, F), (\lambda y.\, \mathsf{one}))$$

$$
\begin{aligned}
\mathsf{option_enc}\; d\; (g,h) \quad &:= \\
&(\\
&\quad (\lambda x.\, \mathsf{CASE_option}(x, (\mathsf{F}, d), \lambda s. (\mathsf{T}, g(s)))), \\
&\quad (\lambda (b,s).\, \mathsf{MUX}(b, \mathsf{any}(h(s)), \mathsf{none})) \\
&) \\
\mathsf{sum_enc}\; d_1\; d_2\; (g_1,h_1)\; (g_2,h_2) \quad &:= \\
&(\\
&\quad (\lambda x.\, \mathsf{CASE_sum}\;(\, x,\; (\lambda a.\, (\mathsf{F}, g_1(a), d_1)), (\lambda b.\, (\mathsf{T}, d_2, g_2(b))))), \\
&\quad (\lambda (b,p,q).\, \mathsf{MUX}(b, \mathsf{INR}(h_2(q)), \mathsf{INL}(h_1(p)))) \\
&)
\end{aligned}
$$

Die Ausdrücke auf der rechten Seite der Definition beschreiben jeweils eine Kodierung in Form eines Pärchens aus Kodierungs- und Dekodierungsfunktion. Durch die beiden Funktionen option_enc und sum_enc werden die Typen $(\alpha)\mathsf{option}$ bzw. $\alpha+\beta$ kodiert, die selbst von Typen abhängen (Typoperatoren). Aus diesem Grunde bauen die durch diese Ausdrücke beschriebenen Kodierungen auf Kodierungen für die Untertypen α und β auf. Die Kodierung der Gesamttypen ist mit den Kodierungen der Untertypen (g,h) bzw. (g_1,h_1) und (g_2,h_2) parametrisiert.

Eine Kodierung mit one_enc, option_enc und sum_enc führt für beliebige zusammengesetzte Typausdrücke aus diesen Typkonstanten und -operatoren one, $(\alpha)\mathsf{option}$ und $\alpha+\beta$ zu einer booleschen Kodierung. Dabei wird i. allg. die Kardinalität des Datentyps erhöht. Es entstehen zusätzliche Werte, die nicht zur Bildmenge der Kodierungsfunktion gehören. Die Parameter d, d_1 und d_2 beschreiben für diese zusätzlichen Werte den Funktionswert der Dekodierungsfunktion. Diese können frei gewählt werden.

Die folgenden Theoreme besagen, daß die oben definierten Kodierungen tatsächlich Kodierungen im Sinne von is_encoding sind. Aus den Theoremen läßt sich auch leicht ablesen, wie Kodierungen für zusammengesetzte Kodierungen gewonnen werden können: zunächst werden die Kodierungen für die Unterausdrücke gewonnen, um daraus dann unter Zuhilfenahme der entsprechenden Implikation via Modus Ponens die Kodierung für den Gesamttyp abzuleiten.

$$
\begin{aligned}
&\vdash \mathsf{is_encoding}(\mathsf{one_enc}) \\
&\vdash \mathsf{is_encoding}(e) \Rightarrow \mathsf{is_encoding}(\mathsf{option_enc}\; d\; e) \\
&\vdash \mathsf{is_encoding}(e_1) \wedge \mathsf{is_encoding}(e_2) \\
&\qquad \Rightarrow \mathsf{is_encoding}(\mathsf{sum_enc}\; d_1\; d_2\; e_1\; e_2)
\end{aligned}
$$

Die drei Theoreme sind hinreichend, um beliebige Typausdrücke mit diesen Typen zu kodieren. Im allgemeinen sind sie jedoch nicht optimal. So würde beispielsweise der Typ $\mathsf{one}+\mathsf{one}$ durch $\mathsf{bool}\times\mathsf{bool}\times\mathsf{bool}$ statt durch bool kodiert werden. Aus diesem Grunde wurden einige Umkodierungen bewiesen. Der Typ $\alpha+\alpha$ läßt sich so beispielsweise in den Typ $\alpha\times\mathsf{bool}$ überführen und $\alpha\times\mathsf{one}$ in α. Durch Anwendung dieser beiden Kodierungen gelangt man von $\mathsf{one}+\mathsf{one}$ nach bool.

Felder $(\alpha)\mathsf{array}_n$ können direkt durch n-Tupel boolescher Werte kodiert werden. Kodierungen müssen dabei für jedes n separat definiert und das Kodierungs-

theorem abgeleitet werden. Dies gilt in gleicher Weise für Aufzählungsdatentypen enum_n.

6.2.6 Strukturelle Umformungen auf regulären Strukturen

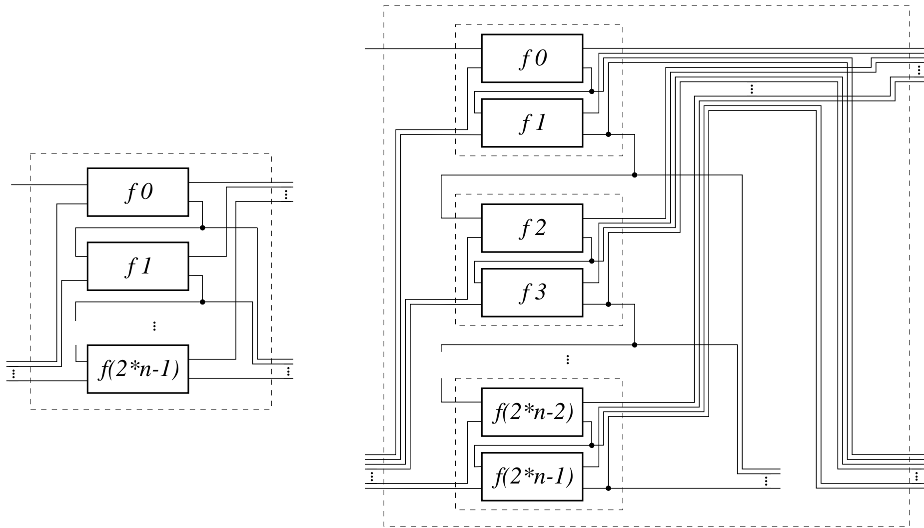
Reguläre Schaltungsbeschreibungen werden von zahlreichen Syntheseprogrammen unterstützt. Oft wird die Schaltungsbeschreibung jedoch bei der Konvertierung in das interne Datenformat ausgeflacht und in eine Schaltungsstruktur mit einer i. allg. großen Anzahl von Teilkomponenten überführt. Dies ist zum Beispiel immer dann zwingend erforderlich, wenn ein Syntheseergebnis in einer Schaltungssprache dargestellt werden soll, in denen Regularität nicht explizit beschrieben werden kann (BLIF, KISS etc.). Dieser Schritt ist aus zweierlei Gründen ungünstig. Zum einen geht dabei die Information über die explizit beschriebene Regularität verloren und zum anderen sind reguläre Schaltungsbeschreibungen wie etwa Schaltungsbeschreibungen mit ripple bei weitem kompakter als die entsprechenden ausgeflachten Strukturen. Die Kompaktheit der Schaltungsdarstellung wirkt sich während der Synthese nicht nur in einem geringeren Speicherbedarf aus, sondern gestattet auch effizientere Schaltungstransformationen und Kostenbewertungsfunktionen. Der hier vorgestellte Ansatz unterstützt gezielt ein solches Vorgehen.

Die Hardwarekosten für abgeleitete reguläre Strukturen (Abbildungen 4.7, 4.8) lassen sich direkt aus den Kosten der Elementarkomponenten und den natürlichzahligen Parametern zur Dimensionierung ableiten. So ergeben sich beispielsweise bei der Parallelschaltung $\mathsf{par}\, n\, (\mathsf{constantly}(f))$ die Hardwarekosten als das n-fache der Hardwarekosten von f und die kombinatorische Tiefe der Gesamtschaltung als die kombinatorische Tiefe von f. Analog können auch zu den anderen regulären Beschreibungen die Kosten der Gesamtschaltung auf die Elementarkomponenten zurückgeführt werden.

Abbildung 6.6 zeigt ein Beispiel für eine Transformation auf einer regulären Struktur. Durch die Transformation werden in der regulären Struktur ripple benachbarte Teilkomponenten paarweise zusammengefaßt. Diesem Schritt kann eine Optimierung der neuen Teilkomponenten, die sich aus zwei alten Teilkomponenten zusammensetzen, nachgeschaltet werden. Der Optimierungsschritt muß bei dieser regulären Beschreibungsform nur einmal angewandt werden. Bei den üblichen konventionellen Syntheseprogrammen würde zunächst die reguläre Struktur ausgeflacht werden, und anschließend müßte man den Optimierungsschritt n mal ausführen.

Durch den in Abbildung 6.6 dargestellten Schritt werden je zwei benachbarte Komponenten gruppiert. Wendet man diesen Schritt mehrfach hintereinander an, so entstehen Gruppierungen mit 4, 8, 16,... benachbarten Komponenten.

Durch Optimierungen in den zusammengefaßten Grundkomponenten kann versucht werden, die Gesamtkosten zu optimieren. Handelt es sich bei den Grundkomponenten um rein boolesche Komponenten, so kann dies z. B. durch eine zweistufige Minimierung erfolgen. Bei der zweistufigen Logikminimierung wird eine Schaltung, die auch mehrstufig sein darf, in eine zweistufige Form gebracht, und es wird dabei



$$
\begin{array}{l}
\vdash\ \mathsf{ripple}\ (2*n)\ f\ (c,x) \\
\quad = \\
\quad \mathsf{let}\ (y,z)\ = \\
\qquad \mathsf{ripple}\ n \\
\qquad\quad (\lambda i.\ \lambda(r,(s,t)). \\
\qquad\qquad \mathsf{let}\ (s',r')\ =\ f\ (2*i)\ (r,s)\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ (t',r'')\ =\ f\ (2*i+1)\ (r',t)\ \mathsf{in} \\
\qquad\qquad\quad (((s',t'),(s,r'')),r'') \\
\qquad\quad) \\
\qquad\quad (\,c,\ \mathsf{shrink}\ n\ x\,) \\
\quad \mathsf{in} \\
\qquad (\ \mathsf{unshrink}\ n\ (\mathsf{splitfst}\ n\ y),\ \mathsf{unshrink}\ n\ (\mathsf{splitsnd}\ n\ y)\ )
\end{array}
$$

Abbildung 6.6: Transformation auf einer ripple-Struktur

versucht, mit einem minimalen Aufwand an Gattern auszukommen. Mehrstufige Schaltungen sind i. allg. langsamer als zweistufige Schaltungen. Bezüglich des Hardwareaufwands kann es jedoch sein, daß ein minimaler Hardwareaufwand nur mit Schaltungen zu erzielen ist, die eine größere kombinatorische Tiefe haben und daß deshalb die zweistufige Minimierung einer mehrstufigen Schaltung zu einem erhöhten Hardwareaufwand führt.

Durch abwechselndes Anwenden der Schaltungstransformation und Optimierung der Teilkomponenten (z.B. durch zweistufige Logikminimierung) kann versucht werden, die kombinatorische Tiefe zu reduzieren. Wie oft die Schaltungstransformation in Abbildung 6.6 wiederholt werden soll, wieviele benachbarte Teilkomponenten also gruppiert und anschließend optimiert werden sollen, ist eine Abwägung zwischen Hardwareaufwand einerseits und kombinatorischer Tiefe andererseits.

Die Transformation in Abbildung 6.6 stellt einen sehr allgemeinen Mechanismus dar, da alle regulären Strukturen aus dem Konstrukt ripple abgeleitet wurden und diese Transformation somit zur Zusammenfassung benachbarter Komponenten in verschiedenartigsten Strukturen verwendet werden kann. Um bei abgeleiteten regulären Strukturen wie rippleb, ser, par etc. diese Transformation anwenden zu können, müssen lediglich die Definitionsgleichungen (Abbildung 4.7) angewandt werden. Man erhält eine Struktur, bei der Regularität nur noch durch ripple-Ausdrücke beschrieben wird und kann auf diese das Theorem in Abbildung 6.6 anwenden.

Das nachfolgende Theorem beschreibt eine Umformung auf einer zweidimensionalen Struktur. Eine m-fach-Parallelschaltung einer n-fach-Serienschaltung wird in eine n-fach-Serienschaltung einer m-fach-Parallelschaltung überführt. Theoreme wie das in Abbildung 6.6 dargestellte beziehen sich auf eine eindimensionale Struktur. Mit Hilfe des nachfolgenden Theorems wird es möglich, diese eindimensionalen Transformationen auf die einzelnen Dimensionen (horizontal, vertikal) der Gesamtstruktur anzuwenden.

$$\vdash \ \mathsf{par}\, m\, (\lambda i.\, \mathsf{ser}\, n\, (\lambda j.\, f\ i\ j))\ x \ = \ \mathsf{cut}\, m\, (\mathsf{ser}\, n\, (\lambda j.\, \mathsf{par}\, m\, (\lambda i.\, f\ i\ j))\ x)$$

Die oben definierten regulären Strukturen bilden eine solide logische Grundlage zur expliziten Beschreibung von Regularität. Für spezielle Werte der Skalierungsparameter können aus ihnen konkrete Netzlistenstrukturen abgeleitet werden. Beispielsweise kann aus einem n-Bit-Ripple-Carry-Addierer eine konkrete Netzlistenstruktur für $n = 32$ abgeleitet werden. In einer solchen Netzlistenstruktur ist die Information darüber, daß die Schaltung regulär ist, nur noch implizit enthalten. Wendet man auf dieser Struktur weitere Umformung an, so geht die Regularität i. allg. verloren. Führt man beispielsweise in der Netzstruktur des 32-Bit-Addierers im vierten Volladdierer eine boolesche Umformung durch, so ist die Regularität der Struktur zerstört. Eine Netzlistenstruktur aus einer regulären Beschreibung abzuleiten und danach auf dieser weitere Umformungen durchzuführen, führt i. allg. dazu, daß zu einer expliziten regulären Beschreibung nicht mehr zurückgefunden werden kann.

Schaltungstransformationen auf regulären Strukturen, in denen die Regularität explizit beschrieben ist, sind offensichtlich kompakter als entsprechende Netzlistenstrukturen. Dies führt in Syntheseprogrammen nicht nur zu speichereffizienteren Darstellungen der Schaltungsstruktur, sondern auch zu schnelleren Syntheseprogrammen.

6.2.7 Arithmetisch motivierte Umformungen

Eine besondere Klasse von Schaltungen sind arithmetische Einheiten. Natürliche Zahlen in einem Intervall $[0, 2^n - 1]$ werden durch Felder aus booleschen Werten modelliert. Den numerischen Einheiten, die auf den Feldern mit booleschen Werten arbeiten, „entsprechen" arithmetische Operationen auf natürlichen Zahlen. Diese „Entsprechung" soll nun für die Synthese ausgenutzt werden. Statt neue Theoreme zur Umformung von Strukturen mit arithmetischen Einheiten zu beweisen, sollen diese aus entsprechenden Theoremen für natürliche Zahlen abgeleitet werden. Der Theorembeweiser HOL stellt für natürliche Zahlen bereits eine große Anzahl an Theoremen zur Verfügung.

Die Funktion, die einen Zusammenhang zwischen Feldern und natürlichen Zahlen herstellt, heißt bools_to_num. Sie interpretiert ein Feld x als eine Binärzahl wobei $x(0)$ das Bit mit der niedrigsten und $x(n-1)$ das Bit mit der höchsten Wertigkeit ist. Bevor die Funktion bools_to_num eingeführt wird, werden zunächst die beiden Hilfsfunktionen bool_to_num und sum definiert. Die Funktion bool_to_num ordnet einem einzelnen booleschen Wert eine natürliche Zahl zu:

$$\begin{array}{rcl} \mathsf{bool_to_num}\ \mathsf{F} & := & 0 \\ \mathsf{bool_to_num}\ \mathsf{T} & := & 1 \end{array}$$

Die Funktion sum berechnet zu einer Folge x mit den Folgenelementen $x(0), x(1), \ldots$ die Summe $\sum_{i=0}^{n-1} x(i)$. Sie ist wie folgt definiert:

$$\begin{array}{rcl} \mathsf{sum}\ 0\ x & := & 0 \\ \mathsf{sum}\ (n+1)\ x & := & x(n) + (\mathsf{sum}\ n\ x) \end{array}$$

Die Funktion bools_to_num wird nun definiert als die Summe der mit dem Faktor 2^r gewichteten Folgenwerte $x(r)$:

$$\mathsf{bools_to_num}\ n\ x \quad := \quad \mathsf{sum}\ n\ (\lambda r.\ \mathsf{bool_to_num}(x(r)) * 2^r)$$

Zur Funktion bools_to_num wird nun eine inverse Funktion num_to_bools definiert. num_to_bools bildet eine natürliche Zahl a auf eine Folge boolescher Werte ab, wobei das i-te Folgenelement den Wert $\mathsf{ODD}(a\ \mathsf{DIV}\ 2^i)$ bekommt. ODD bestimmt dabei, ob der Ausdruck $(a\ \mathsf{DIV}\ 2^i)$ ungerade ist. Ist der Ausdruck ungerade, dann ist ihr Funktionswert T und ansonsten F. Die Funktion for sorgt dafür, daß alle Folgenwerte ab dem Index n den Wert E annehmen.

$$\mathsf{num_to_bools}\ n\ a \quad := \quad \mathsf{for}\ n\ (\lambda i.\ \mathsf{ODD}(a\ \mathsf{DIV}\ 2^i))$$

Die nachfolgenden Definitionen zeigen eine Auswahl von Schaltungsstrukturen, die arithmetische Operationen realisieren. Zunächst werden die beiden Konstanten bools_zero und bools_one definiert. Beide Konstanten sind Felder einer Länge n und entsprechen den natürlichzahligen Werten 0 und 1. Die Konstante bools_zero beschreibt ein n Bit breites Signal, dessen Einzelsignale alle den Wert F haben. Die Konstante bools_one wird aus bools_zero abgeleitet, indem das Bit an der Position 0 modifiziert wird: es erhält den Wert T. Die Funktion bool_fulladder beschreibt einen 1-Bit-Volladdierer — ein einfaches Schaltnetz. Die Schaltung bools_add realisiert einen n-Bit-Volladdierer. bools_add ist eine reguläre Struktur, die durch Aneinanderreihung von n 1-Bit-Volladdierern entsteht. Die Schaltung bools_less steht für die <-Operation. Sie ordnet zwei Feldern boolescher Werte einen booleschen Wert zu, der aussagt, ob das erste Feld, als natürliche Zahl betrachtet, kleiner als das zweite Feld ist. Die Schaltung bools_less entsteht analog zu bools_add durch Aneinanderreihung der entsprechenden 1-Bit-Version (hier: bool_less). Die Funktion bools_move ist eine Hilfsfunktion für den Multiplizierer bools_mult. Sie bildet ein Signal a der Länge m auf ein Signal der Länge $k + m + n$ ab, wobei links k und rechts n Werte F angefügt werden (bools_zero k und bools_zero n). Der Multiplizierer bools_mult multipliziert zwei Zahlen a und b mit den Längen m und n. Das Ergebnis hat die Länge $m+n$. Die Multiplikation erfolgt nach der klassischen Schulmethode. Zunächst werden Teilprodukte berechnet. Für jedes $i = 0 \ldots (n-1)$ wird der Faktor a mit Hilfe von bools_move um i Stellen nach links verschoben und anschließend mit der i-ten Stelle von b multipliziert. Die Berechnung der Teilprodukte gestaltet sich aufgrund der binären Darstellung sehr einfach: für $b(i) = \mathsf{T}$ ist das Produkt gerade der andere Faktor und für $b(i) = \mathsf{F}$ ist das Produkt null (bools_zero). Anschließend werden diese Teilprodukte aufsummiert.

$$
\begin{array}{rcl}
\mathsf{bools_zero}\; n & :\hat{=} & \mathsf{spread}\; n\; \mathsf{F} \\
\mathsf{bools_one}\; n & :\hat{=} & \mathsf{modify}\; n\; (\mathsf{enum}\; n\; 0, \mathsf{T}, \mathsf{bools_zero}\; n) \\
\mathsf{bool_fulladder}(c,(a,b)) & :\hat{=} & (\\
 & & \quad \mathsf{EQ}(\mathsf{EQ}(a,b),c), \\
 & & \quad \mathsf{OR}(\mathsf{AND}(c,a), \mathsf{OR}(\mathsf{AND}(c,b), \mathsf{AND}(a,b))) \\
 & &) \\
\mathsf{bools_add}\; n\; (a,b) & :\hat{=} & \mathsf{FST}(\mathsf{rippleb}\; n\; (\mathsf{constantly}\; \mathsf{bool_fulladder}) \\
 & & \quad (\mathsf{F}, \mathsf{comb}\; n\; (a,b))\;\;) \\
\mathsf{bool_less}(c,(a,b)) & :\hat{=} & \mathsf{OR}(\mathsf{AND}(\mathsf{INV}(a),b), \mathsf{AND}(c, \mathsf{EQ}(a,b))) \\
\mathsf{bools_less}\; n\; (a,b) & :\hat{=} & \mathsf{ripplec}\; n \\
 & & \quad (\mathsf{constantly}\; \mathsf{bool_less})\; (\mathsf{F}, \mathsf{comb}\; n\; (a,b)) \\
\mathsf{bools_move}\; k\; m\; n\; a & :\hat{=} & \mathsf{append}\; (k+m)\; n \\
 & & \quad (\mathsf{append}\; k\; m\; (\mathsf{bools_zero}\; k, a), \\
 & & \quad\quad \mathsf{bools_zero}\; n\;\;)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{bools_mult}\; m\; n\; (a,b) & :\hat{=} & \mathsf{rippled}\; n\; (\mathsf{constantly}(\mathsf{bools_add}(m+n))) \\
 & & \quad (\\
 & & \quad \mathsf{bools_zero}(m+n), \\
 & & \quad \mathsf{par}\; n \\
 & & \qquad (\lambda i\, y.\, \mathsf{MUX}(y, \mathsf{bools_move}\; i\; m\; (n-i)\; a, \\
 & & \qquad\quad \mathsf{bools_zero}\; (m+n))\ \ )\; b \\
 & & \quad)
\end{array}
$$

Die obigen Schaltungsstrukturen wurden zwar mit der Absicht konstruiert, daß sie arithmetische Operationen durchführen, bewiesen ist dies jedoch noch nicht. Es wurden deshalb die Theoreme (6.1) bis (6.5) bewiesen, die mit Hilfe der Funktionen bools_to_num und num_to_bools die Brücke zu den entsprechenden natürlichzahligen Konstanten und Funktionen 0, 1, $+$, $<$ und $*$ schlagen. Die ersten beiden Theoreme (6.1) und (6.2) beschreiben, daß bools_zero und bools_one genau der 0 bzw. der 1 in den natürlichen Zahlen entsprechen. Das Theorem (6.3) besagt, daß der Addierer bools_add eine Addition modulo 2^n durchführt. Dies wird in Theorem (6.3) wie folgt ausgedrückt: konvertiert man die beiden Summanden des Addierers a und b in natürliche Zahlen, addiert diese, bildet dann den modulo-2^n-Wert und konvertiert das Ergebnis wieder zurück in ein Feld boolescher Werte, so erhält man den Wert, der genau dem Ausgangssignal des Addierers entspricht. Nach dem gleichen Schema besagt Theorem (6.4), daß die Schaltung bools_less den booleschen Wert berechnet, der entsteht, wenn man zunächst die beiden Operanden a und b in natürliche Zahlen konvertiert und sie danach mit der natürlichzahligen $<$-Operation vergleicht. Das Theorem zur Multiplikationseinheit (6.5) hat das gleiche Schema wie das Theorem zum Addierer (6.5). Es besagt, daß die Multiplikationseinheit bei einer entsprechenden Konvertierung der Ein- und Ausgangswerte tatsächlich bezüglich den natürlichen Zahlen eine Multiplikation durchführt.

$$\vdash\ \mathsf{bools_zero}\; n = \mathsf{num_to_bools}\; n\; 0 \tag{6.1}$$

$$\vdash\ \mathsf{bools_one}\; n = \mathsf{num_to_bools}\; n\; 1 \tag{6.2}$$

$$\begin{array}{l}\vdash\ \mathsf{bools_add}\; n\; (a,b) = \\ \quad \mathsf{num_to_bools}\; n \\ \qquad (((\mathsf{bools_to_num}\; n\; a) + (\mathsf{bools_to_num}\; n\; b))\ \mathsf{MOD}\ 2^n)\end{array} \tag{6.3}$$

$$\vdash\ \mathsf{bools_less}\; n\; (a,b) = (\mathsf{bools_to_num}\; n\; a) < (\mathsf{bools_to_num}\; n\; b) \tag{6.4}$$

$$\begin{array}{l}\vdash\ \mathsf{bools_mult}\; m\; n\; (a,b) = \\ \quad \mathsf{num_to_bools}\; (m+n) \\ \qquad ((\mathsf{bools_to_num}\; m\; a) * (\mathsf{bools_to_num}\; n\; b))\end{array} \tag{6.5}$$

Dadurch, daß diese Theoreme bewiesen wurden, gewinnt man für die Schaltungen die Sicherheit, daß sie tatsächlich arithmetische Operationen ausführen. Diese Theoreme können jedoch auch für die Formale Synthese verwendet werden, um natürlichzahlige Umformungen zur Schaltungstransformation zu verwenden — und dies ist der eigentliche Grund, weshalb sie bewiesen wurden.

