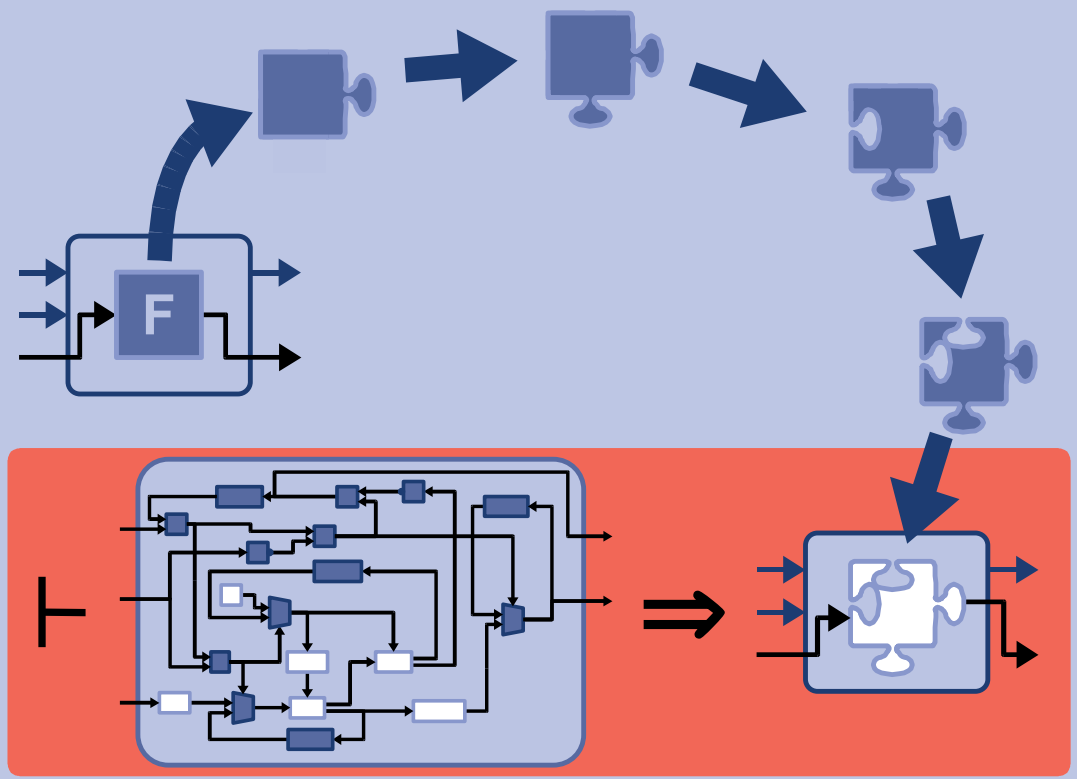
Universität Karlsruhe (TH)
Institut für Technische Informatik

Kai Kapp

Eine formale algorithmische Synthese digitaler Schaltungen



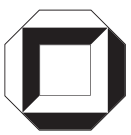
universitätsverlag karlsruhe

Kai Kapp

Eine formale algorithmische Synthese digitaler Schaltungen

Eine formale algorithmische Synthese digitaler Schaltungen

von
Kai Kapp



universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH), Fakultät für Informatik, 2005
u.d.T.: Eine automatisierte optimierende formale Synthese steuerfluss-behafteter Schaltungsbeschreibungen

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2005
Print on Demand

ISBN 3-937300-90-2

Danksagung

Die vorliegende Arbeit entstand in den Jahren 1999 bis 2005 am Institut für Technische Informatik* der Fakultät für Informatik an der Universität Karlsruhe (TH).

An dieser Stelle möchte ich mich besonders bei Herrn Prof. Dr. Detlef Schmid für seine Unterstützung, konstruktive Betreuung sowie für die Bereitstellung ausgezeichneter Arbeitsbedingungen bedanken. Auch Herrn Prof. Dr. Peter Schmitt gilt mein herzlicher Dank für die Übernahme des Korreferates und die sorgfältige Begutachtung meiner Arbeit.

Des Weiteren möchte ich allen, die mich bei meiner Forschungstätigkeit unterstützt haben, meinen Dank aussprechen. Besonders zu erwähnen ist hierbei Viktor Sabelfeld, dem ich für die vielen fruchtbaren Diskussionen und Ratschläge, aber auch für die angenehme Zusammenarbeit danke. Sehr dankbar bin ich ebenfalls Mattias Ulbrich sowie Florian Widmann für die wertvolle Unterstützung meiner Forschungstätigkeit. Ebenso gilt Frederik Beutler, Fridtjof Feldbusch, Tobias Schüle, Michael Syrjakow und Roberto Ziller mein besonderer Dank für die sehr gute und freundschaftliche Zusammenarbeit. Für die orthographische Durchsicht der Arbeit bin ich insbesondere Angelika Scherhaufer sehr dankbar.

Meinen Eltern danke ich für ihren Rückhalt und ihren Beistand, ohne die die Durchführung meines Studiums und meiner Promotion kaum möglich gewesen wäre. Der größte Dank aber gilt meiner lieben Frau Isabel, welche mir durch ihre Kraft und ihre liebevolle Unterstützung die Vollendung der Promotion ungemein erleichtert hat.

Karlsruhe, im August 2005 *Kai Kapp*

*ehemals Institut für Rechnerentwurf und Fehlertoleranz

Inhaltsverzeichnis

Kapitel 1

Einleitung

Computersysteme halten immer mehr Einzug in alle Bereiche des Lebens. Dabei nimmt vor allem die Verbreitung eingebetteter Systeme ständig zu. Eingebettete Systeme sind Computersysteme, die Bestandteil eines übergeordneten Systems sind, dessen primäre Funktion nicht rechenorientiert ist. Beispielsweise enthalten Mobiltelefone, Fernseher und Fotokopierer eingebettete Systeme. Auch in sicherheitskritischen Bereichen, wie der Luft- und Raumfahrttechnik, der Medizintechnik oder der Automobiltechnik, kommen digitale Systeme verstärkt zum Einsatz. Gerade in diesen Bereichen ist es von höchster Wichtigkeit, dass die eingesetzten Systeme fehlerfrei arbeiten. Funktionsfehler können hier nicht nur immense Kosten, sondern sogar Personenschäden nach sich ziehen.

Fehler können in Computersystemen in der Hardware oder, falls vorhanden, in der Software auftreten. Während bei sicherheitskritischen Anwendungen Fehler von vornherein weitestgehend ausgeschlossen werden müssen, ist in weniger kritischen Bereichen eine nachträgliche Fehlerkorrektur denkbar. Allerdings ist selbst in diesen Fällen häufig nur eine Korrektur der Software durchführbar, beispielsweise durch Aktualisierung der Firmware eines Systems. Eine Nachbesserung der Hardware ist dagegen im Allgemeinen ausgeschlossen oder mit einem inakzeptablen Kostenaufwand verbunden. Aus diesem Grund ist die Korrektheit der Hardwareanteile digitaler Systeme von besonderer Bedeutung.

Grundvoraussetzung für die Herstellung fehlerfreier Hardware ist zunächst die fehlerfreie Spezifikation der gewünschten Schaltung. Dabei handelt es sich keineswegs um eine triviale Aufgabe. Ganze Forschungsbereiche, wie etwa die Modellprüfung [ClGP99, BBFL01, Schn03], widmen sich der Überprüfung von Spezifikationen auf bestimmte Korrektheitseigenschaften. Untersucht werden beispielsweise deren Lebendigkeit (*liveness*), Sicherheit (*safety*) und Fairness. Genauso wichtig wie die Korrektheit der Spezifikation ist deren fehlerfreie Umsetzung. Die resultierende Schaltung muss exakt die in der Spezifikation definierten Eigenschaften aufweisen. Die Erfüllung dieser zweiten Anforderung ist der zentrale Punkt der vorliegenden Arbeit.

Auf Grund der Größe und der Komplexität heutiger digitaler Systeme ist ein manueller Hardwareentwurf in der Regel nicht praktikabel. Stattdessen kommen Synthesewerkzeuge zum Einsatz, die eine abstrakte Verhaltensspezifikation der Schaltung erlauben und somit deren Größe und Komplexität handhabbar machen [McPC90]. Aus der abstrakten Spezifikation leitet das

Synthesesystem eine konkrete Implementierung ab, welche als Vorlage für die Produktion des digitalen Systems dient.

Neben der Möglichkeit einer abstrakteren Schaltungsspezifikation ist ein weiterer Vorteil der maschinellen Synthese deren Zuverlässigkeit gegenüber einem manuellen Entwurf. In diesem Zusammenhang wurde der Begriff *correctness by construction* geprägt, welcher die Annahme ausdrückt, dass durch den Einsatz eines Entwurfssystems das Auftreten von Fehlern während der Synthese ausgeschlossen wird und infolgedessen eine Implementierung grundsätzlich exakt die anfänglich spezifizierten Eigenschaften aufweist. Diese Annahme gilt jedoch nur, solange das Entwurfssystem selbst mit absoluter Sicherheit keinen Fehler enthält. Im Laufe der Jahre wurden die Entwurfssysteme allerdings ebenfalls immer größer und komplexer. Entsprechend ist auch die Wahrscheinlichkeit von Fehlern in diesen Systemen gewachsen. Fehlerhafte Entwurfssysteme können jedoch zu inkorrekten Implementierungen führen, so dass heutzutage eine Verifikationsphase nach Abschluss eines Syntheselaufs unumgänglich ist. Nur so kann sichergestellt werden, dass die Implementierung tatsächlich den Vorgaben der Spezifikation entspricht.

Die Verifikationsphase ist allerdings sehr aufwändig und kostenintensiv. Zwar kann die Einhaltung spezifizierter Randbedingungen, wie die Ausführungsdauer eines Taktes und die Leistungsaufnahme oder Größe der Schaltung, meist verhältnismäßig einfach durch eine Analyse der Implementierung überprüft werden. Die Verifikation ihrer Funktionalität ist jedoch sehr aufwändig. Daher beschränken sich viele Verfahren auf die Entdeckung möglichst vieler Fehler in der Implementierung. Die wenigsten haben den Anspruch, die Korrektheit der Implementierung nachzuweisen. Einen automatischen Korrektheitsnachweis können entsprechende Verfahren in der Regel auch allenfalls für kleinere Schaltungen auf den unteren Abstraktionsebenen erbringen. Zudem ist deren Anwendung meistens sehr zeit- und speicheraufwändig. Da aus Komplexitätsgründen die Verifikation des Synthesewerkzeugs an sich ebenfalls im Allgemeinen ausgeschlossen ist, ist die ideale Lösung dieses Problems eine Entwurfsmethodik, die von vornherein Fehler während der Synthese ausschließt und somit eine nachträgliche Gegenüberstellung des spezifizierten Verhaltens und des Verhaltens der resultierenden Implementierung überflüssig macht.

Der Ansatz der sogenannten *transformationsbasierten Synthese* versucht diesen Anspruch zu erfüllen, indem die Implementierung aus der Spezifikation ausschließlich durch verhaltenserhaltende Transformationen abgeleitet wird. Wenn jeder der Transformationsschritte das Verhalten seines jeweiligen Ausgangspunktes erhält, so weist jedes Zwischenergebnis der Synthese und infolgedessen auch die resultierende Implementierung automatisch das ursprünglich spezifizierte Verhalten auf. Dieser Ansatz funktioniert selbstverständlich nur, solange tatsächlich keine der Transformationen das Verhalten seiner Eingabe ändert, also korrekt ist. Die Sicherstellung der Korrektheit jeder einzelnen Transformation ist daher unabdingbar [McFa93]. Naheliegend ist ein entsprechender Nachweis in Form eines formalen Beweises.

In vielen Ansätzen wird die Korrektheit der Transformationen nur manuell auf dem Papier bewiesen, so dass eine Überprüfung der Beweise nur durch mehrfaches Gegenlesen erfolgen kann. Menschliche Fehler lassen sich auf diese Weise kaum ausschließen. Als äußerst sicher kann dagegen die Durchführung der Beweise mit Hilfe maschineller Unterstützung innerhalb sogenannter *Theorembeweiser* angesehen werden. Theorembeweiser sind Programme zur Er-

zeugung formaler Spezifikationen und Beweise. Sie basieren für gewöhnlich auf einem kleinen Kalkül aus wenigen Axiomen und Inferenzregeln und gelten daher als extrem zuverlässig.

Aber selbst wenn die Korrektheit aller Transformationen nachgewiesen wurde, kann der transformationsbasierte Ansatz immer noch fehlschlagen, wenn die Implementierungen der Transformationen fehlerhaft sind. Außerdem müssen diese Implementierungen in ein Synthesesystem integriert werden, was auf Grund der Komplexität dieser Systeme ebenfalls eine sicherheitskritische Aufgabe darstellt.

Durch die Einbettung des gesamten Ableitungsprozesses in einen Theorembeweiser ist allerdings eine Gewährleistung der korrekten Ausführung der Transformationen möglich. Dieser Ansatz wird als *formale Synthese* bezeichnet. Schaltungen werden als Terme und Formeln innerhalb eines Theorembeweisers spezifiziert und ausschließlich durch Transformationen umgeformt, deren Korrektheit mit Hilfe des Theorembeweisers nachgewiesen wurde. Als Ergebnis eines jeden Transformationsschritts erhält man durch diesen Ansatz zusätzlich zu der transformierten Variante der Schaltung automatisch den formalen Beweis, dass die transformierte Schaltung funktional korrekt bezüglich ihres Ausgangspunktes ist.

Die meisten Arbeiten im Bereich der formalen Synthese befassen sich lediglich mit der Synthese von Schaltungen auf den unteren Abstraktionsebenen (Register-Transfer- und Gatterebene) oder sind auf der algorithmischen Ebene auf die Synthese reiner Datenflussbeschreibungen beschränkt. Überdies eignen sich nur wenige dieser Ansätze für eine Automatisierung der Synthese, so dass dem Schaltungsentwerfer die interaktive Steuerung des Entwurfsablaufs abverlangt wird. In diesem Fall sind vom Entwerfer fundierte Kenntnisse im Bereich der formalen Methoden gefordert, was die Akzeptanz eines entsprechenden Werkzeugs maßgeblich erschwert [KBES96].

Das bisher umfassendste System zur formalen Synthese digitaler Schaltungen ist das HASH (**H**igher order logic **A**pplied to **S**ynthesis of **H**ardware) Synthesesystem. HASH wurde im Rahmen eines von der Deutschen Forschungsgemeinschaft (DFG) unterstützten Forschungsprojekts am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe entwickelt. In diesem Projekt wurde eine Basis für die formale Repräsentation digitaler Schaltungen auf verschiedenen Abstraktionsebenen sowie für eine durchgehende formale Synthese von der System- bis zur Gatterebene geschaffen [Eise99, Blum00].

Im Gegensatz zu anderen Ansätzen der formalen Synthese unterstützt HASH nicht nur die formale Spezifikation und Synthese reiner Datenflussbeschreibungen. Steuerflussbehaftete Verhaltensbeschreibungen mit zyklischem Steuerfluss können ebenfalls in dem System spezifiziert und verarbeitet werden [Blum00].

Die Synthese steuerflussbehafteter Schaltungsbeschreibungen unterliegt allerdings bislang gravierenden Einschränkungen. Eine Einflussnahme auf qualitative Eigenschaften der resultierenden Schaltung, wie deren Größe und Geschwindigkeit, ist kaum möglich. So werden steuerflussbehaftete Spezifikationen im Zuge der algorithmischen Synthese (*high-level synthesis*) in eine einzige Schleife mit einem steuerflussbereinigten Rumpf überführt. Dadurch gehen die expliziten Steuerinformationen der Spezifikation verloren und stehen für Optimierungen während des restlichen Syntheseprozesses nicht zur Verfügung. Als Folge dieser Transformation wird beispielsweise die gemeinsame Nutzung von Funktionseinheiten, welche in der Spezifikation exklusiv in unterschiedlichen Programmzweigen zum Einsatz kommen, verhindert. Außerdem wird

durch diese Vorgehensweise implizit eine Unterschranke für die Ausführungszeit des Schleifenrumpfes festgelegt, welche der Ausführungszeit des längsten Grundblocks der Spezifikation entspricht. Diese Ausführungszeit bestimmt letztendlich auch die Dauer eines Taktes auf der Register-Transfer-Ebene (RT-Ebene), da die resultierende Implementierung auf der Register-Transfer-Ebene pro Takt genau eine Schleifeniteration ausführt. Somit wird die Ausführung aller kürzeren Grundblöcke infolge dieser Vorgehensweise unnötig ausgebremst.

In [Blum00] werden zwar einige Transformationen vorgestellt, die vor der Überführung der Spezifikation in die Ein-Schleifen-Form eine begrenzte Einflussnahme auf die Qualität des Syntheseergebnisses erlauben. Eine ähnliche Qualität der Implementierungen wie in der konventionellen Synthese ist mit diesen Transformationen allerdings nicht zu erreichen. Ein Konzept für den automatisierten Einsatz dieser Transformationen existiert nicht, so dass die Anwendung der Transformationen interaktiv erfolgen muss und somit vom Entwerfer fundierte Kenntnisse in formalen Methoden gefordert sind. Auch weicht das Konzept so stark von der konventionellen Synthese ab, dass bewährte Verfahren aus diesen Bereichen nicht eingesetzt werden können.

1.1 Zielsetzung der Arbeit

In dieser Arbeit wird ein neuer Ansatz für die formale Synthese steuerflussbehafteter Schaltungsbeschreibungen vorgestellt. Dieser Ansatz verfolgt zwei grundlegende Ziele: die Erhöhung der Qualität der resultierenden Implementierungen sowie die vollständige Automatisierung des Syntheseprozesses.

Durch den Einsatz einer neuen formalen Schaltungsrepräsentation bleiben die Steuerinformationen, die der Entwerfer in der Spezifikation vorgegeben hat, während des gesamten Syntheseverlaufs erhalten. Das Optimierungspotential der Implementierungen erhöht sich dadurch beträchtlich. Beispielsweise ergeben sich weitreichende Möglichkeiten für die gezielte Einordnung der einzelnen Operationen der Spezifikation in bestimmte Takte der geplanten RT-Ebenen-Implementierung. Durch eine günstige Verteilung der Operationen lässt sich die durchschnittliche Taktzahl, welche zur Ausführung der Spezifikation erforderlich sein wird, minimieren bzw. eine optimale Ausnutzung der eingesetzten Funktionseinheiten erreichen.

Als Grundlage für die formale Spezifikation und Synthese von Schaltungsbeschreibungen kommt in HASH *Logik höherer Ordnung* [Chur40] zum Einsatz. Logik höherer Ordnung ist nicht entscheidbar, so dass sich für eine Synthese auf Basis Logik höherer Ordnung ein konstruktiver Ansatz wie der transformationsbasierte Entwurf anbietet. Um eine Automatisierung dieser formalen Synthese zu erreichen, muss ein Synthesesystem geschaffen werden, welches im Stande ist, selbstständig geeignete Transformationen zur Ableitung einer RT-Ebenen-Implementierung aus einer Schaltungsspezifikation auszuwählen und anzuwenden. Vor der Anwendung bestimmter Transformationen muss das System überdies eine automatische Analyse und gegebenenfalls eine Umformung der jeweiligen Schaltungsrepräsentation durchführen können, damit ein Gelingen entsprechender Transformationen gewährleistet werden kann. Darüber hinaus muss für jede Transformation, deren Anwendung an bestimmte Vorbedingungen geknüpft ist, eine Strategie für den automatischen Beweis der entsprechenden Vorbedingungen existieren.

Eine Methodik, welche die vollständig automatisierte Durchführung der formalen algorithmischen Synthese realisiert, wird in der vorliegenden Arbeit präsentiert. Es werden die Transformationen und verschiedenen Schaltungsrepräsentationen vorgestellt, welche die Basis für die formale algorithmische Synthese bilden, und gezeigt, wie durch die gezielte Anwendung der Transformationen automatisch die verhaltenserhaltende Ableitung einer RT-Ebenen-Implementierung aus einer steuerflussbehafteten Schaltungsspezifikation erfolgen kann. Durch die vollständige Automatisierung der formalen Synthese wird auch Entwerfern ohne Kenntnisse in formalen Methoden ermöglicht, Implementierungen zu entwickeln, die bewiesenermaßen funktional korrekt bezüglich ihrer Spezifikation sind.

Um eine Verbesserung der Qualität der resultierenden Implementierungen zu erlangen, wird überdies ein Rahmenwerk zur Integration konventioneller Synthesewerkzeuge bzw. Synthesealgorithmen in den formalen Syntheseprozess angeboten. Auf diese Weise können ausgereifte, hochwertige Synthesetechniken eingesetzt werden, um den formalen Ableitungsprozess zu steuern und dadurch Implementierungen hoher Qualität zu erzeugen. Ein Risiko stellt die Anbindung unverifizierter Syntheseverfahren nicht dar. Fehlerhafte Steuerdirektiven können im Ansatz der formalen Synthese ebenso wenig wie potentielle Fehler im HASH Synthesesystem zu unkorrekten Implementierungen führen. Die Ableitung der Implementierungen erfolgt vollständig innerhalb eines Theorembeweisers und dieser stellt die Korrektheit jedes einzelnen Transformationsschritts sicher.

Insgesamt ergibt sich ein Entwurfssystem für die formale algorithmische Synthese, mit dessen Hilfe garantiert korrekte Implementierungen hoher Qualität vollautomatisch aus steuerflussbehafteten Schaltungsbeschreibungen abgeleitet werden können.

1.2 Gliederung der Arbeit

Die Arbeit ist in zwei Teile unterteilt. Im Teil I der Arbeit wird zunächst in Kapitel 2 eine Übersicht über die Synthese digitaler Schaltungen gegeben. Anschließend werden verschiedene Techniken zur Sicherstellung der Korrektheit des Syntheseergebnisses diskutiert. Es folgt in Kapitel 3 eine Beschreibung des in dieser Arbeit eingesetzten Theorembeweisers und eine ausführliche Darstellung der prinzipiellen Durchführung der formalen Synthese. Im Anschluss daran werden die formalen Schaltungsrepräsentationen vorgestellt, die im Zuge der formalen algorithmischen Synthese zum Einsatz kommen.

In Teil II der Arbeit wird das neue Konzept zur formalen Synthese steuerflussbehafteter Schaltungsbeschreibungen präsentiert. Das Rahmenwerk zur Integration konventioneller Optimierungs- und Syntheseverfahren wird in Kapitel 4 beschrieben und die vollständige Automatisierung der formalen algorithmischen Synthese wird anschließend in Kapitel 5 erläutert.

Die Ausarbeitung endet mit einigen experimentellen Ergebnissen in Kapitel 6 und einer Zusammenfassung in Kapitel 7.

Teil I

Grundlagen

Kapitel 2

Synthese digitaler Schaltungen

Die heute zur Verfügung stehenden Technologien und Methoden erlauben die Produktion von Schaltungen einer Größe und Komplexität, deren Entwurf ohne eine maschinelle Unterstützung nicht mehr handhabbar wäre. Schaltungen werden heutzutage im Allgemeinen auf einem hohen abstrakten Niveau in sogenannten *Hardware-Beschreibungssprachen*, wie z. B. VHDL [VHDL02] oder Verilog [ThMo02], spezifiziert. Mit Hilfe von *Synthese*-Werkzeugen wird aus der Schaltungsbeschreibung ein konkretes Layout der Schaltung erzeugt, welches als Vorlage für eine anschließende Chip-Fertigung verwendet wird.

Als Eingabe erhalten die Synthese-Werkzeuge entweder eine strukturelle oder verhaltensorientierte Beschreibung der gewünschten Schaltung und zusätzlich einige physikalische Randbedingungen, welche die resultierende Schaltung einzuhalten hat. Diese Randbedingungen beziehen sich meistens auf die Geschwindigkeit, die Fläche sowie den Leistungsverbrauch der gewünschten Schaltung. Die Fläche der resultierenden Schaltung steht dabei in direktem Zusammenhang mit deren Produktionskosten.

Zur Bewältigung der Komplexität des Syntheseprozesses wird dieser in der Regel schrittweise durchgeführt. Ausgangspunkt eines Syntheseschritts ist eine *Spezifikation*, welche die zu synthetisierende Schaltung beschreibt. Ein Syntheseschritt bewirkt entweder eine *Verfeinerung* oder eine *Optimierung* der Spezifikation. Bei einer Verfeinerung wird die Schaltungsbeschreibung von einer Abstraktionsebene auf eine darunter liegende Abstraktionsebene abgebildet. Einige abstrakt spezifizierte Eigenschaften werden dabei konkretisiert. Bei einer Optimierung werden Eigenschaften der Spezifikation innerhalb einer Abstraktionsebene verbessert.

Das Ergebnis eines Syntheseschritts wird als *Implementierung* bezeichnet. Eine Implementierung repräsentiert wiederum die Basis für den gegebenenfalls anschließend auszuführenden Syntheseschritt, in welchem Fall sie als dessen Ausgangsspezifikation angesehen wird.

In dieser Arbeit wird zwischen folgenden Abstraktionsebenen unterschieden: der *Gatterebene*, der *Register-Transfer-Ebene*, der *algorithmischen Ebene* sowie der *Systemebene*. Auf der Gatterebene wird eine Schaltung als Struktur bestehend aus Logik-Gattern und Flip-Flops, welche durch boolesche Signalleitungen miteinander verbunden sind, dargestellt. Die Logik-Gatter realisieren boolesche Operationen und die Flip-Flops boolesche Speicherelemente. Auf dieser Ebene ist das Zeitverhalten der Elemente von großer Bedeutung. In der Regel kommen approximative Verzögerungsmodelle zur Anwendung. Auf der nächsthöheren Abstraktionsebe-

ne, der Register-Transfer-Ebene, entspricht die Schaltung einer Struktur aus (z. B. arithmetisch-logischen) Funktionseinheiten, Speichereinheiten (z. B. Registern) und Verbindungseinheiten (z. B. Multiplexern). Während auf der Gatterebene die Laufzeiten der Gatter relevant sind, ist die kleinste Zeiteinheit der RT-Ebene üblicherweise der *Takt*. Innerhalb eines Taktes werden die Register gelesen, Operationen ausgeführt und die Ergebnisse wieder in die Register geschrieben. Es findet also in jedem Takt genau ein Register-Transfer statt. Auf der algorithmischen Ebene wird die Schaltung durch nebenläufige Algorithmen beschrieben. Von der Zeit wird weiter abstrahiert. Es spielt nur noch die Kausalität eine Rolle. Die Operationen der Algorithmen müssen in einer Reihenfolge ausgeführt werden, die deren Steuer- und Datenabhängigkeiten Rechnung trägt. Auf der Systemebene werden Schaltungen durch ein System untereinander kommunizierender Prozesse beschrieben. Die Prozesse bestehen einerseits aus funktionalen Prozessen, welche die auf der algorithmischen Ebene definierten Algorithmen ausführen, und andererseits aus Prozessen, die die Kommunikation zwischen den funktionalen Prozessen steuern. Auf dieser Ebene ist nur noch die Kausalität bezüglich der auszuführenden Prozesse von Bedeutung.

Im Folgenden liegt der Schwerpunkt auf der sogenannten *algorithmischen Synthese*, in welcher Schaltungsspezifikationen auf der algorithmischen Ebene auf Implementierungen auf der Register-Transfer-Ebene abgebildet werden [McPC90, GaRa94, Lin97].

2.1 Algorithmische Synthese

Oft wird vor Beginn der algorithmischen Synthese eine Optimierung der algorithmischen Schaltungsbeschreibung vorgenommen. In [NiBr97a, NiBr97b] wird gezeigt, dass dieser Schritt in der Regel eine starke Verbesserung der Qualität der daraus synthetisierten Schaltung mit sich bringt. Die meisten dieser Optimierungen sind aus dem Bereich der optimierenden Software-Compiler bekannt [AhSU86, Much97]. Einige Beispiele sind die Propagierung von Konstantenwerten, das Zusammenfassen gemeinsamer Teilausdrücke, die Vereinfachung boolescher und algebraischer Ausdrücke, die Extraktion von Schleifeninvarianten und die Entfernung überflüssiger Ausdrücke und Variablen. Eine umfassende Übersicht entsprechender Transformationen speziell für die Optimierung von Schaltungsbeschreibungen auf der algorithmischen Ebene gibt [GeRo99].

Die gegebenenfalls optimierte Verhaltensbeschreibung bildet schließlich den Ausgangspunkt für die Abbildung der Verhaltensspezifikation auf eine Implementierung auf der Register-Transfer-Ebene. Die Implementierung besteht in der Regel aus einem *Operationswerk* (*datapath*) und einem *Steuerwerk* (*control unit*). Das Operationswerk ist eine Struktur aus Funktions-, Speicher- und Verbindungseinheiten, dessen spezielle Funktionsweise vom Steuerwerk konfiguriert werden kann. Aufgabe des Steuerwerks ist es, auf Basis des Operationswerks die taktweise Abarbeitung einer Berechnung zu realisieren.

Bei der Durchführung der algorithmischen Synthese werden im Allgemeinen vier Aufgabenstellungen unterschieden [DeMi94]:

- die Ablaufplanung (*scheduling*),
- die Ressourcenbereitstellung (*allocation*),

- die Ressourcenzuweisung (*binding*),
- und die Steuerwerksynthese (*control synthesis*).

Während der Ablaufplanung werden die Operationen einer Spezifikation in einzelne Steuerschritte (*control steps*) eingeordnet. Ein Steuerschritt entspricht einem Takt auf der Register-Transfer-Ebene. Man unterscheidet die *ressourcenbeschränkte* und die *zeitbeschränkte* Ablaufplanung. Die ressourcenbeschränkte Ablaufplanung hat das Ziel, die Anzahl an Steuerschritten zu minimieren, die zur Ausführung einer Berechnung unter Verwendung einer fest vorgegebenen Menge von Ressourcen benötigt werden. Im Gegensatz dazu ist bei der zeitbeschränkten Ablaufplanung eine feste Grenze für die Anzahl an Steuerschritten vorgegeben, in welcher eine Berechnung mit einer möglichst kleinen Menge von Ressourcen durchgeführt werden muss.

Im Rahmen der Ressourcenbereitstellung wird festgelegt, welche und wie viele der Ressourcen, also Funktions-, Speicher- und Verbindungseinheiten, zur Erzeugung eines geeigneten Operationswerks zur Verfügung gestellt werden müssen. Bei den einzelnen Ressourcen existiert häufig eine Auswahl von verschiedenen Realisierungen, welche die gleiche Aufgabe erfüllen, aber unterschiedliche physikalische Eigenschaften, wie z. B. Laufzeit, Flächen- und Energiebedarf, aufweisen. Werden mehr Ressourcen bereitgestellt, so werden in der Regel weniger Takte zur Durchführung einer Berechnung benötigt. Allerdings führt der Einsatz einer größeren Anzahl von Ressourcen auch zu einer Vergrößerung der Fläche der Implementierung und dadurch zu einer Erhöhung der Produktionskosten der Schaltung.

Wenn auch bei der Ablaufplanung eine Einteilung der Operationen in verschiedene Steuerschritte vorgenommen wird, so bleibt dennoch zunächst offen, welche Funktionseinheit konkret die Realisierung einer bestimmten Operation übernimmt. Diese Zuordnung erfolgt im Zuge der sogenannten Ressourcenzuweisung. Während der Ressourcenzuweisung werden Operationen bestimmten Funktionseinheiten und Variablen den bereitgestellten Speicherelementen zugeordnet. Weiter werden alle benötigten Datenübertragungen auf eine geeignete Weise den Verbindungseinheiten zugewiesen, so dass die Daten innerhalb jedes Taktes den Vorgaben der Ablaufplanung entsprechend und im Einklang mit den übrigen Ressourcenzuweisungen übertragen werden können.

Obwohl diese drei Aufgaben hier einzeln aufgelistet werden, sind sie doch stark miteinander verflochten. Sie müssen oft gleichzeitig oder iterativ bearbeitet werden, um ein zufriedenstellendes Ergebnis bzgl. der physikalischen Eigenschaften der Implementierung, wie deren Schnelligkeit, Energiebedarf und Fläche bzw. Kosten, zu erzielen. Ihre Ergebnisse stellen die Basis für den Aufbau des Operationswerks dar.

Die letzte Aufgabe der algorithmischen Synthese ist die Erzeugung eines Steuerwerks. Das Steuerwerk regelt die schrittweise Durchführung einer Berechnung nach den Vorgaben der Ablaufplanung. Dazu gibt es über Steuersignale dem Operationswerk in jedem Takt eine bestimmte Funktionsweise vor und ermittelt in Abhängigkeit von dessen Statusausgabe den anschließend auszuführenden Steuerschritt. Zusätzlich regelt das Steuerwerk die Kommunikation mit der Umgebung der Schaltung. Dazu enthält es einige Steuereingänge zum Anstoßen oder zum Abbrechen einer Berechnung und ebenso Steuerausgänge, über die es den Status der Berechnung seiner Umgebung mitteilt.

2.2 Sicherstellung der Korrektheit des Syntheseergebnisses

Ein wichtiger Bestandteil der Synthese digitaler Schaltungen ist die Sicherstellung der Korrektheit des Syntheseergebnisses. Eine Schaltung mit den besten physikalischen Eigenschaften ist wertlos, falls sie sich nicht den Vorgaben ihrer Spezifikation entsprechend verhält.

Wie bereits eingangs erwähnt, kann die Einhaltung physikalischer Randbedingungen in der Regel verhältnismäßig einfach mit analytischen Mitteln überprüft werden, so dass sich die Betrachtungen in dieser Arbeit auf die funktionale Korrektheit der Implementierung bezüglich ihrer Spezifikation konzentrieren.

Um die Korrektheit einer Implementierung sicherzustellen, bieten sich prinzipiell drei Zeitpunkte an: *vor*, *während* oder *nach* Durchführung der Synthese [KBES96]. Vor der Durchführung eines Syntheselaufs kann die Korrektheit aller resultierenden Ergebnisse gewährleistet werden, indem ein Syntheseprogramm eingesetzt wird, das garantiert fehlerfrei arbeitet. Ein entsprechender Nachweis kann nur durch eine Verifikation des Synthesewerkzeugs erbracht werden. Während des Syntheseprozesses kann die Korrektheit der Implementierung sichergestellt werden, indem ausschließlich verhaltenserhaltende Transformationen zur Ableitung der Implementierung aus der Spezifikation eingesetzt werden. Als letzte Möglichkeit bietet sich eine Verifikation der Implementierung nach Durchführung der Synthese an. In diesem Fall wird die resultierende Implementierung ihrer ursprünglichen Spezifikation gegenübergestellt und überprüft, ob beide das gleiche Verhalten aufweisen.

Im Folgenden werden einige Ansätze aus dem Bereich der algorithmischen Synthese vorgestellt, welche die Korrektheit ihrer Ergebnisse vor, während oder nach Durchführung der Synthese sicherzustellen beabsichtigen.

2.2.1 Sicherstellung der Korrektheit nach der Synthese

Nach Abschluss des Syntheseprozesses steht im Allgemeinen keine Information mehr über die Art und Weise, wie sich die Implementierung aus der Spezifikation ergeben hat, zur Verfügung. Daher muss allein auf Basis der Implementierung und ihrer Spezifikation überprüft werden, ob die Implementierung die vorgesehene Funktionalität aufweist.

Die gebräuchlichste Methode zur Überprüfung der Korrektheit eines Syntheseergebnisses ist hierbei die Simulation. Um nachzuweisen, dass sich eine Implementierung korrekt verhält, müssen alle möglichen Kombinationen von Eingabewerten simuliert und die aus ihnen resultierenden Ausgaben der Implementierung und der Spezifikation gegenübergestellt werden. Bei Schaltwerken muss eine entsprechende Simulation überdies für jeden möglichen Zustand des Systems durchgeführt werden. Die Anzahl der zu überprüfenden Eingabekombinationen wächst somit exponentiell mit der Zahl der Schaltungseingänge bzw. der Zahl der eingesetzten Speicherelemente, so dass eine erschöpfende Simulation – wenn überhaupt – lediglich für sehr kleine Schaltungen mit endlichen Datentypen in Frage kommt. Bei größeren Schaltungen kann nur eine Validierung der Implementierung auf Basis einer Teilmenge der möglichen Eingabesequenzen durchgeführt werden. In diesem Fall stellt die Simulation jedoch nicht die Korrektheit der Implementierung sicher, sondern erhöht allenfalls das Vertrauen des Entwerfers in die resultierende Schaltung.

Eine automatische Verifikation auf Basis formaler Methoden zum nachträglichen Beweis der funktionalen Korrektheit der RT-Ebenen-Implementierung bezüglich ihrer algorithmischen Spezifikation scheidet im Allgemeinen ebenfalls auf Grund der Komplexität dieses Vorhabens aus. Selbst wenn nur endliche Datentypen in der Spezifikation eingesetzt werden und infolgedessen die Schaltung nur begrenzt viele unterschiedliche Zustände einnehmen kann, so wächst dennoch die Anzahl der zu berücksichtigenden Fälle exponentiell mit der Bitbreite der Eingangssignale bzw. der Speicherelemente an (*state explosion problem*). Aus diesem Grund ist eine entsprechende Verifikation für größere Schaltungen kaum durchführbar, beim Einsatz unendlicher Datentypen überhaupt nur eingeschränkt möglich, und gilt daher als kaum praktikabel [Gupt92, KBES96, Krop99].

2.2.2 Sicherstellung der Korrektheit vor der Synthese

Eine Möglichkeit, bereits vor der Durchführung der Synthese die Korrektheit aller Ergebnisse zu gewährleisten, besteht theoretisch in dem Einsatz eines verifizierten Synthesesystems. Für den Schaltungsentwerfer wäre diese Lösung optimal. Ohne Kenntnisse im Bereich der formalen Methoden könnte er allein durch Nutzung eines entsprechenden Werkzeugs garantiert korrekte Implementierungen entwickeln.

Die Verifikation von Software ist jedoch extrem aufwändig und infolgedessen meist nur für kleinere Programme praktikabel. Synthesewerkzeuge sind aber für gewöhnlich große, komplexe Programme, so dass deren Verifikation in aller Regel ausgeschlossen ist. Entsprechende Ansätze beschränken sich daher auf die Verifikation einzelner Optimierungs- und Synthesealgorithmen*.

FDLS-Verifikation: In [NTRG01] wird die formale Spezifikation und Verifikation des *Force-Directed List Scheduling*-Algorithmus [PaKn89] in dem Theorembeweiser PVS (Prototype Verification System) [OwRS92] beschrieben. Dabei handelt es sich um einen Algorithmus zur ressourcenbeschränkten Ablaufplanung.

Die Verifikation beginnt mit der formalen Spezifikation einiger Eigenschaften, die generell für ressourcenbeschränkte Ablaufplanungen gelten müssen: Jede Operation wird einem bestimmten Steuerschritt zugewiesen, die Einordnung der Operationen berücksichtigt alle Steuer- und Datenabhängigkeiten und die Ressourcenbeschränkungen werden strikt eingehalten. Der Algorithmus zur Ablaufplanung selbst wird ebenfalls formal spezifiziert.