Wie nun diese Theoreme dazu verwendet werden können, um Umformungen auf Schaltungsstrukturen auf natürlichzahlige Umformungen zurückzuführen, soll an einem kleinen Beispiel erläutert werden. Es sei beabsichtigt, auf einen Addierer bools_add das Kommutativgesetz anzuwenden, es soll also ein Term der Form bools_add $n\ (a, b)$ in einen Term der Form bools_add $n\ (b, a)$ überführt werden. Ein derartiges Theorem steht bisher nicht zur Verfügung. Normalerweise muß dazu über die Struktur, den Aufbau der Schaltung ein Beweis geführt werden. Dadurch, daß jedoch das Theorem (6.3) zur Verfügung steht und man auch bereits das Kommutativgesetz für natürliche Zahlen in HOL zur Verfügung hat, geht dies bedeutend einfacher:

bools_add $n\ (a, b)$

= *Theorem (6.3)*

num_to_bools n
  $(((\text{bools_to_num}\ n\ a) + (\text{bools_to_num}\ n\ b))\ \text{MOD}\ 2^n)$

= *Kommutativgesetz für natürlichen Zahlen*

num_to_bools n
  $(((\text{bools_to_num}\ n\ b) + (\text{bools_to_num}\ n\ a))\ \text{MOD}\ 2^n)$

= *Theorem (6.3) in entgegengesetzter Richtung*

bools_add $n\ (b, a)$

In analoger Weise können mit Hilfe der Theoreme (6.1) bis (6.5) auch für die anderen arithmetischen Schaltungen Schaltungstransformationen auf die entsprechenden natürlichzahligen Umformungen zurückgeführt werden.

6.2.8 Schaltnetzimplementierungen als ROM

Gewöhnlich werden Schaltnetze durch eine Struktur elementarer Gatter wie NAND und NOR realisiert. Ein ganz anderer Weg besteht darin, die Schaltnetzfunktion in einem ROM-Baustein abzulegen. Die Grundlage hierfür bildet der Baustein MUXT, ein 2^n:1-Multiplexer. Der Baustein MUXT wurde bereits in Abbildung 4.7 definiert. Er setzt sich gemäß Abbildung 6.7 aus mehreren polymorphen Multiplexern zusammen. Die oberen n Eingänge vom Typ bool dienen zur Auswahl eines der 2^n Dateneingänge. Alle Dateneingänge sowie der Datenausgang haben wie beim 2:1 Multiplexer MUX den variablen Typ α. Damit kann MUXT sowohl in bezug auf die Größe (Parameter n) als auch in bezug auf den Typ der Datenleitungen (Typvariable α) beliebig instantiiert werden.

Es soll nun mathematisch beschrieben werden, wie der 2^n:1-Multiplexer MUXT mit Hilfe der Steuereingänge einen Datenausgang auswählt. Man betrachte dazu

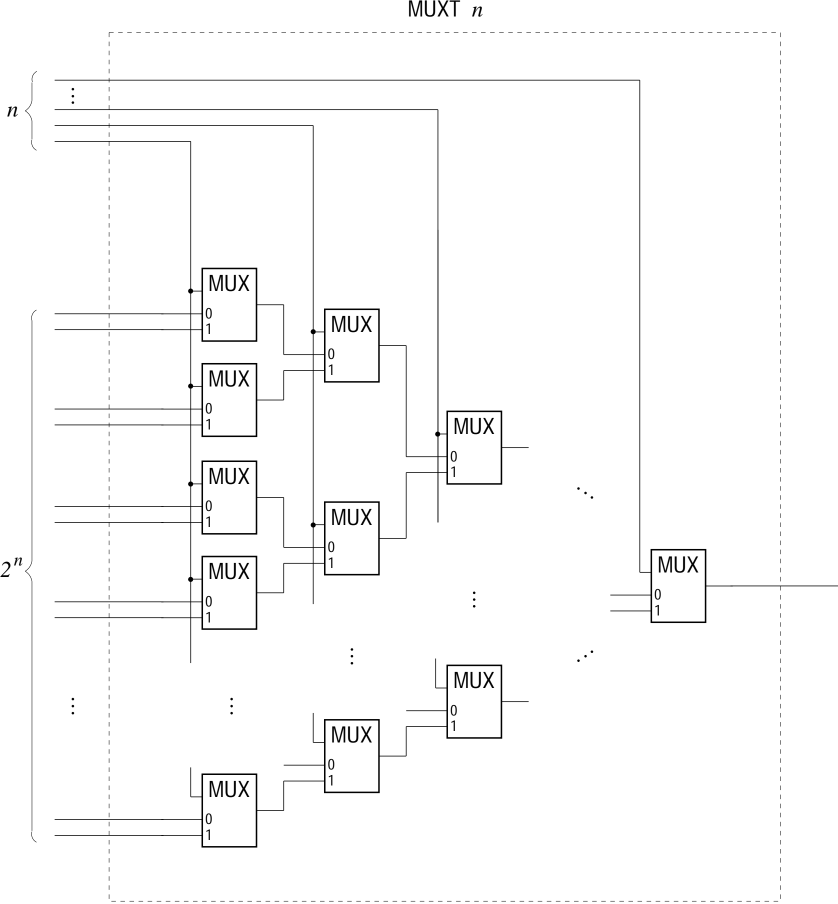


Abbildung 6.7: Der 2^n:1-Multiplexer MUXT

den Ausdruck $\mathsf{MUXT}\,n\ (a,b)$. Dabei sind a und b Felder. Die Elemente von a sind boolesch, die Elemente von b haben den variablen Typ α. Eine der Leitungen des Feldes b wird durch den MUXT-Baustein an den Ausgang geschaltet. Den Index dieser Leitung erhält man, wenn man den Steuereingang a als natürliche Zahl interpretiert. Genau diese Aussage macht das nachfolgende Theorem:

$$\vdash\ \mathsf{MUXT}\,n\ (a,b)\ =\ b(\mathsf{bools_to_num}\ n\ a)$$

Für den Fall, daß b konstant ist, beschreibt MUXT einen ROM-Baustein, wobei sich die gespeicherte Information in b befindet. Mit ROM-Bausteinen lassen sich beliebige Schaltnetze realisieren. Man muß dazu nur die Funktionswerte des Schaltnetzes an der entsprechenden Adresse im ROM-Baustein ablegen. Genau darum geht es in Theorem (6.6). Gegeben sei eine Funktion f vom Typ $(\mathsf{bool})\mathsf{array}_n \to \alpha$. Von der Eingangsbelegung wird also gefordert, daß sie ein boolesches Feld einer beliebigen aber festen Länge n sei. Der Ausgang kann einen beliebigen Typ haben. An den Dateneingang b werden die Funktionswerte von f angelegt und zwar so, daß am Eingang $b(i)$ der Funktionswert $f(\mathsf{num_to_bools}\,n\,i)$ angeschlossen wird. In Theorem (6.6) beschreibt der Ausdruck

$$\mathsf{for}\ 2^n\ (\lambda i.\ f(\mathsf{num_to_bools}\ n\ i))$$

ein Feld der Länge 2^n mit den Funktionswerten $f(\mathsf{num_to_bools}\,n\,i)$, $i = 0 \ldots 2^n - 1$. Die Einträge des Feldes sind jeweils konstant und können durch Auswertung (siehe Abschnitt 4.4) bestimmt werden. Dieses Feld wird am Dateneingang angelegt, um f als ROM-Baustein zu implementieren.

In Theorem (6.6) wird nicht direkt die Funktion $f(a)$, sondern $f(\mathsf{cut}\,n\,a)$ implementiert. Dazu beachte man, daß das Feld a in HOL durch Folgen, also einen unendlichen Datentyp, realisiert wird. Eine ROM-Implementierung ist jedoch nur für eine endliche Anzahl von Eingängen möglich. Der Eingangswert von f muß auf Felder der Länge n beschränkt werden. Dazu wird zunächst die Funktion cut auf die Eingabe angewandt.

$$\vdash\ f(\mathsf{cut}\ n\ a)\ =\ \mathsf{MUXT}\ n\ (a, \mathsf{for}\ 2^n\ (\lambda i.\ f(\mathsf{num_to_bools}\ n\ i))) \qquad (6.6)$$

6.3 Sequentielle Schaltungstransformationen

Zur Transformation sequentieller Schaltungen wurden mehrere Automatengleichungen bewiesen. Mit ihnen lassen sich elementare Syntheseschritte wie Zustandskodierung und Retiming durchführen.

Die im vorangegangen Abschnitt vorgestellten kombinatorischen Schaltungstransformationen bezogen sich auf ganz bestimmte kombinatorische Operationen mit ganz bestimmten Datentypen. Für zahlreiche Syntheseverfahren wie etwa bei der Logikminimierung auf Gatterebene kann es sinnvoll sein, diese Menge z. B. auf rein boolesche Grundoperationen einzuschränken.

Es gibt sehr viele verschiedene Möglichkeiten für sequentielle Schaltungstransformationen. Insbesondere entstehen durch die Kombination von sequentiellen Schaltungstransformationen mit kombinatorischen Schaltungstransformationen neue sequentielle Schaltungstransformationen. In diesem Kapitel sollen „reine" sequentielle Schaltungstransformationen vorgestellt werden. Sie sollen keine bestimmten kombinatorischen Grundoperationen enthalten und sind somit mit jeder Teilmenge an Grundoperationen verträglich. Alle kombinatorischen Anteile einer Schaltung sind durch variable Funktionen beschrieben, deren Ein- und Ausgangssignale variable Typen (Polymorphie) haben.

Betreibt man beispielsweise Schaltungssynthese allein auf Gatterebene, so muß man sich bei den kombinatorischen Grundoperationen auf rein boolesche Grundoperationen beschränken. Die hier betrachteten sequentiellen Schaltungstransformationen sind mit dieser Anforderung verträglich. Sie setzen weder bei der Eingabeschaltung voraus, daß sie bestimmte nichtboolesche Grundoperationen enthalten muß, noch produzieren sie ein Ergebnis, das bestimmte nichtboolesche Operationen enthält, wenn diese nicht schon in der Eingabeschaltungsbeschreibung enthalten sind.

6.3.1 Allgemeine Definitionen

In den nachfolgenden Abschnitten werden die drei Theoreme (6.7), (6.8) und (6.10) beschrieben, mit denen die Zustandsrepräsentation einer Schaltung geändert werden kann: die Zustandskodierung, die Eliminierung unerreichbarer Zustände und die Klassifikation äquivalenter Zustände.

Die Zustandskodierung ist bei der RT-Synthese unabdingbar. Will man auf höheren Abstraktionsebenen auch Schaltungsbeschreibungen mit nichtbooleschen Signalen zulassen, so müssen diese durch die Synthese in rein boolesche Schaltungen überführt werden. Dazu müssen nicht nur, wie in Abschnitt 6.2.5 beschrieben, die kombinatorischen Einheiten kodiert werden, sondern auch die Zustände in den Speichereinheiten. Das Theorem (6.7) deckt diese Aufgabenstellung ab.

Die Theoreme (6.8) und (6.10) dienen zur Reduktion der Anzahl der Zustände. Das Theorem (6.8) wird zur Eliminierung unerreichbarer Zustände verwandt, das Theorem (6.10) zur Zusammenfassung äquivalenter Zustände. Wie noch genauer ausgeführt werden soll, sind diese beiden Theoreme hinreichend, um zu jeder Schaltung eine äquivalente Schaltung mit einer minimalen Anzahl von Zuständen abzuleiten.

Alle drei Theoreme (6.7), (6.8) und (6.10) haben eine Gemeinsamkeit: sie verändern eine Schaltung $\mathsf{automaton}(f, q)$ dadurch, daß sie in die zum Speicherbaustein hin und vom Speicherbaustein weg führenden Signalleitungen eine Kodierungsfunktion g bzw. eine Dekodierungsfunktion h einfügen. Gleichzeitig wird der Initialzustand q in den neuen Initialzustand $g(q)$ überführt. Abbildung 6.8 beschreibt diesen Vorgang.

In diesem Schritt entsteht eine neue kombinatorische Einheit bestehend aus bisheriger kombinatorischer Einheit f, Kodierungsfunktionen g und Dekodierungs-

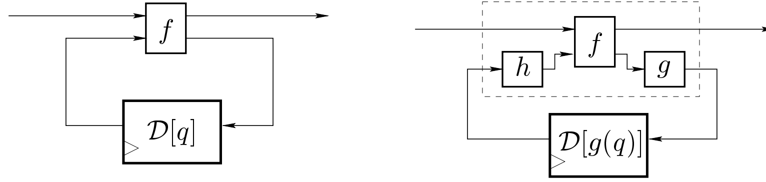
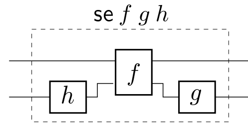


Abbildung 6.8: Allgemeine Zustandskodierung

funktion h. Da diese kombinatorische Struktur aus f, g und h im weiteren verwendet wird, soll sie gemäß Abbildung 6.9 mit $\mathsf{se}\ f\ g\ h$ abgekürzt werden.



$$\begin{aligned} \mathsf{se}\ f\ g\ h\,(i,s) :\hat{=}\ & \mathsf{let}\ a\ =\ h(s)\ \mathsf{in} \\ & \mathsf{let}\ (b,c)\ =\ f(i,a)\ \mathsf{in} \\ & \mathsf{let}\ d\ =\ g(c)\ \mathsf{in} \\ & \ \ (b,d) \end{aligned}$$

Abbildung 6.9: Schaltungsstruktur $\mathsf{se}\ f\ g\ h$

Bei dem Schritt in Abbildung 6.8 ändert sich der Zustandstyp der Schaltung von einem Zustandstyp σ zu einem Zustandstyp σ'. g ist eine Abbildung von σ nach σ' und h eine Abbildung von σ' nach σ. Der ursprüngliche Initialwert des Speichers q hat den Typ σ, der Initialwert des Speichers in der neuen Schaltung $g(q)$ hat den Typ σ'.

Die nachfolgende Gleichung macht die Aussage, daß die beiden Automaten in Abbildung 6.8 äquivalent sind. Diese Aussage ist jedoch nur unter bestimmten Voraussetzungen gültig. Die Theoreme (6.7), (6.8) und (6.10) haben diese Gleichung jeweils als Konklusion — die Prämissen sind jedoch unterschiedlich.

$$\mathsf{automaton}\,(f,q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))$$

6.3.2 Zustandskodierung

Das Theorem (6.7) beschreibt die Kodierung des Speichers. Für die Zustandskodierung wird, wie bei der Kodierung kombinatorischer Einheiten in Abschnitt 6.2.5, ein Pärchen (g,h) bestehend aus Kodierungsfunktion g und Dekodierungsfunktion h benötigt. Gefordert wird, daß die Eigenschaft $\mathsf{is_encoding}(g,h)$ erfüllt ist. Die Eigenschaft $\mathsf{is_encoding}(g,h)$ wurde in Abschnitt 6.2.5 wie folgt definiert als (siehe Abbildung 6.4):

$$\mathsf{is_encoding}(g,h) \quad := \quad \forall x.\, h(g(x)) = x$$

Das Theorem (6.7) besagt, daß die beiden Schaltungen in Abbildung 6.8 unter der Voraussetzung $\mathsf{is_encoding}(g,h)$ äquivalent sind.

$$\begin{aligned} \vdash\ \ & \mathsf{is_encoding}(g,h) \\ & \Rightarrow\ (\,\mathsf{automaton}(f,q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))\,) \end{aligned} \tag{6.7}$$

Die beiden Möglichkeiten, Kodierungen zu einem vorgegebenen Zustandstyp zu finden, wurden bereits in Abschnitt 6.2.5 erläutert. Hat man mit einem der beschriebenen Verfahren eine solche Kodierung

$$\vdash \ \mathsf{is_encoding}(g, h)$$

gefunden, dann ist die Prämisse des Zustandskodierungstheorems (6.7) erfüllt. Via Modus Ponens erhält man dann aus dem Kodierungstheorem $\vdash \ \mathsf{is_encoding}(g, h)$ und dem Zustandskodierungstheorem (6.7) die eigentliche Schaltungstransformationsgleichung für den konkreten Anwendungsfall:

$$\vdash \ \mathsf{automaton}(f, q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))$$

6.3.3 Eliminierung unerreichbarer Zustände

Das Theorem (6.8), das in diesem Abschnitt vorgestellt wird, dient dazu, die Anzahl der Zustände zu reduzieren. Dabei werden unerreichbare Zustände eliminiert. In einem Automaten sind unerreichbare Zustände prinzipiell entbehrlich. Es liegt deshalb nahe, *alle* unerreichbaren Zustände zu eliminieren. Darauf ist das Theorem (6.8) jedoch nicht beschränkt, es macht eine stärkere Aussage. Mit ihm ist es nicht nur möglich, einen Automaten auf genau die Menge der erreichbaren Zustände zu reduzieren, sondern es ist auch möglich, ihn auf jede Obermenge der erreichbaren Zustände zu reduzieren. Außerdem ist es, wie noch gezeigt werden soll, technisch einfacher, für eine Menge nachzuweisen, daß sie eine Obermenge der erreichbaren Zustände ist, als nachzuweisen, daß sie genau die Menge der erreichbaren Zustände ist.

Erreichbarkeit von Zuständen

Bevor das Theorem (6.8) selbst vorgestellt wird, soll zunächst die Erreichbarkeit von Zuständen in einem Automat definiert werden. Die Definition der Erreichbarkeit basiert auf der Funktion state, die in Abschnitt 4.3.2 eingeführt wurde. Mit $\mathsf{state}\,(f, q)\,\mathsf{input}\,t$ wird der Zustand beschrieben, den ein Automat (f, q), an dessen Eingang das Signal input anliegt, zum Zeitpunkt t annimmt.

Das Prädikat $\mathsf{reachable}(f, q)$ bezieht sich auf einen Automaten (f, q). Der Ausdruck $\mathsf{reachable}\,(f, q)\,z$ besagt, daß der Zustand z innerhalb des Automaten (f, q) erreichbar ist. Gemäß der folgenden Definition ist z genau dann als erreichbar definiert, wenn es eine Eingangsfolge *input* und einen Zeitpunkt t gibt, so daß der Automat zum Zeitpunkt t den Zustand z annimmt.

$$\mathsf{reachable}\,(f, q)\,z \quad := \quad \exists input, t.\, \mathsf{state}\,(f, q)\,input\,t = z$$

Abbildung 6.10 stellt diesen Zusammenhang graphisch dar. Die Kreise stehen für Zustände, die Pfeile für die möglichen Zustandsübergänge. Der Initialzustand q ist mit einem gefüllten Kreis gekennzeichnet.

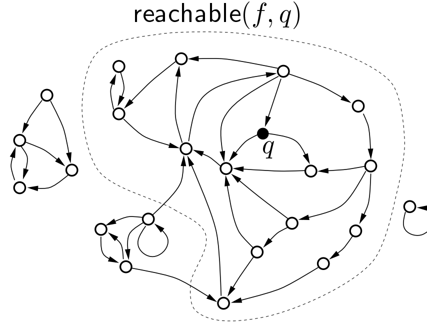


Abbildung 6.10: Erreichbarkeit von Zuständen in einem Automaten (f, q)

Das Theorem zur Eliminierung unerreichbarer Zustände

Das Theorem (6.8) setzt einen Automaten voraus, dessen Zustand den Typ $\sigma_1 + \sigma_2$ hat.[†] In Theorem (6.8) wird gefordert, daß σ_1 für eine Obermenge der erreichbaren Zustände und σ_2 für die Menge der restlichen Zustände steht. Durch die Zustandsminimierung wird die Zustandsmenge von $\sigma_1 + \sigma_2$ auf σ_1 reduziert (siehe Abbildung 6.11).