Die Verifikation des Algorithmus erfolgt durch den formalen Beweis, dass bei seiner Ausführung alle zuvor spezifizierten Eigenschaften erfüllt werden. Im Zuge dessen werden zahlreiche Lemmata bewiesen, welche invarianten Eigenschaften des Algorithmus entsprechen, die zu bestimmten Zeitpunkten seiner Ausführung gelten.

Nach der Verifikation des Algorithmus wird dessen formale Spezifikation als Vorlage für eine Implementierung in C++ verwendet und die Implementierung in das bereits bestehende

*Zur Vereinfachung soll im weiteren Verlauf der Arbeit der Begriff „Synthesealgorithmen“ neben Synthesealgorithmen auch Optimierungsalgorithmen umfassen. Entsprechend soll der Begriff „Syntheseverfahren“ auch Optimierungsverfahren einschließen.

Synthesewerkzeug *DSS* [RKDV92] integriert. Die während der Verifikation entdeckten invarianten Eigenschaften werden zusätzlich als sogenannte *Assertions* „sorgfältig“ implementiert und in die Implementierung des Algorithmus eingefügt. Die Assertions überprüfen während der Ausführung des Algorithmus, ob die geforderten Eigenschaften auch tatsächlich eingehalten werden, und unterstützen eine gegebenenfalls erforderliche Fehlersuche.

In [NaVe98] wird das gleiche Verfahren bei der Ressourcenzuweisung von Speichereinheiten angewandt.

Man beachte, dass in diesem Ansatz kein wirklich verifiziertes Programm zur Ausführung kommt, sondern nur ein Programm, das zu bestimmten Zeitpunkten untersucht, ob ein Fehler in seiner Berechnung aufgetreten ist. Die Autoren bezeichnen daher die resultierende Implementierung als „selbst-verifizierend“. Der Erfolg des Verfahrens ist jedoch abhängig von der Vollständigkeit der bewiesenen Eigenschaften und von der fehlerfreien Implementierung des Algorithmus sowie der Assertions, welche die korrekte Ausführung des Algorithmus sicherstellen sollen.

Bedroc: Das *Bedroc* Projekt der Cornell Universität in Ithaca, NY befasst sich ebenfalls mit der Sicherstellung der Korrektheit des Synthesewerkzeugs [LeAL91, LCAL93]. Ziel des Projekts ist die durchgängige Anwendung formaler Methoden während des gesamten Entwurfsprozesses. Das Bedroc Synthesesystem erzeugt automatisch aus einer Verhaltensspezifikation in der Sprache *HardwarePal* eine Netzliste in XNF (Xilinx Network Format) für die Programmierung eines FPGAs (Field Programmable Gate Array). HardwarePal ist eine im Zuge des Bedroc Projekts entwickelte imperative Sprache, die stark an Pascal angelehnt ist, und zusätzlich einige Konstrukte zur Spezifikation von Hardware bereitstellt.

Der Entwurfsprozess in Bedroc beginnt mit der Spezifikation des Verhaltens der gewünschten Schaltung in HardwarePal. Anschließend wird die Spezifikation in einen sogenannten *Abhängigkeitsflussgraph* (AFG) übersetzt. Die Knoten des AFG stehen für funktionale Operationen, Steueroperationen und Ein-/Ausgabeoperationen. Die Kanten zwischen den Knoten repräsentieren den Steuer- bzw. den Datenfluss zwischen den Operationen. Auf Basis des AFG wird nach einigen hardwareunabhängigen Optimierungen die algorithmische Synthese durchgeführt. Es ergeben sich ein Steuerwerk, welches als Menge logischer Gleichungen dargestellt wird, und ein Operationswerk, das einer Netzliste aus Hardwareressourcen entspricht.

Die nachfolgende Logiksynthese besteht aus mehreren Schritten: Zunächst werden das Steuerwerk sowie das Operationswerk anhand einer Hardwarebibliothek in eine Netzliste primitiver Basiszellen (Addierzellen, Multiplexer, Logikgatter, Flipflops, ...) übersetzt und die kombinatorische Logik des Operationswerks mit Hilfe des Werkzeugs *PBS* (Proven Boolean Simplifier) minimiert. Anschließend werden die kombinatorische Logik des Operationswerks sowie die sequentielle Logik des Steuerwerks wieder zusammengeführt und auf das konkrete Ziel-FPGA abgebildet. Während der Technologieabbildung auf den FPGA werden noch einige hardwareabhängige Optimierungen durchgeführt.

Bisher sind nur einige Bestandteile des Bedroc Systems formal verifiziert worden. Bewiesen wurde beispielsweise die Korrektheit des Algorithmus zur Übersetzung der HardwarePal-Spezifikation in einen AFG sowie die Korrektheit der AFG-basierten Optimierungsalgorithmen.

Diese Beweise wurden jedoch nur auf dem Papier durchgeführt. Im Gegensatz dazu wurde beim Logikminimierer PBS sogar dessen konkrete *Implementierung* mit Hilfe eines Theorembeweisers verifiziert (siehe unten). Der Beweis der Korrektheit der eigentlichen Syntheseschritte, die eine Verfeinerung von der algorithmischen Ebene auf die RT-Ebene bzw. von der RT- auf die Gatterebene bewerkstelligen, steht noch aus.

PBS: Die Verifikation des Logiksynthese-Werkzeugs *PBS* [AaLe91, AaLe94, AaLe95] wurde mit Hilfe des Theorembeweisers Nuprl [CABC86, Lees92] durchgeführt. PBS implementiert den *Weak Division*-Algorithmus, welcher die Fläche der resultierenden Schaltung minimiert, indem er redundante kombinatorische Logik eliminiert [BrMc82].

PBS ist in der Sprache *SML* (Standard Meta Language) implementiert, einer funktionalen Sprache mit einer formal definierten Semantik [MiTM97]. Zu Beginn der Verifikation wird eine für die Implementierung von PBS ausreichende Teilmenge der Sprache SML in dem Theorembeweiser Nuprl definiert. Dadurch kann im Anschluss an die Verifikation des Algorithmus dessen Quelltext unverändert eingesetzt werden. Eine fehlerträchtige nachträgliche Implementierung entfällt. Nach der Spezifikation des Algorithmus im Theorembeweiser werden einige Eigenschaften modelliert, die der Algorithmus zu erfüllen hat. Diese beinhalten, dass der Algorithmus die Funktionalität der Schaltung nicht verändert und dass die Größe der resultierenden Schaltung minimal ist, also keinerlei redundante Logik mehr enthält. Anschließend beweisen die Autoren interaktiv in Nuprl, dass ihre konkrete Implementierung von PBS die geforderten Eigenschaften erfüllt.

Der Quelltext von PBS hat eine Länge von etwa 1000 Zeilen. Zur Verifikation von PBS war der Beweis von ca. 500 Theoremen erforderlich. Der damals in Softwareverifikation unerfahrene Autor Mark Aagaard brauchte 8 Monate zum Beweis des Algorithmus. Die Autoren schätzen, dass geübte Entwickler einen Algorithmus ähnlicher Komplexität und Größe sonst in 3 Monaten verifizieren könnten.

2.2.3 Sicherstellung der Korrektheit während der Synthese

Aus den bisher aufgeführten Ansätzen wird ersichtlich, dass eine Gewährleistung der Implementierungskorrektheit vor bzw. nach der Synthese im Allgemeinen undurchführbar oder mit einem kaum vertretbaren Aufwand verbunden ist. Als Alternative verbleibt die Sicherstellung der Korrektheit während der Durchführung der Synthese.

In diesem Zusammenhang bietet sich der transformationsbasierte Entwurf an. Bei der transformationsbasierten Synthese werden Implementierungen ausschließlich auf Basis verhaltenserhaltender Transformationen aus der Spezifikation abgeleitet. Fehler werden auf diese Weise während des Syntheseprozesses vermieden. Ändert keine der Transformationen das Verhalten der Spezifikation, so weist letztendlich auch die resultierende Implementierung genau das spezifizierte Verhalten auf.

FRESH: Ein Beispiel für ein transformationsbasiertes Synthesesystem ist *FRESH* (FRom EquationS to Hardware). In [MeHF96, MeHF97, MeHF98, MeHe98, MHMP02, MeHP02] wird

eine automatische, transformationsbasierte Synthese von der algorithmischen Ebene auf die RT-Ebene beschrieben. Als Spezifikationssprache wird die eigens entwickelte Sprache *ES* (*Equational Specification*) verwendet. ES ist an die Sprache Lucid [WaAs85] angelehnt, bietet aber keinerlei Steuerkonstrukte an. So sind Schaltungsspezifikationen in ES auf reine – wenn auch zyklenbehaftete – Datenflussbeschreibungen beschränkt.

Die ES-Spezifikation einer gewünschten Schaltung wird zunächst in eine VHDL-Spezifikation, bestehend aus einem einzigen Prozess, überführt und mittels eines konventionellen Synthesewerkzeugs auf eine Implementierung auf der RT-Ebene abgebildet. Die resultierende Implementierung wird daraufhin als Vorlage für die transformationsbasierte Herleitung einer entsprechenden ES-Implementierung innerhalb von FRESH verwendet. Schließlich wird die in FRESH erzeugte RT-Ebenen-Implementierung wieder von der Sprache ES in VHDL übersetzt und mit konventionellen Synthesewerkzeugen weiterverarbeitet.

In FRESH werden alle zur Durchführung einer algorithmischen Synthese erforderlichen Schritte unterstützt. Welchen Einschränkungen jedoch die einzelnen Synthesealgorithmen unterliegen, so dass deren Ergebnisse noch in FRESH umsetzbar sind, geht aus den Veröffentlichungen des Projekts leider nicht hervor.

Da die Korrektheit der Transformationen in dieser Arbeit nur manuell bewiesen und deren Implementierung überhaupt keiner Überprüfung unterzogen wurde, kann allein der Einsatz dieses Systems die Korrektheit der resultierenden Implementierungen nicht gewährleisten. Eine nachträgliche Überprüfung der Ergebnisse bleibt nach wie vor unerlässlich.

CBS: Ein weiteres Verfahren aus dem Bereich der transformationsbasierten Synthese wird in [HiRE99, HiER99] präsentiert. Dabei handelt es sich um einen Algorithmus namens *Case-Based-Scheduling* (CBS) zur Synthese von Pipelinesystemen aus einer sequentiellen Prozessorbeschreibung. Die Spezifikation der Prozessorbeschreibung erfolgt in der Sprache *Language of Labelled Segments* (LLS), einer Sprache zur Spezifikation synchroner Transitionssysteme. Bei dem vorgestellten Verfahren wird die Spezifikation der sequentiellen Prozessorbeschreibung ausschließlich durch die Anwendung von Transformationen, welche die funktionale Korrektheit erhalten, in ein Pipelinesystem überführt.

Um Implementierungsfehler des Synthesewerkzeugs aufzudecken, wird die Korrektheit jedes einzelnen Transformationsschritts durch den externen Äquivalenzprüfer TUD [RiEH99, RiHE99, BELR01] kontrolliert. Zusätzlich wird nach der Durchführung aller Transformationen nochmals die Korrektheit des gesamten Syntheseschritts mit dem in [BuDi94] geschilderten Verfahren überprüft.

Aus der Tatsache, dass durch diese zusätzlichen Verifikationsschritte einige Fehler in dem Synthesewerkzeug aufgedeckt werden konnten [Hinr98], wird ersichtlich, dass eine „sorgfältige“ Entwicklung und Implementierung der Transformationen alleine nicht ausreicht, um die Korrektheit der Ergebnisse sicherzustellen. Ein erster Schritt in Richtung eines zuverlässigeren transformationsbasierten Entwurfssystems ist daher zunächst einmal die formale Verifikation der Transformationen.

Einsatz formal verifizierter Transformationen

In vielen transformationsbasierten Systemen werden Transformationen eingesetzt, die bei ihrem Entwurf intuitiv als „offensichtlich korrekt“ angesehen [McFa93] und höchstens informell verifiziert wurden. Um sicherzugehen, dass die Transformationen tatsächlich korrekt arbeiten, bietet sich eine Verifikation der Transformationen im Rahmen eines Theorembeweisers an. Durch die maschinelle Unterstützung wird jegliche Intuition bei der Beurteilung der Korrektheit der Transformationen ausgeschlossen und die Korrektheitsbeweise können jederzeit objektiv nachvollzogen werden. Auf die Zuverlässigkeit mittels eines Theorembeweisers geführter Beweise wird in Abschnitt 3.1 näher eingegangen.

TRADES: In [Raja95a, Raja95b, Raja95c, MiRa96] wird die Verifikation einiger graphenbasierter Optimierungstransformationen auf der algorithmischen Ebene, wie z. B. das Entfernen gemeinsamer Teilausdrücke oder Konstantenpropagierung, präsentiert, die in dem Synthese-System *TRADES* (TRAnsformationel DErivation System) zum Einsatz kommen. Eine formale Spezifikation der Semantik der Graphen, welche in einer Variante der Sprache *SPRITE Input Language* (SIL) [KMNS92] modelliert werden, wird nicht durchgeführt. Es werden lediglich einige Axiome intuitiv aufgestellt, die grundlegende Eigenschaften der Graphen beschreiben, welche sowohl vor als auch nach deren Transformation zutreffen müssen. Die Axiome werden in dem Theorembeweiser PVS (Prototype Verification System) [OwRS92] spezifiziert und bilden die Basis für die Verifikation der ebenfalls in PVS spezifizierten Transformationen.

Wie sich später herausstellen sollte, war das zunächst entwickelte Graphenmodell offenbar zu abstrakt. Nach der Entwicklung eines detaillierteren Modells der Graphen, wurden Unzulänglichkeiten in den bislang als „korrekt bewiesenen“ Transformationen gefunden [MHMP96, Midd97]. In diesem Zusammenhang stellt sich die Frage, ob das neu geschaffene Modell nun alle erforderlichen Aspekte hinreichend in Betracht zieht.

DSS: Teica und Vemuri stellen in [TeVe01, TeVe01b] den Beweis für die Korrektheit von sieben Transformationen auf der RT-Ebene vor. Die verwendete RT-Modellierung ist auf reine Datenflussbeschreibungen beschränkt. Schleifen und Verzweigungen sind nicht darstellbar. Die Transformationen sind nicht-interpretierend, d. h. die Funktionen der verwendeten Operationen werden außer Acht gelassen. Somit sind Optimierungen, die sich beispielsweise Eigenschaften wie Assoziativität oder Kommutativität arithmetischer Operationen zu Nutze machen, nicht durchführbar. Die Transformationen sind rein struktureller Natur.

Die Semantik der RT-Beschreibungen und alle Transformationen wurden formal in dem Theorembeweiser PVS definiert. Anschließend wurde die Korrektheit und die Vollständigkeit der Transformationen mit Hilfe des Theorembeweisers nachgewiesen.

Die Transformationen kommen in dem Synthesewerkzeug *DSS* [RKDV92] zum Einsatz. Dort werden sie allerdings nicht direkt für die Durchführung der Synthese verwendet, sondern ausschließlich für die Überprüfung der Ergebnisse der Ablaufplanung und der Ressourcenbereitstellung [RaTV00, TeRV01, RaTv01].

Die formale Verifikation der in einem System verwendeten Transformationen trägt beträchtlich zur Erhöhung der Zuverlässigkeit des Systems bei. Es besteht aber immer noch das Risiko, dass sich im Zuge der Implementierung der Transformationen und deren Integration in ein zumeist komplexes Synthesesystem Fehler einschleichen. Entsprechende Fehler können wiederum die Funktionsweise der Transformationen beeinträchtigen. Mit Hilfe eines Theorembeweisers kann jedoch auch diese Fehlerquelle ausgeschlossen werden.

Formale Synthese

In der formalen Synthese wird die Korrektheit der Syntheseergebnisse sichergestellt, indem der komplette Ableitungsprozess in einen Theorembeweiser eingebettet wird. Der Theorembeweiser wird nicht nur genutzt, um die Korrektheit der Transformationen nachzuweisen, sondern auch um deren korrekte Ausführung zu überwachen. Zusätzlich zu einer Implementierung liefert der formale Syntheseprozess automatisch den formalen Beweis, dass die Implementierung die Vorgaben ihrer Spezifikation erfüllt.

Während es auf den unteren Abstraktionsebenen, der RT- und Gatterebene, bereits einige Arbeiten im Bereich der formalen Synthese gibt (siehe [Eise99]), befassen sich bis heute nur wenige Ansätze mit der formalen Synthese algorithmischer Schaltungsbeschreibungen.

MINI-ADDL: Ein formales Synthesesystem zur Optimierung algorithmischer Schaltungsbeschreibungen wird in [Lars96] vorgestellt. Als formale Hardwarebeschreibungssprache bettet der Autor die imperative Sprache *MINI-ADDL* in den Theorembeweiser HOL [GoMe93] ein. MINI-ADDL entspricht einer leicht erweiterten Untermenge der Sprache ADDL (Algorithmic Design Description Language), welche ursprünglich als Spezifikationssprache für das Synthesesystem CAMAD [PeKu94] entwickelt wurde.

In seiner Arbeit präsentiert der Autor einige Transformationen zur Modifizierung von MINI-ADDL-Programmen, welche es dem Benutzer ermöglichen, interaktiv algorithmische Optimierungen an einer Spezifikation vorzunehmen. Eine Abbildung der Spezifikation auf niedrigere Abstraktionsebenen wird nicht durchgeführt.

Die Transformationen basieren auf in HOL bewiesenen Theoremen und werden durch verhaltenserhaltende Termersetzungen von Untertermen der Spezifikation realisiert. Für die Anwendung mancher Transformationen müssen bestimmte Vorbedingungen erfüllt sein, deren Gültigkeit in dem System aber nur teilweise automatisch bewiesen wird. Oft muss der Benutzer eingreifen und die Erfüllung entsprechender Vorbedingung interaktiv im Theorembeweiser nachweisen.

Die Anwendung dieses Systems setzt weitreichende Kenntnisse in der Durchführung formaler Beweise mit dem Theorembeweiser HOL voraus. Eine automatisierte Anwendung der Transformationen ist nicht vorgesehen.

Larsson hatte früher bereits einen anderen Ansatz zur formalen Synthese in HOL [Lars94, Lars95] vorgestellt. Dabei handelt es sich allerdings um eine rein exemplarische und zudem sehr eingeschränkte Synthese einfacher, steuerflussloser Schaltungsbeschreibungen. Dieser Ansatz soll daher nicht weiter betrachtet werden.

Wenn auch die Arbeiten von Larsson nur einen kleinen Ausschnitt der im Zuge einer algorithmischen Synthese anfallenden Aufgaben behandeln, so kann er immerhin deren fehlerfreie

Ausführung garantieren. Dies ist ein großer Vorteil im Vergleich zu den bisher beschriebenen Arbeiten.

Ein wesentlich umfassenderes System zur formalen Synthese digitaler Schaltungen wurde im Rahmen eines Forschungsprojekts am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe entwickelt. Das System trägt den Namen HASH (**H**igher order logic **A**pplied to **S**ynthesis of **H**ardware).

HASH: Mit der Entwicklung des Synthesesystems HASH wurde eine Basis für die formale Repräsentation digitaler Schaltungen auf allen vier Abstraktionsebenen sowie für eine durchgehende formale Synthese von der System- bis zur Gatterebene geschaffen [Eise99, Blum00]. In HASH erfolgt die Synthese größtenteils automatisch, eine interaktive Steuerung ist aber ebenfalls möglich.

Die Synthese beginnt mit der Spezifikation der gewünschten Schaltung als logischer Term direkt im Theorembeweiser HOL (siehe Abschnitt 3.1). Als Spezifikationssprache wird die formale Hardwarebeschreibungssprache GROPIUS verwendet [BlEi98]. Alle Konstrukte von GROPIUS sind in Logik höherer Ordnung definiert, so dass die Sprache eine mathematisch exakte Semantik hat.

Die Korrektheit sämtlicher Transformationen, welche bei der Durchführung der Synthese zum Einsatz kommen, wurde im Theorembeweiser nachgewiesen. Das Synthesesystem liefert zu jedem Optimierungs- oder Syntheseschritt gleichzeitig den Beweis für dessen Korrektheit. Die genaue Vorgehensweise wird ausführlich in Abschnitt 3.2 beschrieben.

In HASH ist die formale Spezifikation und Synthese von Schaltungen nicht auf reine Datenflussbeschreibungen beschränkt. Steuerflussintensive Verhaltensbeschreibungen mit zyklischem Steuerfluss können ebenfalls in dem System spezifiziert und verarbeitet werden. So gibt es auf der algorithmischen Ebene in GROPIUS u. a. das Konstrukt **`WHILE`** zur Spezifikation μ-rekursiver bzw. while-berechenbarer Funktionen (siehe Abschnitt 3.3.3).

Im Folgenden wird näher auf das in [Blum00] vorgestellte Verfahren zur algorithmischen Synthese in HASH eingegangen. Dieses stellt den Ausgangspunkt der vorliegenden Arbeit dar.

Auf der algorithmischen Ebene besteht eine Schaltungsspezifikation in HASH aus zwei Teilen: aus einer algorithmischen GROPIUS-Beschreibung, welche die Funktionalität der gewünschten Schaltung festlegt, sowie einem Schnittstellenverhaltensmuster, das vorgibt, wie die Steuerung der algorithmischen Beschreibung erfolgt und wie deren Kommunikation mit der Außenwelt abläuft.

Bei der algorithmischen Synthese wird in [Blum00] strikt zwischen der Synthese reiner Datenflussbeschreibungen und der Synthese steuerflussbehafteter Beschreibungen unterschieden. Während die Synthese von Datenflussbeschreibungen weitestgehend automatisch abläuft, erfordert die Synthese steuerflussbehafteter Schaltungsbeschreibungen den manuellen Eingriff des Schaltungsentwicklers.

Im Falle steuerflussbehafteter Schaltungen erfolgt die algorithmische Synthese in vier Phasen. Zuerst wird im Zuge einer sogenannten Optimierungs-Programm-Transformation (OPT) die Verhaltensbeschreibung der Schaltung in eine äquivalente, optimierte Beschreibung überführt.

Die sich ergebende Schaltungsbeschreibung wird in einem zweiten Schritt, der sogenannten Transformation in Ein-Schleifen-Form (ESF), in eine wiederum äquivalente Darstellung überführt, die aus einer einzigen Schleife mit einem steuerflusslosen Rumpf besteht: Steuerstrukturen in der Verhaltensbeschreibung werden im Verlauf dieser Transformation schrittweise entfernt und durch boolesche Hilfsvariablen zur Speicherung der Steuerinformationen und Multiplexern, welche im Rumpf der resultierenden Schleife die Steuerinformationen umsetzen, ersetzt. In diesen ersten beiden Schritten erfolgt implizit eine zeitliche Einordnung der Operationen in Steuerschritte sowie die Bereitstellung und Zuordnung von Registern. Die resultierende Ein-Schleifen-Form der Spezifikation wird daraufhin durch Anwenden eines sogenannten Implementierungstheorems auf die Register-Transfer-Ebene abgebildet. Abschließend wird im kombinatorischen Teil der RT-Implementierung die Bereitstellung und Zuordnung von Funktionseinheiten durchgeführt.

Dieses Konzept zur formalen algorithmischen Synthese weist einige schwerwiegende Nachteile auf. Um diese nachvollziehen zu können, ist eine detailliertere Betrachtung des Verfahrens erforderlich. So wird im Folgenden zunächst die Transformation der Spezifikation in Ein-Schleifen-Form näher beschrieben und im Anschluss daran die aus dieser Vorgehensweise resultierenden Anforderungen an die zuvor auszuführende Optimierungs-Programm-Transformation.

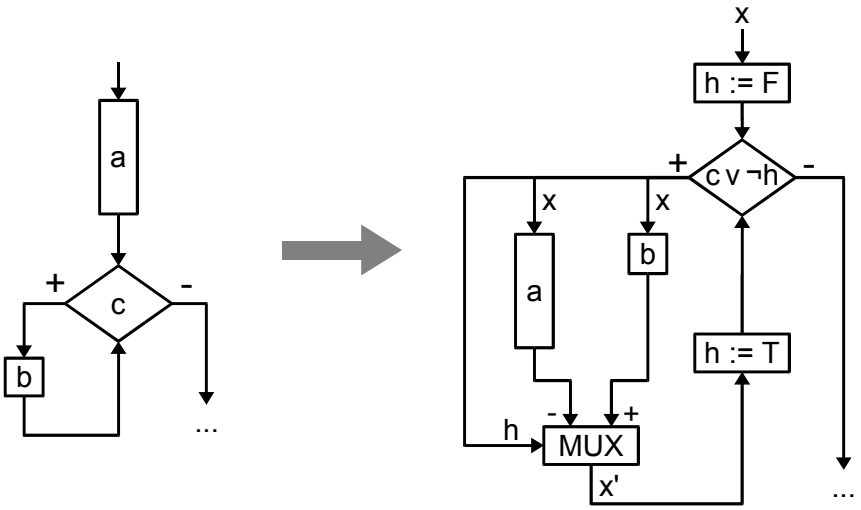


Abbildung 2.1: Beispiel einer Umformung während der ESF-Transformation

Ein Beispiel für eine bei dem Übergang zur ESF verwendete Transformation ist in Abbildung 2.1 schematisch dargestellt. Links ist eine Programmstruktur bestehend aus einem Block a und einer nachfolgenden Schleife mit einer Bedingung c und einem Rumpf b dargestellt. Bei dem Übergang zur ESF wird eine solche Struktur in die auf der rechten Seite dargestellte Programmstruktur überführt. In der neuen Struktur wird zunächst eine boolesche Hilfsvariable h eingeführt, welche mit dem Wert **F** für *falsch* initialisiert wird. Der Körper der neuen Schleife wird genau dann ausgeführt, wenn entweder die ursprüngliche Schleifenbedingung c *wahr* ist oder wenn die Hilfsvariable h den Wert **F** hat. Somit bewirkt die Initialisierung der Hilfsvariable

auf den Wert `F`, dass der Schleifenkörper mindestens einmal durchlaufen wird. Im Körper der Schleife werden sowohl der Block a als auch der Block b ausgeführt. Anschließend selektiert ein im Rumpf neu hinzugefügter Multiplexer in Abhängigkeit von der Hilfsvariable h eines der beiden Ergebnisse der Blöcke a und b. Bei der ersten Iteration ist der Wert der Hilfsvariable `F`, so dass in diesem Fall stets das Ergebnis des Blocks a selektiert und weitergeleitet wird. Am Ende des Schleifenrumpfes wird der Wert der Hilfsvariable auf `T` (*wahr*) gesetzt. Dies bewirkt, dass ab diesem Zeitpunkt die Durchführung aller weiteren Iterationen allein von der ursprünglichen Schleifenbedingung c abhängt, und der Multiplexer im Rumpf der Schleife grundsätzlich nur noch das Ergebnis des ursprünglichen Schleifenrumpfes b selektiert.

Die Überführung anderer Steuerstrukturen in die ESF läuft nach dem gleichen Prinzip ab: Es werden zusätzliche boolesche Hilfsvariablen eingeführt und in deren Abhängigkeit das Ergebnis des passenden Blocks am Ende des Schleifenrumpfes mit Hilfe zusätzlicher Multiplexer selektiert.

Ein entscheidender Nachteil dieser Vorgehensweise ist, dass alle Steuerstrukturen der Spezifikation, wie im Beispiel die Ausführung der Schleife *nach* Ausführung des Blocks a, in Datenselektionen durch Multiplexer überführt werden. Aus Steuerabhängigkeiten werden also Datenabhängigkeiten. So müssen im Rumpf der im Beispiel resultierenden Schleife immer sowohl der Block a als auch der Block b parallel ausgeführt werden, bevor der Multiplexer das passende Ergebnis selektieren kann. Folglich müssen ausreichend Hardwareressourcen bereitgestellt werden, um sowohl Block a als auch Block b gleichzeitig berechnen zu können. Wird dagegen der im Beispiel vom Entwerfer vorgegebene Steuerfluss während der Synthese aufrechterhalten, so kann eine exklusive Ausführung der beiden Blöcke implementiert und somit eine gemeinsame Nutzung von Hardwareressourcen durch die beiden Blöcke realisiert werden.

Weiter hat die ESF-Transformation den Nachteil, dass die Ausführungszeit des Schleifenrumpfes der Ausführungszeit des längsten enthaltenen Blocks entspricht. Eine Schleifeniteration entspricht wiederum nach der Abbildung auf die Register-Transfer-Ebene genau einem Takt, so dass der längste enthaltene Block im Schleifenrumpf letztendlich eine Unterschranke für die Dauer eines Taktes auf RT-Ebene darstellt. Ist wie im Beispiel angedeutet der Block a sehr lange und der Block b nur kurz, so muss im Schleifenrumpf dennoch grundsätzlich auf das Ende der Berechnung des längeren Blocks a gewartet werden, da erst dann der Multiplexer seine Selektion tätigen kann.

Die Überführung einer Spezifikation in Ein-Schleifen-Form ist eindeutig. Es ist Aufgabe der vorher auszuführenden Optimierungs-Programm-Transformation die Spezifikation derart umzuformen und zu optimieren, dass die aus der ESF-Transformation resultierende Schaltung für eine gegebene Kostenfunktion optimal wird. Als Grundlage für die Optimierungs-Programm-Transformation existieren zwar bereits einige Optimierungstheoreme, welche bekannte Umformungen aus dem Bereich des Compiler-Baus, wie z. B. das Aufrollen von Schleifen, nachbilden [AhSU86]. Allerdings gibt es bisher weder auf diesen Ansatz abgestimmte Entwurfsraumuntersuchungsmethoden noch ein Konzept für eine zielgerichtete Anwendung der Optimierungs-Transformationen. Auf Grund der großen Abweichung dieses Ansatzes von gängigen Syntheseverfahren ist überdies der Einsatz konventioneller Synthesealgorithmen kaum möglich.

2.2.4 Zusammenfassung

Aus den vorgestellten Ansätzen geht hervor, dass sich korrekte Syntheseergebnisse im Rahmen einer algorithmischen Synthese am ehesten durch die Vermeidung von Fehlern während der Durchführung der Synthese gewährleisten lassen. Eine nachträgliche Verifikation der Implementierung bietet sich aus Komplexitätsgründen in aller Regel nicht an. Auch ist eine Softwareverifikation der Synthesewerkzeuge auf Grund deren Größe und Komplexität im Allgemeinen nicht möglich. So erweist sich der transformationsbasierte Entwurf als der vielversprechendste Ansatz zur Erzeugung garantiert korrekter Implementierungen.

Grundvoraussetzung für die fehlerfreie Funktionsweise eines transformationsbasierten Systems ist die Korrektheit der eingesetzten Transformationen und deren korrekte Ausführung. Zur Sicherstellung der Korrektheit der Transformationen bietet sich ein formaler Beweis an. Um menschliche Fehler bei der Beweisführung auszuschließen, empfiehlt sich der Einsatz eines Theorembeweisers. Neben der Korrektheit der Transformationen an sich muss zusätzlich deren fehlerfreie Ausführung garantiert werden. Eine „sorgfältige“ Implementierung der Transformationen reicht dazu nicht aus. Jedoch bietet auch in diesem Fall der Einsatz eines Theorembeweisers die Möglichkeit, die korrekte Ausführung der Transformationen zu gewährleisten.

In der formalen Synthese wird der komplette Ableitungsprozess, im Zuge dessen eine Implementierung aus einer Spezifikation abgeleitet wird, in einen Theorembeweiser eingebettet. Die Korrektheit jedes einzelnen Transformationsschritts wird während der Durchführung der Synthese mit Hilfe des Theorembeweisers formal nachgewiesen. Durch diese Vorgehensweise erhält man als Ergebnis der Synthese den formalen Beweis, dass die resultierende Implementierung ihre Spezifikation erfüllt.

Das bisher am weitesten entwickelte System zur formalen Synthese digitaler Schaltungen ist HASH. Im Bereich der algorithmischen Synthese steuerflussbehafteter Schaltungsbeschreibungen weist dieses System allerdings noch einige Unzulänglichkeiten auf. Zum einen schränkt die bisherige Vorgehensweise die Optimierungsmöglichkeiten, die sich normalerweise im Lauf einer algorithmischen Synthese ergeben, stark ein. Zum anderen weicht diese Vorgehensweise so stark von gängigen Verfahren ab, dass eine Anbindung konventioneller Synthese- und Optimierungsalgorithmen kaum in Frage kommt. Die einzigen Optimierungstransformationen, mit denen Einfluss auf die Qualität der Implementierung genommen werden kann, sind im Wesentlichen einige interaktiv anzuwendende algorithmische Optimierungen. Eine zielgerichtete, automatisierte Anwendung dieser Optimierungstransformationen ist nicht vorgesehen.

In dieser Arbeit wird daher ein neues Konzept zur Durchführung der formalen algorithmischen Synthese steuerflussbehafteter Schaltungsbeschreibungen vorgestellt. Es erlaubt die vollständige Automatisierung der formalen algorithmischen Synthese und lehnt sich dabei stark an gängige Syntheseverfahren an, so dass auch klassische Synthesealgorithmen zur Steuerung des Entwurfsprozesses hinzugezogen werden können. Ziel ist eine vollautomatische algorithmische Synthese, welche bewiesenermaßen korrekte Implementierungen hoher Qualität erzeugt.

Kapitel 3

Grundlagen der formalen Synthese

Als Fundament für die Durchführung der formalen Synthese kommt in der vorliegenden Arbeit der Theorembeweiser HOL [GoMe93] (**H**igher **O**rder **L**ogic) zum Einsatz. Nach einer kurzen Einführung in HOL wird gezeigt, wie die Korrektheit der einzelnen Synthesetransformationen mit Hilfe dieses Theorembeweisers sichergestellt und auf deren Basis eine korrekte formale Synthese durchgeführt werden kann. Anschließend werden die verschiedenen Schaltungsrepräsentationen vorgestellt, die in dieser Arbeit zur formalen Darstellung digitaler Schaltungen auf der Gatter-, RT- und algorithmischen Ebene eingesetzt werden.

3.1 Der Theorembeweiser HOL

Die im Theorembeweiser HOL verwendete Logik höherer Ordnung wurde von Mike Gordon an der Universität Cambridge entwickelt [Gord85]. Sie basiert auf fünf Axiomen und acht Inferenzregeln. Ursprünglich war HOL primär für die Spezifikation und Verifikation von Hardware gedacht, mittlerweile wird HOL aber auch in vielen anderen Bereichen eingesetzt.

HOL ist ein direkter Abkömmling des Theorembeweisers LCF (**L**ogic for **C**omputable **F**unctions), welcher Anfang der 70er an den Universitäten Edinburgh und Stanford von Robin Milner entwickelt wurde [GMWa79]. Mit LCF wurde damals die Programmiersprache ML (**M**eta-**L**anguage) entworfen [GMMN78, MiTM97], auf die das HOL-System auch heute noch aufbaut. Da Logik höherer Ordnung nicht entscheidbar ist, ist HOL grundsätzlich für eine interaktive Nutzung ausgelegt. Mit Hilfe der zu Grunde liegenden Programmiersprache ML ist allerdings eine Automatisierung bestimmter Beweisverfahren möglich.

ML ist eine universell einsetzbare, streng-typisierte, funktionale Programmiersprache. Das HOL-System stellt als Sammlung einiger Bibliotheken eine Erweiterung der Sprache ML dar. Mit diesen Bibliotheken wird unter anderem der abstrakte Datentyp thm für Theoreme der Logik höherer Ordnung eingeführt. Es gibt nur fünf vordefinierte Werte des Typs thm. Diese entsprechen den von Gordon ausgewählten fünf grundlegenden Axiomen [Gord85] seiner Logik höherer Ordnung. Weitere Theoreme können ausschließlich auf Basis der in den Bibliotheken zur Verfügung gestellten Funktionen erzeugt werden. Deren korrekter Einsatz wird durch die strenge Typüberprüfung von ML sichergestellt.

Es stehen acht Funktionen zur Ableitung von Theoremen zur Verfügung, welche wiederum Gordons grundlegenden Inferenzregeln entsprechen. Zusätzlich werden zahlreiche komplexere Ableitungsfunktionen angeboten, welche aber alle letzten Endes auf die acht grundlegenden Ableitungsfunktionen zurückgreifen. Durch das Abspeichern abgeleiteter Theoreme in sogenannten Theorien kann die Basis der zur Verfügung stehenden Theoreme erweitert werden. Auf diese Weise müssen bereits abgeleitete Theoreme nicht bei jedem Start des Systems neu erzeugt werden.

Der kleine Kern von HOL, bestehend aus den fünf Axiomen und den acht Ableitungsfunktionen, gepaart mit der strengen Typisierung von ML und vor allem auch dem Einsatz des abstrakten Datentyps thm für Theoreme, führen zu einer extrem hohen Zuverlässigkeit der in HOL geführten Beweise. Laut Konrad Slind, einem der Entwickler von HOL, wurde in den letzten 10 Jahren der HOL-Entwicklung nur ein einziger schwerwiegenderer Fehler gefunden, der gegebenenfalls zu einem ,,falschen Beweis“ hätte führen können.

Die in dieser Arbeit verwendete Version des Theorembeweisers ist *HOL98-Taupo 6*. Ihr liegt die ML-Version *Moscow ML 2.01* zu Grunde, welche voll kompatibel zu *Standard ML* [MiTM97] ist.

Terme in HOL

Terme sind in HOL gemäß dem λ-Kalkül nach Church [Chur40] aufgebaut und sind streng typisiert. Sie basieren auf Variablen, Konstanten, Applikationen (Funktionsanwendungen) und λ-Abstraktionen. Abbildung 3.1 zeigt den Aufbau von Termen in HOL*.

```
λ-Abstraktion = "( λ" Variable "." Term ")"
Applikation   = "(" Term Term ")"
Term          = Konstante |
                Variable |
                Applikation |
                λ-Abstraktion
```

Abbildung 3.1: Syntax der Terme in HOL

Eine Konstante bezeichnet entweder einen konstanten Wert, wie z. B. die boolesche Konstante **`T`** für *wahr*, oder einen fest definierten Term. So steht beispielsweise die Konstante **`AND`** für eine Funktion, welche die boolesche Konjunktion ausführt. Eine Variable ist ein Platzhalter für einen beliebigen Term.

Eine Applikation $(t_1\ t_2)$ bezeichnet das Ergebnis der Anwendung der Funktion t_1, dem *Operator*, auf den Wert des Terms t_2, dem *Operanden*. Applikationen sind linksassoziativ, so dass $t\ t_1\ t_2 \ldots t_n$ als Abkürzung für $(\ldots((t\ t_1)\ t_2)\ldots t_n)$ gilt.