In Theorem (6.8) wird das Prädikat ISL verwendet. Das Prädikat ISL macht vereinfacht gesagt die Aussage, daß ein Element von $\sigma_1 + \sigma_2$ in der Teilmenge σ_1 liegt. Mathematisch bedeutet dies, daß ein Element x vom Typ $\sigma_1 + \sigma_2$ durch den Konstruktor INL entstanden ist, daß es also ein y vom Typ σ_1 gibt mit $x = \mathsf{INL}(y)$. Die Funktion ISL ist wie folgt definiert:

$$\mathsf{ISL}(x) \quad :\hat{=} \quad \mathsf{CASE_sum}(x, (\lambda r.\mathsf{T}), (\lambda r.\mathsf{F}))$$

Die Kodierungs- und Dekodierungsfunktionen g und h (siehe Abbildung 6.8) sind in Theorem (6.8) fest vorgegeben. Für die Dekodierungsfunktion h wird der Konstruktor INL verwendet, und die Kodierungsfunktion g ist als eine bzgl. der Menge, die das Prädikat ISL charakterisiert, inverse Funktion zu h spezifiziert.

$$\begin{array}{ll} \vdash & (\\ & \quad (\ \forall s.\ \mathsf{reachable}\ (f,q)\ s\ \Rightarrow\ \mathsf{ISL}(s)\ )\ \wedge \\ & \quad (\ \forall x.\ h(x) = \mathsf{INL}(x)\ )\ \wedge \\ & \quad (\ \forall x.\ g(\mathsf{INL}(x)) = x\ ) \\ &) \\ & \Rightarrow\ (\ \mathsf{automaton}\ (f,q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))\ ) \end{array} \tag{6.8}$$

[†] Die Typensumme $\sigma_1 + \sigma_2$ steht für die disjunkte Vereinigung mit Umbenennung (siehe auch Abschnitt 4.2.3)

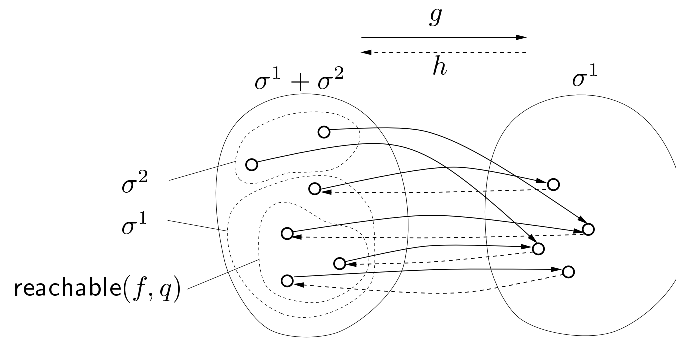


Abbildung 6.11: Eliminierung unerreichbarer Zustände

Anwendung des Theorems zur Eliminierung unerreichbarer Zustände

Es muß nun noch die Frage beantwortet werden, wie dieses Theorem zur Zustandsminimierung auf eine beliebige Schaltung angewandt werden kann. Das Theorem macht eine sehr spezielle Annahme über die Schaltung: die Zustandsmenge soll bereits in zwei disjunkte Teilmengen σ_1 und σ_2 aufgeteilt sein, und es wird gefordert, daß alle erreichbaren Zustände in σ_1 liegen. Es soll nun ein Verfahren erläutert werden, mit dem man ausgehend von einem beliebigen Automaten zu dieser Form zu gelangen.

Zunächst werden die erreichbaren Zustände eines Automaten bestimmt. Dies geschieht außerhalb der Logik. Daß die gefundene Menge tatsächlich alle erreichbaren Zustände enthält, wird später in der Logik verifiziert. Die erreichbaren Zustände eines Automaten zu bestimmen, ist vergleichsweise einfach durch Zustandstraversierung möglich (siehe Abbildung 6.12). Ausgehend von der Menge $\{q\}$, die nur den Initialzustand enthält, erweitert man in mehreren Schritten diese Zustandsmenge um die Elemente, die durch einen einfachen Zustandsübergang von den bisherigen Zuständen aus erreicht werden können. Kommen in einem Schritt keine neuen Zustände mehr hinzu, so ist man fertig, und die aktuelle Menge ist genau die Menge der erreichbaren Zustände. Die Bestimmung der erreichbaren Zustände geschieht außerhalb der Logik — eine Absicherung dafür, daß diese Menge tatsächlich die Menge der erreichbaren Zustände überdeckt, erfolgt erst nach dem nächsten Schritt, der Umkodierung der Zustände.

Im nächsten Schritt werden die Zustände des Automaten, die anfänglich den Typ σ haben, in den Typ $\sigma_1 + \sigma_2$ umkodiert. Der ursprüngliche Zustandstyp des Automaten σ, habe die Kardinalität n. Die bei der Zustandstraversierung bestimmte Menge erreichbarer Zustände habe die Kardinalität n_1. Es sei $n_2 := n - n_1$. Nun müssen zwei Datentypen σ_1 und σ_2 gefunden werden, die genau die Kardinalitäten n_1 bzw. n_2 haben. Dies ist sehr einfach möglich: Der Datentyp one hat die Kardinalität 1, und $(\alpha)\mathsf{option}$ hat eine um 1 höhere Kardinalität als α. Somit lassen sich mit Typen der Form $(\ldots((\mathsf{one})\mathsf{option})\mathsf{option}\ldots)\mathsf{option}$ Mengen beliebiger Kardinalität ausdrücken. Hat man die Typen σ_1 und σ_2 festgelegt, so

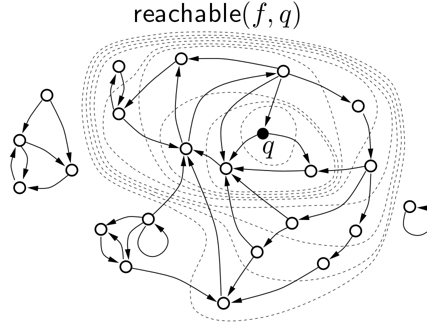


Abbildung 6.12: Zustandstraversierung in einem Automaten (f, q)

muß man mit einer Kodierungsfunktion g jedem Element aus σ ein Element aus $\sigma_1 + \sigma_2$ so zuordnen, daß alle erreichbaren Elemente in σ zu liegen kommen. Bei dieser Zustandskodierung handelt es sich um einen ganz gewöhnlichen Zustandskodierungsschritt gemäß Theorem (6.7).

Jetzt liegt ein zum ursprünglichen Automaten äquivalenter Automat vor, dessen Zustände den Typ $\sigma_1 + \sigma_2$ haben. Bei der Kodierung wurde darauf geachtet, daß alle erreichbaren Zustände in σ_1 zu liegen kommen — bewiesen wurde dies jedoch nicht. Es liegt deshalb auch für die erste Prämisse von Theorem (6.8)

$$(\forall s.\, \mathsf{reachable}\,(f, q)\; s \;\Rightarrow\; \mathsf{ISL}(s) \,)$$

noch kein Beweis vor. Dieser Beweis gestaltet sich jedoch vergleichsweise einfach. Man beachte dazu Theorem (6.9).

$$\begin{array}{ll} \vdash & (\forall s.\, \mathsf{reachable}\,(f, q)\; s \Rightarrow P(s)\,) \\ & = \\ & (\\ & \quad P(q) \wedge \\ & \quad (\forall i, s.\, P(s) \Rightarrow \forall i. P(\mathsf{SND}(f(i, s)))) \\ &) \end{array} \tag{6.9}$$

Es macht die Aussage, daß ein Prädikat P gerade dann eine Obermenge der erreichbaren Zustände charakterisiert ($\forall s.$ $\mathsf{reachable}\,(f, q)\; s \Rightarrow P(s)$), wenn zum einen der Initialzustand in dieser Teilmenge enthalten ist und zum anderen für alle Zustände s der Teilmenge auch alle Nachfolgezustände $\mathsf{SND}(f(i, s))$ in der Teilmenge liegen. Dazu müssen für jeden Zustand aus P die Funktionswerte für alle Eingaben berechnet werden. Theorem (6.9) wurde für beliebige P bewiesen, die Aussage gilt also auch dann, wenn man P mit INL instantiiert. Auf diese Weise wird Theorem (6.9) dazu verwendet, die erste Prämisse von Theorem (6.8) nach obigem Schema durch Fallunterscheidung und Auswertung zu beweisen.

Es wird deutlich, daß dieser Beweis leichter zu erbringen ist als der Beweis dafür, daß σ_1 genau die Menge der erreichbaren Zustände ist. Um zu zeigen, daß ein Zustand erreichbar ist, müßte eine Eingangsfolge *input* gefunden werden,

so daß der Automat nach einer gewissen Anzahl von Takten t in diesen Zustand übergeht. Der Beweis dafür müßte für jeden Zustand erbracht werden.

Die beiden anderen Prämissen von Theorem (6.8) sind leicht erfüllbar. Es wird gefordert, daß für die Kodierungsfunktion h gerade die Funktion INL verwendet werden soll, und es wird gefordert, daß $\forall x.\, g(\mathsf{INL}(x)) = x$ gilt. Die Funktion g läßt sich durch Fallunterscheidung über den Werten von σ_1 definieren, und der Beweis für $\forall x.\, g(\mathsf{INL}(x)) = x$ durch Fallunterscheidung über σ_1 erbringen.

Jetzt sind alle Voraussetzungen erbracht, und der Automat, dessen Zustände bereits den Typ $\sigma_1 + \sigma_2$ haben, kann durch das Anwenden der Konklusion von Theorem (6.8)

$$\mathsf{automaton}\ (f, q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))$$

in einen äquivalenten Automaten überführt werden, bei dem alle unerreichbaren Zustände eliminiert sind.

Verfahren zur Eliminierung unerreichbarer Zustände werden in der Praxis nur partiell angewandt. Der Aufwand für die Bestimmung aller erreichbaren Zustände ist für größere Schaltungen erheblich. Es gilt auch zu bedenken, daß die Halbierung der Zustandsmenge nur zu einem Bit an Speichererersparnis führt. Ferner ist zu bedenken, daß der Automat mit kleinstmöglicher Zustandsanzahl nicht unbedingt der kostengünstigste in bezug auf die benötigten kombinatorischen Hardwareressourcen sein muß, denn unter Umständen ergibt sich daraus ein aufwendigeres Schaltnetz. In der Praxis wird die Eliminierung unerreichbarer Zustände nur auf Schaltungen mit einer geringen Zustandsanzahl (z. B. Kontrolleinheiten) angewandt.

6.3.4 Zustandsklassifikation

Die bisher vorgestellten Theoreme zur Zustandskodierung können verwendet werden, um unerreichbare Zustände zu eliminieren und um Zustände umzukodieren. Die Kodierungsfunktion g war dabei in Bezug auf die Menge der erreichbaren Zustände stets injektiv. Es soll jetzt eine Zustandskodierung vorgestellt werden, die darüber hinausgeht. Mit ihr wird es möglich, auch die Menge der erreichbaren Zustände zu reduzieren. Dies geschieht dadurch, daß mehrere äquivalente Zustände zu einem einzigen Zustand zusammengefaßt werden.

Das Theorem (6.10) reduziert die Menge der erreichbaren Zustände, indem es Klassen von Zuständen mit äquivalentem Verhalten jeweils auf einen Repräsentanten abbildet. Gefordert wird, daß es eine Klassifikation der Zustände gibt, so daß der Automat bei äquivalenten Zuständen und gleicher Eingabe immer die gleiche Ausgabe produziert und die Nachfolgezustände wiederum äquivalent sind.

Das Theorem zur Zustandsklassifikation

In Theorem (6.10) wird dies mit zwei Funktionen g und h beschrieben. g ist die charakteristische Funktion der Äquivalenzrelation. Sie bildet jede Klasse auf einen

Repräsentanten ab (Abbildung 6.13). Die Funktion h ist dazu „invers": Sie bildet jeden Repräsentanten auf ein Element der Klasse ab. In der neuen, kodierten Automatenbeschreibung übernimmt g die Rolle der Kodierungsfunktion und h die Rolle der Dekodierungsfunktion (siehe Abbildung 6.8).

$$\begin{array}{ll}\vdash & (\ \forall s.\, g(h(s)) = s\ )\wedge \\ & (\ \forall s_1, s_2.\, g(s_1) = g(s_2) \\ & \ \Rightarrow (\forall i. \\ & \ \ \mathsf{FST}(f(i,s_1)) = \mathsf{FST}(f(i,s_2)) \wedge \\ & \ \ g(\mathsf{SND}(f(i,s_1))) = g(\mathsf{SND}(f(i,s_2))) \\ & \ )) \\ & \Rightarrow\ (\ \mathsf{automaton}\ (f,q) = \mathsf{automaton}(\mathsf{se}\ f\ g\ h, g(q))\ )\end{array} \tag{6.10}$$

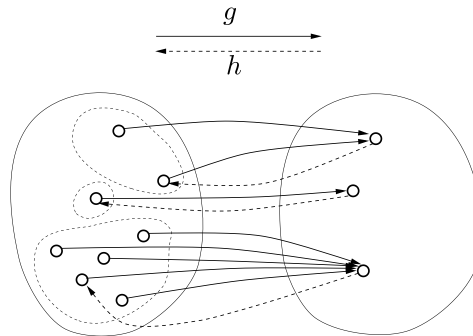


Abbildung 6.13: Zustandsklassifikation

Anwendung des Theorems zur Zustandsklassifikation

Um dieses Theorem anwenden zu können, benötigt man eine Klassifikationsfunktion g. Diese bestimmt man vorab ohne logische Argumentation. Hat man eine solche gefunden, so ist der weitere Weg sehr einfach: man bestimmt per Fallunterscheidung eine dazu passende Dekodierungsfunktion h und beweist dann die Prämissen von Theorem (6.10) durch Fallunterscheidung und Auswertung. Ist die Funktion g passend gewählt, so gelingt der Beweis und man gewinnt aus der Konklusion von Theorem (6.10) die gewünschte Schaltungstransformation.

Das Problem besteht nun darin, eine Klassifikationsfunktion g effizient zu bestimmen. Dazu ist der Satz von Huffman/Mealy hilfreich [Brau84]. Es sei vorausgesetzt, daß bereits alle unerreichbaren Zustände gemäß Theorem (6.8) eliminiert wurden. Der Satz von Huffman/Mealy besagt, daß zwei Zustände schon dann äquivalent sind, wenn sich ihre sog. „Leistungen" in den ersten k-1 Takten nicht unterscheiden. Dabei ist k die Anzahl der Zustände des Mealy-Automaten. Mit Leistung ist dabei die Ein-/Ausgabefunktion gemeint, die der Automat realisiert, wenn man den betreffenden Zustand als Anfangszustand nimmt. Im Satz von Huffman/Mealy werden Zustände als äquivalent bezeichnet, wenn sich ihre Leistungen

nicht unterscheiden. Dieser Äquivalenzbegriff ist konform mit der Äquivalenz, die durch die Prämisse in Theorem (6.10) gefordert wird. Dazu wählt man für g gerade die charakteristische Funktion der über den Leistungsbegriff definierten Äquivalenzrelation. Die Prämisse von Theorem (6.10) wird dadurch erfüllt. Gefordert wird in dieser Prämisse nicht unbedingt genau die Äquivalenzrelation — eine Verfeinerung der Äquivalenzrelation [HoUl79] wäre auch möglich gewesen. Das Ergebnis von Huffman/Mealy ist jedoch besonders günstig, denn es beschreibt den Automaten mit der minimalen Anzahl an Zuständen. Dieser Automat ist bis auf Isomorphie (Umbenennung der Zustände) eindeutig!

Diese Aussage hat eine konkrete technische Bedeutung. Sie besagt, daß durch diesen Schritt nicht nur ein in bezüglich der Anzahl der Zustände verbessertes Ergebnis, sondern gar das Optimum erreicht wird. Man beachte ferner, daß die über die Leistung definierte Äquivalenz von Zuständen zwei äquivalenten Automaten jeweils Äquivalenzrelationen von gleichem Index (mit gleicher Anzahl von Äquivalenzklassen) zuordnet. Dies bedeutet, daß zwei äquivalente Automaten durch dieses Verfahren in Automaten minimaler Zustandsanzahl überführt werden, die sich nur noch in der Benennung der Zustände unterscheiden können.

Um mit Hilfe von Hufman/Mealy die Menge der Zustände in Äquivalenzklassen aufzuteilen, muß die Schaltung also mit allen Zuständen als Anfangszustand und für alle möglichen Eingaben $k - 1$ Takte lang simuliert werden. Der Aufwand dafür ist sehr groß: er wächst exponentiell mit der Anzahl der Speicherelemente und exponentiell mit der Anzahl der Eingangssignale. In [Brau84] werden zwar noch Optimierungen vorgeschlagen, die zu einer schnelleren Bestimmung der Äquivalenzklassen führen können. Der Aufwand dafür bleibt jedoch erheblich und ist deshalb ähnlich wie die Eliminierung unerreichbarer Zustände nur für sehr sehr kleine Schaltungen vertretbar.

6.3.5 Retiming

Retiming ist eine sequentielle Transformation, die auch für große Schaltungen effizient durchgeführt werden kann. Sie wird deshalb gerne als Ergänzung zur rein kombinatorischen Optimierung verwendet. Beschränkt man sich auf rein kombinatorische Umformungen, so bleibt man in einer kleinen Teilmenge der zur Ausgabeschaltung äquivalenten Schaltungen hängen. Oft befindet sich das Optimum nicht in dieser Teilmenge. Durch rein kombinatorische Umformungen ändert sich die Zustandsrepräsentation nicht — Retiming sprengt diese Grenzen.

Vereinfacht gesagt wird beim Retiming eine kombinatorische Einheit f_1 über eine Speichereinheit hinweggeschoben (siehe Abbildung 6.14). Dazu sei der kombinatorische Anteil einer Schaltung aufgeteilt in zwei Teile f_1 und f_2. Gleichzeitig wird während des Retiming-Schritts für die neue Speicherkomponente ein neuer Initialzustand $f_1(q)$ berechnet, wobei q der Initialzustand der alten Speicherkomponente ist.

Durch Retiming kann in signifikanter Weise die Verzögerungszeit des kombinatorischen Anteils einer Schaltung reduziert und damit die Taktfrequenz erhöht werden. Durch Retiming ändert sich, ähnlich wie bei den bereits beschriebenen

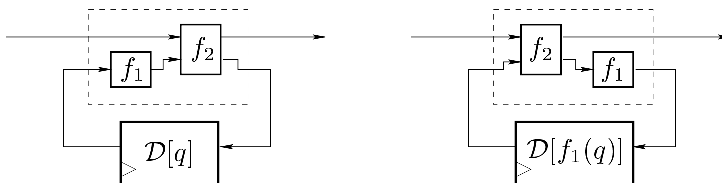


Abbildung 6.14: Retiming

Zustandskodierungsverfahren, die Zustandsrepräsentation. Das hat Einfluß auf die Anzahl der benötigten Speichereinheiten und auf die Anzahl der Zustandswechsel der einzelnen Speichereinheiten im Betrieb. Diese beiden Faktoren bestimmen wesentlich den Energieverbrauch bei CMOS-Schaltungen.

Durch einen Retiming-Schritt wird die Größe des kombinatorischen Anteils einer Schaltung nicht verändert. Der kombinatorische Anteil wird durch diesen Schritt lediglich umgeordnet. Daraus können sich jedoch auch weitere Potentiale zur Optimierung des kombinatorischen Anteils ergeben. Retiming wird deshalb gern mit anschließenden booleschen Optimierungen kombiniert, oder es findet gar ein Iterationsprozeß mit abwechselnden Retiming- und Logikminimierungsschritten statt.

Theorem (6.11) beschreibt den Retiming-Schritt in formaler Weise. Es besagt, daß die beiden Schaltungsstrukturen in Abbildung 6.14 äquivalent sind. In beiden Schaltungen besteht der kombinatorische Anteil aus zwei Teilkomponenten f_1 und f_2.

$$\begin{array}{ll}\vdash & \mathsf{automaton}(\\ & \quad (\lambda(i,s).\, f_2(i, f_1(s)))\ , \\ & \quad q \\ &) \\ & = \\ & \mathsf{automaton}(\\ & \quad (\lambda(i,s).\, \mathsf{let}\ (x,y) = f_2(i,s)\ \mathsf{in}\ (x, f_1(y)))\ , \\ & \quad f_1(q) \\ &)\end{array} \tag{6.11}$$

Beim Retiming wird, ähnlich wie bei den bereits beschriebenen Verfahren zur Zustandskodierung, die Zustandsrepräsentation verändert. Ist f_1 bijektiv, so kann der Retiming-Schritt durch eine Zustandskodierung gemäß Theorem (6.7) ausgedrückt werden. Dabei verwendet man als Kodierungsfunktion g gerade f_1 und als Dekodierungsfunktion h die Inverse. Bei der Kodierung werden die Funktion $g = f_1$ und h eingefügt. Die eingefügte Funktion h und die bereits vorhandene Funktion f_1 heben sich auf, es verbleibt das gleiche Ergebnis wie beim Retiming. Im Gegensatz zur Zustandskodierung gemäß Theorem (6.7) ist f_1 beim Retiming jedoch beliebig und muß weder injektiv noch bijektiv sein.

Um das Retiming-Theorem bei der Synthese anwenden zu können, muß zunächst lediglich der kombinatorische Anteil der Schaltung in zwei Teile f_1 und

f_2 zerlegt werden. Die Zerlegung geschieht in der Logik durch eine einfache kombinatorische Umformung. Es stellt sich jedoch die Frage, wie denn diese Aufteilung am besten zu bestimmen ist. Diese Frage ist i. allg. sehr schwer zu beantworten. Zwar kann direkt bestimmt werden, wie sich der Retiming-Schritt selbst auf die Anzahl der Register und die kombinatorische Tiefe der Schaltung auswirkt. Es ist jedoch zu beachten, daß dem Retiming auch weitere logische Optimierungen nachgeschaltet werden können, und es ist schließlich auch sinnvoll, Retiming und Logikminimierung mehrfach hintereinander im Wechsel durchzuführen. Da gute Verfahren zur Bestimmung eines möglichst optimalen Schnittes alles andere als einfach sind, und man sich hier sehr unterschiedliche Vorgehensweisen vorstellen kann, soll in Abschnitt 6.4 erläutert werden, wie Implementierungen derartiger Verfahren flexibel an den in der Logik durchgeführten formalen Retiming-Schritt angebunden werden können.