Eine λ-Abstraktion $\lambda x.t$ bezeichnet eine Funktion $v \mapsto t[v/x]$ mit dem *Parameter* x und dem *Funktionsrumpf* t. Der Term $t[v/x]$ steht für den Term t, in dem alle *freien* Vorkommen

*In dieser Arbeit werden alle Syntaxdefinitionen in der EBNF, der *erweiterten Backus-Naur-Form*, dargestellt.

der Variablen x durch v substituiert wurden. Der Parameter einer λ-Abstraktion wird auch als *Abstraktionsvariable* bezeichnet. Eine Variable x wird innerhalb eines Terms als *gebunden* bezeichnet, wenn sie im Geltungsbereich eines vorangehenden λx liegt, also zuvor als Abstraktionsvariable deklariert wurde. Ansonsten wird sie als *frei* bezeichnet. Zum Beispiel ist das Auftreten der Variablen x in dem Term $\lambda x.(f(x,y))$ durch das λx gebunden, während das Auftreten der Variablen y frei ist. Ein Term, in welchem alle Variablen gebunden sind, wird als *geschlossen* bezeichnet. Zur Abkürzung von Termen der Form $(\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.t)\ldots))$ wird auch die Notation $\lambda x_1 x_2 \ldots x_n.t$ verwendet.

Eine erhebliche Verbesserung der Lesbarkeit von λ-Abstraktionen ermöglicht die Verwendung des **`let`**-Konstrukts. Es gilt: $(\texttt{let}\ x = p\ \texttt{in}\ M) := ((\lambda x.M)\ p)$. Dem Parameter x wird also explizit durch die **`let`**-Anweisung der Wert des Terms p zur Verwendung im *Rest* M des Terms zugewiesen. Die vereinfachende Wirkung dieser Schreibweise wird vor allem bei geschachtelten Abstraktionen deutlich, wie das folgende Beispiel zeigt:

$$\begin{aligned}((\lambda x_1.(\lambda x_2.(\lambda x_3.M)\ p_3)\ p_2)\ p_1) \quad = \quad & \texttt{let}\ x_1 = p_1\ \texttt{in} \\ & \texttt{let}\ x_2 = p_2\ \texttt{in} \\ & \texttt{let}\ x_3 = p_3\ \texttt{in}\ M\end{aligned}$$

Ein Term der Form $(\lambda x.t)\ s$, also eine Applikation, deren Operator eine λ-Abstraktion ist, wird als β-*Redex* (**red**ucible **ex**pression) bezeichnet und kann β-*reduziert* werden. Eine β-Reduktion bedeutet informell das „Ausführen“ der Applikation und entspricht dem Ersetzen des ursprünglichen Terms $(\lambda x.t)\ s$ durch den Term $t[s/x]$. Eine solche Konversion ist allerdings nur dann zulässig, wenn durch die Substitution von x durch s keine freien Variablen in s gebunden werden. Beispielsweise ist die Substitution der Variablen y in dem Term $(\lambda x.x + y)$ durch den Term $(x + 2)$ nicht zulässig, da die freie Variable x in $(x + 2)$ im resultierenden Term $(\lambda x.x+(x+2))$ durch die vorangehende Deklaration der gleichlautenden Abstraktionsvariable x gebunden wäre. Das Problem kann leicht umgangen werden, indem vor der β-Reduktion die betroffene Abstraktionsvariable umbenannt wird. Diese Art der Umformung wird als α-*Konversion* bezeichnet. Nach der Umbenennung der Abstraktionsvariable x im Term $(\lambda x.x+y)$ in beispielsweise x' erhält man den Term $(\lambda x'.x'+y)$. Eine Substitution von y durch $(x+2)$ ist nun zulässig. Entsprechende Umbenennungen nimmt der Theorembeweiser HOL bei Bedarf automatisch vor.

Eine weitere Art der Umformung von Termen ist die sogenannte η-*Konversion*. Sie reduziert einen η-*Redex*, einen Term der Form $(\lambda x.R\ x)$, auf den Term R, falls x in R nicht frei vorkommt.

Typen in HOL

In der Logik höherer Ordnung sind alle Terme streng typisiert. Das folgende Beispiel soll die Wichtigkeit einer strengen Typisierung deutlich machen. Das Beispiel trägt den Namen „Russells Paradoxon“. Gegeben sei eine Definition der Funktion Ω wie folgt:

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

Die Logik höherer Ordnung erlaubt eine Anwendung der Funktion Ω auf sich selbst. In diesem Fall lautete das Ergebnis:

$$\begin{aligned} \Omega\,\Omega &= (\lambda P.\neg(P\ P))\ \Omega \\ &= \neg(\Omega\,\Omega) \end{aligned}$$

Im ersten Schritt wird das linke Ω durch seine Definition ersetzt. Anschließend wird eine β-Reduktion durchgeführt. Die resultierende Aussage $\Omega\ \Omega = \neg(\Omega\ \Omega)$ ist offensichtlich widersprüchlich. Durch eine strenge Typisierung der Terme in der Logik höherer Ordnung werden derartige Inkonsistenzen unterbunden. Eine Definition der Funktion Ω wie oben ist mit typisierten Termen erst gar nicht möglich.

In der Logik höherer Ordnung hat jeder Term einen fest zugeordneten Typ, der seine Wertemenge definiert. Abbildung 3.2 zeigt die Syntax der Typen in HOL.

Typkomposition	= "**(**" Typ {"**,**" Typ } "**)**" Typoperator
Funktionstyp	= Typ "→" Typ
Typ	= Atomarer_Typ |
	Typvariable |
	Typkomposition |
	Funktionstyp

Abbildung 3.2: Syntax der Typen in HOL

Atomare Typen sind konstante Typen und bezeichnen einen festen Wertebereich. Die Typkonstante $bool$ bezeichnet beispielsweise den booleschen Wertebereich $\{\mathbf{T}, \mathbf{F}\}$. Typvariablen dagegen sind Platzhalter für einen beliebigen, festen Typ und können mit konkreten Typen instantiiert werden. Ein Term, der eine Typvariable enthält, wird als *polymorph* bezeichnet. Ansonsten heißt der Term *monomorph*.

Aus existierenden Typkonstanten können mit Hilfe sogenannter *Typoperatoren* neue Typen erzeugt werden. Entsprechende Typkompositionen haben die Form $(\sigma_1, \ldots, \sigma_n)op$, wobei op einen Typoperator der Wertigkeit n bezeichnet und $\sigma_1, \ldots, \sigma_n$ die zugehörigen Typoperanden. Ein wichtiger Typoperator ist beispielsweise $prod$. Er besitzt die Wertigkeit 2 und bildet das kartesische Produkt zweier Typen. Ein Typ $(\sigma_1, \sigma_2)prod$ wird im Allgemeinen als $\sigma_1 \times \sigma_2$ dargestellt.

Der Funktionstyp $\sigma_1 \rightarrow \sigma_2$ bezeichnet den Typ einer (totalen) Funktion mit dem Definitionsbereich σ_1 und dem Wertebereich σ_2. Der Typoperator $\rightarrow$ ist rechtsassoziativ, so steht $\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$ abkürzend für $\sigma_1 \rightarrow (\sigma_2 \rightarrow (\cdots \rightarrow (\sigma_n \rightarrow \sigma) \ldots))$.

Wird der Typ eines Terms explizit dargestellt, so wird er durch einen Doppelpunkt getrennt hinter dem Term angegeben, wie zum Beispiel in $\lambda(a, b).\mathbf{AND}(a, b){:}bool \times bool \rightarrow bool$.

Nach dieser kurzen Einführung in den Theorembeweiser HOL wird im Folgenden beschrieben, wie auf Basis von HOL die Durchführung einer formalen Synthese realisiert werden kann.

3.2 Durchführung der formalen Synthese

Der Begriff *Formale Synthese* bezeichnet die transformationsbasierte Ableitung einer Implementierung aus einer Schaltungsspezifikation innerhalb eines Theorembeweisers. Sie beginnt mit der Spezifikation der gewünschten Schaltung als logischer Term im Theorembeweiser. Auf der algorithmischen Ebene erfolgt die Spezifikation der Schaltung in der Regel als Verhaltensbeschreibung in GROPIUS (siehe Abschnitt 3.3.3). Im Laufe der Synthese wird die Spezifikation durch verhaltenserhaltende Transformationen optimiert und in eine für den Wechsel der Abstraktionsebene geeignete Form gebracht. Während dieser Phase handelt es sich bei allen Transformationen um Äquivalenzumformungen der Schaltung. Im Folgenden werden zunächst die verschiedenen Transformationsvarianten innerhalb einer Abstraktionsebene beschrieben. Anschließend werden die Transformationen, welche einen Wechsel der Abstraktionsebene bewirken, erläutert.

3.2.1 Transformationen innerhalb einer Abstraktionsebene

Jede Transformation entspricht der Anwendung einer speziellen ML-Funktion, einer sogenannten *Konversion*. Konversionen, welche keinen Wechsel der Abstraktionsebene bewirken, haben als Eingabe einen Term t, transformieren diesen in einen Term t', beweisen im Zuge dessen gegebenenfalls existierende Vorbedingungen, und geben letztendlich ein Theorem $\vdash t = t'$ aus. Dieses Theorem besagt, dass der ursprüngliche Term t äquivalent zu der transformierten Variante t' ist. Die transformierte Variante t' ist der Ausgangspunkt für die nächste Transformation, welche wiederum ein Theorem $\vdash t' = t''$ ergibt. Beide Theoreme werden anschließend auf Basis der Transitivität der Äquivalenzrelation zu einem Theorem $\vdash t = t''$ zusammengefasst. Damit erhält man den Beweis, dass der ursprüngliche Term t äquivalent zum Ergebnis t'' der letzten Transformation ist.

Entsprechend dieser Vorgehensweise wird der Ausgangsterm S_n einer Abstraktionsebene (siehe Abbildung 3.3) optimiert und transformiert, bis er in einer für den Wechsel der Abstraktionsebene geeigneten Form vorliegt. Ergebnis dieser Transformationen ist ein Theorem $\vdash S_n = S'_n$, welches die Äquivalenz zwischen dem ursprünglichen Ausgangsterm S_n und dem resultierenden Term S'_n bestätigt.

Im Folgenden wird im Detail auf die verschiedenen Arten von Transformationsschritten, welche Transformationen einer Schaltung innerhalb einer Abstraktionsebene bewirken, eingegangen.

Transformationen durch direkte Theoremanwendung

Die einfachste Art der Umformung eines Terms ist die Anwendung eines Theorems der Form $\vdash t_1 = t_2$, also eines Theorems, welches die Äquivalenz zweier Terme ausdrückt. Beispiel eines entsprechenden Theorems ist das Theorem der Kommutativität der Multiplikation:

$$\texttt{MULT\_COMM} := \vdash !\, m\, n.\, m * n = n * m \tag{3.1}$$

Wird eine Transformation auf Basis eines solchen Theorems auf einem bestimmten Term angestoßen, so versucht HOL durch Mustererkennung (*pattern-matching*) den linken Teil t_1 der

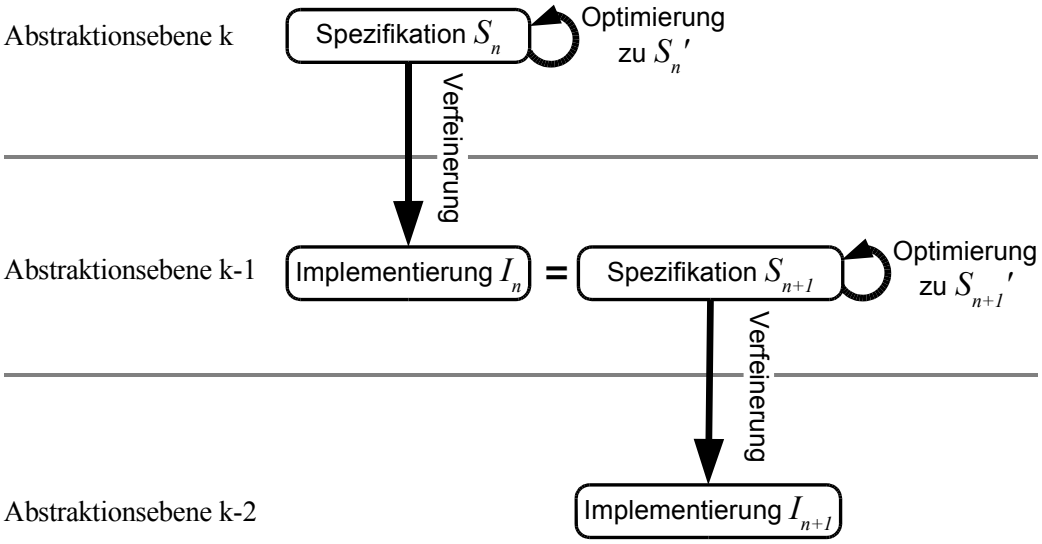


Abbildung 3.3: Optimierungen und Verfeinerungen in der formalen Synthese

im Theorem enthaltenen Gleichung mit dem gegebenen Term in Übereinstimmung zu bringen, indem die allquantifizierten Variablen des Theorems mit passenden Werten instantiiert werden. Gelingt dies, so ist eine Anwendung des Theorems auf den gegebenen Term möglich. Das Ergebnis einer solchen Konversion entspricht der auf den Eingabeterm spezialisierten Variante des eingesetzten Theorems.

Gegeben sei beispielsweise ein Term $7 * (b - 2)$. Sein Bezeichner sei t. Die Operanden der Multiplikation in t sollen mit Hilfe des Theorems 3.1 vertauscht werden. Zur direkten Anwendung eines Theorems dient in HOL die Funktion **REWR_CONV**. Eine Anwendung des Theorems **MULT_COMM** auf t ergibt:

$$(\textbf{REWR\_CONV MULT\_COMM t}) \ = \vdash 7 * (b - 2) = (b - 2) * 7$$

Es wurde erfolgreich eine Mustererkennung durchgeführt: Die Variable m des Theorems MULT_COMM wurde mit 7 und die Variable n mit dem Teilterm $(b - 2)$ instantiiert. Ergebnis ist ein Theorem, welches besagt, dass der ursprüngliche Term $7 * (b - 2)$ äquivalent zu dem resultierenden Term mit den vertauschten Operanden ist.

Transformationen durch Theoreme mit Annahmen/Prämissen

Etwas komplizierter gestaltet sich der Einsatz von Theoremen der Form $A \vdash t_1 = t_2$, also Theoremen mit nichttrivialen Annahmen, bzw. Theoremen der Form $\vdash C \Rightarrow (t_1 = t_2)$, Theoremen mit Prämissen. Sie sollen ebenfalls der Überführung eines Terms der Form t_1 in einen Term der Form t_2 dienen. Für eine automatisierte Anwendung dieser Art von Theoremen müssen spezielle Konversionen entwickelt werden, in deren Verlauf die im Theorem enthaltenen Annahmen bzw. Prämissen automatisch bewiesen werden.

Ein Beispiel für ein Theorem mit einer Prämisse ist das Theorem **MOD_MULT'** (**MOD** steht in HOL für den Modulo-Operator):

$$\textbf{MOD\_MULT'} \;:= \vdash !\, n\, r\, q.\, r < n \Rightarrow ((q * n + r)\, \textbf{MOD}\, n = r) \tag{3.2}$$

Ein allgemeiner Ansatz für Konversionen, welche auf Theoremen dieser Form basieren, beginnt mit der Analyse des zu transformierenden Terms. Es werden die Werte des Eingabeterms ermittelt, welche den Variablen des anzuwendenden Theorems entsprechen. Mit diesen konkreten Werten werden schließlich die Variablen des Theorems instantiiert und die Annahmen bzw. Prämissen des nun spezialisierten Theorems mit Hilfe existierender Konversionen bewiesen. Das resultierende Theorem enthält daraufhin keine Annahmen oder Prämissen mehr und kann, wie im letzten Abschnitt beschrieben, auf den ursprünglichen Eingabeterm angewandt werden.

In Abbildung 3.4 ist als Beispiel eine Konversion dargestellt, welche die automatische Umformung eines Terms auf Basis des Theorems **MOD_MULT'** durchführt. Die Funktionsweise der Konversion soll anhand des Eingabeterms $((b-2) * 7 + 3)$ **MOD** 7 demonstriert werden.

```
fun MOD_MULT'_CONV st =
  let
    val n = rand st
    val r = rand(land st)
    val q = land(land(land st))
    val specThm = ISPECL [n,r,q] MOD_MULT'
    val rwThm = CONV_RULE ((LAND_CONV ARITH_CONV)
                                THENC IMP_CONV) specThm
  in
    REWR_CONV rwThm st
  end
```

Abbildung 3.4: Beispiel einer Konversion zum Theorem **MOD_MULT'**

Als Erstes findet in der Konversion die Analyse des Eingabeterms statt, in welcher die Werte ermittelt werden, die den Variablen des Theorems **MOD_MULT'** entsprechen. Die Funktionen rand und land liefern dabei den rechten bzw. linken Operanden eines Terms. Für den Eingabeterm in diesem Beispiel ergeben sich für die Variable n der Wert 3, für r der Wert 7 und für die Variable q der Wert $b-2$. Mit Hilfe der Funktion ISPECL wird das Theorem **MOD_MULT'** mit den ermittelten Werten instantiiert und so auf den zu transformierenden Term spezialisiert. Das resultierende Theorem wird der Variablen specThm zugeordnet:

$$\text{specThm} = \vdash\; 3 < 7 \Rightarrow (((b-2) * 7 + 3)\, \textbf{MOD}\, 7 = 3)$$

Im nächsten Schritt wird die Prämisse des nun spezialisierten Theorems bewiesen. Die Funktion CONV_RULE erlaubt die Umformung eines Theorems mit Hilfe von Konversionen. Durch die Funktion LAND_CONV wird die Konversion ARITH_CONV nur auf den linken Operanden

der Implikation, den Teilterm $3 < 7$, angewandt. ARITH_CONV ist eine Konversion zur Auswertung bestimmter mathematischer Ausdrücke. In diesem Fall ersetzt sie den Teilterm $3 < 7$ durch **T**, also den Wert *wahr*. Man erhält: $\vdash$ **T** $\Rightarrow$ $(((b-2)*7+3)$ **MOD** $7 = 3)$. Der Operator THENC ermöglicht die sequentielle Ausführung von Konversionen, so dass die direkt nach ARITH_CONV ausgeführte Funktion IMP_CONV die Vereinfachung der Implikation **T**$\Rightarrow f$ zu f bewirkt. Resultat ist das Theorem $\vdash ((b-2)*7+3)$ **MOD** $7 = 3$. Dieses Theorem weist keine Prämisse mehr auf und wird abschließend mit dem Befehl REWR_CONV auf den ursprünglichen Eingabeterm angewandt. Man erhält: $\vdash ((b-2)*7+3)$ **MOD** $7 = 3$.

Das Ergebnis dieser Konversion gleicht zwar in diesem Fall dem in der Konversion abgeleiteten Theorem rwThm. Das Theorem rwThm wird am Schluss aber nochmals explizit auf den Eingabeterm angewandt, um sicherzustellen, dass bei der Analyse des Eingabeterms bzw. bei der Spezialisierung des Theorems **MOD_MULT'** keine Fehler unterlaufen sind.

Transformationen durch konstruierte Theoreme

Eine weitere Möglichkeit der Transformation bieten Konversionen, die ausgehend von einem Eingabeterm einen neuen Term in einer bestimmten Form aufbauen, und anschließend beweisen, dass der neue Term äquivalent zum Eingabeterm ist. Ergebnis der Konversion ist ein speziell konstruiertes Theorem, welches besagt, dass der ursprüngliche Term äquivalent zu seiner transformierten Variante ist.

Ein Beispiel für eine entsprechende Transformation ist in Abbildung 3.5 gegeben. Ziel dieser Konversion ist das Herausziehen des Operanden einer Funktion, so dass aus einer Applikation $f\ opr$ ein Term der Form $(\lambda v.f\ v)\ opr$ (bzw. **let** $v = opr$ **in** $(f\ v)$) wird. Diese Transformation wird später bei der Überführung einer GROPIUS-Spezifikation in eine bestimmte Darstellungsform benötigt (siehe Abschnitt 3.3.1).

```
fun EXTR_OPR_CONV ft =
  let
    val (f,opr) = dest_comb ft
    val v = genvar(type_of opr)
    val ft' = mk_comb(gen_mk_abs (var,mk_comb(f,v)),opr)
    val z = mk_eq (ft,ft')
  in
    prove(z, GEN_BETA_TAC THEN REFL_TAC)
  end
```

Abbildung 3.5: Beispiel einer konstruktiven Konversion

Die Funktionsweise der Konversion soll anhand eines Eingabeterms $p\ (u * 3)$ dargelegt werden. Am Anfang der Konversion wird der Eingabeterm mit der Funktion dest_comb in den Operator p und Operanden $(u * 3)$ unterteilt. Der Operator wird der Variablen f zugewiesen und der Operand der Variablen opr. Als nächstes wird die spätere Abstraktionsvariable erzeugt, welche den gleichen Typ wie der Operand (type_of opr) haben muss. Die Funktion genvar wählt in

diesem Zusammenhang einen eindeutigen, bisher nicht verwendeten Namen für die Abstraktionsvariable. In diesem Beispiel sei dieser Name s. Anschließend wird der Zielterm konstruiert. Die Funktion mk_comb erzeugt aus einer Funktion und einem Operanden eine Applikation und die Funktion gen_mk_abs erzeugt aus einer Variable und einem Term eine Abstraktion (siehe auch Abschnitt 3.1). Der Zielterm ergibt sich zu $(\lambda s.p\ s)(u * 3)$ und wird in der Variable ft' gespeichert.

Nun ist zu beweisen, dass dieser Term äquivalent zum ursprünglichen Eingabeterm ist, dass also gilt: $(p(u*3)) = ((\lambda s.p\ s)(u*3))$. Dieses Beweisziel wird aus dem ursprünglichen Eingabeterm ft und seiner neu konstruierten Variante ft' mit Hilfe der Funktion mk_eq konstruiert. In der prove-Anweisung wird mit Hilfe sogenannten Taktiken der entsprechende Beweis durchgeführt. Die Taktik GEN_BETA_TAC bewirkt eine β-Reduktion auf dem neu konstruierten Term, woraufhin dieser dem Eingabeterm gleicht. Zu beweisen bleibt somit $(p(u * 3)) = (p(u * 3))$. Die Gleichheit der beiden Terme wird durch die Taktik REFL_TAC bestätigt. Ausgabe der Konversion ist schließlich das Theorem $\vdash (p(u * 3)) = ((\lambda s.p\ s)(u * 3))$.

3.2.2 Transformationen zum Wechsel der Abstraktionsebene

Beim Wechsel der Abstraktionsebene findet eine Verfeinerung der Spezifikation statt. Zum Beispiel wird beim Wechsel von der algorithmischen Ebene zur Register-Transfer-Ebene die Zeit konkretisiert. Während auf der algorithmischen Ebene nur die Kausalität relevant ist, werden auf der RT-Ebene einzelne Takte betrachtet (siehe Kapitel 2). In diesem Fall ist die Implementierung nicht äquivalent zur Spezifikation. Stattdessen impliziert sie das in der Spezifikation definierte Verhalten. Entsprechend kommen spezielle *Implementierungstheoreme* der Form $\vdash Impl. \Rightarrow Spez.$ zum Einsatz. Diese Theoreme besagen, dass die verfeinerte Implementierung die abstraktere Spezifikation erfüllt.

Vor dem Wechsel der Abstraktionsebene muss ein Term S_n, welcher hier den Ausgangspunkt auf der abstrakteren Ebene repräsentiert, (vgl. Abbildung 3.3), in eine für das anzuwendende Implementierungstheorem geeignete Form überführt werden. Diese Überführung erfolgt auf Basis von Transformationen, wie sie im letzten Abschnitt beschrieben wurden. Ergebnis dieser Transformationen ist ein Theorem $\vdash S_n = S'_n$, wobei der Term S'_n die von dem anzuwendenden Implementierungstheorem geforderte Form aufweist.

Transformationen, welche einen Wechsel der Abstraktionsebene bewirken, ähneln den Transformationen, welche Theoreme mit Annahmen und Prämissen einsetzen. Sie beginnen ebenfalls mit der Analyse des Eingabeterms, welcher in diesem Fall der speziell geformte Term S'_n ist. Im Term S'_n werden die konkreten Werte ermittelt, welche den Variablen des anzuwendenden Implementierungstheorems entsprechen. Anschließend werden die Variablen des Implementierungstheorems mit den ermittelten Werten instantiiert und so das Implementierungstheorem auf den Eingabeterm spezialisiert. Ergebnis dieser Transformation ist ein Theorem $\vdash I_n \Rightarrow S'_n$. Der Term I_n entspricht der Implementierung auf der niedrigeren Abstraktionsebene.

Ersetzt man in dem resultierenden Theorem nun den Term S'_n durch den äquivalenten Ausgangsterm S_n, so erhält man das Theorem $\vdash I_n \Rightarrow S_n$. Dieses Theorem besagt, dass die resultierende Implementierung I_n die durch den Term S_n definierte Spezifikation erfüllt. Der Term I_n wird im weiteren Verlauf als Spezifikation S_{n+1} auf der tieferen Abstraktionsebene angese-

hen. Die beiden verschiedenen Bezeichner I_n und S_{n+1} werden auf Grund der unterschiedlichen Betrachtungsweisen verwendet, bezeichnen aber exakt den gleichen Term.

Eine weitere Optimierung und Verfeinerung der Spezifikation S_{n+1} ergibt wiederum ein Theorem $\vdash I_{n+1} \Rightarrow S_{n+1}$, bzw. $\vdash I_{n+1} \Rightarrow I_n$. Darin steht I_{n+1} für die Implementierung auf der unter dem Term S_{n+1} befindlichen Abstraktionsebene. Die Ergebnisse zweier Verfeinerungen, hier $\vdash I_{n+1} \Rightarrow I_n$ und $\vdash I_n \Rightarrow S_n$, werden auf Basis der Transitivität der Implikation zusammengefasst. Man erhält: $\vdash I_{n+1} \Rightarrow S_n$. Somit ergibt sich als Endergebnis des gesamten Syntheseprozesses stets ein Theorem, welches besagt, dass die letztendlich resultierende Implementierung die ursprüngliche Spezifikation erfüllt.

Die in dieser Arbeit eingesetzten Implementierungstheoreme und deren konkrete Anwendung werden ausführlich in Abschnitt 5.6 erläutert.

3.2.3 Programmierfehler und die formale Synthese

Ebenso wie konventionelle Synthesewerkzeuge (und Software im Allgemeinen) ist auch das formale Synthesesystem nicht vor Programmierfehlern sicher. Programmierfehler führen allerdings in diesem Ansatz im ungünstigsten Fall zu einem Syntheseprozess, der eventuell abbricht oder niemals terminiert. Weiter existiert die Möglichkeit, dass die durch Synthesealgorithmen getroffenen Entwurfsentscheidungen nicht korrekt umgesetzt werden, so dass die resultierende Implementierung nicht die beabsichtigten physikalischen Eigenschaften aufweist. Sie kann sich beispielsweise als Folge einer unkorrekten Umsetzung als zu langsam erweisen oder eine zu große Fläche erfordern. Fehler dieser Art sind aber in der Regel verhältnismäßig einfach durch eine Gegenüberstellung der ermittelten Entwurfsziele und der resultierenden Implementierung aufzudecken.

Anders als bei konventionellen Synthesewerkzeugen kann jedoch in der formalen Synthese niemals eine Implementierung erzeugt werden, deren Verhalten von dem in der Spezifikation definierten abweicht: Mit jeder Implementierung wird stets der Beweis geliefert, dass diese ihre Spezifikation erfüllt. Dieser Beweis wird automatisch durch das formale Synthesesystem innerhalb des Theorembeweisers HOL geführt. Infolgedessen entspricht die Zuverlässigkeit dieses Entwurfssystems exakt der extrem hohen Zuverlässigkeit des Theorembeweisers HOL (siehe Abschnitt 3.1).

3.3 Formale Repräsentation von Hardware

Grundvoraussetzung für eine formale Synthese ist eine formale Darstellung der Hardware. Ohne eine entsprechende Repräsentation ist ein formales Argumentieren über die Korrektheit von Transformationen nicht möglich. Da die Semantik gängiger Hardwarebeschreibungssprachen wie VHDL [VHDL02] oder Verilog [ThMo02] nur umgangssprachlich definiert ist, bieten sich diese als Spezifikationssprache eines formalen Synthesesystems nicht an. Aus diesem Grund wurde die formale Hardwarebeschreibungssprache GROPIUS entwickelt, deren Syntax und Semantik mathematisch exakt in der Logik höherer Ordnung des Theorembeweisers HOL definiert sind.

Auf Grund der unterschiedlichen Abstraktionsgrade der Entwurfsebenen existiert für jede Ebene eine spezielle Darstellungsform. Im Folgenden wird die formale Schaltungsrepräsentation auf der algorithmischen, der RT- sowie der Gatterebene ausführlich beschrieben. Da in dieser Arbeit die algorithmische Synthese im Vordergrund steht, wird auf Spezifikationen auf der Systemebene nicht näher eingegangen. In diesem Zusammenhang sei auf [Blum99, Blum00, SaBK01] verwiesen.

3.3.1 Kombinatorische Schaltungsbeschreibungen

Kombinatorische Schaltungen stellen einen funktionalen Zusammenhang zwischen ihren Ein- und Ausgängen her. Sie *kombinieren* die an ihrem Eingang anliegenden Größen zu einer entsprechenden Ausgangsgröße. Graphisch werden kombinatorische Schaltungen als zyklenfreie **D**aten**f**luss**g**raphen (DFG) dargestellt. Die gerichteten Kanten eines Datenflussgraphen beschreiben den Datenfluss von den Eingängen der Schaltung über Knoten, welche die auszuführenden Operationen repräsentieren, zu den Ausgängen. In der Realität entsprechen die Datenflusskanten den Signalleitungen, die die Funktionseinheiten miteinander verbinden.

Ein Beispiel für einen Datenflussgraphen zeigt Abbildung 3.6. Die Signale ***a***, ***b***, ***c*** und ***d*** stellen die Eingänge des Datenflussgraphen dar. Die Signale ***g***, ***h*** und ***i*** entsprechen den internen Signalen und ***r***, ***s***, ***t***, ***u*** und ***v*** den Ausgängen des Graphen.

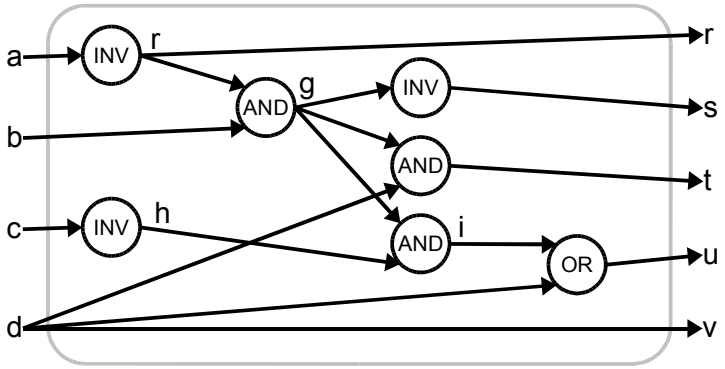


Abbildung 3.6: Ein Datenflussgraph

Die Operationen **`AND`**, **`OR`** und **`INV`**, welche in dem Datenflussgraphen zum Einsatz kommen, bilden in HASH die Grundlage für kombinatorische Schaltungsbeschreibungen auf der Gatterebene. **`AND`** bezeichnet die boolesche Konjunktion, **`OR`** die Disjunktion und **`INV`** die Negation. Auf der Gatterebene sind alle Signale booleschen Typs, können also ausschließlich die Werte **`T`** für *wahr* und **`F`** für *falsch* annehmen. Mit dieser Menge von Konstrukten kann jede beliebige boolesche Funktion beschrieben werden, d. h. sie bilden eine Basis im Sinne der Aussagenlogik. Die Auswahl gerade dieser Konstrukte als Basis geschah willkürlich. Genauso wären andere Basen, bestehend beispielsweise aus der Operation **`NAND`** (der negierten Konjunktion), denkbar gewesen.

DFG-Terme

Kombinatorische Schaltungen werden in GROPIUS durch sogenannte *DFG-Terme* dargestellt [EiKu95]. Wie alle Terme in HOL basieren auch DFG-Terme auf dem λ-Kalkül. Abbildung 3.7 zeigt einen DFG-Term, welcher dem Datenflussgraphen in Abbildung 3.6 entspricht.

```
λ(a,b,c,d). let r = INV a in
            let g = b AND r in
            let h = INV c in
            let i = g AND h in
            let s = INV g in
            let t = d AND g in
            let u = d OR i in
            let v = d in
            (r,s,t,u,v)
```

Abbildung 3.7: Ein DFG-Term

Der funktionale Zusammenhang zwischen den Ein- und Ausgangssignalen einer kombinatorischen Schaltung wird in DFG-Termen als λ-Abstraktion modelliert. Der Parameter der λ-Abstraktion, in diesem Fall die *Variablenstruktur* (a,b,c,d), entspricht den Eingangssignalen der Schaltung. Daher werden die Variablen a, b, c und d auch als *Eingangs-* bzw. *Eingabevariablen* des DFG-Terms bezeichnet.

Eine *Variablenstruktur* ist eine Struktur beliebig verschachtelter Variablen. Man beachte, dass es in HOL keine n-Tupel gibt sondern nur Paare; n-Tupel werden durch verschachtelte Paare nachgebildet. Dabei wird das Komma zur Trennung eines Paars als rechtsassoziativ angesehen, so dass bei rechtsgeklammerten Paaren die Klammern im Allgemeinen weggelassen werden. Die Termstruktur $(t_1, t_2, \ldots, t_{n-1}, t_n)$ steht also zum Beispiel für $(t_1, (t_2, \ldots (t_{n-1}, t_n) \ldots))$. In dem konkreten Beispiel ist (a,b,c,d) eine Abkürzung für (a,(b,(c,d))).

Der DFG-Term in Abbildung 3.7 endet mit der Variablenstruktur (r,s,t,u,v). Die Ausgangsstruktur eines DFG-Terms legt die Ausgangssignale der Schaltung fest. Entsprechend werden die Variablen r, s, t, u und v auch als *Ausgangs-* bzw. *Ausgabevariablen* des DFG-Terms bezeichnet. Im Rumpf des DFG-Terms können mit Hilfe des **let**-Konstrukts interne Signale explizit spezifiziert und mit einem Namen versehen werden. In dem vorliegenden Beispiel wurde die Ausgabe jeder Operation explizit gekennzeichnet. Für jeden Knoten in dem Datenflussgraphen existiert genau ein **let**-Term. Zusätzlich wurde das Durchschleifen des Signals d durch einen weiteren **let**-Term deutlich gemacht.

Die Reihenfolge der **let**-Terme in einem DFG-Term hängt allein vom Datenfluss zwischen dessen Komponenten ab. Die Verwendung freier Variablen ist in DFG-Termen nicht zulässig. Sie entsprächen Eingangssignalen ohne Signalquellen und somit ohne definierten Wert.

Der Datenfluss impliziert eine partielle Ordnung der **let**-Terme. Da die Ordnung der **let**-Terme ausschließlich durch Datenflussabhängigkeiten eingeschränkt ist, können beispielsweise in dem DFG-Term in Abbildung 3.7 die **let**-Terme, welche die internen Signale g und h definieren, vertauscht werden. Ihre beiden Komponenten sind voneinander unabhängig und würden

in Hardware parallel ausgeführt werden. Auch der letzte **let**-Term, der das Durchschleifen des Signals d spezifiziert, kann im Rumpf des DFG-Terms beliebig platziert werden. Dagegen ist beispielsweise das Vertauschen der **let**-Terme mit den Parametern h und i nicht zulässig, da das Signal h zur Berechnung des Signals i benötigt wird.

Alle Variablen in dem DFG-Term in Abbildung 3.7 haben booleschen Typ. Der Typ des DFG-Terms ist somit $bool \times bool \times bool \times bool \rightarrow bool \times bool \times bool \times bool \times bool$.

Eine genaue Definition der Syntax von DFG-Termen ist in Abbildung 3.8 gegeben.

KonstVar	= Konstante | Variable
VarStrukt	= Variable | "**(**" VarStrukt "**,**" VarStrukt "**)**"
AusdrStrukt	= Ausdruck | "**(**" AusdrStrukt "**,**" AusdrStrukt "**)**"
Ausdruck	= KonstVar |
	Präfix-Operator AusdrStrukt |
	AusdrStrukt Infix-Operator AusdrStrukt |
	DFG-Term AusdrStrukt
Ausgabe	= AusdrStrukt
Rumpf	= { "**let**" VarStrukt "**=**" AusdrStrukt "**in**" } Ausgabe
DFG-Term	= "λ" VarStrukt "**.**" Rumpf

Abbildung 3.8: Syntax der DFG-Terme

Der Rumpf eines DFG-Terms besteht aus einer Reihe von **let**-Termen und schließt mit einer Ausgabe ab. Die Ausgabe eines DFG-Terms ist eine beliebige Struktur von Ausdrücken. Ausdrücke können Konstanten, Variablen, Operationen oder auch Applikationen ganzer DFG-Terme sein. Die Möglichkeit der Verschachtelung von DFG-Termen erlaubt eine hierarchische Schaltungsbeschreibung.

Operatoren kommen in Präfix- oder Infix-Notation vor und müssen vor ihrer Verwendung in HOL definiert werden. Tabelle 3.1 listet die Operatoren und Konstanten auf, welche zur Beschreibung von Schaltungen auf der Gatterebene definiert wurden.

Bezeichner	Typ	Funktion
AND	$bool \times bool \rightarrow bool$	boolesche Konjunktion
OR	$bool \times bool \rightarrow bool$	boolesche Disjunktion
INV	$bool \rightarrow bool$	boolesche Negation
T	$bool$	boolesche Konstante *wahr*
F	$bool$	boolesche Konstante *falsch*

Tabelle 3.1: Operatoren und Konstanten in GROPIUS auf der Gatterebene

In den **let**-Termen im Rumpf eines DFG-Terms werden Variablen die Werte beliebiger Ausdrücke zugewiesen. Die Variablen können als Operanden in nachfolgenden **let**-Termen oder in der Ausgabe des DFG-Terms weiterverwendet werden. Wie bereits angemerkt entsprechen diese Variablen internen Signalen der Schaltung, denen explizite Bezeichner zugewiesen wurden.

Flache DFG-Terme

Eine Unterklasse der DFG-Terme bilden sogenannte *flache DFG-Terme* (FDFG-Terme) [Eise99]. Der in Abbildung 3.7 dargestellte DFG-Term gehört beispielsweise dieser Unterklasse an. Flache DFG-Terme haben folgende Einschränkungen gegenüber allgemeinen DFG-Termen:

- Die Ausgabe besteht nur aus einer Struktur von Konstanten und Variablen und enthält keine Operationen.
- In jedem **`let`**-Ausdruck darf maximal ein Operator enthalten sein.
- Der Typ jeder Variable muss atomar oder eine Typvariable sein.