6.3.6 Eliminierung redundanter Schaltungsteile

In bestimmten Situationen können Schaltungsteile entfernt werden, ohne daß dadurch die Schaltungsfunktion im ganzen verändert wird. Das Theorem (6.12) beschreibt einen solchen Fall (siehe Abbildung 6.15). Dabei wird ein kombinatorischer Schaltungsteil f_2 und ein daran angeschlossener Teil des Speichers entfernt. f_2 hat keine Verbindung zum Gesamtausgang der Schaltung. Der Anteil des Speichers, auf den f_2 schreibt, wird nur von f_2 selbst gelesen, nicht jedoch vom Rest der Schaltung.

$$\begin{array}{l} \vdash \ \mathsf{automaton}(\\ \quad (\lambda(i,(s_1,s_2)).\,\mathsf{let}\,(x,y) = f_1(i,s_1)\,\mathsf{in}\,(x,(y,f_2(i,s_1,s_2))))\ , \\ \quad (q_1,q_2) \\ \quad) \\ \quad = \mathsf{automaton}(f_1,q_1) \end{array} \tag{6.12}$$

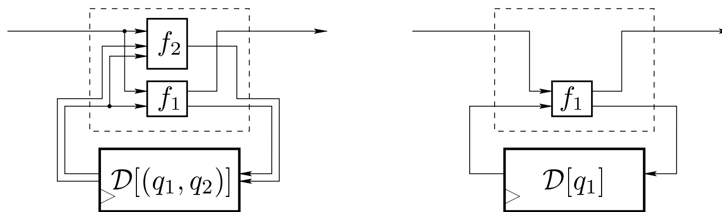


Abbildung 6.15: Eliminierung redundanter Schaltungsanteile

Bei der Synthese synchroner VHDL-Beschreibungen können Schaltungsstrukturen wie im linken Teil von Abbildung 6.15 entstehen [EiKu95c, EiKu96]. VHDL-Variablen müssen in der Regel durch Speicherelemente realisiert werden, da auf diesen im allgemeinen auch über Taktgrenzen hinweg Daten gehalten werden. Für

den speziellen Fall, daß ein Prozeß innerhalb eines Taktzyklus immer erst schreibend auf eine Variable zugreift, bevor er von ihr liest, ist es möglich, diese Variable einfach durch eine Verbindungsleitung statt durch eine Speicherkomponente zu realisieren. Diese Optimierung kann mit Theorem (6.12) erfolgen.

Das Theorem (6.12) kann als ein Spezialfall des Theorems (6.10) zur Zustandskodierung durch Klassifikation betrachtet werden. In der betrachteten Schaltung sind Zustände Paare (x_1, x_2) vom Typ $\sigma_1 \times \sigma_2$, wobei die Schaltung für alle Zustände mit gleichem x_1-Wert das gleiche Verhalten hat. Die charakteristische Funktion für die Äquivalenzklassenbildung ist somit $g = \mathsf{FST}$. Führt man die Zustandskodierung mit Theorem (6.10) durch, so kommt man zu einer kombinatorischen Struktur, in der der Speicher auf seinen ersten Teil reduziert wird und in der innerhalb der kombinatorischen Struktur das Ausgangssignal von f_2 keine Senke hat. f_2 kann also gemäß Abschnitt 6.2.3 eliminiert werden und man gelangt zu dem gleichen Ergebnis wie mit Theorem (6.12).

6.4 Einbindung der Schaltungstransformationen in einen Syntheseablauf

In den vorangegangenen Abschnitten dieses Kapitels wurden verschiedene elementare Schaltungstransformationen vorgestellt. Diese bilden die Basis für eine Formale Schaltungssynthese. Am Beispiel des Retiming-Schritts soll erläutert werden, wie diese elementaren Transformationen als Grundlage für die Formale Synthese verwendet werden können, und es soll erörtert werden, wie bestehende Syntheseverfahren integriert werden können.

Um eine vorgegebene Schaltungsbeschreibung mittels Retiming zu transformieren, sind die folgenden vier Schritte erforderlich:

I. Zunächst muß ein Schnitt für den kombinatorischen Anteil bestimmt werden, der jede kombinatorische Einheit entweder f_1 oder f_2 zuordnet.

II. Gemäß dieser Zuordnung wird dann die kombinatorische Funktion durch eine Äquivalenzumformung in die beiden Teilfunktionen zerlegt.

III. Nun kann das das Retiming-Theorem (6.11) angewandt werden.

IV. Schließlich muß noch der neue Initialwert $f_1(q)$ per Auswertung bestimmt werden.

Abbildung 6.16 zeigt den Ablauf an einem kleinen Beispiel. Im ersten Schritt wird festgelegt, wie der Retiming-Schritt ausgeführt wird. Dieser Schritt beeinflußt die Qualität des damit zu erzielenden Ergebnisses. Eine gute Zuordnung ist nicht leicht zu finden. Zu berücksichtigen sind dabei vor allem Kosten in Form von Hardwareaufwand und kombinatorischer Tiefe. Da Retiming in der Regel mit weiteren kombinatorischen Optimierungen oder sequentiellen Optimierungen kombiniert wird, sind die zu erwartenden Kosten nur grob abschätzbar. Um zu einer

guten Zuordnung der kombinatorischen Einheiten zu gelangen, sind aufwendige Verfahren notwendig.

Durch die Schritte II, III und IV wird die eigentliche Transformation von Eingabeschaltung auf Ausgabeschaltung durchgeführt. Wie dieser Schritt durchgeführt wird, ist durch die in Schritt I bestimmte Zuordnung eindeutig festgelegt. Dadurch, daß die Schritte in dem Theorembeweiser HOL durchgeführt werden, ist sichergestellt, daß diese die Korrektheit der Transformation gewährleisten. Die Eingabeschaltungsbeschreibung t wird durch die Transformation in ein Theorem der Form $\vdash t = t'$ überführt, wobei t' die während der Transformation entstandene neue Schaltungsbeschreibung ist. In HOL ist gewährleistet, daß Theoreme nur durch die Anwendung von logischen Regeln entstehen können, die auf dem HOL-Kalkül basieren. Es ist also implizit gewährleistet, daß dieser Schritt korrekt ist.

Auf diese Weise erfolgt eine klare Trennung in Entwurfsraumuntersuchung (Schritt I) und eigentliche Schaltungstransformation (Schritte II, III und IV). Bei der implementierten Retiming-Transformation dient die Zuordnung als Parameter, mit der die Transformation gesteuert wird (siehe Abbildung 6.17). Die Schaltungstransformation führt nicht für jede denkbare Zuordnung zu einem Ergebnis. Die Transformation scheitert, wenn die als Parameter übergebene Zuordnung bzgl. der vorgegebenen Schaltung nicht zulässig ist. In diesem Fall kommt es zu einer Ausnahmebehandlung (exception) — ein Ergebnis wird nicht erzeugt. Daß durch eine „falsche" Zuordnung ein nicht korrektes Syntheseergebnis erzeugt wird, ist ausgeschlossen.

Welche Zuordnungen zulässig sind und damit zu einem Ergebnis führen und welche nicht, soll nun am Beispiel der Schaltung in Abbildung 6.18 illustriert werden. Eine Zuordnung ist zulässig genau dann, wenn es keinen kombinatorischen Pfad von einem Eingang der Gesamtschaltung oder einem Ausgang einer Komponente aus f_2 zu einem Eingang einer Komponente aus f_1 gibt. Nur unter dieser Voraussetzung ist es in Schritt II möglich, den kombinatorischen Teil in die für das Retiming-Theorem (6.11) notwendige Form zu bringen. In Abbildung 6.18 ist es beispielsweise zulässig, f_1 die Komponenten *C4* und *C6* zuzuordnen und f_2 die Komponenten *C1*, *C2*, *C3* und *C5*. Prinzipiell nicht zulässig ist es, die Komponenten *C2* oder *C3* der Teilfunktion f_1 zuzuordnen, da sie über kombinatorische Pfade mit den Eingängen verbunden sind. Nicht zulässig wäre beispielsweise auch eine Zuordnung, die f_1 die Komponente *C6* und f_2 die Komponente *C4* zuordnet, da so ein kombinatorischer Pfad von einem Ausgang von f_2 zu einem Eingang von f_1 entstehen würde.

Bei der bisher betrachteten Retiming-Schaltungstransformation wurde das Theorem (6.11) von links nach rechts angewandt. Dieser Schritt wird gern auch als Forward-Retiming bezeichnet. Die umgekehrte Richtung, das sog. Backward-Retiming, ist weit aufwendiger in der Implementierung und wird deshalb selten angewandt. Prinzipiell kann Backward-Retiming einfach durch Anwendung des Retiming-Theorems (6.11) von rechts nach links durchgeführt werden. Was den Backward-Retiming-Schritt aufwendiger macht, ist die Bestimmung des neuen In-

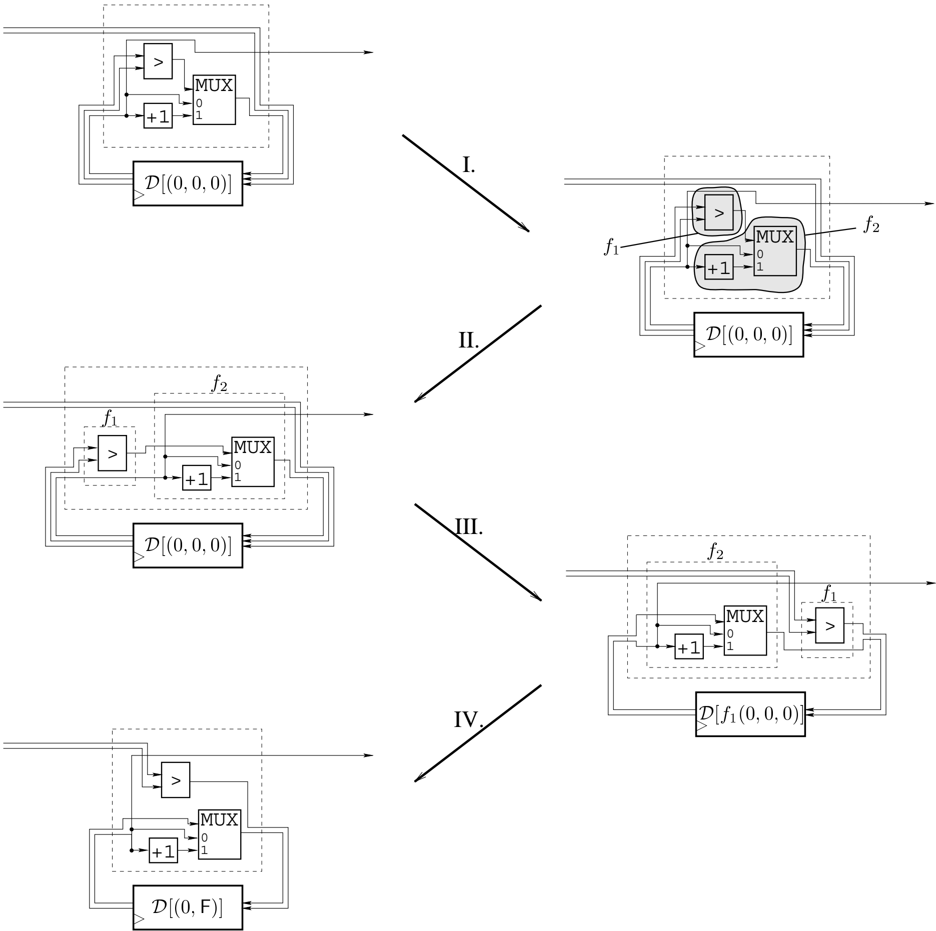


Abbildung 6.16: Beispiel für einen Retiming-Schritt

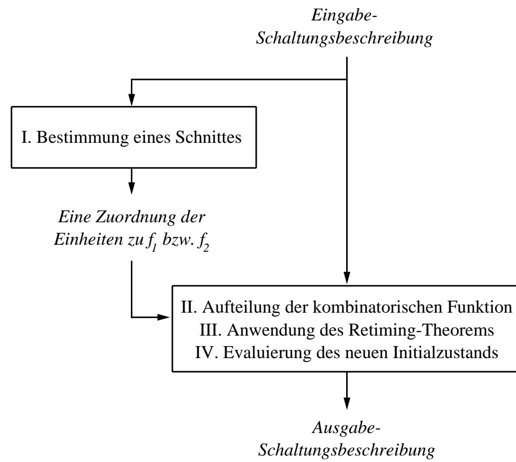


Abbildung 6.17: Entwurfsraumuntersuchung und Schaltungstransformation beim Retiming

itialwertes q' aus dem alten Initialwert q. Es wird gefordert daß $f_1(q') = q$ gilt. Dies ist i. allg. nicht eindeutig, und es ist auch möglich, daß es einen solchen Wert nicht gibt. Im Gegensatz zum Forward-Retiming ist es nicht mehr möglich, den neuen Initialwert einfach durch Auswertung der Funktion f_1 zu bestimmen, stattdessen muß der neue Initialwert — oder genauer: die Menge der neuen Initialwerte — mit aufwendigen Verfahren gesucht werden.

Problematisch ist auch, daß beim Backward-Retiming nicht jede Aufteilung des kombinatorischen Anteils zu einer Lösung führt, und zwar auch dann nicht, wenn die entsprechenden Abhängigkeiten zwischen den Komponenten berücksichtigt werden. Bereits bei der Aufteilung des kombinatorischen Anteils muß die Frage berücksichtigt werden, ob diese Aufteilung überhaupt zu einem neuen Initialwert führen kann. So wäre in Abbildung 6.18 bei einem Backward-Retiming-Schritt eine Aufteilung denkbar, bei der *C5* und *C6* der Funktion f_1 und alle anderen Komponenten der Funktion f_2 zugeordnet werden. Dies scheitert jedoch daran, daß es für diese Aufteilung keinen neuen Initialwert gibt. Man erkennt dies leicht an der Speicherkomponente, die nach diesem Schritt am Ausgang von *C4* zu liegen käme. Deren Initialwert müßte aufgrund der Abhängigkeiten, die durch *C6* hergestellt werden, den Wert F und aufgrund von *C5* den Wert T haben.

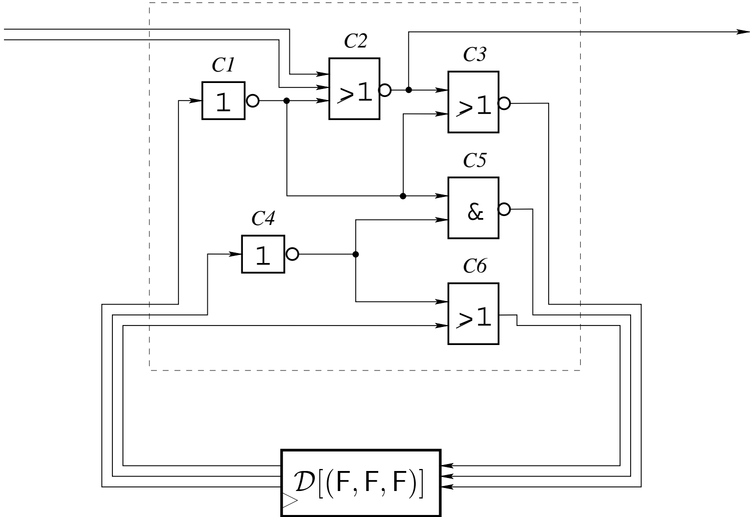


Abbildung 6.18: Beispielschaltung

Kapitel 7

High-Level-Synthese mit Datenpfaden

In diesem Kapitel wird erläutert, wie High-Level-Synthese auf der Basis einer Formalen Synthese durchgeführt werden kann. Ausgehend von einer algorithmischen Beschreibung wird bei der High-Level-Synthese eine Schaltung auf RT-Ebene abgeleitet. Die Zuordnung zwischen einem Algorithmus und einer Schaltung auf RT-Ebene ist nicht eindeutig. Es sind stets verschiedene Implementierungen denkbar. Diese unterscheiden sich vor allem in ihrem zeitlichen Verhalten. Sie unterscheiden sich darin, wie und wann die Eingangswerte von den Signalleitungen gelesen werden, wie und wann die Ausgangswerte auf die Ausgangssignale geschrieben werden, und sie unterscheiden sich darin, wie und ob überhaupt der Status der Berechnung an die Umgebung signalisiert wird.

In diesem Ansatz sollen algorithmische Beschreibung und Schnittstellenverhalten strikt getrennt beschrieben werden. Die algorithmische Beschreibung wird durch eine Funktion repräsentiert, die eine Beziehung zwischen Eingangs- und Ausgangswerten herstellt. Diese Funktion macht keine zeitlichen Aussagen. Sie beschreibt nicht, zu welchen Taktzeitpunkten die Schaltung die Ein-/Ausgangssignale lesen bzw. schreiben soll. Durch das Schnittstellenverhalten wird eine Beziehung zwischen einer Funktion und dem konkreten Verhalten einer Schaltung auf RT-Ebene hergestellt. Üblicherweise beschränken sich High-Level-Syntheseprogramme auf genau ein implizit vorgegebenes Schema von Schnittstellenverhalten. In diesem Ansatz werden dem Anwender mehrere verschiedene Schablonen für das Schnittstellenverhalten zur Verfügung gestellt.

Die hier vorgestellten Arbeiten beschränken sich auf die klassischen Verfahren mit einem reinen Datenflußgraphen als algorithmische Beschreibung [Camp91]. Im Rahmen des Forschungsprojekts (DFG-Projekt SCHM-623/6-1) wird bereits daran gearbeitet, diesen Ansatz auf allgemeine algorithmische Beschreibungen mit beliebigen berechenbaren Funktionen (While-Schleifen etc.) zu erweitern.

In diesem Kapitel werden, wie bereits in Kapitel 6, Syntheseverfahren vorge-

stellt, die im Prinzip bereits bekannt sind. Der Fortschritt besteht darin, daß von diesen Syntheseprogrammen garantiert ist, das sie korrekt arbeiten. Dies ist in objektiver Weise durch den Theorembeweiser sichergestellt. Diesen Beweis für die Korrektheit der Abläufe erbringen konventionelle Syntheseprogramme nicht.

7.1 Vorgehensweise

Die Vorgehensweise ist in diesem Ansatz wie folgt: Zunächst definiert der Schaltungsentwerfer eine Funktion f. Die Funktion f soll gemäß Kapitel 4 durch einen DFG-Term beschrieben werden. Bisher wurden DFG-Terme lediglich zur Modellierung von Schaltnetzen eingesetzt. Jetzt soll eine solche Funktion den Ausgangspunkt für eine High-Level-Synthese bilden. Anschließend wählt der Schaltungsentwerfer eines der vier Schnittstellenverhaltensmuster aus, die im folgenden noch genauer vorgestellt werden. Das Schnittstellenverhaltensmuster beschreibt, auf welche Weise die Schaltung, die konstruiert werden soll, die Funktion f ausführt. Schnittstellenverhaltensmuster sind noch keine Implementierungen. Sie spezifiziert lediglich die zeitlichen und funktionalen Zusammenhänge zwischen Ein- und Ausgangssignalen der Gesamtschaltung.

Die Funktion f und das Schnittstellenverhaltensmuster bilden die Eingabe für die High-Level-Synthese. Die Aufgabe der High-Level-Synthese ist es, aus der Funktion f und dem Schnittstellenverhaltensmuster eine Implementierung auf RT-Ebene abzuleiten. Als Implementierung sind sowohl eine sequentielle Schaltung als auch eine rein kombinatorische Schaltung möglich. In der hier vorgestellten Notation (siehe Kapitel 4) bedeutet dies, daß als Implementierung entweder ein Term der Form $\mathsf{automaton}(f, q)$ oder ein Term der Form $\mathsf{combinatorial_block}(f)$ gefunden werden muß.

Wie schon bei der Formalen Synthese auf RT- und Gatterebene (siehe Kapitel 6), soll auch hier nicht nur irgendwie eine Implementierung erzeugt werden, sondern sie soll in mathematischer Weise abgeleitet werden, um so die Korrektheit der High-Level-Synthese in objektiver und nachvollziehbarer Weise garantieren zu können. Als Ergebnis soll ein Theorem entstehen, das besagt, daß eine solche Implementierung das Verhalten, das durch die Funktion f und durch das Schnittstellenverhaltensmuster spezifiziert wird, erfüllt. In der Logik wird dieses Theorem durch eine Implikation ausgedrückt. Auf der linken Seite der Implikation beschreibt eine Gleichung den durch die Implementierung $\mathsf{automaton}(f, q)$ bzw. $\mathsf{combinatorial_block}(f)$ hergestellten Zusammenhang zwischen Ein- und Ausgabesignalen. Auf der rechten Seite der Implikation befindet sich das Schnittstellenverhaltensmuster mit der Funktion f.

Durch diese Vorgehensweise ist es möglich, aus einer vom Schaltungsentwerfer definierten Funktion f mit Hilfe unterschiedlicher Schnittstellenverhaltensmuster unterschiedliche, an den jeweiligen Verwendungszweck angepaßte Implementierungen abzuleiten.

In den folgenden Abschnitten werden zunächst die vier Schnittstellenverhaltensmuster eingeführt. Anschließend werden Theoreme vorgestellt, mit deren Hil-

fe für die einzelnen Schnittstellenverhaltensmuster zu vorgegebenen Funktionen f Implementierungen auf RT-Ebene abgeleitet werden können. Um mit diesen Theoremen eine Implementierung auf Gatterebene ableiten zu können, muß zunächst die Funktion f in mehrere Teile zerlegt worden sein. Für die Zerlegung der Funktion f gibt es je nach Schnittstellenverhaltensmuster unterschiedliche Randbedingungen. Im nächsten Abschnitt wird erläutert, wie diese Zerlegung durchgeführt werden kann und welchen Einfluß unterschiedliche Zerlegungen auf die Kosten der Implementierung haben. Im letzten Abschnitt soll erläutert werden, wie Verfahren aus der konventionellen High-Level-Synthese eingebunden werden können, um zu einer möglichst kostengünstigen Implementierung zu gelangen.

7.2 Beschreibung von Schaltungen auf Algorithmischer Ebene

Zu einem DFG-Term f hat der Benutzer die Wahl zwischen vier verschiedenen Schnittstellenverhaltensmustern, die beschreiben, wie die Schaltung die Funktion f ausführt: cs_pipeline, cs_cycle, cs_start und cs_reset. Abbildung 7.1 zeigt deren formalen Definitionen, zu denen nun einige Erläuterungen gemacht werden sollen.

Die Schnittstellenverhaltensmuster spezifizieren jeweils eine Schaltung in Abhängigkeit von einem DFG-Term f und einer natürliche Zahl n. Die natürliche Zahl n gibt an, in wievielen Takten die Funktion f durch die Schaltung ausgeführt werden soll. Jede der Schaltungen hat einen Dateneingang *datain* und einen Datenausgang *dataout*, über die die Schaltung die Eingangswerte für f einliest bzw. die Funktionswerte von f ausgibt. Die durch cs_pipeline beschriebene Schaltung verfügt nur über einen Dateneingang und einen Datenausgang. Die durch die anderen Schnittstellenverhaltensmuster beschriebenen Schaltungen verfügen zusätzlich über boolesche Kontrollsignale. Mit dem booleschen Signal *ready* wird signalisiert, daß die Schaltung die Berechnung des Funktionswerts beendet hat und dieser nun am Ausgang anliegt. Die Signale *start* und *reset* dienen in den durch cs_start und cs_reset beschriebenen Schaltungen dazu, eine neue Berechnung anzustoßen, wobei *start* ignoriert wird, wenn gerade eine andere Berechnung durchgeführt wird, wohingegen *reset* bereits laufende Berechnungen abbricht.

Die in den Definitionen von Abbildung 7.1 definierten Schnittstellenverhaltensmuster beschreiben jeweils, wie eine Schaltung die Funktion f ausführt. Dies soll nun erläutert werden. Man betrachte dazu die in Abbildung 7.2 dargestellten Beispielabläufe. In diesen Beispielabläufen wird einheitlich angenommen, daß der Parameter n auf 4 gesetzt sei — die Funktion f soll also jeweils in 4 Takten ausgeführt werden. Die Signale sind jeweils für die ersten 20 Takte dargestellt. Im oberen Bereich sind jeweils die Eingangssignale (*datain*, *start*, *reset*) und im unteren Bereich die Ausgangssignale (*dataout*, *ready*) dargestellt. In den Diagrammen stehen Kreise für die Datenwerte an den *datain*- und *dataout*-Anschlüssen der Schaltungen. Mit den Pfeilen, die über die Funktion f von Eingängen zu Ausgängen führen wird angedeutet, daß durch die Schaltung zu einem bestimmten Zeitpunkt ein Ein-

$$
\begin{array}{l}
\mathsf{cs_pipeline}\ (n,f)\ (\mathit{datain}, \mathit{dataout}) \quad := \\
\quad \forall t_0.\ \mathit{dataout}(t_0+n) = f(\mathit{datain}(t_0))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cs_cycle}\ (n,f)\ (\mathit{datain}, \mathit{dataout}, \mathit{ready}) \quad := \\
\quad (\forall t.\ \mathit{ready}(t) = (t\ \mathsf{MOD}\ (n+1) = 0)) \land \\
\quad (\forall i.\ \mathit{dataout}((n+1)*i+n) = f(\mathit{datain}((n+1)*i)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cs_start}\ (n,f)\ (\mathit{datain}, \mathit{start}, \mathit{dataout}, \mathit{ready}) \quad := \\
\quad \mathit{ready}(0) \land \\
\quad (\forall t_0.\ \mathit{ready}(t_0)\ \Rightarrow\ (\\
\quad\quad \mathsf{if}\ \mathit{start}(t_0)\ \mathsf{then} \\
\quad\quad\quad (\forall i < n.\ \neg(\mathit{ready}(t_0+i+1))) \land \\
\quad\quad\quad \mathit{ready}(t_0+n+1) \\
\quad\quad \mathsf{else} \\
\quad\quad\quad \mathit{ready}(t_0+1) \\
\quad)) \land \\
\quad (\forall t_0.(\mathit{ready}(t_0)\ \land\ \mathit{start}(t_0)) \Rightarrow (\\
\quad\quad \forall \mathit{hold}. \\
\quad\quad\quad (\forall i < \mathit{hold}.\ \neg(\mathit{start}(t_0+n+i+1))) \\
\quad\quad\quad \Rightarrow\ (\forall i \le \mathit{hold}.\ (\mathit{dataout}(t_0+n+i) = f(\mathit{datain}(t_0)))) \\
\quad))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cs_reset}\ (n,f)\ (\mathit{datain}, \mathit{reset}, \mathit{dataout}, \mathit{ready}) \quad := \\
\quad \mathit{ready}(0) \land \\
\quad (\forall t_0. \\
\quad\quad (\mathit{reset}(t_0)\ \land\ (\forall i \le n.\ \neg(\mathit{reset}(t_0+i+1)))) \\
\quad\quad \Rightarrow\ ((\forall i < n.\ \neg(\mathit{ready}(t_0+i+1)))\ \land (\mathit{ready}(t_0+n+1))) \\
\quad) \land \\
\quad (\forall t_0.\ (\neg(\mathit{reset}(t_0))\ \land\ \mathit{ready}(t_0))\ \Rightarrow\ \mathit{ready}(t_0+1)) \land \\
\quad (\forall t_0.((\mathit{reset}(t_0))\ \land\ (\forall i \le n.\ \neg(\mathit{reset}(t_0+i+1)))) \Rightarrow (\\
\quad\quad \forall \mathit{hold}. \\
\quad\quad\quad (\forall i < \mathit{hold}.\ \neg(\mathit{reset}(t_0+n+i+1))) \\
\quad\quad\quad \Rightarrow (\forall i \le \mathit{hold}.\ \mathit{dataout}(t_0+n+i) = f(\mathit{datain}(t_0))) \\
\quad))
\end{array}
$$

Abbildung 7.1: Definition der Schnittstellenverhaltensmuster

gangswerte x eingelesen wird, für diesen der Funktionswert $f(x)$ berechnet wird und das Ergebnis schließlich $n = 4$ Takte später auf den Ausgang gelegt wird. Die Kontrollsignale mit ihren booleschen Werten sind durch kleine Quadrate dargestellt, wobei gefüllte Quadrate für den Wert T stehen und ungefüllte Quadrate für den Wert F.

Alle vier Schaltungen haben eine Gemeinsamkeit: Sie benötigen genau n Takte, um eine Berechnung durchzuführen. Bei cs_pipeline beginnt zu jedem Zeitpunkt eine neue Berechnung, die n Takte später abgeschlossen ist. Die Aufträge werden parallel, versetzt bearbeitet (Pipelining). Bei cs_cycle, cs_start und cs_reset wird dagegen stets nur ein Auftrag gleichzeitig bearbeitet. Die nächste Berechnung beginnt erst nachdem die vorangegangene Berechnung abgeschlossen ist.

Alle Schnittstellenverhaltensmuster haben einen Dateneingang namens *datain*, über den zu bestimmten Zeitpunkten t die Eingangsdaten *datain*(t) eingelesen werden, und ein Ausgangssignal *dataout*, über das die Funktionswerte n Takte später ausgegeben werden. Zusätzlich besitzen die durch die Schnittstellenverhaltensmuster cs_cycle, cs_start und cs_reset spezifizierten Schaltungen Kontrollsignalleitungen, mit denen der aktuelle Zustand der Schaltung abgefragt und neue Berechnungen angestoßen werden können. Der Kontrollausgang *ready* signalisiert in allen drei Fällen mit dem Wert T, daß die letzte Berechnung abgeschlossen ist und daß der Funktionswert der letzten Berechnung am Ausgang anliegt. Hat *ready* den Wert F, so befindet sich die Schaltung gerade im rechnenden Zustand.

Die drei durch cs_cycle, cs_start und cs_reset spezifizierten Schaltungen unterscheiden sich darin, wie die nächste Berechnung angestoßen wird. Bei cs_cycle werden die Berechnungen unmittelbar hintereinander ausgeführt. Bei cs_start und cs_reset verharrt die Schaltung nach dem Abschluß einer Berechnung im Zustand *ready* und startet ihre nächste Berechnung erst dann, wenn sie dazu durch das *start*- bzw. *reset*-Signal angestoßen wird. Solange noch keine neue Berechnung angestoßen wurde, steht der letzte Funktionswert am Ausgang zur Verfügung. Die Kontrolleingänge *start* und *reset* von cs_start bzw. cs_reset unterscheiden sich in ihrer Wirkung für den Fall, daß sie während einer laufenden Berechnung gesetzt werden. Das *start*-Signal wird während einer laufenden Berechnung ignoriert. Das *reset*-Signal unterbricht eine laufende Berechnung und startet sofort die nächste Berechnung.

Es ist anzumerken, daß die vorgestellten Schnittstellenverhaltensmuster das Verhalten einer Schaltung nur partiell spezifizieren. Die Werte des Ausgangssignals *dataout* sind nur zu ganz bestimmten Zeitpunkten spezifiziert, und zwar dann, wenn ein gültiger Wert anliegt. In diesen Fällen liegt am Ausgang der Funktionswerte des um n Zeiteinheiten zurückliegenden Signal *datain* an. Ansonsten können beliebige Werte anliegen. Es gibt also jeweils mehrere Schaltungen mit unterschiedlichem synchronem Verhalten, die diese Spezifikationen erfüllen.

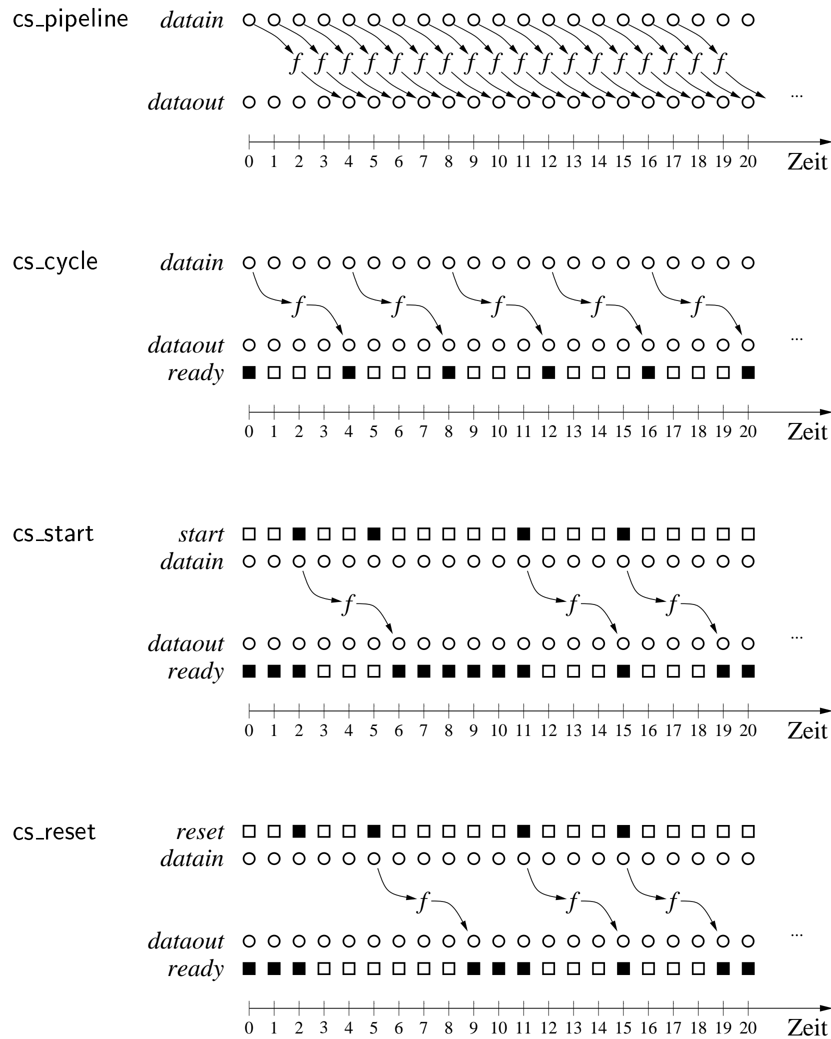


Abbildung 7.2: Beispielabläufe mit den Schnittstellenverhaltensmustern

7.3 Implementierungstheoreme

Es gilt nun, für eine vorgegebene Funktion f, eine beliebige aber feste Zahl n und ein Schnittstellenverhaltensmuster eine Implementierung zu finden. Daß es in einer solche Situation nicht nur eine mögliche Implementierung gibt, ist offensichtlich, denn die Ausführung der Funktion f muß auf mehrere Takte verteilt werden. Unterschiedliche Aufteilungen führen zu unterschiedlichen Implementierungen mit unterschiedlichen Kosten. Es wird deutlich, daß es i. allg. nicht sinnvoll ist, einem Schnittstellenverhaltensmuster genau eine Implementierung fest zuzuordnen.

Mit den Implementierungstheoremen, die in diesem Abschnitt vorgestellt werden, werden Implementierungen in generischer Weise beschrieben. Die Implementierungstheoreme beschreiben für ein Schnittstellenverhaltensmuster jeweils nicht nur eine feste Implementierung, sondern Implementierungen für beliebige Aufteilungen der Funktion f. Wie eine solche Aufteilung aus f gewonnen werden kann, soll im nachfolgenden Abschnitt erläutert werden. Der Grund dafür, daß zunächst die Implementierungstheoreme vorgestellt werden und erst im Anschluß erläutert wird, wie die Aufteilung von f durchgeführt werden kann, ist der, daß die Implementierungstheoreme sowohl Randbedingungen als auch Kostenfunktionen für die Aufteilung der Funktion f festlegen.

Die Implementierungstheoreme sind in mehrere Gruppen aufgeteilt, die jetzt in den folgenden Abschnitten vorgestellt werden sollen. Zunächst einmal ist der Spezialfall $n = 0$ zu nennen. Die Besonderheit ist hier, daß dadurch, daß die gesamte Funktion f innerhalb eines Taktes ausgeführt werden soll, weder die Funktion f aufgeteilt werden muß, noch müssen Speicherbausteine für die Speicherung von Zwischenergebnissen zur Verfügung gestellt werden. Bei den Implementierungen für $n > 0$ muß die Funktion in mehrere Teile aufgeteilt werden, und es entstehen Zwischenergebnisse, die zwischen den Takten in Speicherbausteinen gepuffert werden müssen. Das Schnittstellenverhaltensmuster cs_pipeline gestattet im Gegensatz zu den anderen Schnittestellenverhaltensmustern keine Wiederverwertung von Hardwareressourcen. Diese Problematik soll noch genauer erläutert werden. Als Konsequenz ergibt sich, daß die Implementierung $n > 0$ auf eine andere Art abgeleitet werden muß als die Implementierungen für die anderen Schnittstellenverhaltensmuster.

7.3.1 Implementierungstheoreme für $n = 0$

Die Theoreme, mit denen die Implementierungen für $n = 0$ abgeleitet werden können, sind in Abbildung 7.3 angegeben. Die Theoreme beschreiben jeweils eine Implikation aus RT-Ebenen-Implementierung und Schnittstellenverhaltensmuster mit Funktion f.

Die Implementierungen sind vergleichsweise einfach. Bei cs_pipeline und cs_cycle handelt es sich um Schaltnetze. Die Implementierungen von cs_start und cs_reset sind Schaltwerke, bei denen der Speicher lediglich die Aufgabe hat, die neu berechneten Ausgangswerte gemäß der Spezifikation des Schnittstellenverhaltensmusters solange am Ausgang zu halten, bis die nächste Berechnung gestartet

wird.

Das Einlesen des Datenwerts und das Schreiben des Funktionswerts erfolgt für $n = 0$ im gleichen Takt. Dies gilt unabhängig davon, welches der vier Schnittstellenverhaltensmuster verwendet wird. Im Gegensatz zu $n > 0$, müssen bei $n = 0$ keine Zwischenergebnisse von Takt zu Takt weitergereicht werden. Zur Berechnung des Funktionswerts wird jeweils nur eine einfache kombinatorische Einheit benötigt. Da bei den Schnittstellenverhaltensmustern cs_start und cs_reset zusätzlich gefordert wird, daß der einmal berechnete Funktionswert am Ausgang zur Verfügung steht bis der nächste Auftrag erteilt wird, enthalten diese Implementierungen zusätzlich ein Register zur Pufferung des Datenausgangs sowie einen Multiplexer, um zwischen dem neu berechneten Wert und dem gepufferten Wert hin- und herschalten zu können.

Für $n = 0$ unterscheiden sich die Spezifikationen von cs_start und cs_reset nicht. Die beiden Schnittstellenverhaltensmuster unterscheiden sich nur für den Fall, daß eine neue Berechnung angestoßen werden soll, während eine andere Berechnung, die ein oder mehrere Takte zuvor gestartet wurde, bereits läuft. Dieser Unterschied bezieht sich nur auf Schaltungen mit $n > 0$.

Deshalb kann man im Falle $n = 0$ für cs_start und für cs_reset die gleiche Implementierung verwenden. In der Implementierung taucht die freie Variable *inits* auf, mit der der Initialwert des Puffers beschrieben wird. Die beiden Theoreme besagen, daß die Implementierung die jeweilige Spezifikation für beliebige Werte von *inits* erfüllt, der Wert kann somit beliebig instantiiert werden.

7.3.2 Implementierungstheoreme für $n > 0$

Bevor für die einzelnen Schnittstellenverhaltensmuster Implementierungen für $n > 0$ vorgestellt werden, sollen zunächst einige allgemeine Voraussetzungen erläutert werden.

Im Fall $n > 0$ erfolgt die Berechnung des Funktionswerts in mehreren, aufeinanderfolgenden Takten. Die in dem DFG-Term f enthaltenen Operationen müssen auf die einzelnen Takte verteilt werden. Dazu soll f in Teilfunktionen $f = f_n \circ f_{n-1} \circ \ldots f_1 \circ f_0$, zerlegt werden, die dann jeweils in einzelnen Takten ausgeführt werden. Zu jedem Takt müssen die in den jeweiligen Teilfunktionen benötigten Operationseinheiten zur Verfügung gestellt werden. Dies bedeutet jedoch nicht, daß zu jeder Operation in f auch eine Operationseinheit realisiert werden müßte. Oft kommen innerhalb eines DFG-Terms mehrere Instanzen der gleichen Operation vor. Dann besteht die Möglichkeit, Operationseinheiten wiederzuverwenden. Wiederverwendung bedeutet, daß eine Operationseinheit in den einzelnen Takten zur Ausführung unterschiedlicher Instanzen der Operation verwendet wird.

Die vier vorgestellten Schnittstellenverhaltensmuster spezifizieren zwar lediglich eine Schaltung und beschreiben noch nicht ihre Struktur, sie führen jedoch schon eindeutig zu einer Implementierung mit bzw. ohne Wiederverwendung hin. Bei dem Schnittstellenverhaltensmuster cs_pipeline kann Wiederverwendung nicht

$\vdash$ $(dataout = \mathsf{combinatorial_block}\ f\ datain)$
$\Rightarrow$
$\mathsf{cs_pipeline}\ (0, f)\ (datain, dataout)$

$\vdash$ (
  $(\lambda t.\ (dataout(t), ready(t))) =$
    $\mathsf{combinatorial_block}\ (\lambda x.\ (\mathsf{T}, f(x)))\ datain$
)
$\Rightarrow$
$\mathsf{cs_cycle}\ (0, f)\ (datain, dataout, ready)$

$\vdash$ (
  $(\lambda t.\ (dataout(t), ready(t))) =$
    automaton
      (
        $(\ \lambda((a, b), c).\ \mathsf{let}\ x\ =\ \mathsf{MUX}(a, b, c)\ \mathsf{in}((\mathsf{T}, f(x)), x)\ )$
        ,
        $inits$
)
      $(\lambda t.\ (start(t), datain(t)))$
)
$\Rightarrow$
$\mathsf{cs_start}\ (0, f)\ (datain, start, dataout, ready)$

$\vdash$ (
  $(\lambda t.\ (dataout(t), ready(t))) =$
    automaton
      (
        $(\ \lambda((a, b), c).\ \mathsf{let}\ x\ =\ \mathsf{MUX}(a, b, c)\ \mathsf{in}((\mathsf{T}, f(x)), x)\ )$
        ,
        $inits$
)
      $(\lambda t.\ (reset(t), datain(t)))$
)
$\Rightarrow$
$\mathsf{cs_reset}\ (0, f)\ (datain, reset, dataout, ready)$

Abbildung 7.3: Implementierungstheoreme für $n = 0$

realisiert werden. Die Schnittstellenverhaltensmuster cs_cycle, cs_start und cs_reset eignen sich hingegen für Wiederverwendung.

Die zur Ableitung einer Implementierung bewiesenen Theoreme setzen voraus, daß der DFG-Term f bereits in mehrere Teildatenflußgraphen $f_n \circ f_{n-1} \circ \ldots f_1 \circ f_0$ aufgeteilt wurde. Da die Zwischenergebnisse von Teilfunktion zu Teilfunktion über die Taktgrenzen hinweg weitergereicht werden sollen, müssen für diese Zwischenergebnisse Register zur Verfügung gestellt werden. Die Typen der Zwischenergebnisse seien $\sigma_1, \sigma_2, \ldots \sigma_n$. f_0 hat den Typ $\iota \to \sigma_1$, f_n hat den Typ $\sigma_n \to \omega$ und alle anderen Teilfunktionen f_i mit $0 < i < n$ haben den Typ $\sigma_i \to \sigma_{i+1}$.

Auch bei den Registern ist eine Wiederverwendung möglich, wenngleich sie im Fall cs_pipeline nicht sinnvoll ist. Für alle anderen Schnittstellenverhaltensmuster kann so die Anzahl der benötigten Register minimiert werden. Für die Implementierungen der Schnittstellenverhaltensmuster cs_cycle, cs_start und cs_reset soll deshalb auch bei Registern Wiederverwendung angestrebt werden. Dazu soll die Aufteilung des DFG-Graphen so durchgeführt werden, daß die Typen der Zwischenwerte σ_i mit $1 \leq i < n$ alle gleich σ sind.

Implementierungstheoreme zu cs_cycle, cs_start und cs_reset für $n > 0$

Die Implementierungstheoreme, die nachfolgend vorgestellt werden, setzen voraus, daß der Datenflußgraph f in der Form $h \circ (\mathsf{concat}(L)) \circ g$ vorliegt, wobei n den Wert $\mathsf{LENGTH}(L) + 1$ hat. Die ursprünglich vorgegebene Funktion f wird hier durch ein Tripel (h, L, g) repräsentiert. Dabei entsprechen $h = f_n$, $g = f_0$ und $L = [f_1, \ldots, f_{n-1}]$. Die Funktion concat steht für die Konkatenation aller in der Liste enthaltenen Funktionen $f_{n-1} \circ \ldots \circ f_1$. Auf die Definition dieser Funktion wird nachfolgend noch eingegangen (siehe auch Abbildung 7.4). Alle Elemente der Liste L haben den gleichen Typ $\sigma \to \sigma$, und auch die Gesamtfunktion $\mathsf{concat}([f_{n-1}, \ldots, f_2])$ hat den Typ $\sigma \to \sigma$. Es wird für diese Implementierungen also gefordert, daß die Funktion f so in Teilfunktionen zerlegt wird, daß alle Zwischenergebnisse den gleichen Typ σ haben.

Die Implementierungen für cs_cycle, cs_start und cs_reset bauen auf den Komponenten in Abbildung 7.5 auf. Die Komponente data_flow beschreibt eine Datenflußeinheit, bei der in Abhängigkeit eines Steuersignals q die Funktion f_q ausgeführt wird. Im Fall $q = 0$ wird der Wert von der Eingabe gelesen, darauf f_0 angewandt und in einem Register abgespeichert. In den Fällen $q = 1 \ldots n - 1$ wird der aktuelle Registerwert durch Anwendung der Funktion f_q auf einen neuen Registerwert abgebildet, und für $q = n$ wird der aktuelle Registerwert durch Anwendung von f_n auf den Ausgangswert abgebildet. Solange die Steuervariable q den Wert n hält, steht am Ausgang das Ergebnis der letzten Berechnung zur Verfügung. Für $q \neq n$ liegt am Ausgang ein beliebiger Wert *defo* an. Da zu jedem Zeitpunkt immer nur eine der Funktionen $f_0, \ldots, f_n$ ausgeführt werden muß, benötigt die Implementierung data_flow von jedem Operationstyp jeweils nur die maximale Anzahl der in $f_0, \ldots, f_n$ von dieser Operation enthaltenen Instanzen.