Somit ergibt sich für FDFG-Terme eine Syntax gemäß Abbildung 3.9. In den folgenden Syntaxdefinitionen werden teilweise bereits zuvor definierte Nichtterminale der Übersichtlichkeit halber nochmals aufgeführt. Unveränderte Definitionen von Nichtterminalen sind durch graue Schrift gekennzeichnet.

```
KonstVar       = Konstante | Variable
VarStrukt      = Variable | "(" VarStrukt { "," VarStrukt } ")"
KonstVarStrukt = KonstVar |
                 "(" KonstVarStrukt { "," KonstVarStrukt } ")"
Ausdruck       = KonstVar |
                 Präfix-Operator KonstVarStrukt |
                 KonstVarStrukt Infix-Operator KonstVarStrukt
Ausgabe        = KonstVarStrukt
Rumpf          = { "let" VarStrukt "=" Ausdruck "in" } Ausgabe
FDFG-Term      = "λ" VarStrukt "." Rumpf
```

Abbildung 3.9: Syntax flacher DFG-Terme (FDFG-Terme)

In FDFG-Termen werden alle internen Signale explizit innerhalb von **`let`**-Ausdrücken benannt und können so im Rest des Terms mehrfach verwendet werden. Dies ist beispielsweise im Zusammenhang mit der Optimierung *Zusammenfassen gemeinsamer Teilausdrücke* wichtig: Ein Teilausdruck, der mehrmals in einem DFG-Term vorkommt, wird nach dieser Optimierung nur noch einmalig in einem **`let`**-Term berechnet und über den explizit zugewiesenen Namen im restlichen DFG-Term beliebig oft referenziert (siehe Abschnitt 4.2.4).

Ein weiterer Vorteil von FDFG-Termen gegenüber allgemeinen DFG-Termen ist die bessere Lesbarkeit der Schaltungsbeschreibung. Abbildung 3.10 stellt als Gegensatz die *Normalform* des DFG-Terms in Abbildung 3.7 dar. Die Normalform eines Terms erhält man, indem man alle β-Redizes und η-Redizes durch Anwendung ihrer entsprechenden Konversionen reduziert. Die Überführung in die Normalform ist *konfluent*. Dies bedeutet, dass das Ergebnis (abgesehen von den Namen der Variablen) von der Reihenfolge der Reduktionsanwendungen unabhängig ist.

```
λ(a,b,c,d). (INV a,
             INV(b AND (INV a)),
             d AND (b AND (INV a)),
             d OR ((b AND (INV a)) AND (INV c)),
             d)
```

Abbildung 3.10: Normalform des Terms in Abbildung 3.7

Wie dem Beispiel in Abbildung 3.10 zu entnehmen ist, wurde der DFG-Term durch die Überführung in die Normalform unübersichtlicher. Die Struktur des Datenflussgraphen kann nur noch schwer nachvollzogen werden. Da aber die Normalform für alle DFG-Terme, welche denselben Datenflussgraphen repräsentieren, eindeutig ist, kann sie zum Äquivalenzbeweis zweier DFG-Terme genutzt werden. Diese Vorgehensweise ist allerdings nur bei sehr kleinen DFG-Termen möglich, da deren Größe im Zuge der Umformung in die Normalform mit der kombinatorischen Tiefe der Schaltung exponentiell anwachsen kann. Ursache für ein entsprechendes Anwachsen ist die β-Reduktion von Signalen, die mehrfach als Operand im Rest des Terms zum Einsatz kommen.

Der größte Vorteil einer Darstellung der Schaltung mittels flacher DFG-Terme ist eine Vereinfachung der Automatisierung verschiedener Beweisverfahren. FDFG-Terme erlauben eine einfache Traversierung über alle Operationen des jeweils entsprechenden Datenflussgraphen.

Einen Mittelweg zwischen gänzlich ausgeflachten DFG-Termen und DFG-Termen in Normalform stellen sogenannte *kompakte FDFGs* (KFDFGs) dar [KaSa03].

Kompakte FDFG-Terme

KFDFG-Terme werden stets ausgehend von FDFG-Termen erzeugt. Soll ein DFG-Term in einen KFDFG-Term überführt werden, so muss dieser zunächst in einen FDFG-Term transformiert werden. Kompakte FDFG-Terme erhält man aus allgemeinen FDFG-Termen durch

- die β-Reduktion aller **let**-Terme, welche auf der rechten Seite der Zuweisung keine Operation, sondern nur eine Variable oder Konstante stehen haben.
- die β-Reduktion aller übrigen **let**-Terme, deren Parameter höchstens einmal im Rest des Terms verwendet wird.

Durch die β-Reduktion aller **let**-Terme, welche interne Signale für Variablen oder Konstanten einführen, wird der Term verkleinert und dadurch auch die Übersichtlichkeit des Terms erhöht. Durch die β-Reduktion aller übrigen **let**-Terme, deren Parameter höchstens einmal im Rest des Terms Verwendung finden, wird die Größe des Terms weiter reduziert. Da ausschließlich **let**-Terme reduziert werden, deren Parameter höchstens einmal im Rest des Terms verwendet werden, wird ein Wachsen des Terms, wie es bei der Überführung in die normierte Darstellung auftreten kann, von vornherein unterbunden. Neben der höheren Übersichtlichkeit von KFDFG-Termen ist vor Allem ihre Kompaktheit gegenüber allgemeinen DFG-Termen von Vorteil. Die

Verarbeitungsgeschwindigkeit eines Terms im Theorembeweisers HOL hängt zu einem wesentlichen Teil von dessen Größe ab. Eine Verkleinerung der Terme führt somit zu einer Beschleunigung des formalen Synthesesystems.

Infolge der oben genannten Transformationen, welche einen FDFG-Term als Ausgangspunkt voraussetzen, ergibt sich für einen KFDFG-Term eine Syntax gemäß Abbildung 3.11.

```
KonstVar              = Konstante | Variable
VarStrukt             = Variable | "(" VarStrukt { "," VarStrukt } ")"
KonstVarAusdrStrukt = KonstVar |
                        Ausdruck |
                        "(" KonstVarAusdrStrukt { "," KonstVarAusdrStrukt } ")"
Ausdruck              = Präfix-Operator KonstVarAusdrStrukt |
                        KonstVarAusdrStrukt Infix-Operator KonstVarAusdrStrukt
Ausgabe               = KonstVarAusdrStrukt
Rumpf                 = { "let" VarStrukt "=" Ausdruck "in" } Ausgabe
KFDFG-Term            = "λ" VarStrukt "." Rumpf
```

Abbildung 3.11: Syntax kompakter FDFG-Terme (KFDFG-Terme)

Die kompakte Fassung des FDFG-Terms in Abbildung 3.7 zeigt Abbildung 3.12. Die **let**-Terme mit den Parametern r und g wurden nicht reduziert, da diese beiden Parameter mehrfach im Rest des Terms verwendet werden.

```
λ(a,b,c,d). let r = INV a in
            let g = b AND r in
            (r, INV g, d AND g, d OR (g AND (INV c)), d)
```

Abbildung 3.12: Der kompakte FDFG-Term des Terms in Abbildung 3.7

Definition hierarchischer Strukturen

HOL unterstützt den hierarchischen Aufbau von Funktionseinheiten. Auf Basis der Grundelemente auf der Gatterebene - **AND**, **OR** und **INV** - lassen sich neue Funktionseinheiten zur weiteren Verwendung in Schaltungsbeschreibungen definieren. Ein **NAND**-Baustein zur Berechnung einer negierten Konjunktion lässt sich beispielsweise durch Verknüpfung eines **AND**- und eines **INV**-Bausteins folgendermaßen definieren:

$$\mathbf{NAND} := \lambda(a, b).\mathbf{INV}(\mathbf{AND}(a, b))$$

In dieser Arbeit wird für die Definition neuer Konstanten in HOL das Symbol **:=** verwendet. Es hat die Bedeutung „wird in HOL definiert durch“. Es handelt sich hierbei um eine *konservative Erweiterung* der Logik. Somit kann die Konsistenz der Logik von HOL nicht durch die Definition zusätzlicher Konstanten verletzt werden [Melh93].

Ebenfalls zulässig ist die Definition neuer Konstanten in HOL mit freien Eingangsvariablen. Freie Variablen werden in diesem Zusammenhang implizit als allquantifiziert betrachtet:

$$\texttt{NAND}(a, b) := \texttt{INV}(\texttt{AND}(a, b))$$

Durch die Definition beliebiger DFG-Terme als Konstanten mit einem bestimmten Namen, welche wiederum zur Definition weiterer Komponenten herangezogen werden können, ist ein hierarchischer Systementwurf möglich. In GROPIUS sind schon eine Reihe von zusammengesetzten Bausteinen definiert. Einige wichtige Beispiele aus [Eise99] zeigt Abbildung 3.13.

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

Abbildung 3.13: Einige hierarchisch definierte Elemente

AND3 bezeichnet eine Konjunktion mit drei Eingängen und **EQ1** die Äquivalenz zweier boolescher Signale. **EQ3** verwendet den neu definierten **EQ1**-Baustein zur Berechnung der Äquivalenz zweier Signalbündel mit jeweils drei booleschen Signalen. **XOR** ist der Exklusiv-Oder-Baustein und **MUX1** ein Multiplexer. **HADD** steht für einen Halbaddierer und wird zur Definition des Volladdierers **FADD** verwendet.

Abstrakte kombinatorische Schaltungsbeschreibungen

Komponenten, bei deren Definition Signale variablen Typs verwendet werden, werden als *polymorphe Komponenten* bezeichnet. Sobald eine polymorphe Komponente in einer Schaltungsbeschreibung eingesetzt wird, werden deren variable Typen in Abhängigkeit ihres Kontexts mit konkreten Typen instantiiert. Tabelle 3.2 stellt als Beispiel die Definitionen der polymorphen Verdrahtungskomponenten **FST** und **SND** dar.

Bezeichner	Definition	Typ	Funktion
FST	**FST** (a,b) := a	$\alpha \times \beta \rightarrow \alpha$	Selektion der linken Hälfte eines Paars
SND	**SND** (a,b) := b	$\alpha \times \beta \rightarrow \beta$	Selektion der rechten Hälfte eines Paars

Tabelle 3.2: Polymorphe Verdrahtungskomponenten in GROPIUS auf der Gatterebene

Die Funktion **FST** (*first*) selektiert die linke Hälfte eines Paars und die Funktion **SND** (*second*) die rechte Hälfte. Sie entsprechen den Funktionen λ(f,s).f bzw. λ(f,s).s. Werden diese Komponenten in einer Schaltungsbeschreibung eingesetzt, so werden ihre Typvariablen α und β automatisch mit zu den jeweils angeschlossenen Signalleitungen passenden Typen belegt.

Gegeben sei beispielsweise ein Signalbündel ((a,b),c), dessen einzelne Signale a, b und c alle booleschen Typs seien. Das Signalbündel hat somit den Typ $(bool \times bool) \times bool$. Wird die Funktion **FST** auf dieses Signalbündel angewandt, so wird der Typ dieser **FST**-Instanz zu $((bool \times bool) \times bool) \rightarrow (bool \times bool)$ konkretisiert. Eine Anwendung der Funktion **SND** auf dasselbe Signalbündel ergäbe für diese Instanz von **SND** den Typ $((bool \times bool) \times bool) \rightarrow bool$.

Weitere wichtige polymorphe Komponenten sind der Multiplexer **MUX** und der Operator **EQ**, welcher die Äquivalenz zweier Signale bzw. Signalbündel ausdrückt (siehe Tabelle 3.3). Der Ausdruck **MUX**(a,b,c) wählt in Abhängigkeit des booleschen Signals a das Signal(-bündel) b oder c aus. Hat a den Wert **T**, so wird das Signal(-bündel) b gewählt, anderenfalls das Signal(-bündel) c. Der Ausdruck **EQ**(a,b) liefert den Wert **T**, wenn die beiden eingehenden Signale bzw. Signalbündel a und b bzw. die gleichen Werte enthalten.

Bezeichner	Typ	Funktion
MUX	$bool \times \alpha \times \alpha \rightarrow \alpha$	Multiplexer
EQ	$\alpha \times \alpha \rightarrow bool$	Äquivalenzoperator

Tabelle 3.3: Polymorphe Operatoren in GROPIUS auf der Gatterebene

Sind alle variablen Typen einer polymorphen Komponente mit konkreten Typen belegt, so kann eine Abbildung der Komponente auf eine Schaltungsstruktur aus elementaren Gatterbausteinen erfolgen. Ein Ausdruck **MUX**(a,(s1,s2),(t1,t2)), dessen Signale alle booleschen Typ haben, könnte beispielsweise durch die gepaarte Anwendung der Komponente **MUX1** aus Abbildung 3.13 realisiert werden:

$$\mathbf{MUX}(a,(s1,s2),(t1,t2)) \quad := \quad (\mathbf{MUX1}(a,s1,t1),\mathbf{MUX1}(a,s2,t2))$$

Auf der Gatterebene sind noch einige weitere Typen (u. a. Aufzählungsdatentypen) und Operatoren, beispielsweise zur Bearbeitung von Feldern, definiert, mit deren Hilfe generische reguläre Schaltungsstrukturen definiert werden können. Damit ist beispielsweise die Modellierung generischer n-Bit-Addierer möglich, aus welchen durch Instantiierung der Variablen n konkrete Addierer der gewünschten Bitbreite abgeleitet werden können [SaKa03, SaKa03b]. Die Modellierung generischer Schaltungen wird ausführlich in [Eise99] beschrieben und soll hier nicht weiter betrachtet werden.

3.3.2 Sequentielle Schaltungsbeschreibungen

Ergänzt man eine kombinatorische Schaltung (ein *Schaltnetz*) um eine Rückkopplung, so entsteht eine sequentielle Schaltung (ein *Schaltwerk*). Schaltwerke sind konkrete Implementierungen abstrakter Automaten bestehend aus einem Speicher, der den Zustand des Automaten enthält, sowie

einer kombinatorischen Schaltung, welche die Ausgabe und den Folgezustand des entsprechenden Automaten ermittelt (siehe Abbildung 3.14). Den aktuellen Zustand hat der Automat auf Grund eines Anfangszustandes und der gesamten Vorgeschichte aller bisher eingetroffenen Eingaben erreicht. Man beachte, dass in synchronen Schaltungen eine Rückkopplung der Signale ausschließlich über Speicherbausteine stattfindet. Infolgedessen sind verzögerungsfreie Zyklen ausgeschlossen.

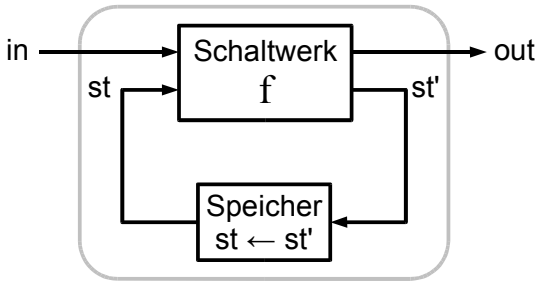


Abbildung 3.14: Ein Schaltwerk

Die oft im Zusammenhang mit Automaten gebräuchliche Aufteilung der Ausgabe- und Übergangsfunktion in zwei strikt getrennte Funktionen wird in GROPIUS nicht vorgenommen. In der Hardware wird dieser kombinatorische Anteil der Automaten letztendlich durch eine Struktur elementarer Gatterbausteine realisiert. Durch ein Zusammenfassen der beiden Funktionen erhöht sich das Potenzial möglicher Optimierungen. Es können beispielsweise gemeinsame Teilausdrücke der Aus- und Übergangsfunktion zusammengefasst werden, so dass sie nur einmalig realisiert werden müssen. Als Folge werden die Fläche der Schaltung und somit auch deren Kosten verringert.

Eine sonst gängige gesonderte Angabe des Ein- und Ausgabealphabets sowie des Zustandsraums ist auf Grund der strengen Typisierung in GROPIUS nicht erforderlich. Durch den Typ des Automaten werden die Wertebereiche der Ein- und Ausgänge sowie der Zustände eindeutig festgelegt. Im Folgenden bezeichnen die Typvariablen ι, ω und σ die Typen der Eingangssignale, der Ausgangssignale und der Zustände.

Ein Automat wird in GROPIUS durch ein Paar (f,q) definiert [Eise97b]. Die Funktion f repräsentiert den kombinatorischen Anteil der Schaltung, also die zusammengefasste Ausgabe- und Übergangsfunktion, und die Variable q steht für den Initialzustand des Automaten (siehe Tabelle 3.4).

Bezeichner	Typ	Funktion
f	$(\iota \times \sigma) \rightarrow (\omega \times \sigma)$	Ausgabe- und Übergangsfunktion
q	σ	Initialzustand

Tabelle 3.4: Bestandteile eines Automaten auf der RT-Ebene

Die Funktion f erwartet als Eingabe ein Paar des Typs $(\iota \times \sigma)$ bestehend aus der aktuellen Eingabe des Automaten sowie dem aktuellen Zustand des Automaten. Die Ausgabe der Funktion f

ist ein Paar, welches den Ausgabewert des Automaten sowie dessen Folgezustand enthält. Der Ausgabetyp von f ist somit $(\omega \times \sigma)$.

Ein Automat wird in GROPIUS durch eine Funktion dargestellt, welche eine Sequenz von Eingabewerten auf eine Sequenz von Ausgabewerten abbildet. Die Zeit wird dabei durch natürliche Zahlen des Typs num modelliert. Eine Zeiteinheit entspricht der Dauer eines Register-Transfers. Ein Register-Transfer umfasst das Lesen des Speichers, die Verarbeitung der Eingabe und des aktuellen Zustands durch die Funktion f sowie die Ausgabe des Automaten und das Speichern des berechneten Folgezustands. Die Sequenz der Eingangssignale ist eine Funktion insq mit dem Typ $num \rightarrow \iota$. Der Ausdruck (insq t) liefert den Eingabewert des Automaten zum Zeitpunkt t. Entsprechend liefert die Funktion outsq:$num \rightarrow \omega$ die Ausgabe des Automaten in Abhängigkeit von der Zeit.

Die Semantik eines Automaten wird in GROPIUS mit Hilfe des Konstrukts **automaton** beschrieben. Die Definition der Funktion **automaton** erfolgt auf Basis der Hilfsfunktion **state**. Die Funktion **state** ist mittels primitiver Rekursion über die Zeit t definiert:

$$\begin{aligned} \textbf{state}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}\ 0 &:= \mathrm{q} \\ \textbf{state}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}\ (\textbf{SUC}\ t) &:= \textbf{SND}(\mathrm{f}(\mathrm{insq}\ t, \textbf{state}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}\ t)) \end{aligned}$$

Die Funktion **SUC** liefert den Nachfolger einer natürlichen Zahl. Der Ausdruck (**SUC** t) entspricht demnach (t+1). Die Funktion **state** liefert für einen Automaten (f,q) zu einer Eingabesequenz insq die zugehörige Sequenz stsq:$num \rightarrow \sigma$ der Zustände, die der Automat einnimmt:

$$\mathrm{stsq} := \textbf{state}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}$$

Der Ausdruck (**state** (f,q) insq t) ergibt beispielsweise den Zustand des Automaten (f,q), in dem er sich infolge der Eingabesequenz insq zum Zeitpunkt t befindet. Zum Zeitpunkt 0 ist der Zustand des Automaten q. Auf Basis der Funktion **state** ist die Funktion **automaton** folgendermaßen definiert:

$$\textbf{automaton}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}\ t := \textbf{FST}(\mathrm{f}(\mathrm{insq}\ t, \textbf{state}\ (\mathrm{f}, \mathrm{q})\ \mathrm{insq}\ t))$$

Der Ausdruck (**automaton** (f,q) insq) liefert die Ausgabesequenz des Automaten (f,q) zu einer gegebenen Eingabesequenz insq. Entsprechend gibt (**automaton** (f,q) insq t) den Ausgabewert des Automaten zum Zeitpunkt t an.

Ein Automat g, welcher auf einer Funktion f und einem Initialzustand q basiert, kann nun unter Verwendung des Konstrukts **automaton** folgendermaßen definiert werden:

$$\mathrm{g} := \textbf{automaton}(\mathrm{f}, \mathrm{q})$$

Durch die Darstellung eines Automaten als ein Paar, bestehend aus einem DFG-Term f und einem Initialzustand q, werden inkonsistente Schaltungsspezifikationen von vornherein ausgeschlossen. DFG-Terme lassen die Spezifikation von Kurzschlüssen nicht zu und verzögerungsfreie Zyklen sind ebenfalls durch die implizite Rückkopplung des Automatenzustands über Speicherelemente bereits syntaktisch ausgeschlossen.

3.3.3 Algorithmische Schaltungsbeschreibungen

Auf der algorithmischen Ebene besteht eine Schaltungsspezifikation aus einer algorithmischen Beschreibung, welche die Funktionalität der gewünschten Schaltung spezifiziert, sowie einem Schnittstellenverhaltensmuster, das die Schnittstelle der Schaltung zur Außenwelt definiert. Das Schnittstellenverhaltensmuster erweitert die algorithmische Schaltungsbeschreibung um einige Steuer- und Statussignale, wie zum Beispiel einem *reset*-, *start*- und *ready*-Signal, und legt fest, wie die Kommunikation der Schaltung mit ihrer Umgebung auf Basis dieser zusätzlichen Signale abläuft.

Durch das Signal *reset* kann die Ausführung einer Schaltung jederzeit abgebrochen und die Schaltung in ihren Anfangszustand versetzt werden. Das *reset*-Signal ist auf der algorithmischen Ebene von besonderer Bedeutung, da für GROPIUS-Spezifikationen auf dieser Ebene nicht mehr grundsätzlich entscheidbar ist, ob die spezifizierte Berechnung für alle möglichen Eingaben terminiert. Die Klasse der spezifizierbaren Algorithmen entspricht der Klasse der berechenbaren bzw. μ-rekursiven Funktionen. Es ist möglich, dass eine Funktion nur für eine bestimmte Eingabemenge ein Ergebnis liefert, während sie für andere Eingaben niemals terminiert. Eine solche Funktion wird als *partielle* Funktion bezeichnet. Andere Funktionen, wie beispielsweise DFG-Terme, terminieren immer. Sie liefern für jeden Eingabewert einen definierten Ausgabewert und werden als *totale* Funktionen bezeichnet.

Um formal über partielle Funktionen auf der algorithmischen Ebene argumentieren zu können, wurde der Datentyp $\alpha\, partial$ für die Ausgabe algorithmischer Schaltungsbeschreibungen eingeführt [BlEi98]. Der Typ ist folgendermaßen definiert:

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

Der Typ $\alpha\, partial$ verfügt über die Konstruktoren $\mathsf{Defined}{:}\alpha \rightarrow \alpha\, partial$ und $\mathsf{Undefined}{:}\alpha$. Ein Objekt des Typs $\alpha\, partial$ kann die Werte $\mathsf{Defined}$ x, wobei x den Typ α hat, und $\mathsf{Undefined}$ annehmen. Aus formaler Sicht bedeutet ein Wert $\mathsf{Defined}$ x als Ausgabe einer algorithmischen Beschreibung, dass die Berechnung zu Ende geführt wurde und das resultierende Ergebnis den Wert x hat. Der Wert $\mathsf{Undefined}$ besagt, dass die Berechnung nicht terminiert. Im Allgemeinen ist für eine Spezifikation auf der algorithmischen Ebene nicht vorhersagbar, ob deren Berechnung terminieren wird oder nicht. Das Halteproblem ist für berechenbare Funktionen unentscheidbar.

Mit der Einführung des neuen Typs $\alpha\, partial$ erzeugt der Theorembeweiser HOL automatisch die Funktion **`PRIMREC_partial`**, welche später zur Definition einiger der GROPIUS-Konstrukte auf der algorithmischen Ebene verwendet wird:

$$\begin{aligned} \vdash\; & (\forall f\, u\, y.\, \texttt{PRIMREC\_partial}\; (\mathsf{Defined}\, y)\; f\; u = f\; y) \wedge \\ & (\forall f\, u.\, \texttt{PRIMREC\_partial}\; \mathsf{Undefined}\; f\; u = u) \end{aligned} \tag{3.3}$$

Für die folgenden Definitionen der GROPIUS-Konstrukte ist die Variable u in Gleichung 3.3 grundsätzlich mit dem Wert $\mathsf{Undefined}$ belegt. Die Semantik des resultierenden Ausdrucks **`PRIMREC_partial`** x f $\mathsf{Undefined}$ lässt sich in einfachen Worten folgendermaßen beschreiben: Ist die Eingabe x definiert, entspricht sie also einem Wert $\mathsf{Defined}$ y, so wird der Wert y

als Operand an die Funktion f weitergereicht. In diesem Fall liefert der **PRIMREC_partial**-Ausdruck den Wert $f\ y$. Ist die Eingabe x allerdings undefiniert, also Undefined, so ist die Ausgabe des **PRIMREC_partial**-Ausdrucks auch stets Undefined.

Für Ausdrücke der Form **PRIMREC_partial** $x\ f$ Undefined wird auf Grund der besseren Lesbarkeit die folgende Schreibweise bevorzugt verwendet:

$$\begin{array}{l} \textbf{case}\ x\ \textbf{of} \\ \quad \mathsf{Defined}\ y\ \ \rightarrow f\ y \\ \quad \|\ \mathsf{Undefined} \rightarrow \mathsf{Undefined} \end{array} \tag{3.4}$$

Algorithmische Verhaltensspezifikation in GROPIUS

Auf der algorithmischen Ebene erfolgt die Schaltungsspezifikation in GROPIUS mittels sogenannter P-Terme (**P**rogramm-**Terme**). Während ein DFG-Term den generischen Typ $\alpha \rightarrow \beta$ hat, haben P-Terme einen Typ $\alpha \rightarrow \beta\, partial$. Abbildung 3.15 zeigt die Syntaxdefinition von P-Termen. Bei den in dieser Arbeit eingesetzten Konstrukten handelt es sich um eine Fortentwicklung der in [Blum00] vorgestellten Konstrukte. Die ursprünglich erforderlichen acht Grundkonstrukte konnten auf fünf reduziert werden [SaBK01]. Durch die Reduktion der Grundkonstrukte sind zur Durchführung der formalen Synthese weniger Theoreme und Konversionen erforderlich, wodurch sich ebenfalls die Automatisierung der formalen Synthese vereinfacht (siehe Abschnitt 4.2.1).

P-Term = "**DEF**" DFG-Term |
P-Term "**SER**" P-Term |
P-Term "**PAR**" P-Term |
"**IF**" DFG-Kond P-Term P-Term |
"**WHILE**" DFG-Kond P-Rumpf

Abbildung 3.15: Syntax der P-Terme

DFG-Kond steht für DFG-Terme mit boolescher Ausgabe, also DFG-Terme des Typs $\alpha \rightarrow bool$. Diese Art von Termen wird zur Spezifikation von Bedingungen in Schleifen und Verzweigungen verwendet. Allgemein hat ein P-Term einen Typ $\alpha \rightarrow \beta\, partial$. Das Nichtterminal P-Rumpf in Abbildung 3.15 steht für eine Untermenge der P-Terme, deren Typ $\alpha \rightarrow \alpha\, partial$ ist. Sie kommen als Rumpf von **WHILE**-Schleifen zum Einsatz. In Tabelle 3.5 sind die Typen der neu eingeführten Konstrukte aufgelistet.

DFG-Terme sind ein wesentlicher Bestandteil von P-Termen. Sie werden eingebunden mit dem Konstrukt **DEF**, welches den generellen Typ $\alpha \rightarrow \beta$ eines DFG-Terms in den Typ $\alpha \rightarrow \beta\, partial$ umwandelt und auf diese Weise den DFG-Term in einen P-Term überführt:

$$\textbf{DEF}\ f\ x := \mathsf{Defined}(f\ x) \tag{3.5}$$

Die Typumwandlung ändert nichts an der Tatsache, dass der DFG-Term für alle Eingaben terminiert. Die Ausgabe einer Funktion der Form **DEF** f angewandt auf einen Operanden x er-

Bezeichner	Typ
DEF	$(\alpha \to \beta) \to (\alpha \to \beta\, partial)$
SER	$(\alpha \to \beta\, partial) \to (\beta \to \gamma\, partial) \to (\alpha \to \gamma\, partial)$
PAR	$(\alpha \to \beta\, partial) \to (\alpha \to \gamma\, partial) \to (\alpha \to (\beta \times \gamma)\, partial)$
IF	$(\alpha \to bool) \to (\alpha \to \beta\, partial) \to (\alpha \to \beta\, partial) \to (\alpha \to \beta\, partial)$
WHILE	$(\alpha \to bool) \to (\alpha \to \alpha\, partial) \to (\alpha \to \alpha\, partial)$

Tabelle 3.5: Typen der GROPIUS-Konstrukte auf der algorithmischen Ebene

gibt grundsätzlich einen Wert Defined (f x). Der Wert Undefined kann durch **DEF**-Terme nicht erreicht werden.

Das Konstrukt **SER** dient zur sequentiellen Ausführung zweier P-Terme. **SER** wird dabei in Infix-Notation verwendet. In einem Ausdruck (A **SER** B) x wird zuerst der P-Term A auf den Operanden x angewandt. Terminiert die Berechnung von A x, so ist das Ergebnis der Berechnung ein Wert Defined y. Der Wert y dient anschließend als Eingabe für den P-Term B. Terminiert einer der beiden P-Terme nicht, so ist das Gesamtergebnis Undefined. Das Konstrukt **SER** ist folgendermaßen definiert:

$$\begin{aligned}(\text{A } \textbf{SER}\ \text{B})\ \text{x} := \textbf{case}\ \text{A x}\ \textbf{of}\ & \\ & \ \ \text{Defined y} \ \to \text{B y} \\ & \|\ \ \text{Undefined} \to \text{Undefined}\end{aligned} \tag{3.6}$$

Die P-Terme, die durch das Konstrukt **SER** miteinander kombiniert sind, werden in Anlehnung an imperative Programmiersprachen auch als *Anweisungen* bezeichnet. Tatsächlich sind P-Terme natürlich Funktionen. Weiter wird in Termen der Form A **SER** B der Term A als *Vorgänger* von B bzw. der Term B als *Nachfolger* von A bezeichnet.

Mit Hilfe des Konstrukts **PAR** kann die Unabhängigkeit zweier P-Terme explizit ausgedrückt werden. Die Unabhängigkeit besagt in diesem Zusammenhang, dass keiner der beiden P-Terme Ergebnisse des anderen P-Terms zu seiner eigenen Berechnung benötigt. Die Funktion **PAR** wird wie das Konstrukt **SER** in Infix-Notation verwendet.

In einem Ausdruck (A **PAR** B) x wird der Operand x:α als Operand an die beiden P-Terme A:$\alpha \to \beta\, partial$ und B:$\alpha \to \gamma\, partial$ weitergereicht. Terminiert die Berechnung von (A x) mit einem Wert Defined (u:$\beta\, partial$) und die Berechnung von (B x) mit einem Wert Defined (v:$\gamma\, partial$), so werden ihre Ergebnisse u und v zu einem Paar zusammengefasst und Defined ((u,v):$(\beta \times \gamma)\, partial$) als Gesamtergebnis des **PAR**-Ausdrucks ausgegeben. Ergibt allerdings einer der P-Terme den Wert Undefined, so ist auch das Ergebnis des gesamten **PAR**-Ausdrucks Undefined. Im Gegensatz zu der Berechnung eines **SER**-Ausdrucks können beim **PAR**-Konstrukt die beiden P-Terme in beliebiger Reihenfolge ausgeführt werden.

Das Konstrukt **PAR** ist mit Hilfe zweier verschachtelter **case**-Funktionen definiert:

$$\begin{aligned}(\text{A } \textbf{PAR}\ \text{B})\ \text{x} := \textbf{case}\ \text{A x}\ \textbf{of}\ & \\ \quad \mathsf{Defined}\ \text{u} \rightarrow \textbf{case}\ \text{B x}\ \textbf{of}\ & \\ \quad\quad \mathsf{Defined}\ \text{v} \rightarrow \mathsf{Defined}\ (\text{u,v}) & \\ \quad\quad \| \ \mathsf{Undefined} \rightarrow \mathsf{Undefined} & \\ \quad \| \ \mathsf{Undefined} \rightarrow \mathsf{Undefined} &\end{aligned} \tag{3.7}$$

Zur Definition des **IF**-Konstrukts wird die HOL-Funktion **COND** benutzt. Sie bewirkt die bedingte Ausführung eines ihrer beiden Zweige und hat den generischen Typ $bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. Im Allgemeinen werden die Notationen (if c then A else B) oder (c $\Rightarrow$ A | B) für den Ausdruck **COND** c A B verwendet. Ist die Bedingung c *wahr*, so ist das Ergebnis des Ausdrucks A, ansonsten B.

Das **IF**-Konstrukt ist auf Basis der Funktion **COND** folgendermaßen definiert:

$$(\textbf{IF}\ \text{c A B})\ \text{x} := \big(\ \textbf{COND}\ (\text{c x})\ (\text{A x})\ (\text{B x})\ \big) \tag{3.8}$$

Ist die Bedingung c für das Argument x *wahr*, so ist das Ergebnis der Ausdruck A x. Ergibt die Berechnung von c x den Wert **F**, wird also die Bedingung nicht erfüllt, ist das Ergebnis des **IF**-Terms der Ausdruck B x. Um die Lesbarkeit des **IF**-Konstrukts zu verbessern, werden im Folgenden den beiden Zweigen des **IF**-Konstrukts die Schlüsselwörter **THEN** und **ELSE** vorangestellt. Es gilt:

$$(\textbf{IF}\ \text{c}\ \textbf{THEN}\ \text{A}\ \textbf{ELSE}\ \text{B}) := (\textbf{IF}\ \text{c A B}) \tag{3.9}$$

Das folgende Theorem beschreibt die Semantik des **WHILE**-Konstrukts. Die konstruktive Definition der **WHILE**-Funktion wird ausführlich in [Blum00] vorgestellt.

$$\begin{aligned}\vdash \forall\, \text{c A x.}\ & \\ \quad (\exists \text{y.}\ (\textbf{WHILE}\ \text{c A x} = \mathsf{Defined}\ \text{y}) \wedge & \\ \quad\quad \exists\, \text{t s.} & \\ \quad\quad\quad (\text{s } 0 = \text{x}) \wedge & \\ \quad\quad\quad (\forall \text{n.}\ \text{n}<\text{t} \Rightarrow (\text{c (s n)} \wedge & \\ \quad\quad\quad\quad\quad \text{A (s n)} = \mathsf{Defined}\ (\text{s (n+1)}))) \wedge & \\ \quad\quad\quad \neg \text{c (s t)} \wedge & \\ \quad\quad\quad (\text{s t} = \text{y})) & \\ \quad \vee \quad (\textbf{WHILE}\ \text{c A x} = \mathsf{Undefined}) &\end{aligned} \tag{3.10}$$

In diesem Theorem bezeichnet die Variable c die Schleifenbedingung, also einen DFG-Term mit boolescher Ausgabe. A steht für den P-Term, welcher den Rumpf der Schleife bildet, und die Variable x für den Eingabewert der Schleife.

Der linke Teil der Disjunktion beschreibt den Fall, in welchem die Berechnung der Schleife terminiert. In diesem Fall gibt es einen Wert y, so dass das Ergebnis der Schleifenberechnung Defined y ist. Zusätzlich existieren ein Wert t, welcher die Anzahl der Iterationen repräsentiert, die bis zum Terminieren der Schleife durchgeführt werden, sowie eine Funktion s, welche

den Stand der Berechnung in Abhängigkeit einer bestimmten Iterationszahl liefert. Der Stand der Berechnung entspricht vor der Durchführung der ersten Iteration dem Eingabewert x der **WHILE**-Schleife (s 0 = x). Solange die Bedingung der Schleife für den aktuellen Stand der Berechnung *wahr* ist, also (c (s n)) gilt, iteriert die Schleife weiter. In jeder Iteration wird einmalig der Rumpf der Schleife ausgeführt. Der Rumpf wird auf den aktuellen Stand der Berechnung angewandt und ergibt so den neuen Stand der Berechnung. Terminiert die Schleife, so muss auch die Berechnung jeder einzelnen Iteration terminieren und infolgedessen einen definierten Wert ergeben (A (s n) = Defined (s (n+1))). Die Schleife terminiert nach der t-ten Iteration. Zu diesem Zeitpunkt ist die Schleifenbedingung nicht mehr *wahr* ($\neg$c (s t)). Das Ergebnis der gesamten Schleifenberechnung entspricht schließlich dem letzten Stand der Berechnung (s t = y).

Im Theorem 3.10 beschreibt der rechte Teil der Disjunktion den Fall, in welchem die Schleife nicht terminiert. Das kann zweierlei Ursachen haben: Entweder ergibt die Schleifenbedingung niemals den Wert *falsch*, es handelt sich also um eine Endlosschleife, oder in einer Iteration führt die Berechnung des Rumpfes zu einem undefinierten Wert, d. h. der Rumpf selbst terminiert nicht. Terminiert die **WHILE**-Schleife nicht, so ist das Ergebnis der Schleifenberechnung undefiniert (Undefined).

Durch das Einfügen des Schlüsselworts **DO** zwischen der Bedingung und dem Rumpf einer Schleife soll im Folgenden die Lesbarkeit des **WHILE**-Konstrukts verbessert werden. Es gilt:

$$(\textbf{WHILE}\ \text{c}\ \textbf{DO}\ \text{A}) := (\textbf{WHILE}\ \text{c A}) \tag{3.11}$$

Ein Beispiel für eine algorithmische Verhaltensbeschreibung in GROPIUS wird in Abbildung 3.16 gegeben. Bei dieser Spezifikation handelt es sich um einen effizienten Algorithmus zur Berechnung von u^n. Abbildung 3.17 stellt zum Vergleich das entsprechende Programm in der Programmiersprache C dar.

```
DEF (λ(u, n).
      let v = 1 in (u, v, n))
SER
(WHILE (λ(u, v, n).n> 0) DO
        ( IF (λ(u, v, n).ODD n)
             THEN (DEF (λ(u, v, n).
                          let v' = u∗v in
                          let n' = n−1 in (u, v', n')))
             ELSE (DEF (λ(u, v, n).
                          let u' = u∗u in
                          let n' = n DIV 2 in (u', v, n')))) )
SER
DEF (λ(u, v, n).v)
```

Abbildung 3.16: Algorithmische GROPIUS-Beschreibung zur Berechnung von u^n

```
int uhochn (int u, int v)
    { int v = 1;
      while (n>0)
            { if (n%2)
                  { v = u*v;
                    n = n-1;}
              else
                  { u = u*u;
                    n = n/2;} }
      return v;
    }
```

Abbildung 3.17: C-Programm zur Berechnung von u^n

Ableitung weiterer algorithmischer Konstrukte

Um den Komfort von Schaltungsspezifikationen in GROPIUS zu erhöhen, können auf Basis der fünf vorgestellten algorithmischen Grundkonstrukte weitere Hilfskonstrukte definiert werden. Es folgen einige Bespiele nützlicher Definitionen, die im weiteren Verlauf der Arbeit Verwendung finden.

$$\textbf{NOP} \;:=\; \textbf{DEF}(\lambda(x.x)) \tag{3.12}$$

$$\textbf{IFT c THEN A} \;:=\; \textbf{IF c THEN A ELSE NOP} \tag{3.13}$$

Das Konstrukt **NOP** (*no operation*) wird im Wesentlichen als Hilfskonstrukt für weitere Definitionen verwendet. Die zur Definition von **NOP** verwendete Funktion λ(x.x) entspricht der Identitätsfunktion, welche u. a. durch die Funktion **I** repräsentiert wird. Sie führt keine Operation durch und gibt ihren Eingabewert unverändert weiter. Auf Basis des **IF**-Konstrukts und **NOP** ist das Konstrukt **IFT** definiert, welches die bedingte Ausführung eines P-Terms A realisiert.