Die Komponente genimp ist eine generische Implementierung für alle drei Schnittstellenverhaltensmuster. Sie setzt sich aus der Datenflußeinheit data_flow,

$$
\begin{aligned}
\mathsf{LENGTH}([]) &:= 0 \\
\mathsf{LENGTH}(\mathsf{CONS}\, h\, t) &:= 1 + \mathsf{LENGTH}(t) \\[1ex]
\mathsf{APPEND}\, []\, b &:= b \\
\mathsf{APPEND}\, (\mathsf{CONS}\, h\, t)\, b &:= \mathsf{CONS}\, h\, (\mathsf{APPEND}\, t\, b) \\[1ex]
\mathsf{MAP}\, f\, [] &:= [] \\
\mathsf{MAP}\, f\, (\mathsf{CONS}\, h\, t) &:= \mathsf{CONS}\, (f(h))\, (\mathsf{MAP}\, f\, t) \\[1ex]
\mathsf{concat}\, [] &:= (\lambda x.\, x) \\
\mathsf{concat}\, (\mathsf{CONS}\, h\, t) &:= (\mathsf{concat}\, t) \circ h
\end{aligned}
$$

Abbildung 7.4: Hilfsfunktionen

einer Kontrolleinheit *control* sowie einer EQ-Komponente zusammen. Dabei ist *control* ein Parameter von genimp. *control* steuert mit seinem Ausgang vom Typ enum_{n+1} die Datenflußeinheit data_flow. Die Typen für Eingang und Zustand von *control* sind variabel. Die Komponente *control* hat selbst einen Parameter m. Diesem wird in der Struktur von genimp der Wert $\mathsf{LENGTH}(L)+2 = n+1$ übergeben. Die Implementierungen für die drei Schnittstellenverhaltensmuster unterscheiden sich lediglich im Aufbau der Kontrolleinheit *control*. Je nach gewünschtem Schnittstellenverhalten wird der generischen Komponente genimp eine der Kontrolleinheiten cycle_control, start_control bzw. reset_control übergeben.

In den Implementierungen werden fünf Funktionen verwendet, die nicht zu den bisher vorgestellten Mitteln zur Schaltungsbeschreibung gehören: LENGTH, APPEND, CONS, MAP und concat. Diese Funktionen sind durch primitive Rekursion über Listen definiert (siehe Abbildung 7.4).

LENGTH bestimmt die Länge einer Liste, APPEND kettet zwei Listen aneinander, CONS hängt ein einzelnes Element vorne an eine Liste an, und MAP wendet eine Funktion auf jedes Element einer Liste an. Die Funktion concat beschreibt die Konkatenation einer Liste von Funktionen. Für den Fall einer leeren Liste von Funktionen ist er definiert als die identische Funktion $(\lambda x.\, x)$. Im anderen Fall, wenn die Liste also aus mindestens einem Element h und einem Rest t besteht, dann ergibt sich die Funktion, die entsteht, wenn man concat t mit h konkateniert.

Diese Funktionen beziehen sich durchweg auf die Liste L. Man muß sich vor Augen halten, daß für L eine wohlbekannte Liste $[f_1, \ldots, f_{n-1}]$ mit einer endlichen Anzahl von Elementen instantiiert wird. Sobald die Liste L instantiiert wurde, können die oben aufgeführten Funktionen direkt ausgewertet werden und verschwinden damit aus der Beschreibung der Implementierung. Dies geschieht durch einfache Termersetzung mit den Definitionen dieser Funktionen (Abbildung 7.4).

$$\mathsf{CASE}\ l\ i \quad :\hat{=} \quad \mathsf{pick}\ (\mathsf{LENGTH}(l))\ (i, \mathsf{mkarray}(l))$$

$$\mathsf{data_flow}\ \mathit{defo}\ g\ L\ h\ ((i,q),y) \quad :\hat{=}$$
$$(\ (\ \mathsf{if}\ (q = \mathsf{LENGTH}(L)+1)\ \mathsf{then}\ h(y)\ \mathsf{else}\ \mathit{defo}\ )\ ,\ (\ \mathsf{CASE}(\mathsf{APPEND}\ (\mathsf{CONS}\ (g(i))\ (\mathsf{MAP}(\lambda p.\, p(y))L))\ [y])\ q\ )\ )$$

$$\mathsf{genimp}\ \mathit{defo}\ g\ L\ h\ \mathit{control}\ ((\mathit{cin}, \mathit{din}), (\mathit{cs}, \mathit{ds})) \quad :\hat{=}$$
$$\mathsf{let}\ ((\mathit{cout}, c), \mathit{cs'}) \ = \ \mathit{control}\, (\mathsf{LENGTH}(L)+2)\ (\mathit{cin}, \mathit{cs})\ \mathsf{in}$$
$$\mathsf{let}\ (\mathit{dout}, \mathit{ds'}) \ = \ \mathsf{data_flow}\ \mathit{defo}\ (g, L, h)\ ((\mathit{din}, c), \mathit{ds})\ \mathsf{in}$$
$$(\ (\mathsf{EQ}(\mathit{cout}, \mathsf{maxenum}\ \mathsf{LENGTH}(L)+2), \mathit{dout}),\ (\mathit{cs'}, \mathit{ds'})\ )$$

$$\mathsf{cycle_control}\, m\ (c_{\mathsf{one}}, q) \quad :\hat{=}$$
$$\mathsf{let}\ q' = \mathsf{next}\, m\ (q)\ \mathsf{in}\ ((q,q'),q')$$

$$\mathsf{start_control}\, m\ (c,q) \quad :\hat{=}$$
$$\mathsf{let}\ q' = \mathsf{MUX}(\mathsf{EQ}(q, \mathsf{maxenum}\ m), \mathsf{MUX}(c, \mathsf{enum}\ m\ 0, \mathsf{maxenum}\ m), \mathsf{next}\ m\ q))\ \mathsf{in}\ ((q,q'),q')$$

$$\mathsf{reset_control}\, m\ (c,q) \quad :\hat{=}$$
$$\mathsf{let}\ q' = \mathsf{MUX}(c, \mathsf{enum}\ m\ 0, \mathsf{MUX}(\mathsf{EQ}(q, \mathsf{maxenum}\ m), \mathsf{maxenum}\ m, \mathsf{next}\ m\ q))\ \mathsf{in}\ ((q,q'),q')$$

Abbildung 7.5: Hilfskomponenten

$$
\begin{array}{ll}
\vdash & (\\
& \quad (\lambda t.\,(dataout(t), ready(t))) = \\
& \quad\quad \mathsf{automaton} \\
& \quad\quad\quad (\\
& \quad\quad\quad\quad (\mathsf{genimp}\ defo\ g\ L\ h\ \mathsf{cycle_control}), \\
& \quad\quad\quad\quad (\mathsf{maxenum}\ (\mathsf{LENGTH}(L) + 2), initds) \\
& \quad\quad\quad) \\
& \quad\quad\quad (\lambda t.\,(dummy(t), datain(t))) \\
&) \\
& \Rightarrow \\
& \mathsf{cs_cycle}\ (\ \mathsf{LENGTH}(L) + 1,\ h \circ (\mathsf{concat}(L)) \circ g\ ) \\
& \quad (datain, dataout, ready)
\end{array}
$$

$$
\begin{array}{ll}
\vdash & (\\
& \quad (\lambda t.\,(dataout(t), ready(t))) = \\
& \quad\quad \mathsf{automaton} \\
& \quad\quad\quad (\\
& \quad\quad\quad\quad (\mathsf{genimp}\ defo\ g\ L\ h\ \mathsf{start_control}), \\
& \quad\quad\quad\quad (\mathsf{maxenum}\ (\mathsf{LENGTH}(L) + 2), initds) \\
& \quad\quad\quad) \\
& \quad\quad\quad (\lambda t.\,(start(t), datain(t))) \\
&) \\
& \Rightarrow \\
& \mathsf{cs_start}\ (\ \mathsf{LENGTH}(L) + 1,\ h \circ (\mathsf{concat}(L)) \circ g\ ) \\
& \quad (datain, start, dataout, ready)
\end{array}
$$

$$
\begin{array}{ll}
\vdash & (\\
& \quad (\lambda t.\,(dataout(t), ready(t))) = \\
& \quad\quad \mathsf{automaton} \\
& \quad\quad\quad (\\
& \quad\quad\quad\quad (\mathsf{genimp}\ defo\ g\ L\ h\ \mathsf{reset_control}), \\
& \quad\quad\quad\quad (\mathsf{maxenum}\ (\mathsf{LENGTH}(L) + 2), initds) \\
& \quad\quad\quad) \\
& \quad\quad\quad (\lambda t.\,(reset(t), datain(t))) \\
&) \\
& \Rightarrow \\
& \mathsf{cs_reset}\ (\ \mathsf{LENGTH}(L) + 1,\ h \circ (\mathsf{concat}(L)) \circ g\ ) \\
& \quad (datain, reset, dataout, ready)
\end{array}
$$

Abbildung 7.6: Implementierungstheoreme zu cs_cycle, cs_start und cs_reset für $n > 0$

Implementierungstheoreme zu cs_pipeline für $n > 0$

Ausgangspunkt für eine Implementierung zu cs_pipeline ist ein Term der Form $f = f_n \circ \ldots \circ f_0$. Für die Zwischenergebnisse wird bewußt nicht gefordert, daß deren Typen gleich sind. Anders als bei den anderen Schnittstellenprotokollen, wo bei diesem Term gefordert wurde, daß die Zwischenergebnisse alle den gleichen Typ haben sollen, gibt es hier keine geschlossene Darstellung für ein Implementierungstheorem.

Anstatt nun ein Theorem mit einer allgemeinen Implementierung anzugeben, werden in Abbildung 7.7 zwei Theoreme angegeben, mit denen Theoreme für ein jeweils beliebiges, aber festes n abgeleitet werden können. Das erste Theorem beschreibt eine Implementierung für $n = 1$. Das zweite Theorem leitet aus einer Implementierung für ein beliebiges n eine Implementierung für $n + 1$ ab. Ausgehend von dem ersten Theorem können so mit Hilfe des zweiten Theorems durch sukzessive Anwendung von Modus Ponens Implementierungstheoreme für beliebige n abgeleitet werden.

$$
\begin{array}{ll}
\vdash & \forall datain, dataout. \\
& \quad (\ dataout = \mathsf{automaton}\ (\ (\lambda(i,s).\,(h(s), g(i))),\ initds)\ datain\ ) \\
& \quad \Rightarrow \\
& \quad \mathsf{cs_pipeline}\ (\ 1\ ,\ h \circ g\ )\ (datain, dataout)
\end{array}
$$

$$
\begin{array}{ll}
\vdash & (\\
& \quad \forall datain, dataout. \\
& \quad\quad (\ dataout = \mathsf{automaton}\ (f, initds)\ datain\ ) \\
& \quad\quad \Rightarrow \\
& \quad\quad \mathsf{cs_pipeline}\ (\ n\ ,\ h\ )\ (datain, dataout) \\
&) \\
& \Rightarrow \\
& (\\
& \quad \forall datain, dataout. \\
& \quad\quad (\ dataout = \\
& \quad\quad\quad \mathsf{automaton}\ (\\
& \quad\quad\quad\quad (\lambda(i,(s,s')).\,\mathsf{let}\ a = g(i)\ \mathsf{in}\ \mathsf{let}\ (b,c) = h(s,s')\ \mathsf{in}\ (b,(a,c))\ ) \\
& \quad\quad\quad\quad , \\
& \quad\quad\quad\quad (initds', initds) \\
& \quad\quad\quad)\ datain \\
& \quad\quad) \\
& \quad\quad \Rightarrow \\
& \quad\quad \mathsf{cs_pipeline}\ (\ n+1\ ,\ h \circ g\ )\ (datain, dataout) \\
&)
\end{array}
$$

Abbildung 7.7: Implementierungstheoreme zu cs_pipeline für $n > 0$

$$
\begin{array}{l}
(\lambda(a,b,c). \\
\quad \mathsf{let}\ p\ =\ \mathsf{MULT}(a,b)\ \mathsf{in} \\
\quad \mathsf{let}\ s\ =\ \mathsf{ADD}(b,c)\ \mathsf{in} \\
\quad \mathsf{let}\ t\ =\ \mathsf{SUB}(p,s)\ \mathsf{in} \\
\quad \mathsf{let}\ q\ =\ \mathsf{ADD}(s,c)\ \mathsf{in} \\
\quad \mathsf{let}\ r\ =\ \mathsf{MULT}(p,q)\ \mathsf{in} \\
\quad \mathsf{let}\ x\ =\ \mathsf{ADD}(r,t)\ \mathsf{in} \\
\quad \mathsf{let}\ y\ =\ \mathsf{MULT}(r,t)\ \mathsf{in} \\
\quad\quad (x,y)\ )
\end{array}
\quad = \quad
\begin{array}{l}
(\lambda(a,b,c). \\
\quad \mathsf{let}\ s\ =\ \mathsf{ADD}(b,c)\ \mathsf{in} \\
\quad\quad (a,b,s,c)\ ) \\
\circ \\
(\lambda(a,b,s,c). \\
\quad \mathsf{let}\ p\ =\ \mathsf{MULT}(a,b)\ \mathsf{in} \\
\quad \mathsf{let}\ q\ =\ \mathsf{ADD}(s,c)\ \mathsf{in} \\
\quad\quad (p,q,s,z_1)\ ) \\
\circ \\
(\lambda(p,q,s,z_1). \\
\quad \mathsf{let}\ r\ =\ \mathsf{MULT}(p,q)\ \mathsf{in} \\
\quad \mathsf{let}\ t\ =\ \mathsf{SUB}(p,s)\ \mathsf{in} \\
\quad\quad (r,t,z_2,z_3)\ ) \\
\circ \\
(\lambda(r,t,z_2,z_3). \\
\quad \mathsf{let}\ x\ =\ \mathsf{ADD}(r,t)\ \mathsf{in} \\
\quad \mathsf{let}\ y\ =\ \mathsf{MULT}(r,t)\ \mathsf{in} \\
\quad\quad (x,y)\ )
\end{array}
$$

Abbildung 7.8: Beispiel für die Aufteilung eines DFG-Terms

7.4 Die Aufteilung des DFG-Terms und die Qualität der Implementierung

Die Anwendung der Theoreme zur Ableitung von Implementierungen setzen voraus, daß der DFG-Term f zunächst in mehrere Teilfunktionen $f = f_n \circ f_{n-1} \circ \ldots f_1 \circ f_0$ aufgeteilt wurde. Zur Aufteilung des DFG-Terms f sind grundlegend verschiedene Verfahren notwendig, je nachdem, ob das Schnittstellenverhaltensmuter cs_pipeline oder eines der drei Schnittstellenverhaltensmuster cs_cycle, cs_start oder cs_reset angestrebt wird. Zum einen betrifft dies die Form des Ergebnisses, bei dem für cs_cycle, cs_start oder cs_reset die Einschränkung gemacht wird, daß alle Zwischenergebnisse den gleichen Typ haben, und zum anderen sind auch die bei der Implementierung entstehenden Kosten unterschiedlich zu beurteilen.

Die Gleichung in Abbildung 7.8 beschreibt einen derartigen Aufteilungsschritt, der zur Synthese eines der Schnittstellenverhaltensmuster cs_cycle, cs_start oder cs_reset verwendet werden kann. Die linke Seite beschreibt den Zustand vor der Aufteilung des DFG-Terms f, die rechte Seite den Zustand danach. Nach der Aufteilung besteht der DFG-Term aus 4 DFG-Termen f_0, f_1, f_2 und f_3, die durch Konkatenation $f_3 \circ f_2 \circ f_1 \circ f_0$ miteinander verbunden sind. Eine graphische Darstellung der Gleichung aus Abbildung 7.8 findet man in Abbildung 7.9: der Zustand I steht für die linke Seite der Gleichung und der Zustand IV für die rechte Seite.

Die verwendeten Grundoperationen sind mit MULT, ADD und SUB bezeichnet. Der Einfachheit halber sind die Typen der Datenwerte in diesem Beispiel

alle gleich, was bei dem hier vorgestellten Ansatz jedoch keineswegs eine Voraussetzung ist. Der in diesem Beispiel durchgeführte Zerteilungsschritt führt zu einem Syntheseergebnis mit einer guten Qualität. Wie ein derartiger Zerteilungsschritt in der Logik durchgeführt werden kann und wie dabei Kostengesichtspunkte berücksichtigt werden können, soll im folgenden erläutert werden.

Die wesentlichen Größen zur Beurteilung der Qualität des Syntheseergebnisses sind:

- der Aufwand an Hardwarekomponenten
- die Anzahl der Takte n, die zur Ausführung von f benötigt wird
- der Durchsatz in ausgeführten Funktionen pro Zeiteinheit
- die kombinatorische Tiefe bzw. die erzielbare Taktfrequenz

Die Aufteilung des DFG-Terms hat maßgeblichen Einfluß auf die erzielbare Qualität des Synhteseergebnisses. Es muß einschränkend gesagt werden, daß in diesem frühen Stadium der Schaltungssynthese die oben genannten Kosten natürlich nur grob abgeschätzt werden können. In der Regel wird versucht, die Hardwarekosten durch die Anzahl der benötigten Operationseinheiten und Register abzuschätzen. Weitere Kosten, die etwa durch die Generierung von Kontrolleinheiten oder durch die Kommunikationssynthese entstehen, werden in der Regel vernachlässigt oder nur grob abgeschätzt. Auch sind in diesen Kostenschätzungen weitere Optimierungen auf tieferen Ebenen (Retiming, Logikminimierung etc.) nicht berücksichtigt, geschweige denn genaue Verzögerungszeiten, wie sie sich oft erst nach der Plazierung und Verdrahtung ergeben.

Trotzdem fallen bei der High-Level-Synthese bereits wesentliche Entscheidungen im Spannungsfeld zwischen Laufzeiteffizienz und Hardwarebedarf. Die Teilentscheidungen, die sich bei der Aufspaltung eines DFG-Terms ergeben, sind:

1. Ablaufplanung (scheduling): Welche Operation wird zu welcher Zeitphase ausgeführt? Die Anzahl der Takte ist dabei entweder vorgegeben oder muß ebenfalls während dieses Schritts bestimmt werden.

2. Allokierung (allocation): Wieviele Operationseinheiten und Register müssen zu welchem Taktschritt zur Verfügung gestellt werden?

3. Zuordnung (binding): Welche Operation wird welcher Operationseinheit und welche Hilfsvariable wird welchem Register zugeordnet?

Ablaufplanung und Allokierung bestimmen den Hardwareaufwand in bezug auf die Anzahl der benötigten Operationseinheiten und Register. Gleichzeitig wird aber auch — und diese Kosten konkurrieren hier — die Anzahl n der benötigten Takte und die kombinatorische Tiefe der Schaltung festgelegt. Durch geschickte Zuordnung kann der Kommunikationsaufwand zwischen den Operationseinheiten untereinander sowie mit den Registern optimiert werden.

Für diese Syntheseschritte gibt es i. allg. keine optimale Lösung und zwar schon deshalb nicht, weil die verschiedenen Kosten miteinander konkurrieren und deshalb i. allg. nicht gleichzeitig ihr absolutes Minimum erreichen. Der Entwerfer muß zwischen den verschiedenen Forderungen abwägen. In der Literatur existieren verschiedene automatisierte Heuristiken wie List-Scheduling und Force-Directed-Scheduling [GDWL94, CaWo91]. Im wesentlichen unterscheiden sich die Verfahren darin, daß der Entwerfer entweder die Kosten für die Hardware oder zeitliche Beschränkungen vorgibt. Die Verfahren versuchen dann, unter diesen Beschränkungen die jeweils anderen Kosten zu minimieren.

Die gängigen Verfahren können auch bei der Formalen Synthese zum Einsatz kommen. Eine sehr einfache Möglichkeit, den Syntheseschritt durch logische Umformungen auszudrücken, besteht darin, daß man zunächst zu dem vorgegebenen DFG-Term f mit konventionellen Verfahren Ablaufplan, Allokierung und Zuordnung berechnet, dies dann umsetzt, indem man den Term f entsprechend in seine Teile $f_n \circ \ldots \circ f_0$ zerlegt und schließlich das Theorem $\vdash f = f_n \circ \ldots \circ f_0$ logisch ableitet. Eine Möglichkeit dieses Theorem abzuleiten besteht darin, zunächst den Term aufzustellen und ihn danach zu beweisen. Der Beweis von $f = f_n \circ \ldots \circ f_0$ ist vergleichsweise einfach dadurch möglich, daß man beide Seiten normiert (siehe Abschnitt 4.1.2). Bei größeren DFG-Termen und häufiger Mehrfachverwendung von Zwischenergebnissen kann es dabei jedoch zu einem exponentiellen Anwachsen des normierten Ausdrucks und auch der Zeit für die Normierung kommen. Effizienter ist es dann, die Abspaltung der Teilfunktionen einzeln vorzunehmen und immer nur die let-Ausdrücke der angehängten Teilfunktion zu expandieren.

In Abbildung 7.9 sind die Schritte beschrieben, mit denen eine Aufteilung wie in Abbildung 7.8 erzielt werden kann. Die Grundoperationen ADD, MULT und SUB sind der Übersicht halber mit $+$, $*$ beziehungsweise $-$ dargestellt. Bei der Ablaufplanung wurde hier die Entscheidung getroffen, auf Chaining zu verzichten. Unter Chaining versteht man die Hintereinanderschaltung von Operationseinheiten innerhalb eines Taktes. Dies hat zur Konsequenz, daß lediglich Kommunikationseinheiten zwischen Registern und Operationseinheiten, jedoch keine Kommunikationseinheiten zwischen den Operationseinheiten untereinander benötigt werden. Die Operationen wurden so auf die Takte verteilt, daß die Schaltung mit je einem ADD-, MULT- und SUB-Baustein sowie mit vier Registern auskommt (Allokierung). Die Register sind in Abbildung 7.9 mit schwarzen Quadraten angedeutet. Die zur Verfügung zu stellenden Hardwareressourcen, Operationseinheiten und Register, werden nicht in jedem Takt benötigt. Der Übersicht halber sind in Abbildung 7.9 lediglich die unbenutzten Register, nicht jedoch die unbenutzten Operationseinheiten dargestellt. Bei der Zuordnung werden den Operationen und Zwischenergebnissen konkrete Instanzen von Operationseinheiten bzw. Registern zugeordnet. Da in dem Beispiel jeweils von jedem Operationstyp genau eine Operationseinheit zur Verfügung gestellt wird, ist hier die Zuordnung eindeutig. Bei der Registerzuordnung können Spielräume genutzt werden. Die Zahlen in den Registern bezeichnen die Nummern der Instanzen der Register. So wird bei der hier durchgeführten Zuordnung das Zwischenergebnis s, das zweimal hintereinader ge-

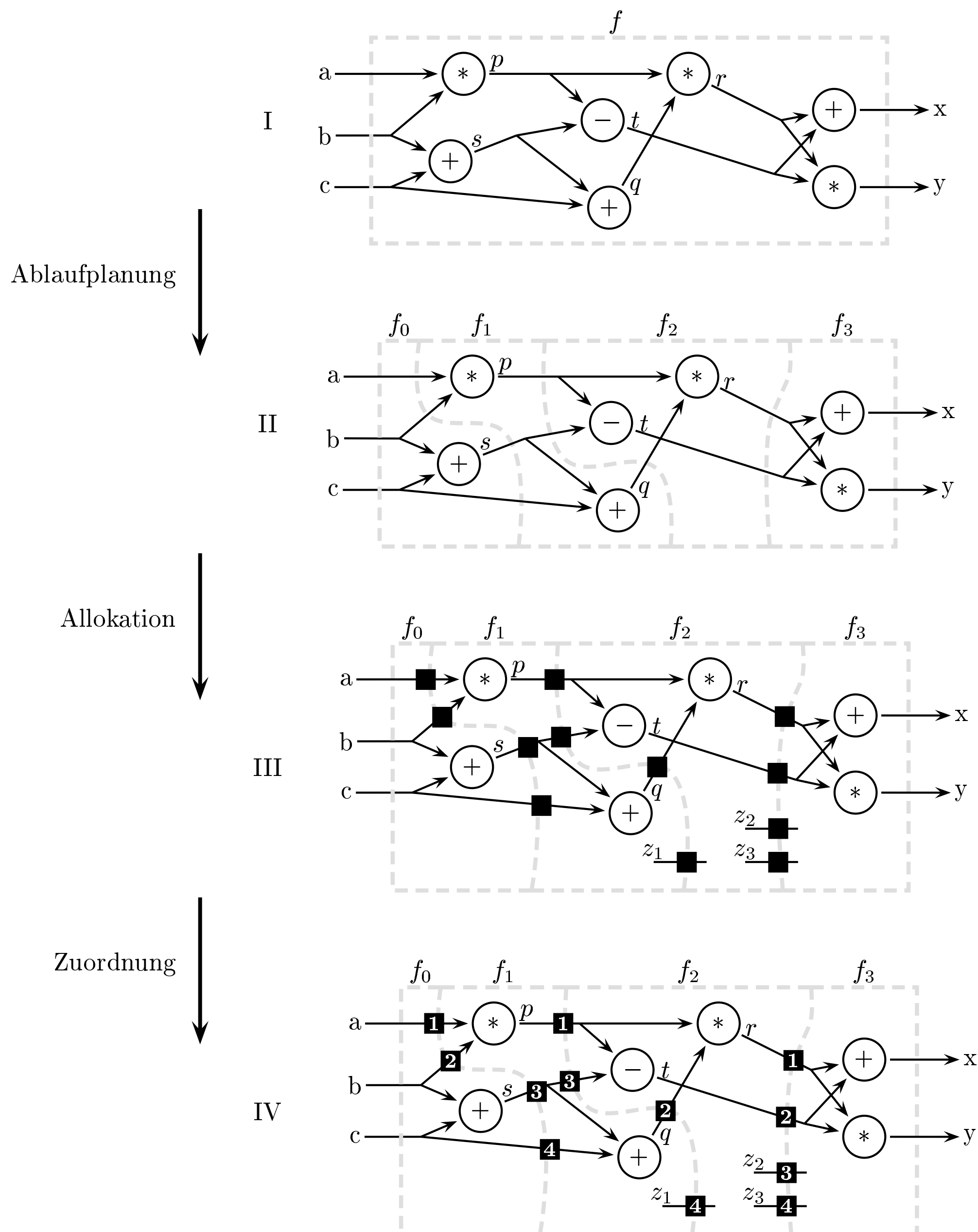


Abbildung 7.9: Schritte der High-Level Synthese

speichert werden muß, immer dem gleichen Register zugeordnet. Dadurch wird ein unnötiger Kommunikationsaufwand vermieden, der bei einem Transport des Wertes von einem zum nächsten Register entstehen würde. In Abbildung 7.8 sind die Zwischenergebnisse durch Quadrupel dargestellt. Die Registerzuordnung kommt durch die Anordnung der Variablen innerhalb des Quadrupels zum Ausdruck.

7.5 Entwurfsraumuntersuchung und logische Transformation

Es wurde ein Syntheseschritt vorgestellt, bei dem die Kostensituation kompliziert ist: Die Kosten sind einerseits zu diesem frühen Synthesestadium nur ungenau abschätzbar, und andererseits konkurrieren die verschiedenen Kosten miteinander. Dies führt dazu, daß in den Syntheseprogrammen komplexe Heuristiken zum Einsatz kommen. Aufgrund der hohen Kosten einer nachträglichen Verifikation stellt sich hier in besonderer Weise die Forderung, die Korrektheit der Synthese zu gewährleisten.

Mit dem hier vorgestellten Verfahren ist es möglich, die Implementierung in der Logik in korrekter Weise durchzuführen. Dabei können die bisher bereits implementierten Syntheseverfahren voll zum Einsatz kommen. Die wesentlichen Entscheidungen spiegeln sich hier in der Aufteilung des DFG-Terms wider. Die Aufteilung des DFG-Terms kann aus Ablaufplan, Allokierung und Zuordnung, wie sie in klassischen Syntheseverfahren bestimmt werden, direkt abgeleitet werden. Es liegt deshalb nahe, analog zum Retiming, den Syntheseschritt in zwei Phasen, nämlich Entwurfsraumuntersuchung und logische Transformation, aufzuteilen (siehe Abbildung 7.10). Bei der Entwurfsraumuntersuchung werden mit geeigneten Heuristiken Ablaufplan, Allokierung und Zuordnung für den DFG-Term bestimmt. Die nachgeschaltete logische Transformation teilt den DFG-Term entsprechend auf und leitet daraus mit Hilfe der oben beschriebenen Theoreme eine Implementierung auf RT-Ebene ab.

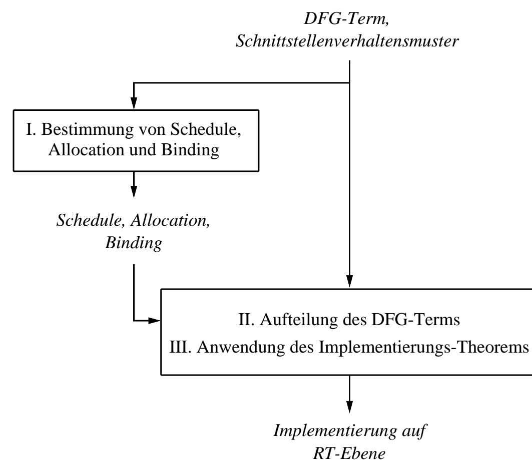


Abbildung 7.10: Entwurfsraumuntersuchung und Schaltungstransformation bei der High-Level Synthese

Kapitel 8

Experimentelle Ergebnisse

Die Akzeptanz eines Systems zur Formalen Synthese hängt unter anderem auch davon ab, wie effizient die Schaltungstransformationen durchgeführt werden können. Hier konkurriert das Konzept der Formale Synthese mit einer nachgeschalteten Post-Synthese-Verifikation. Am Beispiel des Retiming-Syntheseschritts sollen anhand experimenteller Ergebnisse die Vorteile des verwendeten Ansatzes erläutert werden.

8.1 Bedeutung von Retiming

Bei vielen älteren Syntheseprogrammen ist die Kodierung der Zustände fest vorgegeben. Die Synthese besteht lediglich darin, die Signal- und Zustandswerte nach einem starren Schema zu kodieren und die verwendeten Operatoren direkt auf Elemente der Schaltungsbibliothek abzubilden oder sie mit Hilfe von Modulgeneratoren erzeugen zu lassen. Schließlich wird die so erzeugte Schaltung noch auf boolescher Ebene optimiert. Die vom Entwerfer vorgegebenen nichtbooleschen Datentypen werden dabei in einer starr vorgegebenen Weise kodiert. Während des Optimierungsvorgangs wird lediglich der kombinatorische Teil der Schaltung verändert, wohingegen die Zustandsrepräsentation nicht verändert wird.

Für den modernen Schaltungsentwurf ist diese Beschränkung oft nicht mehr zu akzeptieren. Insbesondere Verfahren zur Reduktion des Energieverbrauchs verlassen den Bereich der rein booleschen Optimierung [MSBS91, FeRF93, Mart96]. Durch eine veränderte Zustandsrepräsentation, wie sie durch Retiming- oder Reencoding-Verfahren erzeugt wird, ist es möglich, Schaltungen zu synthetisieren, die mit einer im Mittel geringeren Anzahl von Zustandswechseln bei den Speicherbausteinen und damit mit einem geringeren Energieverbrauch auskommen.

Ein großes Problem beim Einsatz derartiger Verfahren ist die Überprüfung des Syntheseergebnisses. Die Syntheseabläufe sind oft sehr komplex und allein anhand des Syntheseergebnisses nicht mehr nachvollziehbar. Verschiedene Syntheseverfahren kombinieren „sequentielle” Transformationen wie Retiming oder Reenco-

ding mit klassischen kombinatorischen Optimierungsverfahren. In [FeRF93] wird beispielsweise ein Verfahren vorgestellt, bei dem Retiming- und Logikminimierungsschritte abwechselnd und iterativ durchgeführt werden. Der Entwurf und das von dem Syntheseprogramm erzeugte Ergebnis haben nur noch gemein, daß das Verhalten an der Schnittstelle übereinstimmt. Entwurfshierarchien werden aufgelöst, interne Variablennamen werden umbenannt oder verschwinden. Die internen Abläufe sind nicht mehr miteinander vergleichbar. Da der Zusammenhang zwischen der ursprünglichen Zustandsrepräsentation und der während der Synthese erzeugten Kodierung verloren geht, sind die Schaltungen nur noch an den Außenanschlüssen miteinander vergleichbar. Dies erschwert den Vergleich durch Simulation erheblich, und auch im Betrieb sind so erzeugte Schaltungen nur schwer beobachtbar. Für größere Schaltungen lassen sich in bezug auf die Korrektheit allein durch Simulation nur sehr vage Vermutungen anstellen. Vor diesem Hintergrund bekommt die Sicherstellung der Korrektheit bereits durch die Synthese eine besondere Bedeutung.

8.2 Konzeptionelle Unterschiede der Ansätze

Zur Problematik der Korrektheit des Retiming-Syntheseschritts gibt es bereits verschiedene Arbeiten. Der eigene Ansatz soll mit drei weiteren, konzeptionell anderen Ansätzen verglichen werden:

1. die Systeme SMV [McMi92b] bzw. SIS [SSLM92], die zur Verifikation allgemeine Techniken einsetzen

2. das Verfahren von van Eijk zur Optimierung des Model-Checking, das sich für Retiming besonders eignet [EiJe96, Eijk97]

3. das Verfahren von Huang et. al., das speziell zur Verifikation von Retiming-Schritten entwickelt wurde. [HuCC96]

Die Systeme SMV und SIS stellen ganz allgemeine Verifikationsverfahren dar, mit denen beliebige Schaltungen miteinander verglichen werden können. Das Verfahren von van Eijk ist ebenfalls allgemein einsetzbar, es wurde jedoch auf die Problematik des Retiming hin optimiert und liefert deshalb für diese Problemstellungen besonders gute Ergebnisse. Alle drei Ansätze SMV, SIS und das System von van Eijk haben den Anspruch, einen durchgängigen Beweis zu führen. Diesen Anspruch hat das Verfahren von Huang nicht. Das Verfahren von Huang möchte lediglich für einen ganz bestimmten Syntheseschritt, den Retiming-Schritt, versuchen, die Korrektheit möglichst effizient dadurch sicherzustellen, daß ein selbstgeschriebenes Prüfprogramm die Stimmigkeit des Ergebnisses überprüft. Daß die Kriterien, die das Prüfprogramm untersucht, hinreichend für die Korrektheit des Retiming-Schritts sind, wird nicht bewiesen. Ein durchgängiger Beweis liegt somit nicht vor.

Mit den Werkzeugen SMV und SIS ist es möglich, beliebige Syntheseschritte auf der Gatterebene zu verifizieren. Die Verfahren basieren auf Model-Checking-Techniken, bei denen die Äquivalenz der Schaltungen durch Zustandstraversierung bewiesen wird. Typischerweise werden bei der Zustandstraversierung zur Repräsentation von Schaltungsrelationen und Zustandsmengen BDDs oder ähnliche Entscheidungsdiagramme verwendet. Die Zustandstraversierung gestaltet sich dann als eine Folge von Transformationsschritten auf den Entscheidungsdiagrammen. Zwar sind die Verfahren vollständig automatisiert und allgemein einsetzbar, problematisch ist jedoch die Komplexität des Verfahrens bei mittleren und größeren Schaltungen.

Das Verfahren von van Eijk ist eine Ergänzung zum Model-Checking. In einem vorgeschalteten Schritt wird die formale Darstellung optimiert, wobei Abhängigkeiten zwischen den Signalen der beiden Schaltungsbeschreibungen ausgenutzt werden. Es entsteht eine formale Darstellung, die innerhalb des Model-Checking-Schritts i. allg. effizienter abgearbeitet werden kann als die ursprünglich vorgegebene. Bei dem Verfahren von van Eijk handelt es sich weiterhin um ein allgemeines Verifikationsverfahren, in dem Sinne, daß mit ihm beliebige Beweisziele verifiziert werden können. Die spezifischen Vorteile des Verfahrens werden jedoch vor allem bei der Verifikation von Syntheseschritten deutlich, bei denen, wie beim Retiming, strukturelle Ähnlichkeiten zwischen Eingabe- und Ausgabeschaltungsbeschreibung bestehen. Die strukturelle Ähnlichkeiten der beiden Schaltungen äußert sich im Produktautomaten in funktionalen Abhängigkeiten zwischen den Signalen.

Das dritte betrachtete Verfahren, das Verfahren von Huang, ist speziell auf die Verifikation von Retiming-Syntheseschritten zugeschnitten. Bei den beiden Schaltungsstrukturen vor und nach dem Retiming-Schritt wird versucht, Entsprechungen in den Schaltungen in der Form einer Zuordnung von Signalleitungen und Teilkomponenten zu finden. Darauf aufbauend werden die relativen Zeitverzögerungen an den Anschlußleitungen und die Initialwerte der Speicherkomponenten auf Stimmigkeit überprüft. Dies geschieht automatisch mit Hilfe eines Programmes. Das Programm ist hinreichend effizient, um auch größere Schaltungen behandeln zu können. In [HuCC96] werden experimentelle Ergebnisse mit den ISCAS'89-Benchmarks* vorgestellt. Es wird gezeigt, daß sich das Verfahren auch auf Schaltungen mit mehreren hundert Speicherelementen anwenden läßt. Kam es während des Retiming-Syntheseschritts zu einem Fehler, so kann mit diesem Verfahren der Fehler i. allg. sehr schnell aufgespürt werden. Mit einem Beweis im mathematischen Sinne hat dieses Verfahren jedoch nichts zu tun. Der erfolgreiche Ablauf des oben beschriebene Tests stellt lediglich ein Plausibilitätskriterium dar, aber noch keinen durchgängigen Beweis. Es fehlt zum einen der Nachweis dafür, daß die Tests tatsächlich hinreichend sind, um die Korrektheit des Retiming-Schritts zu gewährleisten, und es fehlt der Beweis dafür, daß die Testprogramme korrekt implementiert wurden. Die Vorgehensweise, die Verifikation einer Schal-

*Die ISCAS'89-Benchmarksammlung ist ein Vorläufer der im nächsten Abschnitt betrachteten IWLSI'91-Benchmarksammlung.

tung auf ein selbstgestricktes, effizienter überprüfbares Plausibilitätskriterium zu reduzieren, um daraus dann ohne weiteren Beweis die Korrektheit der Schaltung zu folgern, ist weitverbreitete Praxis. Das Verfahren von Huang hat zwar eine Laufzeiteffizienz, die alle anderen betrachteten Verfahren übertrifft, es eignet sich jedoch tatsächlich nur zur Überprüfung von Retiming-Schritten. Wird Retiming mit anderen Syntheseschritten wie etwa booleschen Optimierungen kombiniert, so scheitert das Verfahren. Eine darauf aufbauende, durchgängige Verifikation eines Syntheseablaufs müßte den Syntheseablauf in seine elementaren Phasen zerlegen und dann zu jedem Syntheseschritt ein geeignetes Plausibilitätskriterium und ein entsprechendes Prüfprogramm finden. Dieser Ansatz erscheint wenig sinnvoll. Aus den aufgeführten Gründen kann dieses Verfahren nicht mit den anderen verglichen werden. Mit diesem Verfahren sollte lediglich angedeutet werden, daß es durchaus auch Verfahren gibt, die von sich in Anspruch nehmen, eine Verifikation des Retimingvorganges effizienter durchführen zu können. Verfahren wie das von Huang stellen jedoch weder einen durchgängigen Korrektheitsbeweis dar, noch bieten sie ein durchgängiges Konzept für die gesamte Synthese.

8.3 Beispielschaltungen

Als Beispiele sollen zum einen die sequentiellen IWLS'91-Benchmarks und zum anderen eine eigens konstruierte generische Beispielschaltung namens R verwendet werden. Bei den IWLS'91-Benchmarks handelt es sich um Schaltungsstrukturen auf Gatterebene, die speziell zum Zweck des Vergleichs von Ergebnissen im Bereich Schaltungsentwurf zusammengestellt wurden. Die Schaltung R ist eine Struktur auf RT-Ebene ist schematisch in Abbildung 8.1 dargestellt. Sie ist mit zwei Parametern m und n, den Bitbreiten der Eingangssignale und der Bitbreite des internen Speichers, skalierbar. Die Schaltung R' (Abbildung 8.1, unten) beschreibt die Schaltung nach einem einfachen Retiming-Schritt, bei dem die Register am Eingang der $\geq$-Komponente an deren Ausgang verschoben wurden. Der Retiming-Schritt bewirkt, daß die Anzahl der benötigten Bitspeicher von $n + 2m$ auf $n + 1$ reduziert wird. Gleichzeitig reduziert sich im kombinatorischen Anteil die Länge des kritischen Pfades, die Taktfrequenz kann erhöht werden.

8.4 Ergebnisse

Tabelle 8.1 zeigt experimentelle Ergebnisse für verschiedene Verfahren, mit denen ein Retiming-Schritt in mathematisch korrekter Weise durchgeführt wird. Die Beispielschaltungen (s208.1, s298, etc.) sind der IWLS'91-Benchmark-Sammlung entnommen. Zu den einzelnen Beispielschaltungen ist jeweils die Anzahl der Flipflops und der Gatter angegeben. Alle Zeiten sind in Sekunden angegeben. Striche in der Tabelle zeigen an, daß die Experimente aufgrund von Speicherüberlauf bzw. einer Laufzeit von mehr als 10 Stunden abgebrochen werden mußten. Mit einem Fragezeichen wurden Stellen gekennzeichnet, zu denen keine Zeiten vorliegen. Die

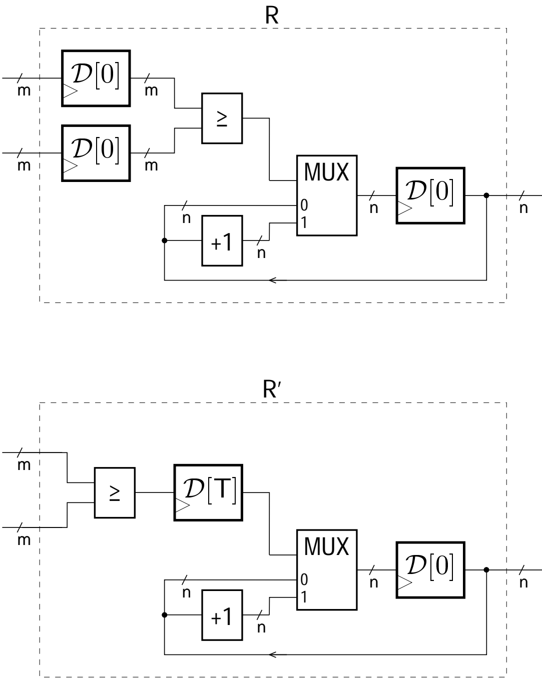


Abbildung 8.1: Retimingschritt mit der Beispielschaltung R

von van Eijk in [EiJe96] vorgestellten Verfahren sind mit Eijk/1 bzw. Eijk/2 benannt. Dabei bezeichnet Eijk/1 ein Verfahren ohne die Ausnutzung von Signalabhängigkeiten und Eijk/2 ein Verifikationsverfahren, bei dem Abhängigkeiten ausgenutzt werden. Die gemessenen Zeiten beziehen sich auf eine HP9000/735. Mit SIS und HOL sind Ergebnisse mit dem Werkzeug SIS bzw. die eigenen Arbeiten zur Formalen Synthese in HOL gekennzeichnet. Diese Experimente wurden auf einer Ultra Sparc mit 200MB Hauptspeicher durchgeführt. Bei den Zeiten in Eijk/1, Eijk/2 und SIS handelt es sich lediglich um die Zeiten zur Verifikation des Syntheseschritts, bei HOL handelt es sich um die Zeit zur Durchführung des formalen Syntheseschritts. Genaugenommen müßten zu den Zeiten in Eijk/1, Eijk/2 und SIS also noch die Zeiten zur Durchführung des Retiming-Schritts addiert werden. Da dieser Aufwand jedoch im Verhältnis zum Aufwand der nachträglichen Verifikation unerheblich ist, wurde dies vernachlässigt.

Name	Flipflops	Gatter	Eijk/1	Eijk/2	SIS	HOL
s208.1	8	104	0.5	0.7	1.2	4.4
s298	14	119	29.6	15.7	1.6	14.3
s420.1	16	218	598.4	94.8	1007.6	17.7
s510	6	211	3.2	42.7	7.3	28.7
s526	21	193	178.2	115.8	49.3	37.6
s838.1	32	288	-	-	-	83.0
s1196	18	529	?	?	3.1	61.1
s1423	74	657	?	?	-	149.5
s1488	6	653	1.6	3.3	1.7	155.1
s1494	6	647	?	?	1.6	153.7

Tabelle 8.1: IWLS'91 Benchmarks

Bei allen betrachteten Ansätzen hat sich herausgestellt, daß der Aufwand stark von der Größe der Schaltung abhängt, jedoch nur unwesentlich davon beeinflußt wird, wie der Retiming-Schritt konkret aussieht, d. h. welche Register wie verschoben werden. Für das eigene Verfahren HOL hat sich herausgestellt, daß der Aufwand mit der Größe des kombinatorischen Anteils steigt, über den die Register hinweg verschoben werden. Da aus Eijk/1 bzw. Eijk/2 nicht eindeutig hervorging, wie der Retiming-Schritt durchgeführt wurde, wurde zum Vergleich für HOL der ungünstigste Fall mit einer maximalen Größe dieses kombinatorischen Teils zugrundegelegt. Die Ergebnisse in SIS beziehen sich auf die gleichen Syntheseschritte wie in HOL.