Zur Einführung einer lokalen Variablenstruktur mit einer Initialisierung n dient das Konstrukt **LOCVAR**. Es ist folgendermaßen definiert:

$$\textbf{LOCVAR n A} \;:=\; \textbf{DEF}(\lambda x.(x, n))\ \textbf{SER A SER DEF FST} \tag{3.14}$$

Der erste **DEF**-Ausdruck erweitert einen Eingabewert x:α um eine lokale Variable bzw. eine Variablenstruktur, welche mit dem Wert n:γ initialisiert wird. Auf den resultierenden Wert wird der P-Term A angewandt. A hat einen Typ $(\alpha \times \gamma) \rightarrow (\beta \times \gamma)\, partial$. Der rechte **DEF**-Ausdruck lässt die lokale Variable schließlich wieder fallen. Der Typ des gesamten Ausdrucks **LOCVAR** n A ist somit $\alpha \rightarrow \beta\, partial$.

Die Konstrukte **LEVA** und **RIVA** ermöglichen die selektive Anwendung eines P-Terms auf den linken bzw. rechten Teil einer gepaarten Eingabe. Es gilt:

$$\textbf{LEVA A} \;:=\; (\textbf{DEF FST SER A})\ \textbf{PAR}\ (\textbf{DEF SND}) \tag{3.15}$$

$$\textbf{RIVA A} \;:=\; (\textbf{DEF FST})\ \textbf{PAR}\ (\textbf{DEF SND SER A}) \tag{3.16}$$

Der Ausdruck **LEVA** A hat einen Typ $(\alpha \times \gamma) \rightarrow (\beta \times \gamma)\, partial$. Eine Eingabe (x,k):$(\alpha \times \gamma)$ wird entsprechend der Semantik des **PAR**-Konstrukts zunächst an beide Zweige von **PAR** weiter geleitet. Im linken Zweig wird mit **DEF FST** der rechte Teil des Paars (x,k) fallen gelassen und anschließend der P-Term A auf den verbleibenden Wert x angewandt. Im rechten Zweig dagegen wird der Wert x fallen gelassen und der verbleibende Wert k unverändert weitergeleitet. Das Ergebnis der Berechnung von A x und der unveränderte Wert k werden schließlich, falls die Berechnung von A x terminiert, entsprechend der **PAR**-Semantik zusammengefasst und ausgegeben. Terminiert die Berechnung von A x nicht, so ist die Ausgabe undefiniert. Bei dem Konstrukt **RIVA** wird analog der P-Term A ausschließlich auf dem rechten Teil des Eingabepaars ausgeführt. Der linke Teil bleibt unverändert.

Um dem Schaltungsentwerfer die Spezifikation von Schaltungen weiter zu erleichtern, ist auf Basis der Grundkonstrukte auch die Herleitung eventuell bereits bekannter Konstrukte aus anderen Programmiersprachen möglich. Im Folgenden werden als Beispiele die Konstrukte **REPEAT** und **FOR** eingeführt, mit deren Hilfe Schleifen mit der Semantik der entsprechenden Konstrukte der Sprache MODULA-2 [Wirt85] dienen können.

Das Konstrukt **REPEAT** erlaubt die Spezifikation einer *Schleife mit nachfolgender Prüfung*. Es findet bei diesem Schleifentyp eine erste Ausführung des Schleifenrumpfes ohne Prüfung der Abbruchbedingung statt. Daher werden entsprechende Schleifen auch als ,,annehmende Schleifen“ bezeichnet. Das Konstrukt ist wie folgt definiert:

$$\textbf{REPEAT}\ \text{A}\ \textbf{UNTIL}\ \text{c} \quad := \quad \text{A}\ \textbf{SER}\ (\textbf{WHILE}\ \text{c}\ \textbf{DO}\ \text{A}) \tag{3.17}$$

Das folgende Beispiel der Definition des Konstrukts **FOR** zeigt, dass auch komplexere und ausgefeiltere Ableitungen aus den Grundkonstrukten einfach möglich sind:

$$\begin{aligned}
&\textbf{FOR}\ \text{i m s A} := \textbf{LOCVAR}\ \text{(i,m,s)} \\
&\qquad (\textbf{IF}\ \ (\lambda\text{(x,(i,m,s))}.\ \text{s} > 0) \\
&\qquad\quad \textbf{THEN}\ (\textbf{WHILE}\ \ (\lambda\text{(x,(i,m,s))}.\ \text{s} \leq \text{m}) \\
&\qquad\qquad\qquad \textbf{DO}\ (\text{A}\ \textbf{PAR}\ (\textbf{DEF}\ (\lambda\text{(x,(i,m,s))}.\text{(i+s,m,s)})))) \\
&\qquad\quad \textbf{ELSE}\ (\textbf{WHILE}\ \ (\lambda\text{(x,(i,m,s))}.\ \text{s} \geq \text{m}) \\
&\qquad\qquad\qquad \textbf{DO}\ (\text{A}\ \textbf{PAR}\ (\textbf{DEF}\ (\lambda\text{(x,(i,m,s))}.\text{(i+s,m,s)}))))
\end{aligned} \tag{3.18}$$

Der Wert i definiert den Initialwert der Zählervariable, m ihren Grenzwert und s ihre Schrittweite. Alle drei Werte haben den Typ num. Der Typ des gesamten Ausdrucks **FOR** i m s A sei $\alpha \rightarrow \alpha\, partial$.

Die Realisierung einer **FOR**-Schleife beginnt in GROPIUS mit der Einführung der Zählervariable, des zugehörigen Grenzwerts sowie der Schrittweite der Zählervariable als lokale Variablen. Im anschließenden **IF**-Ausdruck wird in Abhängigkeit davon, ob die Schrittweite positiv oder negativ ist, zu einer der beiden **WHILE**-Schleifen mit der passenden Abbruchbedingung verzweigt. Nach der Semantik von FOR-Schleifen in MODULA-2 ist die Zählervariable im Rumpf der Schleife zwar lesbar, aber nicht veränderbar. Dieser Tatsache wird im Rumpf der Schleife durch das **PAR**-Konstrukt Rechnung getragen. Im rechten Zweig des **PAR**-Ausdrucks wird separat die Zählervariable i um die Schrittweite s erhöht bzw. vermindert und der Grenzwert m sowie die Schrittweite s unverändert weitergegeben. Der linke Zweig des **PAR**-Ausdrucks erhält wie

der rechte Zweig die gesamte Variablenstruktur (x,(i,m,s)) als Eingabe. Die Funktion A berechnet auf Basis dieser Eingabe einen neuen Wert des Typs $\alpha\, partial$ und lässt den rechten Teil der Eingabe fallen. A hat also den Typ $(\alpha \times num \times num \times num) \rightarrow \alpha\, partial$. Auf diese Weise kann die Funktion A Berechnungen auf Basis der Zählervariable (und auch des Grenzwerts bzw. der Schrittweite) durchführen, diese drei Werte aber nicht verändern.

Schnittstellenverhaltensmuster

Für eine vollständige Schaltungsbeschreibung in GROPIUS ist neben der Spezifikation des funktionalen Verhaltens der gewünschten Schaltung zusätzlich die Auswahl eines bestimmten Schnittstellenverhaltensmuster erforderlich, welches die Kommunikationsweise der Schaltung mit der Außenwelt definiert. Es steht bereits eine Auswahl aus mehreren vordefinierten Schnittstellenverhaltensmustern zur Verfügung.

Ein Schnittstellenverhaltensmuster ergänzt die algorithmische Spezifikation der Schaltung um einige Steuer- und Statussignale, im Allgemeinen um die Signale *start*, *ready* und *reset*. Die Signale *start* und *reset* sind eingehende Signale, während das Signal *ready* ein Ausgangssignal ist (siehe Abbildung 3.18). Mit Hilfe des Eingangssignals *start* wird auf Basis der gerade gültigen Eingabewerte eine Berechnung angestoßen, welche dem spezifizierten funktionalen Verhalten der Schaltung entspricht. Das Signal *ready* gibt an, ob die Schaltung gerade beschäftigt ist, oder ob sie bereit ist, eine neue Berechnung durchzuführen. Mit Hilfe des Signals *reset* kann eine Berechnung jederzeit abgebrochen und die Schaltung in ihren Anfangszustand versetzt werden.

Abbildung 3.18: Vollständige Schaltungsbeschreibung auf der algorithmischen Ebene

Die einfache Syntax vollständiger algorithmischer Schaltungsspezifikationen ist in Abbildung 3.19 dargestellt. Das Nichtterminal SSVM steht für ein bestimmtes Schnittstellenverhaltensmuster, welches dem P-Term VerhSpez zugeordnet wird. Der P-Term VerhSpez entspricht der funktionalen Verhaltensbeschreibung der gewünschten Schaltung.

```
VerhSpez = P-Term
AlgSpez  = SSVM VerhSpez
```

Abbildung 3.19: Syntax algorithmischer Schaltungsbeschreibungen

Definition 3.19 gibt die Spezifikation eines Schnittstellenverhaltensmusters wieder[†]. Die Funktion **GEN_SSVM** beschreibt eine Relation zwischen den Ein- und Ausgangssignalen einer algorithmischen Schaltungsbeschreibung. Aus dem Muster **GEN_SSVM** werden später durch spezielle Belegungen der eingehenden Steuersignale weitere Schnittstellenverhaltensmuster abgeleitet.

```
    GEN_SSVM SPEC (startsq,resetsq,readysq,inputsq,outputsq) :=                (3.19)
 1  ¬(startsq 0) ⇒ readysq 0
 2  ∧
 3  ∀ t.
 4      (readysq t ∧ ¬(startsq (t+1))) ⇒
 5                  readysq (t+1) ∧
 6                  (outputsq (t+1) = outputsq t)
 7      ∧
 8      (resetsq t ∧ ¬(startsq t)) ⇒ readysq t
 9      ∧
10      (((t=0) ∨ readysq (t-1) ∨ resetsq t) ∧ startsq t) ⇒
11          case (SPEC (inputsq t)) of
12                Defined res →
13                    (∃ m.
14                          ∀ n.(n < m ∧ (∀ p.p < n ⇒ ¬(resetsq (t+p+1)))) ⇒
15                              ¬(readysq (t+n))
16                          ∧
17                          (∀ n.n < m ⇒ ¬(resetsq (t+n+1))) ⇒
18                              ((outputsq (t+m) = res) ∧ readysq (t+m)) )
19              || Undefined →
20                    (∀ m.
21                          (∀ n.n < m ⇒ ¬(resetsq (t+n+1))) ⇒
22                              ¬(readysq (t+m)) )
```

In Definition 3.19 werden wie bei den sequentiellen Schaltungsbeschreibungen in Abschnitt 3.3.2 Sequenzen von Signalen, also Signale in Abhängigkeit von der Zeit, betrachtet. Die Zeit wird hier ebenfalls durch natürliche Zahlen des Typs num repräsentiert. Zwar ist auf der algorithmischen Ebene die Zeit ausschließlich in Bezug auf Kausalitäten relevant. Letztendlich wird jedoch eine Zeiteinheit der algorithmischen Ebene in der resultierenden Implementierung auf der RT-Ebene auf genau einen Register-Transfer, also einen Takt, abgebildet.

Die Steuersignale *start*, *reset* und *ready* haben alle den Typ $bool$. Entsprechend haben ihre Wertsequenzen startsq, resetsq und readysq den Typ $num \rightarrow bool$. Hat die Verhaltensspezifikation SPEC den Typ $\alpha \rightarrow \beta\, partial$, so folgen daraus für die Sequenzen der Ein- und Ausgabesignale (inputsq und outputsq) die Typen $num \rightarrow \alpha$ bzw. $num \rightarrow \beta$.

Das Verhalten des Schnittstellenverhaltensmusters **GEN_SSVM** ist beispielhaft in Abbildung 3.20 dargestellt. In dieser Darstellung entspricht ein hoher Signalpegel dem booleschen Wert

[†]Um die Übersichtlichkeit größerer Formeln zu erhöhen, werden im Folgenden Klammerungen teilweise weggelassen und durch Einrückung des Textes ausgedrückt.

wahr (in GROPIUS: **T**) und ein niedriger Signalpegel dem Wert *falsch* (in GROPIUS: **F**). Ein Signal, welches wahr ist, wird auch als *aktiv* bzw. *aktiviert* bezeichnet. Graue Kästen beim Ausgabesignal stehen für einen undefinierten Wert.

Um eine einfache Gegenüberstellung der verschiedenen Schnittstellenverhaltensmuster zu ermöglichen, werden in den folgenden Beispielen ihre Wirkungsweisen jeweils anhand der gleichen Eingabesequenzen (bzw. einer Untermenge davon) veranschaulicht. Weiter soll jedem Beispiel die gleiche algorithmische Verhaltensspezifikation SPEC zu Grunde liegen. Die Zugehörigkeit eines Ausgabewerts zu einem bestimmten Eingabewert wird durch Indizes hervorgehoben. So gilt zum Beispiel SPEC i1 = Defined o1, d. h. die Ausgabe o mit dem Index 1 wurde auf Basis der Eingabe i mit dem gleichen Index 1 ermittelt.

Zum Zeitpunkt 0 befindet sich die Schaltung im Ruhezustand, es sei denn, dass sofort durch ein aktives Signal *start* eine Berechnung angestoßen wird. Im Ruhezustand hat das Signal *ready* den Wert *wahr*. Dieses Verhalten ist in Zeile 1 der Definition 3.19 spezifiziert. Im Beispiel in Abbildung 3.20 ist dieser Zustand zum Zeitpunkt 0 dargestellt.

Ist die Schaltung im Ruhezustand, ist also *ready* aktiv, und wird zum darauf folgenden Zeitpunkt keine neue Berechnung durch Aktivierung des *start*-Signals angestoßen (Zeile 4), so bleibt die Schaltung im Ruhezustand (Zeile 5) und am Ausgang wird der letzte Ausgabewert weiter gehalten (Zeile 6, Zeitpunkte 0, c, e und j).

Mit Hilfe des *reset*-Signals kann eine Berechnung jederzeit abgebrochen und die Schaltung wieder in den Ruhezustand versetzt werden (Zeile 8, Zeitpunkt g). Wird allerdings gleichzeitig mit dem *reset*-Signal das *start*-Signal aktiviert, so wird eine eventuell laufende Berechnung abgebrochen und sofort eine neue Berechnung auf Basis des zu diesem Zeitpunkt gültigen Eingabewerts angestoßen (Zeile 10). In Abbildung 3.20 wird zum Beispiel die Berechnung, die zum Zeitpunkt h auf Basis des Eingabewertes i5 gestartet wurde zum Zeitpunkt i abgebrochen und zeitgleich eine neue Berechnung auf Basis des Eingabewertes i6 angestoßen.

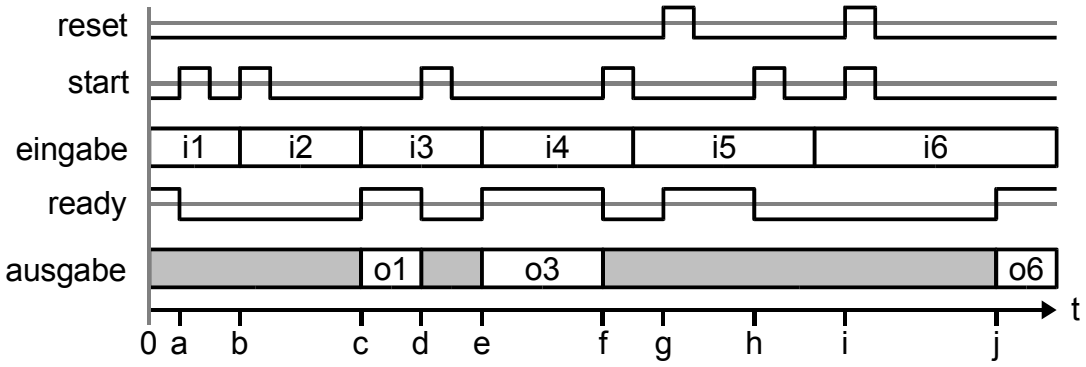


Abbildung 3.20: Verhalten des Schnittstellenverhaltensmusters **GEN_SSVM**

Um bei einem inaktiven *reset*-Signal eine neue Berechnung durch Aktivierung des Signals *start* anstoßen zu können, muss die Schaltung sich entweder im Ruhezustand befinden (Zeitpunkte a, d, f und h) oder gerade erst in Betrieb genommen worden sein (Zeile 10). Ansonsten hat die Aktivierung des Signals *start* keine Auswirkungen auf eine laufende Berechnung. So hat

zum Beispiel das Aktivieren des Signals *start* zum Zeitpunkt b keinerlei Einfluss auf die zum Zeitpunkt a begonnene Berechnung.

Eine Berechnung wird auf Basis des Eingabewertes durchgeführt, welcher zu Beginn der Berechnung am Eingang der Schaltung anliegt (Zeile 11). Zum Zeitpunkt a ist das der Wert i1, bei d i3, bei f i4, bei h i5 sowie zum Zeitpunkt i der Wert i6.

Für den Verlauf einer Berechnung (Zeile 11) gibt es zwei Möglichkeiten: Entweder sie terminiert mit einem definierten Wert res (Zeile 12-18) oder sie terminiert nicht und das Ergebnis ist undefiniert (Zeile 19-22).

Terminiert die Berechnung, so gibt es einen Wert m, welcher der Dauer der Berechnung entspricht (Zeile 13). Startet die Berechnung zum Zeitpunkt t, so endet sie zum Zeitpunkt t+m. Während einer laufenden Berechnung ist das Signal *ready* durchgehend inaktiv (Zeile 15), sofern die Berechnung nicht mit Hilfe des *reset*-Signals abgebrochen wird (Zeile 14). Bleibt während der gesamten Berechnung das *reset*-Signal inaktiv (Zeile 17), so liegt schließlich zum Zeitpunkt t+m das Ergebnis der Berechnung res am Ausgang der Schaltung an (Zeile 18, Zeitpunkte c, e und j). Gleichzeitig wird das *ready*-Signal aktiviert, um anzuzeigen, dass die gerade durchgeführte Berechnung beendet ist und ein gültiger Wert am Ausgang der Schaltung anliegt. Zusätzlich wird dadurch die Bereitschaft der Schaltung für weitere Berechnungen signalisiert.

Falls die Berechnung nicht terminiert, wird das *ready*-Signal niemals aktiviert (Zeile 22). So wird der Umgebung signalisiert, dass die Schaltung die Berechnung noch nicht zu Ende geführt hat. Erst durch die Aktivierung des *reset*-Signals (Zeile 21 bzw. Zeile 8) kann die Schaltung wieder initialisiert werden, damit sie für weitere Berechnungen zur Verfügung steht. Dabei kann, wie bereits erwähnt, durch gleichzeitiges Aktivieren des Signals *start* sofort eine neue Berechnung angestoßen werden (Zeile 10, Zeitpunkt i).

Tabelle 3.6 fasst für das Schnittstellenverhaltensmuster **GEN_SSVM** die Bedeutung der verschiedenen möglichen Kombinationen der Steuersignale nochmals zusammen. Die Werte **T** bzw. **F** stehen für aktive bzw. inaktive Signale. Der Eintrag „–“ steht sowohl für den Wert **T** als auch für den Wert **F**.

start	*reset*	*ready*	Bedeutung
F	–	**T**	Ruhezustand der Schaltung
T	–	**T**	Anstoßen einer Berechnung
T	**T**	**F**	Abbruch einer laufenden und Start einer neuen Berechnung
–	**F**	**F**	Aktiver Zustand der Schaltung
F	**T**	**F**	Abbruch einer laufenden Berechnung

Tabelle 3.6: Bedeutung der Steuersignale des Schnittstellenverhaltensmusters **GEN_SSVM**

Koppelt man im Schnittstellenverhaltensmuster **GEN_SSVM** das *reset*-Signal mit dem *start*-Signal, so erhält man das Schnittstellenverhaltensmuster **START_RESET_SSVM**:

START_RESET_SSVM SPEC (startresetsq,readysq,inputsq,outputsq) := (3.20)
GEN_SSVM SPEC (startresetsq,startresetsq,readysq,inputsq,outputsq)

Abbildung 3.21 stellt das Verhalten dieses Schnittstellenverhaltensmusters exemplarisch dar. Gleichzeitig mit der Aktivierung des *start*-Signals wird auch jeweils das *reset*-Signal aktiviert.

Somit werden die Berechnungen, welche zu den Zeitpunkten a, f und g gestartet und noch nicht zu Ende geführt wurden, jeweils durch die nachfolgenden erneuten Aktivierungen des *start*- bzw. *reset*-Signals zu den Zeitpunkten b, g und h abgebrochen.

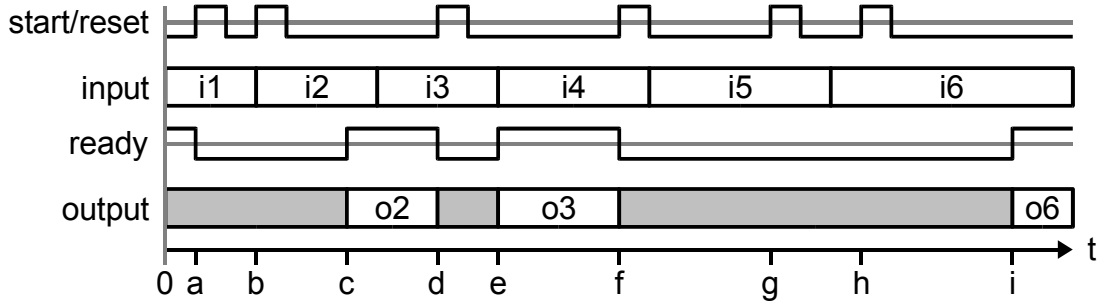


Abbildung 3.21: Verhalten des Schnittstellenverhaltensmusters **START_RESET_SSVM**

Ein weiteres Schnittstellenverhaltensmuster erhält man, indem man im Schnittstellenverhaltensmuster **GEN_SSVM** das *reset*-Signal fest deaktiviert, also konstant durch die Wertesequenz (λx.**F**) auf den Wert *falsch* setzt. Dieses Schnittstellenverhaltensmuster hat den Namen **START_NO_RESET_SSVM**:

START_NO_RESET_SSVM SPEC (startsq,readysq,inputsq,outputsq) := (3.21)
GEN_SSVM SPEC (startsq,(λx.**F**),readysq,inputsq,outputsq)

Das resultierende Verhalten wird in Abbildung 3.22 dargestellt. Im Gegensatz zum Verhalten des Schnittstellenverhaltensmusters **START_RESET_SSVM** wird beim Schnittstellenverhaltensmuster **START_NO_RESET_SSVM** durch Aktivierung des *start*-Signals keine laufende Berechnung abgebrochen. Das erneute Aktivieren des *start*-Signals zum Zeitpunkt b bzw. g hat somit auf die zum Zeitpunkt a bzw. f gestartete Berechnung keinerlei Auswirkung. Zu beachten ist, dass dieses Schnittstellenverhaltensmuster nur bei Verhaltensspezifikationen einsetzen werden sollte, bei denen man sicher weiß, dass die Berechnung für alle möglichen Eingabewerte terminiert. Eine laufende Berechnung kann ansonsten nur noch durch die Außerbetriebname der gesamten Schaltung abgebrochen werden.

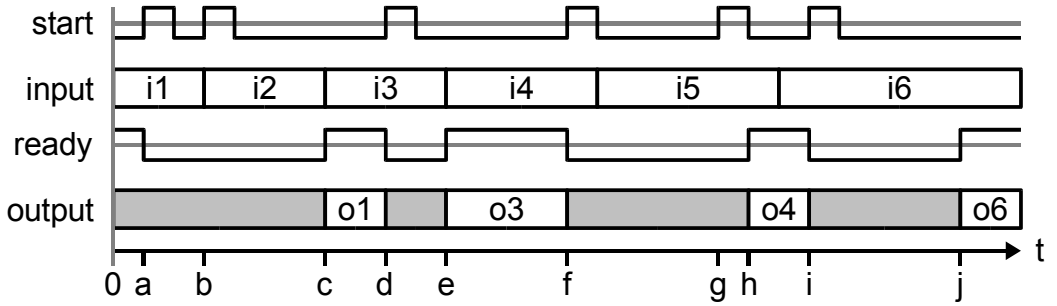


Abbildung 3.22: Verhalten des Schnittstellenverhaltensmusters **START_NO_RESET_SSVM**

Bei den folgenden beiden Schnittstellenverhaltensmustern **RUN_RESET_SSVM** und **RUN_NO_RESET_SSVM** ist das *start*-Signal konstant aktiviert. Ihre Definitionen sind in den Gleichungen 3.22 und 3.23 gegeben.

$$\begin{aligned}&\textbf{RUN\_RESET\_SSVM}\ \text{SPEC (resetsq,readysq,inputsq,outputsq) :=}\\&\textbf{GEN\_SSVM}\ \text{SPEC }((\lambda x.\textbf{T}),\text{resetsq,readysq,inputsq,outputsq})\end{aligned} \tag{3.22}$$

Beim Schnittstellenverhaltensmuster **RUN_NO_RESET_SSVM** ist zusätzlich das *reset*-Signal dauerhaft deaktiviert. Eine laufende Berechnung kann also nicht mehr abgebrochen werden. So gelten für die einzusetzenden algorithmischen Verhaltensbeschreibungen die gleichen Einschränkungen wie bei dem Schnittstellenverhaltensmuster **START_NO_RESET_SSVM**.

$$\begin{aligned}&\textbf{RUN\_NO\_RESET\_SSVM}\ \text{SPEC (readysq,inputsq,outputsq) :=}\\&\textbf{GEN\_SSVM}\ \text{SPEC }((\lambda x.\textbf{T}),(\lambda x.\textbf{F}),\text{readysq,inputsq,outputsq})\end{aligned} \tag{3.23}$$

Abbildung 3.23 beschreibt das Verhalten des Schnittstellenverhaltensmusters **RUN_RESET_SSVM**. Dadurch, dass das *start*-Signal dauerhaft aktiviert wurde, wird umgehend mit der Inbetriebnahme der Schaltung die erste Berechnung gestartet (Zeitpunkt 0). Nach dem Ende einer Berechnung wird sofort eine neue Berechnung angestoßen (Zeitpunkte b, d f und j). Durch Aktivierung des *reset*-Signals wird eine laufende Berechnungen abgebrochen und zeitgleich eine neue Berechnung angestoßen (Zeitpunkte g und h).

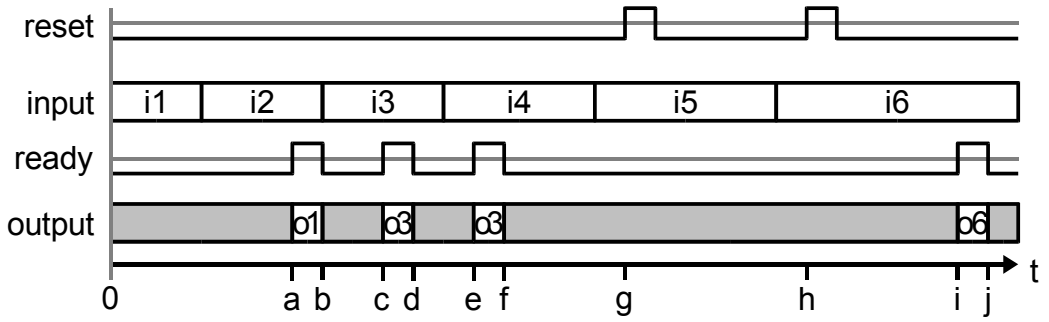


Abbildung 3.23: Verhalten des Schnittstellenverhaltensmusters **RUN_RESET_SSVM**

Das Schnittstellenverhaltensmuster **RUN_NO_RESET_SSVM** enthält keinerlei eingehende Steuersignale mehr. Die erste Berechnung startet mit der Inbetriebnahme der Schaltung (Zeitpunkt 0). Nach Beendigung einer Berechnung wird sofort eine neue Berechnung angestoßen (Zeitpunkte b, d f und h). Da bei diesem Schnittstellenverhaltensmuster wie bei dem Schnittstellenverhaltensmuster **START_NO_RESET_SSVM** das *reset*-Signal dauerhaft deaktiviert ist, ist der Abbruch von Berechnungen nicht möglich. Somit eignet sich auch dieses Schnittstellenverhaltensmuster nur für Berechnungen, die sicher terminieren. Das Verhalten des Schnittstellenverhaltensmusters **RUN_NO_RESET_SSVM** wird in Abbildung 3.24 veranschaulicht.

Eine Schaltung, welche aus einer algorithmischen GROPIUS-Spezifikation hervorgeht, liest beim Start einer Berechnung die anliegenden Eingangssignale, führt auf deren Basis die spezifizierte Berechnung durch und gibt schließlich, falls die Berechnung terminiert, ein Ergebnis

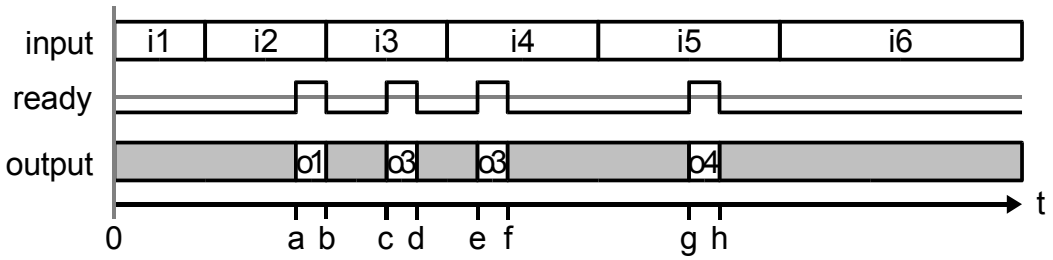


Abbildung 3.24: Verhalten des Schnittstellenverhaltensmusters **RUN_NO_RESET_SSVM**

auf den Ausgangssignalen aus. Um permanente Ströme von Eingabewerten zu verarbeiten, muss man sich der Mittel der Systemebene bedienen. Auf der Systemebene werden funktionale Prozesse über sogenannte Kommunikationsprozesse verknüpft, welche übergeordnet die Steuerung der funktionalen Prozesse übernehmen [Blum99, Blum00, SaBK01]. Die funktionalen Prozesse entsprechen algorithmischen GROPIUS-Spezifikationen und unterscheiden sich nur durch den Einsatz eines an die Systemebene angepassten Schnittstellenverhaltensmusters von den in dieser Arbeit behandelten algorithmischen Spezifikationen. Die algorithmische Synthese dieser beiden Spezifikationsformen läuft nach exakt dem gleichen Prinzip ab.

3.3.4 Die steuerflussorientierte Darstellung

Zusätzlich zu den in [Blum00] vorgestellten Schaltungsrepräsentationen wurde im Rahmen des Forschungsprojekts *Formale Synthese* eine neue Zwischendarstellung für formale Schaltungsbeschreibungen auf der algorithmischen Ebene definiert: die steuerflussorientierte Darstellung. Diese Darstellungsform repräsentiert die Schaltung durch Zustände, welche die Operationen der Spezifikation enthalten, sowie Transitionen zwischen den einzelnen Zuständen und erlaubt somit die formale Beschreibung der Schaltung als Steuer-/Datenflussgraph. Sie wird im Zuge der neu entwickelten algorithmischen Synthese (siehe Kapitel 5) eingesetzt. Durch die Durchführung der algorithmischen Synthese auf Basis der steuerflussorientierten Darstellung wird die Steuerstruktur der Spezifikation nicht mehr wie in der in [Blum00] präsentierten Vorgehensweise zerstört, sondern kann während des gesamten Syntheseprozesses nach Belieben aufrechterhalten werden. Dadurch erhöht sich das Optimierungspotential für die resultierende Schaltung beträchtlich (siehe Abschnitt 2.2.3, Seite 20). Grundlage der steuerflussorientierten Darstellung ist das Konstrukt **LOOP**.

In den folgenden Beschreibungen stehen Kleinbuchstaben für Variablen elementaren Typs (**a**, **b**, **c**, ...). Unterstrichene Buchstaben bezeichnen Strukturen solcher Variablen (**x**, **vsr**, ...). Kleingeschriebene, schräggestellte Buchstaben repräsentieren ganze DFG-Terme (***f***, ***g***, ***cnd***, ...) und schräggestellte Großbuchstaben P-Terme (***L***, ***R***, ***T***, ...). Nicht fett dargestellte Bezeichner stehen für Variablen beliebigen Typs.

Das Konstrukt LOOP

Das Konstrukt **LOOP** bildet die Grundlage der steuerflussorientierten Darstellung. Das Konstrukt hat als Operanden eine Liste L von Zuständen sowie einen Zustandsbezeichner **start**, der den Startzustand definiert:

LOOP start L

Die Bestandteile eines **LOOP**-Ausdrucks haben folgende Typen:

Bezeichner	Typ
LOOP	$\alpha \rightarrow (\alpha \times (\beta \rightarrow (\beta \times \alpha)\, partial))\, list \rightarrow (\beta \rightarrow \beta\, partial)$
start	α
L	$(\alpha \times (\beta \rightarrow (\beta \times \alpha)\, partial))\, list$

Tabelle 3.7: Bestandteile der steuerflussorientierten Darstellung

Listen in HOL: Eine Liste wird von dem HOL-Parser in eckige Klammern eingefasst dargestellt. Als Trennungselement für die Elemente einer Liste wird das Semikolon verwendet. Eine Liste hat somit die Form [el1; el2; ...]. Durch den Listenkonstruktor **::** kann ein Element am Anfang einer Liste hinzugefügt werden. So bezeichnet der Ausdruck (h**::**R) eine Liste mit dem Kopf h und dem Rest R. Die leere Liste wird durch [ ] dargestellt.

Darstellung der Zustände

Ein Zustand in der **LOOP**-Liste L ist ein Paar bestehend aus dem Bezeichner des Zustands und dem sogenannten *Rumpf* des Zustands. Jeder Zustand ist durch seinen Bezeichner eindeutig gekennzeichnet. Als Bezeichner kommen in dieser Arbeit natürliche Zahlen des Typs num zum Einsatz. Prinzipiell können aber auch andere Typen verwendet werden. Im Folgenden wird auch die Abkürzung $\langle \mathbf{s} \rangle$ für einen Zustand mit dem Bezeichner **s** verwendet.

Sind die Bezeichner vom Typ α, so haben die Rümpfe der Zustände einen Typ $\beta \rightarrow (\beta \times \alpha)\, partial$. Der Typ β entspricht dabei dem Eingangstyp des gesamten **LOOP**-Ausdrucks. Im Rumpf eines Zustands wird auf Basis einer Eingabe des Typs β ein neuer Wert desselben Typs berechnet und zusätzlich der Bezeichner des Folgezustands (vom Typ α) ermittelt.

Definition des Konstrukts LOOP

Bei der formalen Definition des **LOOP**-Konstrukts kommen zwei Hilfsfunktionen zum Einsatz: **StateExists** und **StateCase**. Die Funktion **StateExists** überprüft, ob in einer Zustandsliste ein Zustand eines bestimmten Bezeichners existiert. Sie ist folgendermaßen rekursiv definiert:

(**StateExists** (h**::**R) s := ((s = **FST** h) $\vee$ **StateExists** R s)) $\wedge$ (3.24)
(**StateExists** [ ] s := **F**)

Ein Zustand mit dem Bezeichner s ist genau dann in der Liste vorhanden, wenn entweder der gesuchte Bezeichner s mit dem Bezeichner (**FST** h) des ersten Zustands h in der Liste übereinstimmt oder wenn er dem Bezeichner eines der Zustände im Rest R der Liste entspricht. Im rechten Teil der Konjunktion wird der Fall abgedeckt, in dem die Liste leer ist. In diesem Fall terminiert die Funktion **StateExists** mit dem Wert **F** (*falsch*).

Die Funktion **StateCase** wählt aus einer Zustandsliste anhand eines Zustandsbezeichners den Rumpf des entsprechenden Zustands aus. Sie ist ebenso wie die Funktion **StateExists** rekursiv definiert:

$$\textbf{StateCase}\ (h::R)\ s := (\text{if } s = \textbf{FST}\ h \text{ then } \textbf{SND}\ h \text{ else } \textbf{StateCase}\ R\ s) \tag{3.25}$$

Stimmt der Bezeichner (**FST** h) des ersten Zustands h der Liste mit dem gesuchten Bezeichner s überein, so wird der Rumpf (**SND** h) dieses Zustands ausgegeben. Ansonsten wird im Rest R der Liste weiter nach einem entsprechenden Zustand gesucht.

Mit Hilfe der Funktionen **StateExists** und **StateCase** ist das Konstrukt **LOOP** nun folgendermaßen definiert:

$$\begin{aligned}\textbf{LOOP start}\ L := \ & \textbf{DEF}\ (\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\textbf{start})) \\ & \textbf{SER} \\ & (\ \textbf{WHILE}\ (\lambda(\underline{\mathbf{x}},\mathbf{s}).\ \textbf{StateExists}\ \mathbf{L}\ \mathbf{s}) \\ & \qquad \textbf{DO}\ (\lambda(\underline{\mathbf{x}},\mathbf{s}).\ \textbf{StateCase}\ \mathbf{L}\ \mathbf{s}\ \underline{\mathbf{x}})\ ) \\ & \textbf{SER} \\ & \textbf{DEF FST}\end{aligned} \tag{3.26}$$

Der erste **DEF**-Ausdruck führt eine Hilfsvariable ein, welche den Bezeichner des jeweils aktuellen Zustands enthält (siehe auch Abbildung 3.25). Sie wird mit dem Wert **start**, dem Bezeichner des Anfangszustands, initialisiert. Die Hilfsvariable wird nach der Ausführung der **WHILE**-Schleife durch den abschließenden **DEF**-Term wieder fallengelassen.

In der Bedingung der **WHILE**-Schleife wird überprüft, ob in der Zustandsliste ein Zustand existiert, dessen Bezeichner dem aktuellen Inhalt der Hilfsvariable entspricht. Ist dies nicht der Fall, so verweist die Hilfsvariable auf einen sogenannten *Endzustand* und die Schleife terminiert. Ansonsten wird der Rumpf der **WHILE**-Schleife ausgeführt. Dort wird durch den Ausdruck **StateCase L s** der Rumpf des aktuellen Zustands $\langle \mathbf{s} \rangle$ ausgewählt, welcher anschließend auf $\underline{\mathbf{x}}$ angewandt wird. In diesem Zusammenhang wird $\underline{\mathbf{x}}$ als *Eingabe* des Zustands $\langle \mathbf{s} \rangle$ bezeichnet. Die Anwendung des Rumpfs von $\langle \mathbf{s} \rangle$ auf $\underline{\mathbf{x}}$ liefert schließlich den Bezeichner $\mathbf{s}'$ des Folgezustands von $\langle \mathbf{s} \rangle$ sowie dessen Eingabewert $\underline{\mathbf{x}}'$.