Der Tabelle 8.1 ist zu entnehmen, daß die Ergebnisse, die durch die Model-Checking-Ansätze erzielt wurden (Eijk/1, Eijk/2 und SIS), vor allem mit der Anzahl der Flipflops korreliert sind und weniger von der Größe des kombinatorischen Anteils der Schaltung abhängen. Wie sich die Laufzeitkomplexität der Verfahren mit zunehmender Schaltungsgröße entwickelt, wird am besten an den Schaltungen s208.1, s420.1 und s838.1 deutlich. Dabei handelt es sich um die 8-Bit, die 16-Bit und die 32-Bit-Version einer Multipliziereinheit. Bei der Verdopplung der

Bitbreite von 8 auf 16 ist der Faktor in der Laufzeit bei den einfachen Model-Checking-Ansätzen Eijk/1 und SIS ca. 1000, und auch das verbesserte Verfahren Eijk/2 führt zu einem Faktor von ca. 100. Alle drei Verfahren scheitern am Beispiel s838.1.

Bei dem in HOL implementierten Verfahren zur Formalen Synthese ist zu erkennen, daß die Zeiten bei kleinen Schaltungen ungünstiger sind als bei den anderen Ansätzen. Das liegt vor allem an dem Grundaufwand, der durch die Anwendung des Retiming-Theorems entsteht (siehe Abschnitt 6.3.5). Die Komplexität wächst mit zunehmender Schaltungsgröße jedoch weit langsamer als bei den anderen Verfahren. Die Laufzeit nimmt bei den Schaltungen s208.1, s420.1 und s838.1 lediglich mit dem Faktor 4 zu, und auch die Schaltung s838.1 kann noch in vernünftiger Zeit behandelt werden. Bei großen Schaltungsstrukturen wirkt sich in HOL die große Anzahl gebundener Variablen negativ auf die Effizienz bei der Aufteilung der Schaltungsfunktion in ihre beiden Anteile aus.

Die hier vorgestellten experimentellen Ergebnissen spiegeln die allgemein bekannte Tatsache wieder, daß die Komplexität von Model-Checking-basierte Verfahren mit einer größer werdenden Anzahl von Speicherelementen i. allg. exponentiell anwächst. Für den Aufwand sind vor allem zwei Größen relevant: die Anzahl der Speicherelemente und die „Tiefe" der sequentiellen Schaltung. Mit Tiefe ist hier die Anzahl der Schritte gemeint, die mindestens erforderlich ist, um vom Ausgangszustand aus alle anderen Zustände zu erreichen. Im allgemeinen wächst die Tiefe exponentiell mit der Anzahl der Speicherelemente. Die Komplexität der Verfahren hängt noch von anderen Größen wie der Anzahl der Eingänge und der Anzahl der kombinatorischen Einheiten ab. Betrachtet man jedoch die Beispiele in Tabelle 8.1, so erkennt man, daß man bereits aufgrund der Anzahl der Speicherelemente einen ersten Eindruck von der zu erwartenden Komplexität der Verfahren gewinnt.

Wie die Anzahl der Speicherelemente und die Tiefe der Schaltung die Komplexität der Model-Checking-basierten Verfahren beeinflussen, soll anhand des Retiming-Schritts mit der Beispielschaltung R systematisch untersucht werden. Hier lassen sich die Anzahl der Speicherbausteine und die Tiefe der Schaltung direkt aus den Skalierungsparametern m und n ableiten: die Anzahl der Speicherelemente ist $2m+n$, und die Tiefe der Schaltung ist 2^n-1. Es sei angemerkt, daß R keinesfalls eine besonders untypische oder für die Verfahren Eijk/1, Eijk/2 und SIS besonders ungünstige Schaltung ist. Dies wird deutlich, wenn man die mit R erzielten Zeiten in Tabelle 8.2 mit den Zeiten entsprechend großer Schaltungen aus den IWLS'91-Benchmarks in Tabelle 8.1 vergleicht.

Alle drei Verfahren Eijk/1, Eijk/2 und SIS führen Zustandstraversierungen durch. Bei der Zustandstraversierung hat man vor allem mit zwei Problemen zu kämpfen: die Größe der BDDs und die Anzahl der Traversierungsschritte. Die Größe der BDDs hängt maßgeblich von der Anzahl der Variablen (Speicherelemente) ab, und nimmt i.allg. exponentiell mit dieser zu. Einfluß darauf hat neben der Anzahl der Variablen auch die verwendete Variablenordnung, die ggf. manuell optimiert werden sollte. Die Anzahl der Traversierungsschritte ergibt sich als die Anzahl der Takte, in der bei geeigneter Belegung der Eingangssignale frühestens

der letzte Zustand des Produktautomaten erreicht wird. Diese nimmt mit der Größe der Schaltung i.allg. ebenfalls exponentiell zu. In der Schaltung R ist die Anzahl der Variablen $2m + n$ und die Anzahl der Schritte, die notwendig ist, um alle Zustände zu erreichen $2^n - 1$. Die Komplexität der Zustandstraversierung entwickelt sich deshalb sowohl in m als auch in n exponentiell. Dies spiegelt sich in den Ergebnissen mit SIS und SMV wider, die auf einer Sparc-10 mit 96MB Hauptspeicher durchgeführt wurden (Abbildung 8.2 und 8.3). Bei beiden Verfahren wurden sehr schnell die Speichergrenzen des Rechners überschritten. Beispiele mit $n > 16$ oder $m > 16$ waren nicht möglich. SMV scheiterte bereits ab $m = 10$.

Mit der durch Formale Synthese in HOL durchgeführte Schaltungstransformationen war es möglich, bei weitem größere Schaltungen zu behandeln. In Tabelle 8.2 sind experimentelle Ergebnisse dargestellt, bei denen der Einfachheit halber für m und n gleiche Werte genommen wurden.

Bei der Formalen Synthese wurden drei verschiedene Vorgehensweisen unterschieden, die mit HOL-1, HOL-2 und HOL-3 bezeichnet wurden. HOL-1 bezieht sich auf eine Schaltungstransformation, bei der die zugrundeliegenden Schaltungen auf der Gatterebene mit rein booleschen Signalen beschrieben werden. Bei diesem Ansatz wächst mit dem Parameter n die Größe der Ausdrücke zur Darstellung der Typen. Dies führt dazu, daß mit dem Parameter n auch der Aufwand für die Unifikation von Termen vor jeder Gleichungsanwendung zunimmt. Infolgedessen wächst der zeitliche Aufwand für den Syntheseschritt mit größer werdendem n stärker als linear an.

Bei den Ansätzen HOL-2 und HOL-3 wurden anstelle von Tupeln Felder verwendet. Bei diesem Ansatz ist der Typ immer $\mathsf{num} \rightarrow \mathsf{bool}$ und zwar unabhängig vom Parameter n. Die beiden Verfahren HOL-2 und HOL-3 unterscheiden sich dadurch, daß bei HOL-2 die Auswertung des Ausdrucks zur Bestimmung des neuen Initialzustands auf Bit-Ebenen durchgeführt wurde, wohingegen dies bei HOL-3 direkt durch Umformungen auf arithmetischen Ausdrücken erzielt werden konnte. Daß bei HOL-3 die arithmetischen Umformungen wie auch alle anderen Umformungen unabhängig von n sind, spiegelt sich in den erzielten Ergebnissen wider.

Es wird deutlich, daß die hier vorgestellte Formale Synthese durchaus in der Lage ist, auf höheren Abstraktionsebenen zu großen, generisch beschriebenen Schaltungen effiziente Syntheseverfahren anzubieten. Im Gegensatz dazu sind die Verfahren Eijk/1, Eijk/2 und SIS allein auf die Gatterebene beschränkt. Die abstrakteren und kompakteren Schaltungsdarstellungen auf höheren Abstraktionsebenen sind bei weitem effizienter zu handhaben. Dies gilt sowohl für die Synthese als auch für die logische Argumentation. Da die Verfahren Eijk/1, Eijk/2 und SIS ausschließlich auf die Gatterebene beschränkt sind, bleiben ihnen diese Potentiale verschlossen.

8.5 Zusammenfassung

Am Beispiel eines elementaren Syntheseschritts wurde gezeigt, daß das Konzept der Formalen Synthese dem oft praktizierten Verfahren einer nichtformalen Syn-

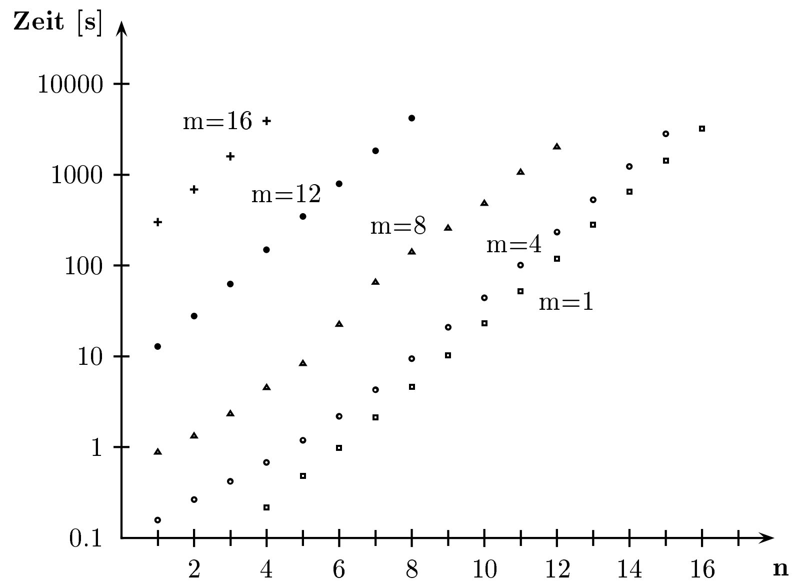


Abbildung 8.2: Verifikation von R ⇒ R' in SIS

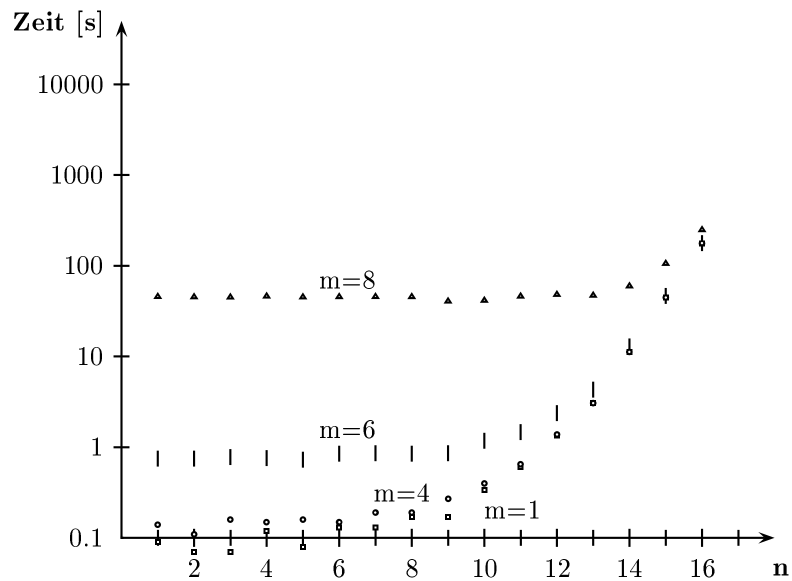


Abbildung 8.3: Verifikation von R ⇒ R' in SMV

m=n	Flipflops	Gatter	SIS	SMV	HOL-1	HOL-2	HOL-3
1	3	4	0.4	0.1	0.2	0.09	0.07
2	6	8	0.4	0.1	0.2	0.09	0.07
3	9	12	0.4	0.1	0.3	0.10	0.07
4	12	16	0.8	0.1	0.3	0.11	0.07
5	15	20	1.2	0.1	0.3	0.11	0.07
6	18	24	2.4	0.3	0.4	0.13	0.07
7	21	28	8.4	2.0	0.4	0.14	0.07
8	24	32	55.3	18.7	0.4	0.15	0.07
9	27	36	284.0	213.3	0.4	0.16	0.07
10	30	40	1487.5	-	0.5	0.18	0.07
40	120	160	-	-	1.5	0.63	0.07
80	240	320	-	-	2.7	1.40	0.07
120	360	480	-	-	4.3	2.56	0.07
160	480	640	-	-	5.8	3.98	0.07
200	600	800	-	-	7.9	5.70	0.07
240	720	960	-	-	9.9	7.84	0.07

Tabelle 8.2: Retiming von R mit $m = n$ (siehe Abbildung 8.1)

these mit nachgeschalteter Verifikation überlegen ist. Dies ergibt sich vor allem daraus, daß zum Zeitpunkt der Verifikation nur noch die Eingabe- und die Ausgabeschaltungsbeschreibung zur Verfügung stehen, nicht jedoch die Information, wie diese abgeleitet wurde. Das Problem, zwei Schaltungen miteinander zu vergleichen, ist bereits bei rein kombinatorischen Schaltungen NP-vollständig. Die Praxis zeigt, daß der Vergleich sequentieller Schaltungen um eine Größenordnung aufwendiger ist als der Vergleich kombinatorischer Schaltungen. Halbwegs effiziente Verfahren gibt es nur für Spezialfälle, etwa dann, wenn sich große Teile der Schaltung nicht geändert haben, wenn die Kodierung sich nicht ändert oder wenn nur ein reiner Retiming-Schritt durchgeführt wurde. Derartige Beschränkungen sind bei der modernen Schaltungssynthese nicht mehr praktikabel. Die Menge der Schaltungen, die durch rein kombinatorische Transformationen aus einer vorgegebenen Schaltung abgeleitet werden können, ist nur eine Teilmenge der zu dieser Schaltung äquivalenten Schaltungen. Oft befindet sich die Schaltung mit den günstigsten Eigenschaften bezüglich Energieverbrauch, Flächenbedarf etc. nicht in dieser Teilmenge. Eine Beschränkung auf rein kombinatorische Transformationen ist mit dem heutigen Stand der Technik nicht vereinbar.

Bei der Formalen Synthese zeigt sich, daß sie vor allem bei abstrakteren Formen der Schaltungsbeschreibung gewinnbringend eingesetzt werden kann. Ein höheres Beschreibungsniveau und die effiziente Ausnutzung von Regularität führen nicht nur zu kompakteren Schaltungsbeschreibungen, sondern erhöhen auch die Effizienz der Schaltungstransformationen.

Literaturverzeichnis

[AaLe91] M. Aagaard and M. Leeser. A formally verified system for logic synthesis. In *ICCD-91*. IEEE, October 1991.

[AaLe94a] M. Aagaard and M. Leeser. PBS: Proven Boolean Simplification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):459–470, April 1994.

[AaLe95] M. Aagaard and M. Leeser. Verifying a logic synthesis algorithm and implementation: A case study in software verification. *IEEE Transactions on Software Engineering*, October 1995.

[AFMF89a] L.J.M. Claesen, editor. *Applied Formal Methods For Correct VLSI Design*, volume 1. IMEC-IFIP, Elsevier Science Publishers, 1989.

[AHL92] AHL. *Lambda Reference Manual*, 1992.

[AnPP93] F. Andersen, K.D. Petersen, and J.S. Peterson. Programm verification using HOL-unity. In T.F. Melham and J. Camileri, editors, *Higher Order Logic Theorem Proving and its Applications*, number 859 in Lecture Notes in Computer Science, pages 1–15, Valletta, Malta, September 1994. Springer-Verlag.

[BBCC94] G. Bezzi, M. Bombana, P. Cavalloro, S. Conigliaro, and G. Zaza. Quantitative Evaluation of Formal Based Synthesis in ASIC Design. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 286–291, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[BoCZ92] M. Bombana, P. Cavalloro, and G. Zaza. Specification and formal synthesis of digital circuits. In [HOL92], pages 475–486.

[Brau84] W. Brauer, editor. *Automatentheorie*. Teubner, 1984.

[BrMc82] R.K.Brayton and C. McMullen. Decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*. IEEE Computer Society, 1982.

[Busc92] H. Busch. Transformational design in a theorem prover. In [TPCD92], pages 175–196.

[Camp91] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer Aided Design*, 10(1):85–93, January 1991.

[CaWo91] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer, Boston, 1991.

[EDTC96] IEEE Computer Society and ACM/SIGDA. *The European Design & Test Conference*, Paris, France, March 1996. IEEE Computer Society Press.

[EiJe96] C.A.J. van Eijk and J.A.G. Jess. Exploiting functional dependencies in finite state machine verification. In [EDTC96], pages 9–14.

[Eijk97] C. A. J. van Eijk. *Formal Methods for the Verification of Digital Circuits*. Universiteitsdrukkerij Technische Universiteit Eindhoven, 1997.

[EiKu95c] D. Eisenbiegler and R. Kumar. Formalizing the semantics for a synchronous subset of VHDL. Technical Report 8/95, Forschungszentrum Informatik (FZI), 1995.

[EiKu96] D. Eisenbiegler and R. Kumar. Synthese von Verhaltensbeschreibungen in VHDL mittels logischer Transformationen. In *Workshop „Methoden des Entwurfs und der Verifikation digitaler Schaltungen"*, Kreischa, Germany, March 1996. GI/ITG/GME.

[EnCS98] H. Engesser, V. Klaus, and A. Schwill, editors. *Duden „Informatik"*. Dudenverlag, 1988.

[FeRF93] E. Fehlauer, S. Rülke, and G. Franke. Design improvement by combining sequential and combinational techniques. In *PATMOS 1993*, pages 21–29, La Grande Motte, France, October 1993.

[FFFH89] S. Finn, M.P. Fourman, M.D. Francis, and B. Harris. Formal system design - interactive synthesis based on computer assisted formal reasoning. In [AFMF89a], pages 97–110.

[FiFM91] S. Finn, M.P. Fourman, and G. Musgrave. Interactive synthesis in HOL-abstract. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.

[FMVD90] J. Staunstrup. *Formal Methods for VLSI Design*. North-Holland, 1990.

[FoMa89] M.P. Fourman and E.M. Mayger. Formally Based System Design - Interactive hardware scheduling. In G. Musgrave and U. Lauther, editors, *Very Large Scale Integration*, pages 101–112, Munich, Federal Republic of Germany, August 1989. IFIP TC 10/WG10.5 International Conference, North-Holland.

[GDWL94] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, 1994.

[GoMe93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GrMT94] W. Grass, M. Mutz, and W. Tiedemann. High level synthesis based on formal methods. In *Proc. EUROMICRO*, pages 83–91, Liverpool, 1994.

[HaDa92] K. Hanna and N. Daeche. The veritas design logic: A users view. In [TPCD92], pages 301–310.

[HaLD89] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In L.J.M. Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, volume 2, pages 532–548. IMEC-IFIP, Elsevier Science Publishers, 1989.

[HFFM92] R.B. Hughes, M.D. Francis, S.P. Finn, and G. Musgrave. Formal tools in tri-state design in busses. In [HOL92], pages 459–475.

[HOL92] L.J.M. Claesen and M.J.C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications*, volume A-20, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland.

[HoUl79] E. Hopcroft and D. Ullmann, editors. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HuCC96] Huang, Cheng, and Chen. On verifying the correctness of retimed circuits. In *Great Lakes Symposium on VLSI*, Ames, USA, March 1996.

[JoBB88] S.D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383, Boston, 1988. Kluwer Academic Publishers.

[John84] S. D. Johnson. Synthesis of Digital Designs from Recursion Equations. MIT Press, 1984.

[JoSh90] G. Jones and M. Sheeran. *Circuit design in Ruby*, chapter 1, pages 13–70. In [FMVD90], 1990.

[JoSh91a] G. Jones and M. Sheeran. Deriving bit-serial circuits in ruby. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 71–80, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[KBES96] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design - A classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design. First International Conference,FMCAD'96*, number 1166 in Lecture Notes in Computer Science, pages 294–309, Palo Alto, CA, USA, November 1996. Springer-Verlag.

[KJBC93] P. Kission, A.A. Jerraya, L. Bergher, and E. Closse. Industrial experimentation of high-level-synthesis. In R. Werner, editor, *European Design Automation Conference with Euro-VHDL*, pages 506–513, Hamburg, Germany, September 1993. IEEE, IEEE Computer Society Press.

[KlBr95] C.D. Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*. Kluwer, Madrid, Spain, March 1995.

[KnWi89] D.W. Knapp and M. Winslett. A formalization of correctness for linked representations of datapath hardware. In [AFMF89a], pages 3–22.

[KnWi92] D.W. Knapp and M. Winslett. A prescriptive model for data-path hardware. *IEEE Transactions on Computer Aided Design*, 11(2), February 1992.

[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, August 1993.

[LeAL91] M.Leeser, M. Aagaard, and M. Linderman. The BEDROC high level synthesis system. In *ASIC'91*. IEEE, 1991.

[Lees92] M. Leeser. Using nuprl for the verification and synthesis of hardware. *Phil. Trans. R. Soc. Lond.*, 339:49–68, 1992.

[LoMu97] T. Lock and M. Mutz. Automatische formale Post-Synthese-Verifikation von High-level Syntheseergebnissen. In R. Hagelauer and M. Pfaff, editors, *Methoden des Entwurfs und der Verifikation digitaler Systeme*, pages 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*, pages 59–70, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[Mart96] H. Martin. Retiming for circuits with enable registers. In *EUROMICRO 1996*, pages 275–280, Prague, October 1996.

[McMi92b] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[Melh93] T. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[MSBS91] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential circuits with combinatorial techniques. In *IEEE Transactions on CAD*, pages 74–91, January 1991.

[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.

[Paul94] L.C. Paulson. *Isabelle - A Generic Theorem Prover.* Springer, 1994.

[RaJe95] M. Rahmouni and A.A. Jerraya. PPS: A pipeline path based scheduler. In *The European Design & Test Conference 1995*, pages 557–561, Paris, France, March 1995. IEEE Computer Society and ACM/SIGDA, IEEE Computer Society Press.

[Ross90a] Lars Rossen. *Formal Ruby*, chapter 4, pages 179–190. In [FMVD90], 1990.

[Ross90b] L. Rossen. Ruby algebra. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 297–312, Oxford, England, September 1990. Universities of Oxford and Glasgow, Springer-Verlag.

[Sent97] E. M. Sentovich. Quick conservative causality analysis. In *10th International Symposium on System Synthesis*, pages 2–8, Antwerp,Belgium, September 1997. IMEC, IEEE Computer Society Press.

[ShBT96] T.R.Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In [EDTC96], pages 328–333.

[ShRa95] R. Sharp and O. Rasmussen. The T-Ruby design system. In *IFIP Conference on Hardware Description Languages and their Applications*, pages 587–596, 1995.

[SSLM92] E.M. Sentovich, K.J. Singh, L. Lavagno, and C. Moon et. al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, 1992.

[TPCD92] V. Stavridou, T.F. Melham, and R.T. Boute, editors. *Theorem Provers in Circuit Design*, volume A-10, Nijmegen, The Netherlands, June 1992. IFIP TC10/WG10.2 International Conference, North-Holland.

[VHDL96] IEEE. *IEEE Standard VHDL Language Reference Manual Std 1076.3*, 1996.