Weitere Hilfsfunktionen

In diesem Abschnitt werden einige zusätzliche Hilfsfunktionen vorgestellt, die später in Verbindung mit den Transformationen, welche auf der steuerflussorientierten Darstellung operieren, zum Einsatz kommen (siehe Abschnitt 5.2). Manche der Funktionen sind den Bibliotheken von HOL entnommen, während andere selbst im Projekt *Formale Synthese* entwickelt wurden.

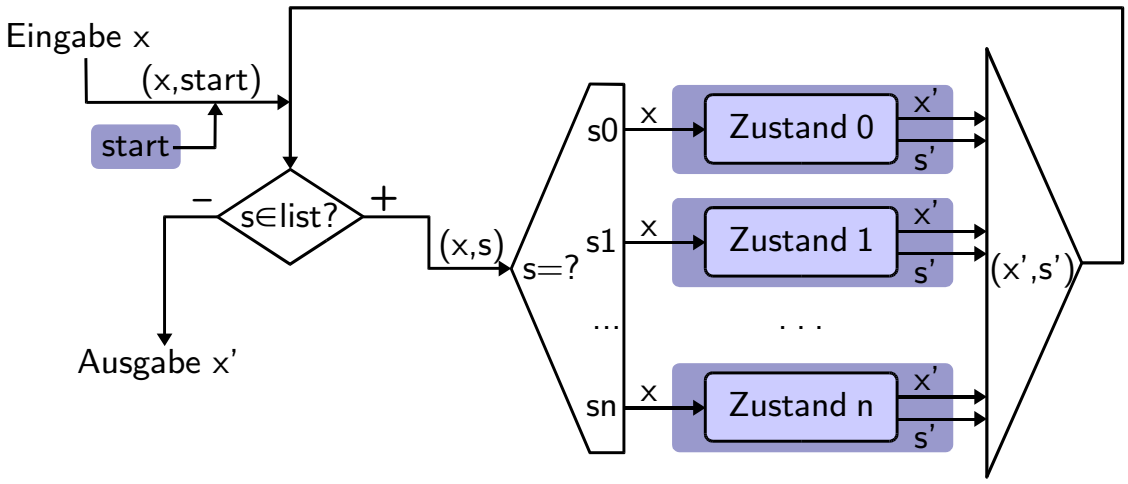


Abbildung 3.25: Die **LOOP**-Schleife

DeleteState: Die Funktion **DeleteState** löscht das erste Auftreten eines Zustands mit einem bestimmten Bezeichner aus der Zustandsliste einer steuerflussorientierten Darstellung. Sie ist folgendermaßen definiert:

$$\begin{aligned}&(\ \textbf{DeleteState}\ (h::R)\ s := (\text{if } s = \textbf{FST}\ h \text{ then } R \text{ else } h::(\textbf{DeleteState}\ R\ s)\ ) \wedge\\ &(\ \textbf{DeleteState}\ [\,]\ s \qquad := [\,]\ )\end{aligned} \tag{3.27}$$

Der Ausdruck **DeleteState** L s hat als Ergebnis die Liste L, aus welcher das erste Auftreten eines Zustand mit dem Bezeichner s entfernt wurde. Existiert in der Liste kein entsprechender Zustand, so liefert die Funktion die unveränderte Liste L.

Da in dieser Arbeit ausschließlich Transformationen eingesetzt werden, die sicherstellen, dass die Bezeichner aller Zustände in der Zustandsliste einer steuerflussorientierten Darstellung eindeutig sind, kann der gesuchte Zustand $\langle s \rangle$ höchstens einmal in einer Zustandsliste vorkommen. Somit entspricht ein „erstes Auftreten“ des Zustands gegebenenfalls dem einzigen Auftreten dieses Zustands.

Die Funktion **DeleteState** erzeugt die neue Liste konstruktiv aus der ursprünglichen Liste. Vom Kopf bis zum Ende der Liste wird rekursiv untersucht, ob der Bezeichner **FST** h des ersten Zustands h der Liste dem gesuchten Bezeichner s entspricht. Ist dies der Fall, so gibt die Funktion **DeleteState** nur den Rest R der Liste (ohne den Zustand h) zurück. Anderenfalls wird eine Liste mit dem Zustand h als Kopf und dem Rest R, in dem weiter rekursiv nach einem Zustand $\langle s \rangle$ gesucht und dieser gegebenenfalls eliminiert wird, zurückgegeben. Ist die zu durchsuchende Liste leer, so gibt **DeleteState** eine leere Liste zurück und terminiert.

EXISTS: Die Funktion **EXISTS** gibt an, ob es in einer Liste ein Element gibt, welches ein bestimmtes Prädikat erfüllt. Ihre Definition lautet:

$$\begin{aligned}&(\ \textbf{EXISTS}\ P\ (h::R) := P\ h \vee \textbf{EXISTS}\ P\ R\ ) \wedge\\ &(\ \textbf{EXISTS}\ P\ [\,] \qquad := \textbf{F}\ )\end{aligned} \tag{3.28}$$

Der Ausdruck **EXISTS** P (h::R) ist genau dann *wahr*, wenn irgendein Element der Liste (h::R) das Prädikat P erfüllt. Zunächst wendet die Funktion das Prädikat P auf das erste Element h der Liste an. Ergibt dieser Ausruck P h den Wert *wahr*, so ergibt der gesamte **EXISTS**-Ausdruck den Wert *wahr*. Ansonsten wird rekursiv der Rest R der Liste untersucht. Findet sich kein Element, auf welches das Prädikat P zutrifft, so ergibt der **EXISTS**-Ausdruck den Wert *falsch*.

ALL_EL: Die Funktion **ALL_EL** gibt an, ob ein bestimmtes Prädikat für alle Elemente einer Liste gilt. Ihre Definition lautet:

$$\begin{array}{l}(\ \textbf{ALL\_EL}\ \mathrm{P}\ (\mathrm{h}::\mathrm{R}) := \mathrm{P}\ \mathrm{h} \wedge \textbf{ALL\_EL}\ \mathrm{P}\ \mathrm{R}\ ) \wedge \\ (\ \textbf{ALL\_EL}\ \mathrm{P}\ [\,] \quad := \textbf{T}\ )\end{array} \tag{3.29}$$

Der Ausdruck **ALL_EL** P (h::R) ist genau dann *wahr*, wenn alle Elemente der Liste (h::R) das Prädikat P erfüllen. Zunächst wendet die Funktion das Prädikat P auf das erste Element h der Liste an. Ergibt dieser Ausdruck P h den Wert *falsch*, so ergibt der gesamte **EXISTS**-Ausdruck den Wert *falsch*. Ansonsten wird rekursiv der Rest R der Liste untersucht. Ist die Liste leer oder trifft das Prädikat P auf alle Elemente zu, so ergibt der **ALL_EL**-Ausdruck den Wert *wahr*.

UNZIP: Die Funktion **UNZIP** erzeugt aus einer Liste von Paaren ein Paar zweier Listen. Die linke bzw. rechte Liste des resultierenden Paars enthält dabei die jeweils linken bzw. rechten Teile aller Paare der ursprünglichen Liste. Die Funktion ist folgendermaßen definiert:

$$\begin{array}{l}(\ \textbf{UNZIP}\ ((\mathrm{l},\mathrm{r})::\mathrm{R}) := (\mathrm{l}::\textbf{FST}\ (\textbf{UNZIP}\ \mathrm{R}), \mathrm{r}::\textbf{SND}\ (\textbf{UNZIP}\ \mathrm{R}))\ ) \wedge \\ (\ \textbf{UNZIP}\ [\,] \quad := ([\,],[\,])\ )\end{array} \tag{3.30}$$

Ähnlich wie die Funktion **DeleteState** baut auch die Funktion **UNZIP** konstruktiv die beiden Listen auf Basis der ursprünglichen Liste auf. Ist das erste Element der Liste ein Paar der Form (l,r), so liefert die Funktion **UNZIP** ein Paar zweier Listen zurück, in welchem der Term l bzw. r das erste Element der linken bzw. rechten Liste bildet. Die Reste der beiden Listen ermittelt die Funktion **UNZIP** entsprechend rekursiv auf Basis des Rests R der ursprünglichen Liste.

MEM: Die Funktion **MEM** überprüft, ob ein bestimmtes Element in einer Liste enthalten ist. Sie ist folgendermaßen definiert:

$$\begin{array}{l}(\ \textbf{MEM}\ \mathrm{x}\ (\mathrm{h}::\mathrm{R}) := (\mathrm{x} = \mathrm{h}) \vee \textbf{MEM}\ \mathrm{x}\ \mathrm{R}\ ) \wedge \\ (\ \textbf{MEM}\ \mathrm{x}\ [\,] \quad := \textbf{F}\ )\end{array} \tag{3.31}$$

Der Ausdruck **MEM** x (h::R) gibt an, ob das Element x in der Liste vorkommt. Es wird überprüft, ob x dem ersten Element h entspricht. Ist dies nicht der Fall, so wird rekursiv der Rest der Liste überprüft. Ist kein entsprechendes Element in der Liste enthalten, terminiert die Funktion **MEM** mit dem Wert *falsch*.

MAP: Die Funktion **MAP** wendet eine bestimmte Funktion auf alle Elemente einer Liste an. Sie ist folgendermaßen definiert:

$$\begin{array}{ll} (\ \textbf{MAP}\ f\ (h::R) & := (f\ h)::(\textbf{MAP}\ f\ R)\ ) \wedge \\ (\ \textbf{MAP}\ f\ [\,] & := [\,]\ ) \end{array} \tag{3.32}$$

Der Ausdruck **MAP** f L wendet die Funktion f einmalig auf jedes Element der Liste L an.

Disjoint: Die Funktion **Disjoint** gibt an, ob alle Elemente einer Liste nur ein einziges Mal in dieser Liste auftreten. **Disjoint** wird im Zusammenhang mit steuerflussorientierten Darstellungen meistens dazu verwendet, die Eindeutigkeit der Bezeichner aller Zustände in der Zustandsliste auszudrücken. Ihre Definition lautet:

$$\begin{array}{ll} (\ \textbf{Disjoint}\ (h::R) & := \neg(\textbf{MEM}\ h\ R) \wedge \textbf{Disjoint}\ R\ ) \wedge \\ (\ \textbf{Disjoint}\ [\,] & := \textbf{T}\ ) \end{array} \tag{3.33}$$

Ausgehend von dem Ausdruck **Disjoint** (h::R) prüft die Funktion **Disjoint** zunächst mit Hilfe der Funktion **MEM**, ob das Kopfelement h im Rest R der Liste enthalten ist. Ist dies nicht der Fall, so wird rekursiv der Rest R der Liste überprüft. Durch diese Vorgehensweise wird paarweise die Ungleichheit aller Elemente der Liste untersucht. Die Funktion ergibt den Wert *wahr*, wenn keine Elemente mehr in der Liste übrig sind.

Teil II

Konzept einer formalen algorithmischen Synthese

Die formale Synthese beginnt mit der Eingabe der Spezifikation der gewünschten Schaltung als Term im Theorembeweiser. Auf der algorithmischen Ebene besteht eine Schaltungsbeschreibung aus der Verhaltensspezifikation der Schaltung in der Sprache GROPIUS sowie einem ausgewählten Schnittstellenverhaltensmuster, welches die Kommunikation der resultierenden Schaltung zur Außenwelt regelt.

Bei der formalen Synthese wird unterschieden zwischen der eigentlichen Durchführung der Synthese, also der Ableitung der Implementierung aus der Spezifikation, und der Ermittlung der Entwurfsziele, welche zur Steuerung des Ableitungsprozesses innerhalb des Theorembeweisers herangezogen werden. Da es effizientere Plattformen als den Theorembeweiser HOL zur Durchführung der oft aufwändigen Optimierungs- und Syntheseverfahren zur Ermittlung der Syntheseziele gibt, werden diese häufig außerhalb des Theorembeweisers ausgeführt [EiBK96, BlEK96]. Die eigentliche Synthese, während der die Implementierung nach den Vorgaben entsprechender Verfahren aus der Spezifikation abgeleitet wird, findet dagegen vollständig innerhalb des Theorembeweisers statt. Durch diese Aufteilung können die Entwurfsentscheidungen auf eine hocheffiziente Weise getroffen werden und gleichzeitig garantiert deren Umsetzung innerhalb des Theorembeweisers, dass die resultierende Implementierung auch tatsächlich das ursprünglich spezifizierte Verhalten aufweist (siehe auch Abschnitt 3.2.3).

Neben der Korrektheit der Implementierung spielt deren Qualität (Fläche, Verzögerung, Leistungsverbrauch) eine ebenso entscheidende Rolle für ihre Verwertbarkeit. Daher ist es von großer Bedeutung, die Entwurfsentscheidungen auf Basis hochwertiger Verfahren zu treffen. Anstatt nun spezielle Entwurfsraumuntersuchungsmethoden für die formale Synthese zu entwickeln, liegt es nahe, auf den riesigen Vorrat bereits existierender, ausgereifter Optimierungs- und Syntheseverfahren aus dem Bereich der konventionellen Synthese zurückzugreifen.

In [Blum00] wurden bereits konventionelle Algorithmen, wie zum Beispiel der *List-Based-Scheduling*-Algorithmus [DLSM81] und der *Force-Directed-Scheduling*-Algorithmus [PaKn87], zur Ermittlung von Ablaufplänen reiner Datenflussbeschreibungen eingesetzt. Die algorithmische Synthese steuerflussbehafteter Schaltungsbeschreibungen unterscheidet sich in [Blum00] jedoch so stark von konventionellen Syntheseverfahren, dass der Einsatz konventioneller Synthesealgorithmen in diesem Zusammenhang nicht praktikabel ist. Zusätzlich ist in [Blum00] die formale algorithmische Synthese nur partiell automatisiert. Viele der Transformationen müssen nach Gutdünken des Entwicklers interaktiv selektiert und angewandt werden, so dass vom Schaltungsentwerfer weitreichende Kenntnisse in formalen Methoden gefordert werden (siehe Abschnitt 2.2.3).

Im Zuge der hier vorliegenden Arbeit wurde ein neues Konzept für die formale algorithmische Synthese digitaler Schaltungsbeschreibungen erarbeitet. Das Konzept ist in der Hinsicht universell anwendbar, dass es sowohl für reine Datenflussbeschreibungen wie auch für steuerflussbehaftete Schaltungsbeschreibungen geeignet ist. Es erlaubt die Integration konventioneller Syntheseverfahren in den formalen Syntheseprozess und die vollautomatische Umsetzung der von den Syntheseverfahren gefällten Entwurfsentscheidungen. Somit wird auch Schaltungsentwicklern ohne Kenntnisse in formalen Methoden ermöglicht, garantiert korrekte Implementierungen hoher Qualität zu entwerfen.

In Kapitel 4 werden einige Maßnahmen beschrieben, welche zur Optimierung der Implementierungsqualität während der formalen algorithmischen Synthese ergriffen werden. Eine der

wichtigsten Errungenschaften ist dabei das in dieser Arbeit entwickelte Rahmenwerk, welches die einfache und effiziente Integration konventioneller Optimierungs- und Syntheseverfahren in den formalen Syntheseprozess erlaubt. Anschließend wird in Kapitel 5 das neu entwickelte Konzept zur Durchführung einer formalen algorithmischen Synthese erläutert. Dieses Konzept ermöglicht die vollautomatische Synthese sowohl reiner Datenflussbeschreibungen wie auch steuerflussbehafteter Schaltungsbeschreibungen unter Berücksichtigung der zuvor getroffenen Entwurfsentscheidungen.

Kapitel 4

Optimierungs- und Syntheseverfahren in der formalen algorithmischen Synthese

Neben der Korrektheit einer aus der formalen Synthese resultierenden Implementierung ist deren Qualität von ebenso großer Bedeutung. Eine Implementierung, die zwar garantiert das in der Spezifikation definierte Verhalten aufweist, aber langsam ist oder eine große Fläche benötigt, kann genauso unbrauchbar sein wie eine fehlerhafte Implementierung.

Da im Bereich der konventionellen Synthese bereits unzählige Verfahren zur Optimierung und Synthese digitaler Schaltungen entwickelt wurden, liegt es nahe, auch in der formalen Synthese zur Erzielung hochwertiger Implementierungen aus diesem enormen Fundus ausgereifter Algorithmen zu schöpfen. Während sich einige dieser Verfahren schnell und einfach in der funktionalen Programmiersprache Moscow ML, welche dem Theorembeweiser HOL zu Grunde liegt, realisieren und so in das formale Synthesesystem integrieren lassen, bietet sich für viele jedoch eher eine Ausführung außerhalb des Theorembeweisers an. Zahlreiche Methoden wurden beispielsweise in prozeduralen oder objektorientierten Umgebungen entworfen. Werden die entsprechenden Algorithmen in ihren ursprünglichen Umgebungen ausgeführt, so können deren paradigmenspezifische und umgebungsabhängige Optimierungen voll zum Tragen kommen. Zusätzlich entfällt deren gegebenenfalls aufwändige Portierung auf die funktionale Sprache Moscow ML.

Zur einfachen und effizienten Integration konventioneller Optimierungs- und Syntheseverfahren in das formale Synthesesystem wurde im Zuge der vorliegenden Arbeit ein Rahmenwerk entwickelt, welches eine Brücke zwischen der konventionellen und der formalen Synthese schlägt. Durch die Möglichkeit, existierende Verfahren außerhalb des Theorembeweisers auszuführen, wird deren Anbindung an das formale Synthesesystem vereinfacht und gleichzeitig deren Effizienz aufrechterhalten. Darüber hinaus erlaubt das Rahmenwerk die Integration kommerzieller Synthesewerkzeuge, deren Quelltext nicht offenliegt und daher deren Reimplementierung im Theorembeweiser von vornherein nicht in Frage kommt.

In Abschnitt 4.1 wird zunächst der Aufbau und die Funktionsweise des neu entwickelten Rahmenwerks beschrieben. Anschließend werden in Abschnitt 4.2 einige algorithmische Transformationen vorgestellt, welche zur Optimierung von Verhaltensspezifikationen dienen bzw. deren Optimierungspotenzial für später auszuführende Optimierungs- und Syntheseverfahren erhöhen.

Die einzelnen Funktionsblöcke des Rahmenwerks werden in den Abschnitten 4.3 bis 4.7 näher betrachtet.

4.1 Anbindung konventioneller Optimierungs- und Syntheseverfahren

In der formalen Synthese wird die Korrektheit des Syntheseergebnisses sichergestellt, indem die Ableitung der Implementierung aus der Spezifikation vollständig innerhalb des Theorembeweisers erfolgt (siehe Abschnitt 3.2). Die Ermittlung der Entwurfsziele auf Basis entsprechender Optimierungs- und Syntheseverfahren kann dabei auch außerhalb des Theorembeweisers durchgeführt werden. Anhand der resultierenden Entwurfsziele werden dann durch das formale Synthesesystem passende Transformationen selektiert und ausgeführt, welche die formale Umsetzung der Entwurfsziele innerhalb des Theorembeweisers bewirken. Die Sicherheit des Ansatzes der formalen Synthese wird durch diese Vorgehensweise nicht gefährdet, denn sollte eine der getroffenen Entwurfsentscheidungen fehlerhaft sein, so wird sich deren Umsetzung im Theorembeweiser als unmöglich erweisen (siehe Abschnitt 3.2.3).

Die Möglichkeit der externen Ausführung von Entwurfsraumuntersuchungen wird in dieser Arbeit genutzt, um die Integration konventioneller Synthesetechniken in den formalen Syntheseprozess zu optimieren. Dabei stehen die Einfachheit deren Integration sowie deren effiziente Ausführung im Vordergrund. Beide Ziele wurden durch das Schaffen geeigneter Schnittstellen an das formale Synthesesystem erreicht. Durch Nutzung dieser Schnittstellen ist es möglich, konventionelle Verfahren in denjenigen Umgebungen auszuführen, für die sie ursprünglich konzipiert waren und für die sie optimiert wurden. So kann beispielsweise ein objektorientierter Algorithmus, der in das formale Synthesesystem eingebettet werden soll, in einer objektorientierten Umgebung (wie C++) ausgeführt werden. Alternativ müsste der Algorithmus auf die funktionale Programmiersprache Moscow ML, welche dem formalen Synthesesystem zu Grunde liegt, portiert werden. Neben einer gegebenenfalls aufwändigen Portierung nähme man damit jedoch auch häufig eine Verringerung der Effizienz des Algorithmus in Kauf. Durch eine Portierung würden in der Regel alle umgebungs- und paradigmenspezifischen Optimierungen des Algorithmus unwirksam.

In den folgenden Beschreibungen des im Zuge dieser Arbeit entwickelten Rahmenwerks bezeichnet der Begriff *formale Umgebung* den Theorembeweiser als Ausführungsumgebung, während *externe Umgebung* für eine Ausführungsumgebung außerhalb des Theorembeweisers bzw. des formalen Synthesesystems steht.

Der Einsatz eines Optimierungs- oder Synthesealgorithmus außerhalb des Theorembeweisers HOL läuft wie folgt ab: Ausgangspunkt eines jeden Schritts der formalen Synthese ist eine formale Spezifikation der Schaltung als logischer Term im Theorembeweiser (siehe Abbildung 4.1). Kommt ein externes Verfahren zum Einsatz, so muss zunächst die formale Schaltungsrepräsentation in eine zur Verarbeitung durch das anzuwendende Verfahren geeignete Form überführt werden. Im Zuge dessen werden nach einigen Umformungen und Optimierungen der Schaltungsbeschreibung (Abschnitt 4.2) deren Steuer- und Datenfluss analysiert (Abschnitt 4.3). Ergebnis

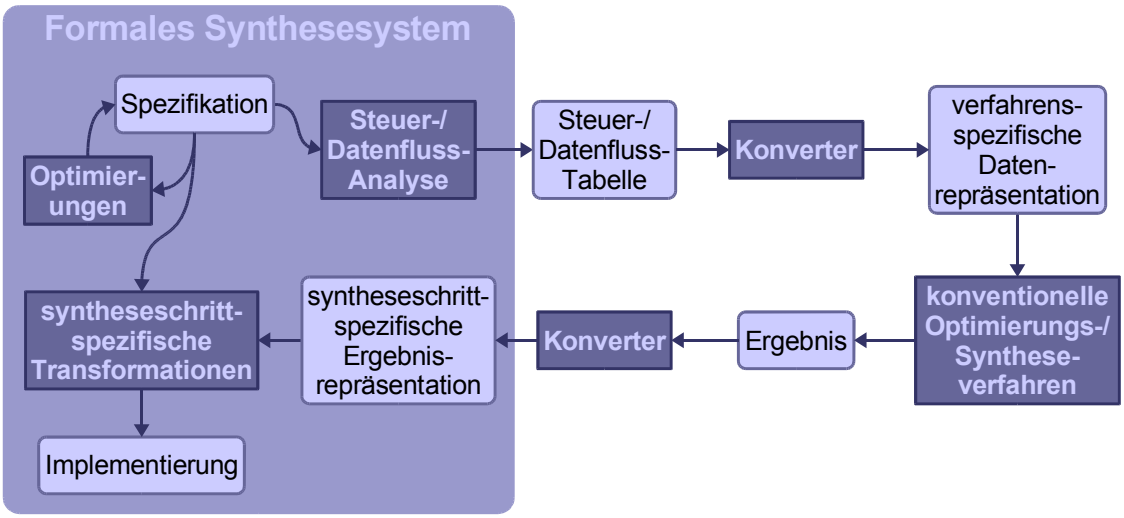


Abbildung 4.1: Anbindung externer Optimierungs- und Syntheseverfahren

dieser Analyse ist eine Tabelle, welche alle in der Spezifikation auftretenden Operationen sowie deren Steuer- und Datenflussabhängigkeiten enthält. Auf Basis dieser Steuer-/Datenflusstabelle wird durch einen Konverter eine für das extern anzuwendende Verfahren geeignete Datenrepräsentation der Schaltung erzeugt (Abschnitt 4.4). Das externe Verfahren (oder mehrere Verfahren, siehe Abschnitt 4.5) wird (werden) ausgeführt und dessen (deren) Ergebnis(-se) durch einen weiteren Konverter in eine für deren Umsetzung im formalen Synthesesystem geeignete Repräsentation überführt (Abschnitt 4.6). Die Form dieser Darstellung ist von dem jeweils durchzuführenden Syntheseschritt abhängig. Anschließend werden in dem formalen System adäquate Transformationen selektiert und ausgeführt, welche eine formale Ableitung der Implementierung aus der Spezifikation nach den Vorgaben der extern ermittelten Entwurfsziele innerhalb des Theorembeweisers bewirken (Abschnitt 4.7).

Wie bereits erwähnt, ermöglicht das Rahmenwerk auch die Integration kommerzieller Synthesewerkzeuge, deren Quelltext nicht verfügbar ist. Es müssen nur geeignete Konverter für die unterschiedlichen Arten der Datenrepräsentation im Theorembeweiser und in dem konventionellen Synthesewerkzeug entwickelt werden.

4.2 Vorbereitung der algorithmischen Synthese

Als Vorbereitung der formalen algorithmischen Synthese wird die Spezifikation der zu synthetisierenden Schaltung in eine Form gebracht, die sich für eine automatisierte Durchführung der Synthese im Theorembeweiser, wie sie in Kapitel 5 beschrieben wird, eignet und gleichzeitig die Analyse des Steuer- und Datenflusses der Spezifikation erleichtert. Es werden einige algorithmische Optimierungen der Spezifikation vorgenommen sowie diverse Umformungen, welche das Potenzial für weitere Optimierungen erhöhen bzw. die anschließende Synthese der Schaltung vereinfachen oder beschleunigen.

4.2.1 Zurückführung auf Grundkonstrukte

Um generell eine Vereinfachung und Effizienzsteigerung der formalen algorithmischen Synthese zu erlangen, werden zunächst alle erweiterten algorithmischen Konstrukte der gegebenen Verhaltensspezifikation durch ihre Definitionen (siehe Abschnitt 3.3.3, Seite 48) ersetzt. Durch diese Maßnahme verkleinert sich die Menge der in der resultierenden Verhaltensspezifikation verwendeten Konstrukte auf die fünf Grundkonstrukte **DEF**, **SER**, **IF**, **PAR** und **WHILE**.

Durch die Beschränkung auf die fünf Grundkonstrukte wird die Anzahl an konstruktspezifischen Transformationen sowie entsprechender Theoreme reduziert. Die Entwicklung von Verfahren, welche auf dieser reduzierten Darstellungsform operieren, vereinfacht sich somit beträchtlich. Die ganze formale algorithmische Synthese wird durch diesen Schritt übersichtlicher, effizienter und leichter wartbar.

4.2.2 Verschieben der Bedingungsberechnungen

Die Transformation *Verschieben der Bedingungsberechnung* von **IF**- und **WHILE**-Ausdrücken erhöht das Optimierungspotenzial einer Spezifikation und vereinfacht die spätere Analyse von deren Steuer- und Datenfluss (siehe Abschnitt 4.3). Zu deren besserem Verständnis sei an dieser Stelle an den syntaktischen Aufbau von P-Termen erinnert (siehe auch Abschnitt 3.3.3):

P-Term = "**DEF**" DFG-Term |
P-Term "**SER**" P-Term |
P-Term "**PAR**" P-Term |
"**IF**" DFG-Kond "**THEN**" P-Term "**ELSE**" P-Term |
"**WHILE**" DFG-Kond "**DO**" P-Rumpf

Im Zuge der Transformation wird die Berechnung der Bedingungen von **IF**- und **WHILE**-Ausdrücken aus deren Bedingungsblöcken DFG-Kond herausgezogen und in separaten **DEF**-Termen ausgeführt. Als Ergebnis der Transformation wird im Falle eines **IF**-Ausdrucks dessen Bedingung in einem **DEF**-Term vor dem entsprechenden Ausdruck ausgewertet, während die Bedingungsberechnung einer **WHILE**-Schleife in einen **DEF**-Term in den Rumpf der Schleife verschoben wird. Zur Speicherung des Ergebnisses der Bedingungsberechnung wird in beiden Fällen eine neue boolesche Hilfsvariable eingeführt. In den aus der Transformation resultierenden Bedingungsblöcken der **IF**- und **WHILE**-Terme werden schließlich keinerlei Berechnungen mehr durchgeführt, sondern nur noch die jeweils neu eingeführte boolesche Variable getestet.

Das Verschieben der Bedingungsberechnungen macht den in den Bedingungsblöcken enthaltenen Code für zahlreiche Optimierungen zugänglich. Beispiele einiger Verfahren, die von diesem Schritt profitieren, sind die Eliminierung toten Codes (siehe Abschnitt 5.1), die Maximierung der Grundblöcke (siehe Abschnitt 4.2.3) sowie das Zusammenfassen gemeinsamer Teilausdrücke (siehe Abschnitt 4.2.4). Von besonderer Bedeutung ist dieser Schritt auch für die zeitliche Einteilung der in den Bedingungsberechnungen verwendeten Operationen, die im Zuge der Ablaufplanung vorgenommen wird. Eine Sonderbehandlung der Bedingungsberechnungen während der Synthese, wie sie noch in [Blum00] stattfindet, ist als Folge dieser Transformation nicht mehr erforderlich.

Verschieben der Bedingungsberechnung von IF-Ausdrücken

Theorem 4.1 bildet die Grundlage für das Verschieben der Bedingungsberechnung von **IF**-Konstrukten. Ausgangspunkt ist ein Term der Form (**IF** cnd **THEN** A **ELSE** B). Nach Ausführung der Transformation wird die Bedingungsfunktion cnd des ursprünglichen **IF**-Ausdrucks zunächst in einem separaten **DEF**-Term ausgewertet. Der **DEF**-Term reicht das Ergebnis der Bedingungsberechnung als booleschen Wert zusätzlich zum ursprünglichen Eingabewert an den aus der Transformation resultierenden **IF**-Term weiter.

$$
\begin{array}{l}
\vdash \forall \text{ cnd A B.} \\
\quad (\textbf{IF } \text{cnd } \textbf{THEN } \text{A } \textbf{ELSE } \text{B}) \\
= \textbf{DEF } (\lambda \text{x.(x,cnd x)}) \\
\quad \textbf{SER} \\
\quad \textbf{IF SND THEN } (\textbf{DEF FST SER } \text{A}) \\
\qquad\qquad \textbf{ELSE } (\textbf{DEF FST SER } \text{B})
\end{array}
\tag{4.1}
$$

Die Bedingungsfunktion des neuen **IF**-Terms entspricht der einfachen Funktion **SND**, welche aus der Ausgabe des vorangehenden **DEF**-Terms das Ergebnis der Bedingungsberechnung auswählt.

In den beiden Zweigen des neuen **IF**-Ausdrucks wird durch die Terme **DEF FST** der neu eingeführte Bedingungswert wieder fallengelassen und anschließend, wie im Ausgangsterm, einer der beiden ursprünglichen **IF**-Zweige ausgeführt.

Verschieben der Bedingungsberechnung von WHILE-Ausdrücken

Das Verschieben der Bedingungsberechnung von **WHILE**-Termen wird mit Hilfe des Theorems 4.2 realisiert. Ausgangspunkt ist hier ein Term der Form **WHILE** cnd **DO** B.

$$
\begin{array}{l}
\vdash \forall \text{ cnd B.} \\
\quad \textbf{WHILE } \text{cnd } \textbf{DO } \text{B} \\
= \textbf{DEF } (\lambda \text{x.(x,}\textbf{T}\text{)}) \\
\quad \textbf{SER} \\
\quad \textbf{WHILE SND} \\
\qquad \textbf{DO } (\textbf{ DEF } (\lambda \text{(x,c).(x,cnd x)}) \\
\qquad\quad \textbf{SER} \\
\qquad\quad \textbf{IF SND THEN } (\textbf{DEF FST SER } \text{B } \textbf{SER DEF } (\lambda \text{x.(x,}\textbf{T}\text{)})) \\
\qquad\qquad\quad \textbf{ELSE } (\textbf{DEF } (\lambda \text{(x,c).(x,}\textbf{F}\text{)}))) \\
\quad \textbf{SER} \\
\quad \textbf{DEF FST}
\end{array}
\tag{4.2}
$$

In dem aus dieser Transformation resultierenden Term erweitert zunächst ein **DEF**-Term die Eingabe um den booleschen Wert **T**, also *wahr*. Die Bedingungsfunktion der nachfolgenden **WHILE**-Schleife entspricht der Funktion **SND**, welche den neu eingeführten booleschen Wert selektiert. Da dieser Wert anfangs immer *wahr* ist, wird der Rumpf der neuen **WHILE**-Schleife grundsätzlich mindestens einmal ausgeführt. Nach der ersten Iteration der Schleife entspricht der

Wert der booleschen Hilfsvariable stets dem Ergebnis der letzten Berechnung der ursprünglichen Schleifenbedingung cnd.

Im Zuge der Transformation wird die Bedingungsfunktion cnd in den Rumpf der **WHILE**-Schleife verschoben. Sie wird am Anfang des Rumpfes in einem separaten **DEF**-Term ausgewertet. Das Ergebnis dieser Bedingungsberechnung wird zusätzlich zu dem aktuellen Stand der Schleifenberechnung an den nachfolgenden **IF**-Ausdruck weitergereicht. Der **IF**-Ausdruck selektiert in seinem Bedingungsblock mit Hilfe der Funktion **SND** das Ergebnis der Bedingungsberechnung und entscheidet in dessen Abhängigkeit, ob der ursprüngliche Schleifenrumpf B auszuführen ist oder nicht.

Ist das Ergebnis der Bedingungsberechnung *wahr*, so wird in dem **THEN**-Zweig des **IF**-Ausdrucks zunächst der zusätzlich eingeführte Bedingungswert durch den Ausdruck **DEF FST** fallengelassen und der Rumpf B der ursprünglichen Schleife ausgeführt. Das Ergebnis dieser Berechnung wird anschließend durch den Ausdruck **DEF** (λx.(x,**T**)) wieder um den zuletzt gültigen Wert der Schleifenbedingung (*wahr*) ergänzt, wodurch eine weitere Iteration der **WHILE**-Schleife bewirkt wird.

Ergab die Bedingungsberechnung im **DEF**-Term am Anfang des Schleifenrumpfes den Wert *falsch*, so kommt der **ELSE**-Zweig des **IF**-Ausdrucks zur Ausführung. In dem dort enthaltenen **DEF**-Term gibt die Funktion (λ(x,c).(x,**F**)) den aktuellen Stand der Berechnung sowie das letzte Ergebnis der Bedingungsberechnung, nämlich den Wert *falsch*, unverändert weiter und bewirkt so den Abbruch der **WHILE**-Schleife.

In einem der **WHILE**-Schleife nachfolgenden **DEF**-Term wird schließlich der Wert der letzten Bedingungsberechnung fallen gelassen und das Endergebnis der Schleifenberechnung ausgegeben.

4.2.3 Maximierung der Grundblöcke

Nach dem Herausziehen der Bedingungsberechnungen von **IF**- und **WHILE**-Ausdrücken wird in der Regel eine Maximierung der in der Verhaltensbeschreibung enthaltenen Grundblöcke durchgeführt. Nach [AhSU86] entspricht ein Grundblock (*basic block*) einer größtmöglichen Folge von Anweisungen, von der jede Anweisung – außer der ersten – genau einen Vorgänger und jede Anweisung – außer der letzten – genau einen Nachfolger hat. Etwas abweichend davon wird in GROPIUS jeder einzelne **DEF**-Term als Grundblock angesehen.

Eine Maximierung der Grundblöcke führt zu einer Vereinfachung der Spezifikation und erhöht gleichzeitig deren Optimierungspotential. Nach Durchführung dieser Transformation können beispielsweise mit der im nächsten Abschnitt beschriebenen Methode auch gemeinsame Teilausdrücke zusammengefasst werden, die ursprünglich in zwei durch das Konstrukt **SER** getrennten **DEF**-Termen auftraten.

Sind zwei **DEF**-Terme mit dem Konstrukt **SER** verknüpft, so können sie zu einem einzigen **DEF**-Term zusammengefasst werden. Fasst man alle durch **SER**-Konstrukte verknüpfte **DEF**-Terme zusammen, so erreicht man eine Maximierung der Grundblöcke der GROPIUS-Spezifikation. Es gilt:

$$\vdash \textbf{DEF}\ a\ \textbf{SER DEF}\ b = \textbf{DEF}\ (b \circ a) \tag{4.3}$$

Der Operator $\circ$ steht für die Komposition zweier DFG-Terme: $(f \circ g) := (\lambda x.f(g\ x))$. Oft ist ein Zusammenfassen entsprechender **DEF**-Terme erst nach einer Änderung der Klammerung möglich, wobei von der Assoziatitvät des **SER**-Konstruktes Gebrauch gemacht wird:

$$\vdash A\ \textbf{SER}\ (B\ \textbf{SER}\ C) = (A\ \textbf{SER}\ B)\ \textbf{SER}\ C \tag{4.4}$$

Der aus dem Zusammenfassen zweier aufeinanderfolgender **DEF**-Terme resultierende DFG-Term $(b \circ a)$ gehört (nur) der allgemeinsten Klasse der DFG-Terme an (siehe Abschnitt 3.3.1). Setzen nachfolgende Transformationen die Form eines FDFG- oder KFDFG-Terms voraus, so muss – auch wenn die beiden Quell-DFG-Terme a und b bereits als FDFG- oder KFDFG-Terme vorliegen – der aus der Verschmelzung resultierende DFG-Term speziell in die geforderte Form überführt werden.

4.2.4 Zusammenfassen gemeinsamer Teilausdrücke

Bei der Transformation *Zusammenfassen gemeinsamer Teilausdrücke* (*common subexpression elimination*) werden Berechnungen, die wiederholt innerhalb eines DFG-Terms auftreten, auf deren einmalige Durchführung reduziert und das Ergebnis dieser Berechnung entsprechend mehrfach verwendet. Auf diese Weise können Operationen eingespart werden und dadurch gegebenenfalls die Kosten für die resultierende Schaltung reduziert und deren Geschwindigkeit erhöht werden.

Die Transformation wird auf jedem DFG-Term einer Spezifikation ausgeführt. Wurden bereits die in Abschnitt 4.2.2 beschriebenen Transformationen zum Verschieben der Bedingungsberechnungen durchgeführt, so finden in den Bedingungsblöcken der **IF**- und **WHILE**-Terme keine Berechnungen mehr statt und das Zusammenfassen gemeinsamer Teilausdrücke beschränkt sich auf die verbleibenden DFG-Terme in den **DEF**-Termen der Spezifikation. Diese werden zur Vorbereitung der Transformation zunächst in flache DFG-Terme umgewandelt (siehe Abschnitt 3.3.1), so dass jeder **let**-Ausdruck maximal einen Operator enthält. Zusätzlich werden alle **let**-Terme β-reduziert, auf deren rechter Seite der Zuweisung keine Operation, sondern nur eine Variable oder Konstante steht. Anschließend werden die **let**-Ausdrücke eines jeden DFG-Terms gemäß dem folgenden Verfahren von oben nach unten abgearbeitet:

Jeder **let**-Ausdruck wird daraufhin untersucht, ob die zu dem entsprechenden **let**-Term (**let** vcs = ausdr **in** R) gehörige Berechnung ausdr im Rest R des entsprechenden DFG-Terms nochmals durchgeführt wird. Ist dies der Fall, so werden alle weiteren Vorkommen im Rest des Terms durch die Variable vcs des **let**-Ausdrucks, welcher das Ergebnis der ersten entsprechenden Berechnung zugewiesen wird, ersetzt. Da es sich um einen flachen DFG-Term handelt, steht auf der rechten Seite der Zuweisung der von der Ersetzung betroffenen **let**-Ausdrücke nur noch die Variable vcs. In diesen **let**-Termen werden somit keine Berechnungen mehr durchgeführt. Sie werden vor der Fortsetzung des Verfahrens β-reduziert und dadurch eliminiert. Das Verfahren wird anschließend auf dem nächsten **let**-Ausdruck, dem ersten im Rest R des DFG-Terms, fortgesetzt, bis alle **let**-Ausdrücke des DFG-Terms abgearbeitet sind.

Im folgenden Beispiel wird bei der Untersuchung des ersten **let**-Terms festgestellt, dass der Ausdruck a $-$ 2 in dem Rest des DFG-Terms (mit der unveränderten Variablen a) nochmals berechnet wird.

```
DEF ( λ(a, b).
        let r = a − 2 in
        let x = r + b in
        let y = a − 2 in
        let r = x ∗ 3 in
        let s = a − 2 in
        let x = r − b in
        let z = s + b in (x, y, z) )
```

Alle weiteren Vorkommen des Ausdrucks werden durch die Variable r ersetzt, welche nach dem ersten **let**-Term das Ergebnis des Ausdrucks a $-$ 2 enthält. Die Substitution ergibt:

```
DEF ( λ(a, b).
        let r = a − 2 in
        let x = r + b in
        let y = r in
        let r′ = x ∗ 3 in
        let s = r in
        let x = r′ − b in
        let z = s + b in (x, y, z) )
```

Man beachte, dass bei dieser Transformation eventuell entstehende Namenskonflikte automatisch durch eine Umbenennung der betroffenen Variablen aufgelöst werden. Im Beispiel wurde die Variable r, welche in dem **let**-Ausdruck **let** r = x $*$ 3 verwendet wurde, in r′ umbenannt. Ohne diese Umbenennung wäre die Transformation nicht korrekt, da durch den **let**-Ausdruck **let** s = r die falsche Variable r referenziert würde.

Im Anschluss an das Zusammenfassen eines gemeinsamen Teilausdrucks werden alle **let**-Terme β-reduziert, deren Berechnung ursprünglich diesem gemeinsamen Teilausdruck entsprach. Im Beispiel erhält man:

```
DEF ( λ(a, b).
        let r = a − 2 in
        let x = r + b in
        let r′ = x ∗ 3 in
        let x = r′ − b in
        let z = r + b in (x, r, z) )
```

Dieser Schritt vereinfacht die Spezifikation und deckt gegebenenfalls weitere gemeinsame Teilausdrücke auf. So wird im Beispiel als Folge diesen Schritts ersichtlich, dass die Operation r + b ebenfalls mehrfach in dem DFG-Term berechnet wird. Dieser gemeinsame Teilausdruck wird im nächsten Schritt des Verfahrens, in welchem der nachfolgende **let**-Ausdruck **let** x = r + b zur Bearbeitung kommt, zusammengefasst.

Die vollständige Eliminierung aller gemeinsamen Teilausdrücke ergibt im Beispiel den folgenden DFG-Term:

```
DEF ( λ(a, b).
        let r = a − 2 in
        let x = r + b in
        let r′ = x ∗ 3 in
        let x′ = r′ − b in (x′, r, x) )
```

Nach der Anwendung des Verfahrens auf alle DFG-Terme der Spezifikation sind alle gemeinsamen Teilausdrücke innerhalb eines jeden DFG-Terms zusammengefasst. Zusätzlich enthält jeder **let**-Ausdruck der Spezifikation genau einen Operator. Letztere Eigenschaft ist für die nachfolgende Analyse des Steuer- und Datenflusses der Spezifikation erforderlich.

4.3 Steuer-/Datenflussanalyse der Verhaltensspezifikation

In diesem Schritt wird der Steuer- und Datenfluss der Verhaltensspezifikation analysiert. Ziel der Analyse ist eine möglichst universell verwendbare Darstellung aller in der Spezifikation enthaltenen Operationen sowie deren Steuer- und Datenflussabhängigkeiten in einer Tabelle. Auf Basis dieser Tabelle erzeugen anschließend Konverter Schaltungsrepräsentationen, welche sich als Eingabe für die außerhalb des Theorembeweisers anzuwendenden Optimierungs- oder Syntheseverfahren eignen (siehe Abschnitt 4.4). Um die Operationen der Spezifikation eindeutig referenzieren zu können, wird zunächst jeder Operation ein eindeutiger Bezeichner zugewiesen.

4.3.1 Benennung der Operationen

Ausgangspunkt diesen Schrittes ist eine Verhaltensspezifikation, welche bereits entsprechend der in Abschnitt 4.2 beschriebenen Maßnahmen auf die algorithmische Synthese vorbereitet wurde. Infolgedessen sind alle in der Spezifikation auftretenden DFG-Terme flach und jeder Ausdruck ausdr in jedem **let**-Term **let** var = ausdr enthält genau einen Operator. Es existiert somit für jede Operation genau eine Variable, der das Ergebnis der Operation zugewiesen wird. Durch eine eindeutige Benennung dieser Variablen können die einzelnen Operationen der Spezifikation über den Namen der jeweils zugehörigen Variable eindeutig referenziert werden. Die eindeutige Referenzierbarkeit der Operationen der GROPIUS-Spezifikation ist Grundvoraussetzung für eine Zuordenbarkeit der entsprechenden Operationen der externen Schaltungsrepräsentation.

In dem Namen der Operation werden alle für das extern anzuwendende Verfahren relevanten Eigenschaften der Operation kodiert. Im Folgenden enthält die Identifikation einer Operation zumeist nur deren Name und die Nummer ihrer Instanz. So wird zum Beispiel das zweite Auftreten der Operation + mit dem Namen „ADD2“ versehen. Oft ist auch der Typ oder die Bitbreite einer Operation von Interesse. So könnte beispielsweise der Name „ADD_32_i_1“ auf die erste 32-Bit-Integer-Addition der Spezifikation verweisen. Die Integration von Informationen, welche über die einzelne Operation hinausgehen, ist ebenfalls denkbar. Der Ablaufplanungsalgorithmus *Wavesched* benötigt zum Beispiel [LaRJ99] die Nummer und Schachtelungstiefe der Schleife, in der eine Operation ausgeführt wird.

Abbildung 4.2 zeigt als Beispiel die Spezifikation aus Abbildung 3.16 auf Seite 47 nach der Durchführung der im letzten Abschnitt aufgeführten Vorbereitungsmaßnahmen zur algorithmischen Synthese und der in diesem Abschnitt beschriebenen Benennung der Operationen. Die Bedingungsberechnungen wurden aus den Bedingungsblöcken der **IF**- und **WHILE**-Ausdrücke herausgezogen, alle Grundblöcke maximiert und den einzelnen Operationen eindeutige Namen zugewiesen.

```
DEF (λ(u, n). ((u, 1, n), T))
SER
(WHILE (λ((u, v, n), GT1).GT1) DO
        ( DEF (λ((u, v, n), GT1).
                let GT1' = n> 0 in ((u, v, n), GT1'))
          SER
          IF (λ((u, v, n), GT1).GT1)
             THEN (DEF (λ((u, v, n), GT1).
                          let ODD1 = ODD n in ((u, v, n), ODD1))
                   SER
                   IF (λ((u, v, n), ODD1).ODD1)
                      THEN (DEF (λ((u, v, n), ODD1).
                                   let MULT1 = u∗v in
                                   let SUB1  = n−1 in
                                   ((u, MULT1, SUB1),T)))
                      ELSE (DEF (λ((u, v, n), ODD1).
                                   let MULT2 = u∗u in
                                   let DIV1  = n DIV 2 in
                                   ((MULT2,v,DIV1),T))))
             ELSE (DEF (λ((u, v, n), GT1).((u, v, n), F))) ))
SER
DEF (λ((u, v, n), GT1).v)
```

Abbildung 4.2: Beispielspezifikation nach den Vorbereitungen zur Flussanalyse

4.3.2 Analyse des Steuerflusses

Nach der eindeutigen Kennzeichnung der Operationen über ihre zugehörigen Variablen kann der Steuerfluss der Spezifikation analysiert werden. Ergebnis dieser Analyse ist eine Relation, welche die Steuerflussabhängigkeiten zwischen den Operationen der Spezifikation beschreibt.

Da manche Optimierungs- und Syntheseverfahren als Eingabe den Steuerflussgraphen einer Spezifikation benötigen, ist für deren Einsatz die Einordnung aller Operationen der Spezifikation in eine bestimmte Reihenfolge erforderlich. Im Zuge dessen wird auch Operationen, die eigentlich steuer- sowie datenunabhängig sind und somit in beliebiger Reihenfolge ausgeführt werden

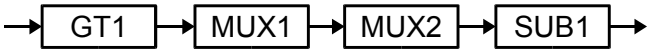
könnten, eine feste Ausführungsreihenfolge zugeordnet. Es hängt von dem jeweiligen Syntheseverfahren ab, ob diese zusätzlichen Steuerflussinformationen verwendet werden oder nicht. Anhand der Spezifikation in Abbildung 4.3 wird im Folgenden erläutert, wie der Steuerfluss innerhalb von GROPIUS-Spezifikationen für den Aufbau einer Steuerflussrelation interpretiert wird.

```
DEF ( λ(i1, i2, i3, i4, i5). let GT1  = i2 > i3 in
                             let MUX1 = MUX(GT1, i2, i3) in
                             let MUX2 = MUX(GT1, i5, i2) in
                             let SUB1  = i3 − i1 in (GT1, SUB1, i2, MUX2) )
SER
IF (λ(ifa, ifb, ifc, ifd). ifa)
      THEN (DEF ( λ(ifa, ifb, ifc, ifd). let ADD1 = ifb + ifc in (ifa, ADD1, ifd)) )
      ELSE (DEF ( λ(ifa, ifb, ifc, ifd). let SUB2  = ifb − ifc in (ifa, SUB2, ifd)) )
SER
DEF  ( λ(da, db, dc). let MULT1 = db ∗ db in (MULT1, dc) )
SER
( DEF ( λ(pa, pb). let ADD2 = pa + pb in ADD2 )
 PAR
 DEF ( λ(pa, pb). let SUB3  = pa − 7 in SUB3 )
SER
DEF ( λ(ea, eb). let LT1 = ea < 128 in (ea, eb, LT1) ) )
SER
WHILE (λ(wa, wb, wc). wc) DO
       ( DEF ( λ(wa, wb, wc).
                 let ADD3 = wa + 2 in
                 let SUB4  = wb − 3 in
                 let LT2   = ADD3 < 312 in (ADD3, SUB4, LT2) ) )
SER
DEF ( λ(fa, fb, fc).
       let ADD4 = fa + 10 in ADD4 )
```

Abbildung 4.3: Beispiel zur Analyse des Steuer- und Datenflusses

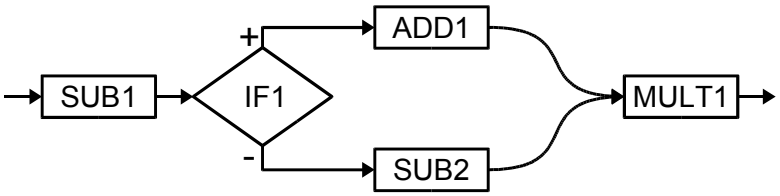
Um den Steuerfluss der Spezifikation in der resultierenden Relation vollständig darstellen zu können, werden zusätzlich zu den bestehenden Operationen sechs spezielle Operationsnamen verwendet: START und ENDE für den Anfang und das Ende der Spezifikation, IF und WHILE für die Verzweigungsoperationen und PAR und PARE für das **PAR**-Konstrukt. Um **IF**-, **PAR**- und **WHILE**-Konstrukte eindeutig referenzieren zu können, wird wie bei den anderen Operationen der spezielle Operationsname jeweils um die Nummer der Instanz des entsprechenden Konstrukts ergänzt. So kennzeichnet beispielsweise eine Operation PAR3 in der sich ergebenden Relation den Beginn des dritten **PAR**-Konstrukts der Spezifikation.

Von den Operationen in den **let**-Ausdrücken von **DEF**-Termen wird angenommen, dass eine Steuerabhängigkeit besteht, die der Reihenfolge ihres Auftretens in den **DEF**-Termen entspricht. Innerhalb des ersten **DEF**-Terms in Abbildung 4.3 ist somit die Operation MUX1 steuerabhängig von GT1, MUX2 von MUX1 und SUB1 von MUX2:



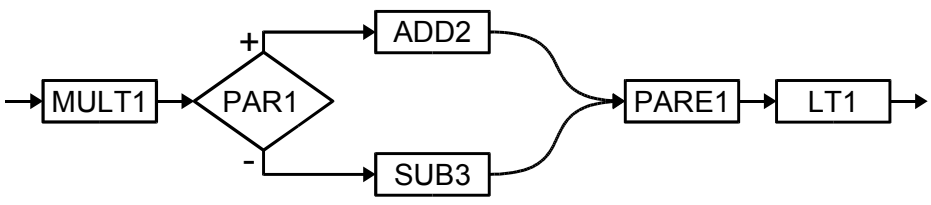
Bei Termen der Form A **SER** B ist die erste Operation in Block B jeweils steuerabhängig von der letzten Operation in Block A. Handelt es sich bei der Anweisung A um einen **IF**- oder **PAR**-Term, so ist die erste Operation in Block B von den jeweils letzten Operationen der beiden Zweige des Vorgängers steuerabhängig.

Die durch ein **IF**-Konstrukt bedingte Verzweigung des Steuerflusses wird durch das Einfügen der speziellen Operation IF gekennzeichnet. Von der IF-Operation sind jeweils die ersten Operationen der beiden Zweige des **IF**-Terms (hier ADD1 und SUB2) steuerabhängig. Durch ein Plus- bzw. Minus-Zeichen wird angezeigt, ob die jeweilige Operation dem **THEN**- oder **ELSE**-Zweig angehört. ADD1 ist somit beispielsweise von +IF1 steuerabhängig, während SUB2 von -IF1 steuerabhängig ist. Die erste Operation der dem **IF**-Term nachfolgenden Anweisung (hier MULT1) ist von den jeweils letzten Operationen der beiden **IF**-Zweige steuerabhängig:

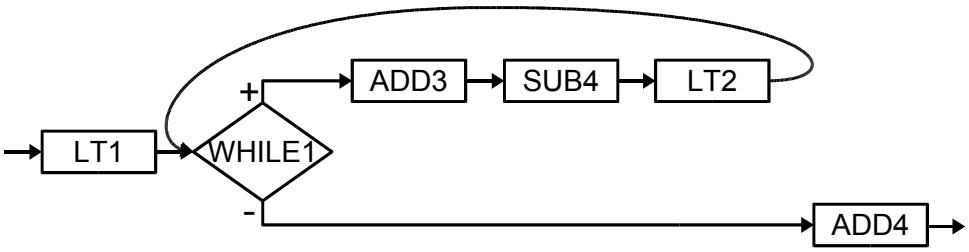


PAR-Terme unterscheiden sich bezüglich des Steuerflusses dadurch von **IF**-Termen, dass stets beide Zweige ausgeführt werden. Die Ausführungsreihenfolge der beiden Zweige ist dabei nicht festgelegt. Als Spezialoperation für **PAR**-Ausdrücke kommen die Bezeichnungen PAR und PARE zum Einsatz. PAR bezeichnet den Beginn einer **PAR**-Anweisung und PARE deren Ende. Von PAR sind jeweils die ersten Operationen der beiden Zweige des **PAR**-Terms steuerabhängig. Hier wird durch ein zusätzliches Plus- bzw. Minus-Zeichen angezeigt, ob die jeweilige Operation dem linken oder rechten Operanden des **PAR**-Konstrukts angehört. Die PARE-Operation kennzeichnet das Ende des **PAR**-Konstrukts. Diese explizite Endmarkierung vereinfacht später ein gegebenenfalls erwünschtes Serialisieren der beiden **PAR**-Zweige (siehe Abschnitt 4.4.1). PARE ist von den jeweils letzten Operationen der beiden Zweige steuerabhängig. Von PARE ist wiederum die erste Operation, welche auf den **PAR**-Term folgt (hier LT1), steuerabhängig.

Der Steuerfluss um die **PAR**-Anweisung gestaltet sich im Beispiel wie folgt:



Für die Beschreibung der Steuerflussabhängigkeiten von **WHILE**-Konstrukten wird die spezielle Operation WHILE verwendet. Diese Operation ist steuerabhängig von der/den letzten Operation/-en der vorangehenden Anweisung (hier LT1) sowie von der/den letzten Operation/-en des Rumpfs der **WHILE**-Schleife (LT2). Von der WHILE-Operation sind wiederum die erste Operation im Schleifenrumpf sowie die erste Operation der nachfolgenden Anweisung steuerabhängig. Durch ein zusätzliches Plus- oder Minus-Zeichen wird angezeigt, ob die jeweils steuerabhängige Operation innerhalb oder außerhalb der Schleife liegt.



Ein WHILE-Knoten hat exakt die gleiche Funktion wie ein IF-Knoten. Beide könnten zu einem *Verzweigungsknoten* zusammengefasst werden. Eine Unterscheidung wurde dennoch vorgenommen, da durch die explizite Kennzeichnung von WHILE-Schleifen eine für die weitere Verarbeitung gegebenenfalls erforderliche Suche nach Zyklen im Steuerfluss erspart werden kann.

Zuletzt werden die beiden Spezialoperationen START und ENDE zur Kennzeichnung des Start- und Endpunkts der Spezifikation eingefügt. Die erste Operation der ersten Anweisung (hier GT1) ist von START steuerabhängig, während ENDE von der/den letzten Operation/-en der letzten Anweisung steuerabhängig ist:

4.3.3 Analyse des Datenflusses

Parallel zum Steuerfluss wird der Datenfluss der Spezifikation analysiert. Die Datenabhängigkeiten aller in der Spezifikation enthaltener Operationen sowie der neu eingeführten Spezialoperationen werden in diesem Schritt ermittelt.

Die Datenabhängigkeiten einer Operation sind durch die Variablen, welche als Operanden der Operation zum Einsatz kommen, definiert. Eine Operation ist von all denjenigen Operationen datenabhängig, deren Ergebnis in einer der Variablen, die einen Operanden der Operation bilden, enthalten sein kann.

Innerhalb eines DFG-Terms entspricht jede Variable, die als Operand in dem Ausdruck ausdr eines **let**-Terms **let** opname = ausdr Verwendung findet, entweder einer der Eingangsvariablen des DFG-Terms oder der Zielvariable einer innerhalb des DFG-Terms vorangehenden Operation. Im letzteren Fall ist die Operation opname (u. a.) von der entsprechenden vorangehenden Operation datenabhängig. Handelt es sich bei dem Operand um eine der Eingangsvariablen des DFG-Terms, so werden die daraus resultierenden Datenabhängigkeiten ermittelt, indem die vorangehenden Operationen über die Grenzen des DFG-Terms hinweg entlang des Steuerflusses

zurückverfolgt werden. Es wird überprüft, welche dieser Operationen potentiell ihr Ergebnis an die betreffende Eingangsvariable des DFG-Terms übertragen. Finden sich entsprechende Operationen, so ist opname von ihnen datenabhängig. Andernfalls entsprechen die Eingangsvariablen des DFG-Terms globalen Eingabewerten der Spezifikation, so dass opname nicht von weiteren Operationen datenabhängig ist.

Im Beispiel in Abbildung 4.3 hängen die Operation GT1 von den Eingangsvariablen i2 und i3 sowie SUB1 von den Eingangsvariablen i1 und i3 des DFG-Terms ab. Da dieser DFG-Term Bestandteil der ersten Anweisung der Spezifikation ist, entsprechen dessen Eingangsvariablen den globalen Eingabewerten der Spezifikation. Diese werden in der resultierenden Steuer-/Datenflusstabelle 4.1 speziell durch START1, START2, START3, START4 und START5 gekennzeichnet. Die Eingangsvariablen i1, i2 und i3 des DFG-Terms entsprechen somit START1, START2 und START3. Die Operationen MUX1 und MUX2 hängen von globalen Eingabewerten und zusätzlich von dem Ergebnis der Operation GT1 ab. Beide Operationen sind somit von der Operation GT1 datenabhängig.

In **DEF**-Anweisungen definieren die Ausgangsvariablen der DFG-Terme (siehe Abschnitt 3.3.1), welche Ergebnisse an die nächste Anweisung übergeben werden. Durch das Konstrukt **SER** werden die Ausgaben des linken Operanden des **SER**-Ausdrucks direkt als Eingabe an die nachfolgende Anweisung, dessen rechten Operanden, weitergereicht. So entspricht die Ausgabe (GT1, SUB1, i2, MUX2) der ersten **DEF**-Anweisung in Abbildung 4.3 den Eingangsvariablen (ifa, ifb, ifc, ifd) der nachfolgenden **IF**-Anweisung.

DFG-Terme, welche den Bedingungsblock einer **IF**- oder **WHILE**-Anweisung bilden, enthalten nach der in Abschnitt 4.2.2 beschriebenen Transformation keine Berechnungen mehr. Sie selektieren nur noch eine der Eingangsvariablen des DFG-Terms, welche das Ergebnis der zugehörigen Bedingungsberechnung enthält. Entsprechend werden die speziellen Verzweigungsoperationen IF bzw. WHILE von denjenigen Operationen als datenabhängig markiert, deren Ergebnis in der durch den zugehörigen Bedingungsblock selektierten Variable enthalten sein kann. In Abbildung 4.3 wird im Bedingungsblock der **IF**-Anweisung dessen Eingangsvariable ifa selektiert. Diese entspricht der Ausgabe GT1 der vorangehenden Anweisung und somit dem Ergebnis der Operation GT1. Infolgedessen ist die Spezialoperation IF1 von GT1 datenabhängig.

Eine **IF**-Anweisung führt in Abhängigkeit ihrer Bedingung jeweils einen ihrer Zweige aus. Eine nachfolgende Anweisung ist gegebenenfalls von Operationen in beiden Zweigen abhängig. So ist die Operation MULT1 in dem der **IF**-Anweisung nachfolgenden **DEF**-Term sowohl von der Operation ADD1 im **THEN**-Zweig der **IF**-Anweisung wie auch von der Operation SUB2 in deren **ELSE**-Zweig datenabhängig. Das Ergebnis beider Operationen wird in der Variable db weitergereicht, welche als Operand der Operation MULT1 verwendet wird.

Bei **PAR**-Anweisungen werden die Ausgaben ihrer beiden Zweige parallel an nachfolgende Anweisungen übergeben. Die Spezialoperationen PAR und PARE selbst befinden sich in keiner Datenabhängigkeit.

Wie bereits erwähnt, ist bei **WHILE**-Anweisungen die Spezialoperation WHILE von all denjenigen Operationen datenabhängig, deren Ergebnisse in der Bedingungsvariable enthalten sein können, welche in dem Bedingungsblock der Schleife selektiert wird. Der Ursprung des Datenflusses einer **WHILE**-Anweisung ist während der ersten Iteration die Ausgabe der vorangehenden Anweisung und nach der ersten Iteration jeweils die Ausgabe des Schleifenrumpfes. So ist in dem

Beispiel in Abbildung 4.3 die Spezialoperation WHILE sowohl von der Operation LT1 der **`DEF`**-Anweisung vor der Schleife als auch von der Operation LT2 am Ende des Rumpfes der Schleife datenabhängig. Entsprechend sind die Operationen im Schleifenrumpf während der ersten Iteration von der Ausgabe der der Schleife vorangehenden Anweisung und danach von der Ausgabe des Schleifenrumpfes selbst datenabhängig. Beispielsweise ist ADD3 sowohl von der Operation ADD2 als auch von sich selbst datenabhängig.

Um die globale Ausgabe der Spezifikation zu kennzeichnen, wird die Spezialoperation ENDE als datenabhängig von den Operationen, welche die Ausgaben der letzten Anweisung der Spezifikation erzeugen, deklariert. Im Beispiel ist die Ausgabe der Spezifikation das Ergebnis der Operation ADD4. Die Operation START steht grundsätzlich in keiner Datenabhängigkeit.

Operation	ist steuerabhängig von	ist datenabhängig von
START		
GT1	START	[START2], [START3]
MUX1	GT1	GT1, [START2], [START3]
MUX2	MUX1	GT1, [START5], [START2]
SUB1	MUX2	[START3], [START1]
IF1	SUB1	GT1
ADD1	+IF1	SUB1, [START2]
SUB2	-IF1	SUB1, [START2]
MULT1	ADD1/SUB2	ADD1/SUB2, ADD1/SUB2
PAR1	MULT1	
ADD2	+PAR1	MULT1, MUX2
SUB3	-PAR1	MULT1, [7]
PARE1	ADD2/SUB3	
LT1	PARE1	ADD2, [128]
WHILE1	LT1/LT2	LT1/LT2
ADD3	+WHILE1	ADD2/ADD3, [2]
SUB4	ADD3	SUB3/SUB4, [3]
LT2	SUB4	ADD3, [312]
ADD4	-WHILE1	ADD2/ADD3, [10]
ENDE	ADD4	ADD4

Tabelle 4.1: Steuer- und Datenfluss des Beispiels in Abbildung 4.3

Tabelle 4.1 gibt das Ergebnis der Steuer- und Datenflussanalyse der Spezifikation aus Abbildung 4.3 an. Ist eine Operation von mehreren Operationen steuerabhängig, so sind diese durch Schrägstriche getrennt angegeben. In der rechten Spalte sind die Datenabhängigkeiten dargestellt. Die einzelnen Operanden einer Operation sind durch Kommata getrennt dargestellt. Kann ein Operand das Ergebnis verschiedener Operationen sein, so sind diese durch Schrägstriche getrennt angegeben. Konstanten und globale Eingabewerte sind in eckigen Klammern dargestellt.

4.4 Erzeugung externer Schaltungsrepräsentationen

Nach der Analyse des Steuer- und Datenflusses der Schaltungsspezifikation werden die Analyseergebnisse in Form einer Tabelle an die externe Umgebung übergeben. Dort werden sie von einem Konverter in eine für das extern auszuführende Verfahren passende Form überführt.

Einige Verfahren zur Analyse toten Codes (siehe Abschnitt 5.1) erwarten beispielsweise eine Beschreibung der Schaltung in Form von DU-/UD-Ketten* (*Definition-Use/Use-Definition chains*) [Much97]. Für den Einsatz kommerzieller Werkzeuge wäre gegebenenfalls eine Überführung der Spezifikation in VHDL [VHDL02] denkbar.

Die genaue Form hängt jeweils von dem einzusetzenden Verfahren ab. Oft ist eine bestimmte Darstellungsform allerdings für mehrere Verfahren geeignet, vor allem wenn diese einen ähnlichen Optimierungs- oder Syntheseschritt realisieren. In diesem Fall können die Verfahren auf einen gemeinsamen Konverter zurückgreifen.

Einer der Konverter, die im Zuge dieser Arbeit entwickelt wurden, erzeugt auf Basis der Steuer-/Datenflusstabelle einer Spezifikation ihren entsprechenden Steuer-/Datenflussgraphen (*control/data flow graph, CDFG*) [DeMi94, GaRa94]. Diese Darstellungsform ist in der algorithmischen Synthese besonders verbreitet. Zum Beispiel basieren zahlreiche Algorithmen zur Ablaufplanung auf CDFGs [RaJe95]. Einer der bekanntesten dieser Algorithmen ist der *Path-Based-Scheduling*-Algorithmus [Camp91]. Ein anderer Vertreter dieser Algorithmen, der *Loop-Directed-Scheduling*-Algorithmus [BhDB94], wurde exemplarisch in dieser Arbeit implementiert. Auf Grund der vielseitigen Anwendungsbereiche von CDFGs soll deren Konstruktion aus einer gegebenen Steuer-/Datenflusstabelle im Folgenden näher betrachtet werden.

4.4.1 Darstellung der Spezifikation als Steuer-/Datenflussgraph

Steuer-/Datenflussgraphen sind gerichtete Graphen, deren Knoten Operationen entsprechen und deren Kanten den Steuer- und Datenfluss zwischen den Knoten beschreiben. Die Kanten sind in zwei Klassen unterteilt: in die Klasse der Steuerflusskanten und die Klasse der Datenflusskanten. Ebenso wie die Kanten eines CDFGs sind auch dessen Knoten in zwei Klassen eingeteilt. Die eine Klasse der Knoten entspricht den speziellen Operationen zur Beschreibung des Steuerflusses (START, ENDE, IF, WHILE, PAR und PARE) und die andere Klasse den in der ursprünglichen Spezifikation enthaltenen Operationen. Unterscheidet das anzuwendende Verfahren nicht zwischen WHILE- und IF-Knoten, so können beide Knotentypen durch einen einheitlichen Verzweigungsknoten dargestellt werden.

Abbildung 4.4 stellt einen Steuer-/Datenflussgraphen dar, der exakt das in Tabelle 4.1 dargestellte Ergebnis der Flussanalyse der Beispielspezifikation aus Abbildung 4.3 widerspiegelt. In der Abbildung sind Steuerflusskanten mittels durchgängiger Pfeile dargestellt, während Datenflusskanten durch gestrichelte Pfeile dargestellt werden.

*Die UD-Kette einer Variablen gibt die Operationen an, deren Ergebnis sie potentiell enthält, und deren DU-Kette die Operationen, die diese Variable potentiell verwenden.

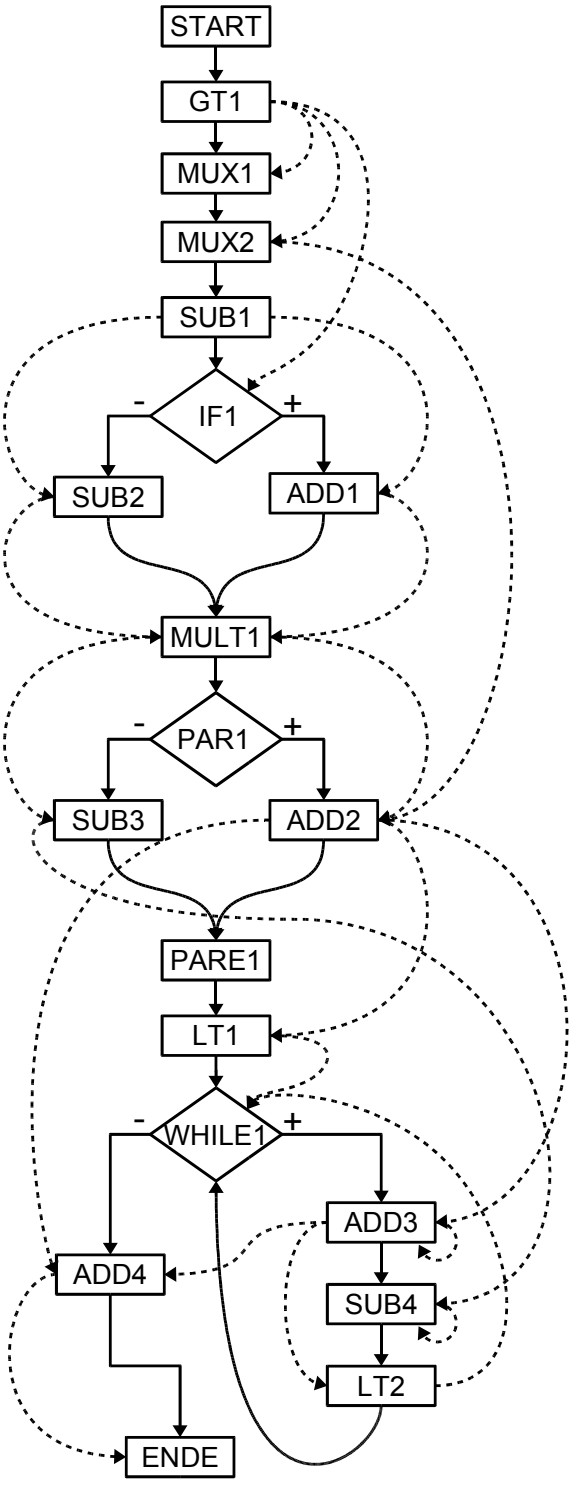


Abbildung 4.4: Steuer-/Datenflussgraph der Spezifikation in Abbildung 4.3

Eliminierung von PAR-Knoten

Oft sind Verfahren, die auf CDFGs operieren, nicht auf PAR-Knoten als Bestandteil der Graphen ausgelegt. In diesen Fällen werden die PAR-Knoten vor dem Einsatz entsprechender Algorithmen eliminiert, indem die Ausführungsreihenfolge der beiden Zweige jeder **PAR**-Anweisung im Voraus festgelegt wird. Dabei kann für jedes **PAR**-Konstrukt separat entschieden werden, welcher der beiden Zweige zuerst ausgeführt werden soll.

Ein einzelner PAR-Knoten wird aus einem CDFG eliminiert, indem zunächst die Steuerflusskante/-n, welche auf den entsprechenden PAR-Knoten verweist/-en, auf die erste Operation desjenigen Zweigs umgelenkt werden, der zuerst ausgeführt werden soll. Die von der letzten Operation dieses Zweigs auf den zugehörigen PARE-Knoten verweisende Steuerflusskante wird auf die erste Operation des anderen **PAR**-Zweiges umgelenkt, so dass der andere Zweig direkt im Anschluss an den ersten Zweig der **PAR**-Anweisung zur Ausführung kommt. Die letzte Operation des **PAR**-Zweiges, welcher als Zweiter zur Ausführung kommt, wird auf den Nachfolger des PARE-Knotens umgeleitet und schließlich der PAR- und der PARE-Knoten entfernt.

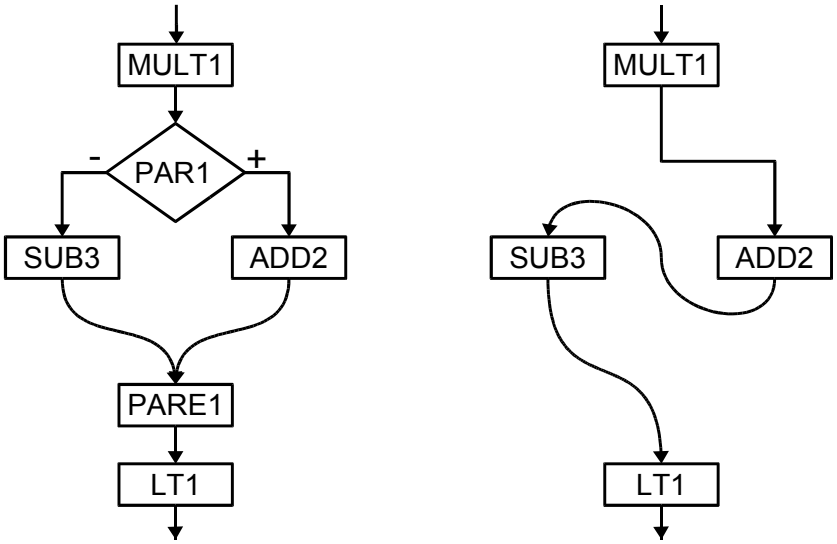


Abbildung 4.5: PAR-Eliminierung im Steuer-/Datenflussgraphen

In Abbildung 4.5 ist die Eliminierung des PAR-Knotens des Steuer-/Datenflussgraphen in Abbildung 4.4 dargestellt. Der mit Plus gekennzeichnete Zweig soll in diesem Fall zuerst zur Ausführung kommen.

Parallel zur Entfernung eines PAR-Knotens aus dem CDFG wird gleichzeitig die entsprechende **PAR**-Anweisungen der GROPIUS-Spezifikation eliminiert. Die beiden Zweige der **PAR**-Anweisung werden mit Hilfe eines der beiden folgenden Theoreme in die gewählte Ausführungsreihenfolge gebracht:

$$\vdash \text{A } \mathbf{PAR} \text{ B} = \mathbf{DEF} \text{ (} \lambda(\text{x. (x, x))) } \mathbf{SER\ LEVA} \text{ A } \mathbf{SER\ RIVA} \text{ B} \quad (4.5)$$

$$\vdash \text{A } \mathbf{PAR} \text{ B} = \mathbf{DEF} \text{ (} \lambda(\text{x. (x, x))) } \mathbf{SER\ RIVA} \text{ B } \mathbf{SER\ LEVA} \text{ A} \quad (4.6)$$

Nach Anwendung des Theorems 4.5 kommt stets der linke Zweig des ursprünglichen **PAR**-Konstrukts zuerst zur Ausführung und nach Anwendung des Theorems 4.6 der rechte. Die neu auftretenden Konstrukte **RIVA** und **LEVA** werden durch die rekursive Anwendung folgender Theoreme eliminiert:

$$
\begin{aligned}
\vdash \textbf{LEVA}\ (\mathrm{A}\ \textbf{SER}\ \mathrm{B}) &= \textbf{LEVA}\ \mathrm{A}\ \textbf{SER}\ \textbf{LEVA}\ \mathrm{B} \\
\vdash \textbf{RIVA}\ (\mathrm{A}\ \textbf{SER}\ \mathrm{B}) &= \textbf{RIVA}\ \mathrm{A}\ \textbf{SER}\ \textbf{RIVA}\ \mathrm{B} \\
\vdash \textbf{LEVA}\ (\textbf{IF}\ \mathrm{c}\ \textbf{THEN}\ \mathrm{A}\ \textbf{ELSE}\ \mathrm{B}) &= \textbf{IF}\ (\lambda(\mathrm{f}, \mathrm{s}).\ \mathrm{c}\ \mathrm{f})\ \textbf{THEN}\ (\textbf{LEVA}\ \mathrm{A})\ \textbf{ELSE}\ (\textbf{LEVA}\ \mathrm{B}) \\
\vdash \textbf{RIVA}\ (\textbf{IF}\ \mathrm{c}\ \textbf{THEN}\ \mathrm{A}\ \textbf{ELSE}\ \mathrm{B}) &= \textbf{IF}\ (\lambda(\mathrm{f}, \mathrm{s}).\ \mathrm{c}\ \mathrm{s})\ \textbf{THEN}\ (\textbf{RIVA}\ \mathrm{A})\ \textbf{ELSE}\ (\textbf{RIVA}\ \mathrm{B}) \\
\vdash \textbf{LEVA}\ (\textbf{WHILE}\ \mathrm{c}\ \textbf{DO}\ \mathrm{A}) &= \textbf{WHILE}\ (\lambda(\mathrm{f}, \mathrm{s}).\ \mathrm{c}\ \mathrm{f})\ \textbf{DO}\ (\textbf{LEVA}\ \mathrm{A}) \\
\vdash \textbf{RIVA}\ (\textbf{WHILE}\ \mathrm{c}\ \textbf{DO}\ \mathrm{A}) &= \textbf{WHILE}\ (\lambda(\mathrm{f}, \mathrm{s}).\ \mathrm{c}\ \mathrm{s})\ \textbf{DO}\ (\textbf{RIVA}\ \mathrm{A}) \\
\vdash \textbf{LEVA}\ (\textbf{DEF}\ \mathrm{a}) &= \textbf{DEF}\ (\lambda(\mathrm{f}, \mathrm{s}).\ (\mathrm{a}\ \mathrm{f}, \mathrm{s})) \\
\vdash \textbf{RIVA}\ (\textbf{DEF}\ \mathrm{a}) &= \textbf{DEF}\ (\lambda(\mathrm{f}, \mathrm{s}).\ (\mathrm{f}, \mathrm{a}\ \mathrm{s}))
\end{aligned}
$$

Abschließend werden die resultierenden Grundblöcke wie in Abschnitt 4.2.3 beschrieben maximiert. Eine Eliminierung des **PAR**-Konstruktes in der Beispielspezifikation aus Abbildung 4.3 entsprechend der Graphentransformation in Abbildung 4.5 ergibt:

```
...
SER
DEF ( λ(da, db, dc). let MULT1 = db ∗ db in
                     let ADD2  = MULT1 + dc in
                     let SUB3  = MULT1 − 7 in
                     let LT1   = ADD2 < 128 in (ADD2, SUB3, LT1) )
SER
...
```

Eliminierung von WHILE-Knoten

Macht ein einzusetzendes CDFG-basiertes Verfahren keinen Gebrauch von der expliziten Kennzeichnung von Schleifen durch WHILE-Knoten, so können diese ebenfalls eliminiert werden. Nach der in Abschnitt 4.2.2 beschriebenen Transformation zum Verschieben der Bedingungsberechnungen von **WHILE**-Konstrukten ist das Sprungverhalten der resultierenden WHILE-Knoten ausschließlich von konstanten Werten abhängig. Durch eine Interpretation dieser konstanten Verzweigungswerte ist es möglich, die WHILE-Knoten aus dem CDFG zu entfernen. Dies soll anhand folgenden Beispiels verdeutlicht werden:

```
DEF ( λx. let MULT1 = x ∗ x in MULT1 )
SER
WHILE (λx. x < 10) DO
        ( DEF ( λx. let ADD1 = x + 3 in ADD1 ) )
SER
DEF ( λx. let MULT2 = x ∗ 4 in MULT2 )
```

Ein Verschieben der Bedingungsberechnung aus dem Bedingungsblock des **WHILE**-Konstrukts (siehe Abschnitt 4.2.2) und eine Maximierung der resultierenden Grundblöcke (Abschnitt 4.2.3) überführt die Spezifikation in folgende Form:

```
DEF ( λx. let MULT1 = x ∗ x in (MULT1, T) )
SER
WHILE (λ(wa, wb). wb) DO
        ( DEF ( λ(wa, wb). let LT1 = wa < 10 in (wa, LT1) )
          SER
          IF (λ(wa, LT1). LT1) THEN ( DEF ( λ(wa, LT1).
                                              let ADD1 = wa + 3 in (ADD1, T) ) )
                                   ELSE ( DEF ( λ(wa, LT1). (wa, F) ) ) )
SER
DEF ( λ(wa, LT1). let MULT2 = wa ∗ 4 in MULT2 )
```

Eine Analyse des Steuer- und Datenflusses dieser Spezifikation ergibt:

Operation	ist steuerabhängig von	ist datenabhängig von
START		
MULT1	START	[START1], [START1]
WHILE1	MULT1 ADD1 -IF1	[**T**]/[**T**]/[**F**]
LT1	+WHILE1	MULT1/ADD1, [10]
IF1	LT1	LT1
ADD1	+IF1	MULT1/ADD1, [3]
MULT2	-WHILE1	MULT1/ADD1, [4]
ENDE	MULT2	MULT2

Der entsprechende Steuerflussgraph ist auf der linken Seite von Abbildung 4.6 dargestellt. Zur Erhöhung der Übersichtlichkeit wurde hier auf die Darstellung der Datenflusskanten verzichtet.

Als Folge des Verschiebens der Bedingungsberechnung des **WHILE**-Konstrukts ist der WHILE-Knoten nur von konstanten Werten datenabhängig: Ist von dem WHILE-Knoten aus gesehen der Ursprung des Steuerflusses der Knoten MULT1 oder ADD1, so ist die Schleifenbedingung (im Graph: wbed) *wahr*. Ist der Vorgänger des WHILE-Knotens dagegen der IF-Knoten, dessen Bedingungswert sich als *falsch* herausgestellt hat, so ist die Schleifenbedingung ebenfalls *falsch*.

Da nach den Knoten MULT1 und ADD1 die Schleifenbedingung grundsätzlich *wahr* ist, kann in diesen Fällen anstatt des WHILE-Knotens auch direkt die erste Operation im Rumpf der Schleife (LT1) angesprungen werden. Entsprechend ist, wenn die Bedingung des **IF**-Konstrukts sich als *falsch* erweist, die Schleifenbedingung immer *falsch*, so dass in diesem Fall stets direkt zu der ersten Operation nach der WHILE-Schleife gesprungen werden kann.

Wird der Steuerfluss entsprechend umgeleitet, so verliert der WHILE-Knoten seine Bedeutung und kann eliminiert werden. Der im Beispiel resultierende Graph ist auf der rechten Seite von Abbildung 4.6 dargestellt.

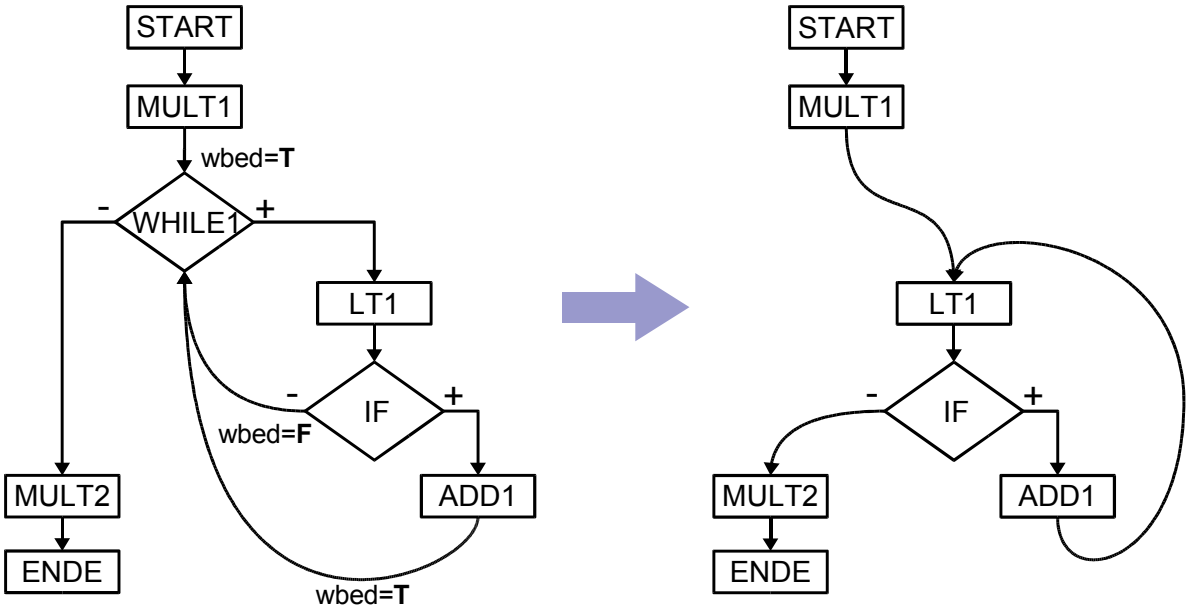


Abbildung 4.6: Eliminierung von WHILE-Knoten

Auf die hier vorgestellte Weise können alle konstanten Verzweigungsbedingungen von WHILE- und IF-Knoten im Voraus ausgewertet und so die Steuerstruktur von CDFGs vereinfacht werden.

In der Verhaltensspezifikation sind die entsprechenden Transformationen der **`WHILE`**-Anweisungen zu diesem Zeitpunkt nicht realisierbar, da Übergänge, wie sie in dem Beispiel infolge der Transformation von der Operation ADD1 zur Operation LT1 stattfinden, in GROPIUS nicht direkt darstellbar sind. Daher werden die entsprechenden Umformungen in der formalen Schaltungsspezifikation erst beim Übergang zu der in Abschnitt 3.3.4 beschriebenen steuerflussorientierten Darstellung, in welcher die Darstellung derartiger Übergänge schließlich möglich ist, nachvollzogen (siehe Abschnitt 5.2.2).

4.5 Ausführung externer Syntheseverfahren

Nach der Konvertierung der Spezifikation in das benötigte Eingabeformat kann schließlich das gewählte Optimierungs- oder Syntheseverfahren zur Ausführung kommen. Welche Verfahren in dem formalen Synthesesystem eingesetzt werden können, hängt davon ab, inwieweit die entsprechenden Module des formalen Systems deren Ergebnisse innerhalb des Theorembeweisers umsetzen können.

Während beispielsweise die Ablaufpläne, die in dem vorgestellten Konzept umgesetzt werden können, und damit die einsetzbaren Ablaufplanungsverfahren, noch gewissen Einschränkungen unterliegen, können zum Beispiel bei der Bereitstellung und Zuweisung der Verbindungs- und Funktionseinheiten (siehe Abschnitte 5.4 und 5.5) weitestgehend alle möglichen (korrek-

ten) Ergebnisse umgesetzt werden. Gegebenenfalls existierende Grenzen der Umsetzbarkeit bestimmter Entwurfsschritte werden ausführlich in Kapitel 5 beschrieben. Da die Entwicklung neuer Optimierungs- und Syntheseheurisitiken nicht Ziel dieser Arbeit war, wird auf die eingesetzten Techniken nicht im Detail eingegangen.

Bei dem Einsatz externer Syntheseverfahren gibt es verschiedene Strategien. Einerseits können nach der Ausführung eines jeden Verfahrens dessen Ergebnisse unverzüglich an das formale Synthesesystem zurückgeliefert werden, um dort sofort umgesetzt zu werden. Auf diese Weise ist es möglich, die Ergebnisse eines Verfahrens umzusetzen, während bereits parallel dazu das nächste externe Optimierungs- oder Syntheseverfahren zur Ausführung kommt. Andererseits bietet es sich aber oft an, mehrere Syntheseverfahren auf einmal extern auszuführen und dann erst deren Ergebnisse an das formale System weiterzuleiten.

Gerade bei den fundamentalen Aufgaben der algorithmischen Synthese – der Erstellung eines Ablaufplans, der Ressourcenbereitstellung und der Ressourcenzuweisung – bietet sich die gemeinsame Ausführung der drei Entwurfsschritte an, da sie stark voneinander abhängig sind. Ein optimales Ergebnis kann oft erst nach deren iterativer Ausführung erzielt werden. Da die Verfahren somit gegebenenfalls mehrfach ausgeführt werden müssen, ist die Effizienz dieser oft aufwändigen Entwurfsraumuntersuchungen von besonderer Bedeutung. Meist sind entsprechende Verfahren bereits hochoptimiert, so dass deren größtmögliche Effizienz dadurch erreicht wird, indem sie in denjenigen Umgebungen ausgeführt werden, für die sie entwickelt und optimiert wurden. Diese Möglichkeit wurde durch das im Zuge dieser Arbeit entwickelte Rahmenwerk geschaffen. Die Verfahren können unverändert in ihrer ursprünglichen Umgebung ausgeführt werden.

Während bis zum Erreichen eines Ergebnisses der gewünschten Qualität einige Verfahren gegebenenfalls mehrfach ausgeführt werden müssen, ist eine Nachbildung der auftretenden Zwischenergebnisse in der formalen Umgebung nicht erforderlich. Erst die endgültigen Ergebnisse eines Entwurfsschritts werden an die formale Umgebung zurückgeliefert. Zuvor müssen sie jedoch mit Hilfe eines weiteren Konverters in eine für das formale Synthesesystem geeignete Form überführt werden.

4.6 Konvertierung der Ergebnisrepräsentation

Die benötigte Darstellungsform der Ergebnisse hängt von dem jeweils durchzuführenden Syntheseschritt ab und wird von der verarbeitenden Einheit in der formalen Umgebung vorgegeben.

Für die Durchführung der Ablaufplanung in dem formalen System wird beispielsweise der extern erzielte Ablaufplan in Form eines Graphen erwartet, welcher – ähnlich wie der Steuer-/Datenflussgraph – als Tabelle an das formale Synthesesystem übertragen wird. Der Ablaufplan ist ein gerichteter Graph, dessen Knoten Zustände und dessen Kanten Übergänge zwischen den Zuständen repräsentieren. Jeder Knoten enthält einen zyklenfreien Steuerflussgraphen bestehend aus denjenigen Operationen, die in dem entsprechenden Zustand in der geplanten RT-Ebenen-Implementierung ausgeführt werden sollen. Die Kanten sind mit den Bedingungen behaftet, unter denen die Übergänge zwischen den RT-Ebenen-Zuständen stattfinden.

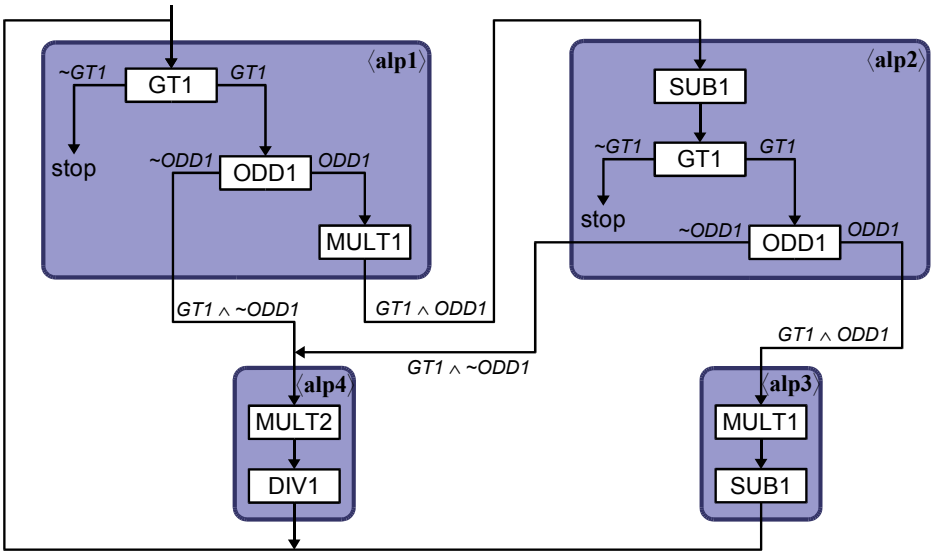


Abbildung 4.7: Ein möglicher Ablaufplan für die Spezifikation in Abbildung 4.2

Abbildung 4.7 stellt einen Ablaufplan für die Spezifikation in Abbildung 4.2 dar. In dieser, wie auch in den folgenden Graphendarstellungen, werden Verzweigungsknoten nicht mehr explizit dargestellt. Stattdessen haben ihre ursprünglich direkt vorangehenden Operationen mehrere ausgehende Kanten, die jeweils mit der maßgeblichen Verzweigungsbedingung behaftet sind. Die einzelnen Verzweigungsbedingungen entsprechen dabei logischen Verknüpfungen von Ausgaben vorangegangener Operationen bzw. globaler Eingabewerte der Spezifikation.

Auf die Eingabedaten der verschiedenen Umsetzungsmodule wird in Kapitel 5 in dem jeweils entsprechenden Zusammenhang näher eingegangen.

4.7 Syntheseschrittspezifische Verarbeitung der Ergebnisse

Liegen die Ergebnisse einer Entwurfsraumuntersuchung dem formalen System schließlich vor, so wird die Spezifikation den Ergebnissen entsprechend innerhalb des Theorembeweisers umgeformt. Dabei wird die Spezifikation im Theorembeweiser durch einzelne Transformationen konstruktiv in die gewünschte Implementierung überführt und nicht, wie es bei der Verifikation nach Durchführung der Synthese (siehe Abschnitt 2.2.1) üblich ist, auf Basis der Analyseergebnisse eine neue Implementierung aufgebaut und nachträglich die Äquivalenz dieser Implementierung zu der Spezifikation bewiesen. Die automatische Umsetzung der einzelnen Syntheseschritte in der formalen Synthese wird ausführlich in Kapitel 5 behandelt.

Kapitel 5

Automatisierung der formalen algorithmischen Synthese

Durch das in Kapitel 4 beschriebene Rahmenwerk wurde eine Möglichkeit zur einfachen Integration und effizienten Durchführung konventioneller Optimierungs- und Syntheseverfahren geschaffen. Die in der externen Umgebung ermittelten Entwurfsziele müssen nun innerhalb des Theorembeweisers umgesetzt werden.

In dieser Arbeit wurde besonders viel Wert auf die vollständige Automatisierung des formalen Syntheseprozesses gelegt. Eine interaktive Steuerung des Entwufsprozesses durch den Entwickler sollte zwar noch möglich, aber nicht erforderlich sein. Eine Automatisierung der formalen Synthese erhöht nicht nur deren Komfort und Akzeptanz, sondern auch deren Geschwindigkeit. Vor allem aber ist der Einsatz einer formalen Synthese nicht mehr auf Entwickler mit ausgeprägten Kenntnissen in formalen Methoden beschränkt. Durch die vollständige Automatisierung wird die Korrektheit des Syntheseergebnisses allein durch das Synthesesystem innerhalb des Theorembeweisers bewiesen. Ein Eingreifen des Entwicklers ist nicht erforderlich.

Voraussetzung für eine durchgängige formale Synthese sind zum einen formale Darstellungsformen, welche die Repräsentation der Spezifikation, der Implementierung sowie aller Zwischenergebnisse während des Syntheseprozesses erlauben, und zum anderen eine ausreichende Menge von Transformationen, die alle für die Realisierung einer Synthese erforderlichen Umformungen abdecken.

Die Automatisierung der Transformationen und der zugehörigen Beweise wird in dieser Arbeit dadurch erreicht, dass der zu transformierende Term, falls er nicht bereits in einer wohldefinierten Form vorliegt, in eine solche überführt wird, so dass stets bekannt ist, an welcher Stelle des Terms welche Information in welcher Form zu erwarten ist. Dies ist zumeist eine Voraussetzung für die Entwicklung von Beweisstrategien, die eine automatisierte Durchführung der im Zusammenhang mit der Transformation stehenden Beweise realisieren sollen. Soweit möglich wurden die Transformationen derart konzipiert, dass deren resultierende Terme bereits eine Form aufweisen, die eine direkte Weiterverarbeitung ohne zusätzliche Umformungen zulässt.

Als Beispiel für entsprechende Umformungen, Beweisstrategien und deren Einsatz folgt in Abschnitt 5.1 eine ausführliche Beschreibung der automatischen Eliminierung toten Codes aus GROPIUS-Spezifikationen. Anschließend wird das im Zuge dieser Arbeit entwickelte Konzept

zur automatisierten Durchführung aller für eine formale algorithmische Synthese erforderlichen Schritte (siehe Abschnitt 2.1) vorgestellt. Die formale algorithmische Synthese beginnt mit der Überführung der Spezifikation in die steuerflussorientierte Darstellung (Abschnitt 5.2). Anschließend wird die Ablaufplanung durchgeführt (Abschnitt 5.3). Nach der Bereitstellung und Zuweisung von Registern (Abschnitt 5.4) wird im Zuge einer Operationswerksynthese (Abschnitt 5.5) sowie der Synthese des Steuerwerks (Abschnitt 5.6) die Bereitstellung und Zuweisung der Funktions- und Verbindungseinheiten umgesetzt.

5.1 Eliminierung toten Codes

Toter Code bezeichnet Operationen und Speicherplätze, die auf das Ergebnis einer Berechnung keinen Einfluss haben [AhSU86, Much97]. Oft wird auch unerreichbarer Code als toter Code bezeichnet. In dieser Arbeit wird jedoch nach [Much97] unter *unerreichbarem Code* Code verstanden, der unabhängig von der Eingabe der Berechnung, niemals zur Ausführung kommt. Im Gegensatz dazu kann toter Code tatsächlich zur Ausführung kommen, hat aber auch in diesem Fall keine Auswirkung auf das Ergebnis einer Berechnung. Für toten Code kommt in dieser Arbeit folgende Definition zum Einsatz [KaSa03]:

Definition 1 (notwendige Variablen und Operationen) *Variablen, die das Endergebnis einer Spezifikation enthalten, sind* notwendig. *Operationen sind notwendig, wenn ihre Ergebnisse notwendigen Variablen zugewiesen werden*[*]. *Variablen, die in notwendigen Operationen verwendet werden, sind ebenfalls notwendig.*

Definition 2 (unnötige Variablen und Operationen, toter Code) *Variablen und Operationen, die nicht notwendig sind, sind* unnötig *und repräsentieren den* toten Code *der Spezifikation.*

Definition 3 (TC-Reduzierbarkeit) *Falls einige der eingehenden Werte eines DFG- oder P-Terms unnötig sind oder der entsprechende Term selbst unnötige Operationen oder Variablen enthält, so heißt dieser Term* TC-reduzierbar *(Toter-Code-reduzierbar). Ansonsten wird der Term als* TC-reduziert *bezeichnet.*

Toter Code tritt meistens infolge anderer Optimierungsverfahren auf, kann jedoch unter Umständen auch (in der Regel unbeabsichtigt) durch den Entwerfer selbst spezifiziert worden sein. Die Eliminierung toten Codes führt oft zu einer Senkung der Kosten der resultierenden Implementierung, da gegebenenfalls einige Funktionseinheiten und auch Speichereinheiten, welche im Zusammenhang mit der Berechnung des toten Codes benötigt werden, eingespart werden können.

Für die folgenden Ausführungen sei daran erinnert, dass in dieser Arbeit Kleinbuchstaben für Variablen elementaren Typs (**a**, **b**, **c**, ...), unterstrichene Buchstaben für Strukturen solcher Variablen (**x**, **vsr**, ...), kleingeschriebene, schräggestellte Buchstaben für ganze DFG-Terme (f, g,

[*]Eine spezielle Stellung nehmen **WHILE**-Schleifen in diesem Zusammenhang ein. Da deren Terminieren im Allgemeinen nicht entscheidbar ist, werden **WHILE**-Operationen in dieser Arbeit grundsätzlich als notwendig erachtet und somit nie eliminiert (siehe auch Seite 98).

cnd, ...) und schräggestellte Großbuchstaben für P-Terme (***L***, ***R***, ***T***, ...) stehen (siehe Abschnitt 3.3.4). Nicht fett dargestellte Bezeichner repräsentieren Variablen beliebigen Typs.

Weiter bezeichnet $\lceil t \rceil$ die Menge aller freien Variablen des Terms t (siehe Abschnitt 3.1). Es gilt also beispielsweise $\lceil \lambda x.(f\ (x,y)) \rceil = y$. Eine beliebige Variablenstruktur bestehend aus einer Variablenmenge `vs` wird durch $\langle$`vs`$\rangle$ dargestellt. So kann zum Beispiel $\langle \{a,b,c\} \rangle$ für die Variablenstrukturen (a,b,c), (b,a,c), ((c,b),a), etc. stehen.

Im Zusammenhang mit der Eliminierung toten Codes kommt eine besondere Variante der DFG-Terme zum Einsatz, die sogenannte *Variableneliminierung*. Eine Variableneliminierung (λ**ev**.**av**) zeichnet sich dadurch aus, dass sie keinerlei Operationen enthält und weniger Aus- als Eingangsvariablen hat, d. h. $\lceil$**av**$\rceil \subset \lceil$**ev**$\rceil$. Als selbständiger Bestandteil eines P-Terms tritt sie entsprechend innerhalb eines **`DEF`**-Terms auf, also in der Form **`DEF`**(λ**ev**.**av**). Als Beispiel für eine Variableneliminierung sei der Term **`DEF`** (λ(d,s3,b).(s3,b)) gegeben. Er bewirkt die Eliminierung der Eingangsvariablen d.

5.1.1 Durchführung der Eliminierung toten Codes

Das Verfahren zur Eliminierung toten Codes bearbeitet jeweils eine Anweisung der Spezifikation nach der anderen (zum Begriff der *Anweisung* in GROPIUS siehe Abschnitt 3.3.3). Es beginnt mit der letzten Anweisung der Spezifikation und endet mit der ersten.

Jede Anweisung wird nach folgendem Prinzip abgearbeitet: Falls der unter Betracht stehenden Anweisung eine Variableneliminierung folgt, so wird diese als Erstes in die Anweisung integriert. Anschließend wird der in der Anweisung enthaltene tote Code eliminiert. Erweisen sich nach diesem Schritt einige der Eingänge der Anweisung als unnötig, so wird die Anweisung aufgeteilt in eine neue Variableneliminierung, gefolgt von der vollständig TC-reduzierten Form der Anweisung. Diese Variableneliminierung wird im darauf folgenden Schritt in den Vorgänger der zuletzt betrachteten Anweisung integriert und dort entsprechend die Eliminierung toten Codes fortgesetzt. Das Verfahren endet nach der Abarbeitung der ersten Anweisung der Spezifikation.

Im Folgenden wird die Eliminierung toten Codes ausführlich für jedes Grundkonstrukt vorgestellt, das nach den in Abschnitt 4.2 beschrieben Vorbereitungsmaßnahmen der algorithmischen Synthese noch eine Anweisung der Spezifikation bilden kann.

Eliminierung toten Codes in `DEF`-Termen

Gegeben sei folgendes Beispiel eines **`DEF`**-Terms gefolgt von einer Variableneliminierung[†]:

```
DEF (λ(a,b,c,d).let s1 = b>c          in
                let s2 = MUX(s1,b,c) in
                let s3 = c−a         in (d,s3,b))
SER
DEF (λ(d,s3,b).(s3,b))                (* Variableneliminierung *)
```

[†]In HOL werden Kommentare durch eine Klammerung in „(*“ und „*)“ gekennzeichnet.

Zuerst wird die Variableneliminierung in den **DEF**-Term eingebunden. Dies erfolgt unter Verwendung des Theorems 4.3 auf Seite 72 nach folgendem Schema:

DEF (λ**ev**.**let** **av** = (***body*** **ev**) **in** **av**)
SER $\Longrightarrow$ **DEF** (λ**ev**.**let** **av** = (***body*** **ev**) **in** **nv**)
DEF (λ**av**.**nv**) (* Variableneliminierung *)

Im Beispiel führt die Integration der Variableneliminierung zur Entfernung der Ausgangsvariablen d:

DEF (λ(a,b,c,d).**let** s1 = b>c **in**
let s2 = **MUX**(s1,b,c) **in**
let s3 = c−a **in** (s3,b))

Nun wird von unten nach oben jeder der **let**-Ausdrücke (**let** var = ausdr **in** R) einzeln abgearbeitet. Es wird überprüft, ob die Variable var des **let**-Ausdrucks im Rest R des DFG-Terms benötigt wird. Dies ist genau dann der Fall, wenn var als freie Variable im Rest des Terms R vorkommt, also falls var $\in \lceil R \rceil$. Kommt var im Rest des Terms R nicht frei vor, so ist die Variable (und damit auch der zugehörige Ausdruck ausdr) unnötig, so dass der gesamte **let**-Ausdruck durch eine β-Reduktion eliminiert werden kann.

Diese Vorgehensweise soll anhand des obigen **DEF**-Terms verdeutlicht werden. Dem Verfahren entsprechend wird zuerst der letzte **let**-Ausdruck (**let** s3 = c−a **in** (s3,b)) untersucht. Die Menge freier Variablen, die im Rest ***r3***:=(s3,b) dieses Ausdrucks vorkommen, ist $\lceil$***r3***$\rceil$={b,s3}. Da s3 $\in$ {b,s3}, ist die Variable s3 notwendig und somit auch der zugehörige Ausdruck c−a.

Die Variable s2 des vorangehenden **let**-Ausdrucks erweist sich dagegen als unnötig: Die Menge freier Variablen im Rest ***r2***:=(**let** s3 = c−a **in** (s3,b)) dieses **let**-Terms ist $\lceil$***r2***$\rceil$={a,b,c} und s2 ist in dieser Menge nicht enthalten. Der **let**-Ausdruck wird daher durch eine β-Reduktion entfernt. Das gleiche gilt für die Variable s1. Der Rest des entsprechenden **let**-Ausdrucks gleicht nach der Entfernung des zweiten **let**-Ausdrucks dem Rest ***r2***. Da s1 ebenfalls nicht in {a,b,c} enthalten ist, ist auch dieser **let**-Ausdruck unnötig und wird daher durch eine β-Reduktion eliminiert. Der resultierende **DEF**-Ausdruck lautet:

DEF (λ(a,b,c,d).**let** s3 = c−a **in** (s3,b))

Als Folge der Eliminierung der beiden **let**-Ausdrücke sind einige der eingehenden Werte des **DEF**-Terms unnötig geworden. Daher wird in einem weiteren Schritt der Term nach folgendem Schema in eine Variableneliminierung λ**ev**.**nv**′, die alle Variablen bis auf die notwendigen Variablen **nv**′ fallen lässt, sowie den vollständig TC-reduzierten **DEF**-Term aufgeteilt:

DEF (λ**ev**.**let** **nv** = (***body*** **nv**′) **in** **nv**) $\Longrightarrow$
DEF (λ**ev**.**nv**′) (* Variableneliminierung *)
SER
DEF (λ**nv**′.**let** **nv** = (***body*** **nv**′) **in** **nv**)

Das Schema auf das Beispiel angewandt ergibt:

```
DEF (λ(a,b,c,d).(a,b,c))        (* Variableneliminierung *)
SER
DEF (λ(a,b,c).let s3 = c−a in (s3, b))
```

Damit ist die Eliminierung des toten Codes aus der gegebenen **DEF**-Anweisung abgeschlossen. Die resultierende Variableneliminierung wird nun mit dem gegebenenfalls existierenden Vorgänger der **DEF**-Anweisung verschmolzen und das Eliminierungsverfahren entsprechend fortgesetzt.

Eliminierung toten Codes in IF-Termen

Folgt einem **IF**-Ausdruck eine Variableneliminierung, so wird diese zunächst mit Hilfe des Theorems 5.1 in den **IF**-Term integriert. Durch Anwendung dieses Theorems wird die nachfolgende Anweisung dupliziert und den beiden Zweigen des **IF**-Terms angehängt:

$$\begin{aligned} &\vdash \forall\ \textit{cnd}\ \textit{T}\ \textit{E}\ \textit{A}. \\ &\quad (\textbf{IF}\ \textit{cnd}\ \textbf{THEN}\ \textit{T}\ \textbf{ELSE}\ \textit{E})\ \textbf{SER}\ \textit{A} \\ &= \textbf{IF}\ \textit{cnd}\ \textbf{THEN}\ (\textit{T}\ \textbf{SER}\ \textit{A})\ \textbf{ELSE}\ (\textit{E}\ \textbf{SER}\ \textit{A}) \end{aligned} \tag{5.1}$$

Anschließend wird das Verfahren zur Eliminierung toten Codes rekursiv auf die beiden Zweige des **IF**-Terms angewandt, woraufhin jeder Zweig, der einige seiner eingehenden Werte nicht benötigt, mit einer Variableneliminierung beginnt. Benötigt mindestens einer der Zweige alle seiner eingehenden Werte, so benötigt implizit der gesamte **IF**-Ausdruck alle eingehenden Werte. In diesem Fall liegt der Term bereits in der TC-reduzierten Form vor. Ansonsten wird untersucht, ob es eingehende Werte gibt, die weder von den beiden Zweigen der **IF**-Anweisung noch von deren Bedingungsberechnung benötigt werden.

Um die Analyse zu vereinfachen, werden den Eingangsvariablen der Bedingungsberechnung und den Variableneliminierungen der beiden **IF**-Zweige die gleichen Namen zugeordnet. Eine Umbenennung der Variablen wird durch eine α-Konversion (siehe Abschnitt 3.1) erreicht. Daraufhin hat der **IF**-Ausdruck folgende Form:

$$\textbf{IF}\ (\lambda\underline{\textbf{ev}}.\textit{cx})\ \textbf{THEN}\ (\textbf{DEF}\ \lambda\underline{\textbf{ev}}.\underline{\textbf{tzv}}\ \textbf{SER}\ \textit{T})\ \textbf{ELSE}\ (\textbf{DEF}\ \lambda\underline{\textbf{ev}}.\underline{\textbf{ezv}}\ \textbf{SER}\ \textit{E})$$

Die Menge der für den **IF**-Term notwendigen Variablen entspricht der Menge aller freier Variablen, die in den Körpern der Bedingungsberechnung sowie der beiden Variableneliminierungen vorkommen: $\mathsf{nv} := \lceil \textit{cx} \rceil \cup \lceil \underline{\textbf{tzv}} \rceil \cup \lceil \underline{\textbf{ezv}} \rceil$. Ist nv eine echte Untermenge der eingehenden Variablen des **IF**-Terms $\mathsf{ev} := \lceil \underline{\textbf{ev}} \rceil$, so ist eine Variablenreduktion möglich.

Gegeben sei folgendes Beispiel:

```
IF (λ(a,b,c,d).b>c) THEN (DEF (λ(a,b,c,d).(d,b,c)) SER T)
                    ELSE (DEF (λ(a,b,c,d).c)        SER E)
```

Die freien Variablen der Unterterme, die den Termen ***cx***, **tzv** und **ezv** entsprechenden, sind: $\lceil b>c \rceil = \{b,c\}$, $\lceil (d,b,c) \rceil = \{b,c,d\}$ und $\lceil c \rceil = \{c\}$. Ihre Vereinigungsmenge ist $\mathsf{nv} := \{b,c,d\}$ und die

Menge der eingehenden Variablen ev:={a,b,c,d}. In diesem Beispiel erweist sich nv als echte Untermenge von ev. Die Menge unnötiger Variablen ist uv:=ev\nv ={a}.

Als Vorbereitung der Anwendung des Theorems 5.2, welches später für das Entfernen der unnötigen Variablen uv eingesetzt wird, wird eine sogenannte *einheitliche Variableneliminierung* ***uvs***:=λ**ev**.**nv** aufgebaut. Diese bildet die Menge der Eingangsvariablen ev des **IF**-Ausdrucks auf eine beliebige Variablenstruktur **nv**:=$\langle$nv$\rangle$ der tatsächlich notwendigen Variablen nv ab. In diesem Beispiel sei (b,c,d) die für $\langle$nv$\rangle$ gewählte Struktur. So ergibt sich die einheitliche Variableneliminierung ***uvs*** in diesem Fall zu (λ(a,b,c,d).(b,c,d)).

Diese Funktion ***uvs*** wird nun aus der Bedingungsberechnung und aus den Variableneliminierungen der beiden Zweige des **IF**-Ausdrucks entsprechend folgender Schemata herausgezogen:

$$\begin{array}{rcl} (\lambda\underline{\mathbf{ev}}.\boldsymbol{cx}) & \Longrightarrow & (\lambda\underline{\mathbf{nv}}.\boldsymbol{cx}) \circ (\lambda\underline{\mathbf{ev}}.\underline{\mathbf{nv}}) \\ (\mathtt{DEF}\ \lambda\underline{\mathbf{ev}}.\underline{\mathbf{tzv}}\ \mathtt{SER}\ \boldsymbol{T}) & \Longrightarrow & (\mathtt{DEF}\ \lambda\underline{\mathbf{ev}}.\underline{\mathbf{nv}}\ \mathtt{SER}\ \mathtt{DEF}\ \lambda\underline{\mathbf{nv}}.\underline{\mathbf{tzv}}\ \mathtt{SER}\ \boldsymbol{T}) \\ (\mathtt{DEF}\ \lambda\underline{\mathbf{ev}}.\underline{\mathbf{ezv}}\ \mathtt{SER}\ \boldsymbol{E}) & \Longrightarrow & (\mathtt{DEF}\ \lambda\underline{\mathbf{ev}}.\underline{\mathbf{nv}}\ \mathtt{SER}\ \mathtt{DEF}\ \lambda\underline{\mathbf{nv}}.\underline{\mathbf{ezv}}\ \mathtt{SER}\ \boldsymbol{E}) \end{array}$$

Daraufhin erhält man im Beispiel:

IF ((λ(b,c,d).b>c) $\circ$ (λ(a,b,c,d).(b,c,d)))
THEN (**DEF** (λ(a,b,c,d).(b,c,d)) **SER** (**DEF** (λ(b,c,d).(d,b,c)) **SER** ***T***))
ELSE (**DEF** (λ(a,b,c,d).(b,c,d)) **SER** (**DEF** (λ(b,c,d).c) **SER** ***E***))

Damit hat nun der resultierende Term die zur Anwendung des Theorems 5.2 erforderliche Form. Mit Hilfe dieses Theorems wird die Funktion ***uvs*** aus dem **IF**-Term herausgezogen und dadurch die unnötigen Variablen uv aus der **IF**-Anweisung entfernt.

$$\begin{array}{l} \vdash \forall\ \boldsymbol{cnd'}\ \boldsymbol{T}\ \boldsymbol{E}\ \boldsymbol{uvs}. \\ \quad \mathtt{IF}\ (\boldsymbol{cnd'} \circ \boldsymbol{uvs})\ \mathtt{THEN}\ (\mathtt{DEF}\ \boldsymbol{uvs}\ \mathtt{SER}\ \boldsymbol{T})\ \mathtt{ELSE}\ (\mathtt{DEF}\ \boldsymbol{uvs}\ \mathtt{SER}\ \boldsymbol{E}) \\ = \mathtt{DEF}\ \boldsymbol{uvs}\ \mathtt{SER}\ (\mathtt{IF}\ \boldsymbol{cnd'}\ \mathtt{THEN}\ \boldsymbol{T}\ \mathtt{ELSE}\ \boldsymbol{E}) \end{array} \tag{5.2}$$

Im Beispiel erhält man nach Anwendung des Theorems 5.2:

DEF (λ(a,b,c,d).(b,c,d))
SER
IF (λ(b,c,d).b>c) **THEN** (**DEF** (λ(b,c,d).(d,b,c)) **SER** ***T***)
ELSE (**DEF** (λ(b,c,d).c) **SER** ***E***)

Der tote Code ist aus dem **IF**-Ausdruck vollständig entfernt und die unnötigen Variablen werden explizit durch die dem **IF**-Term vorangehende Variableneliminierung ***uvs*** ausgedrückt.

Eliminierung toten Codes in PAR-Termen

Falls eine Variableneliminierung einem **PAR**-Ausdruck folgt, so hat diese stets die Form **DEF** (λ(**lv**,**rv**).**nv**). Dabei entspricht **lv** der Ausgabe des linken **PAR**-Zweigs und **rv** der des rechten **PAR**-Zweigs. Um eine solche Variableneliminierung zu integrieren, muss diese zunächst für jeden der beiden Zweige in eine separate Variableneliminierung ***lvs***:=λ**lv**.**lfnv** bzw. ***rvs***:=λ**rv**.**rfnv** aufgeteilt werden. Die Struktur **nv** in der ursprünglichen Variableneliminierung enthält die letztendlich notwendigen Variablen. Die Strukturen **lfnv** und **rfnv** werden aus den für den

linken bzw. rechten Zweig notwendigen Variablen zusammengesetzt: $\underline{\mathbf{lfnv}}{:=}\langle\lceil\underline{\mathbf{lv}}\rceil\cap\lceil\underline{\mathbf{nv}}\rceil\rangle$ und $\underline{\mathbf{rfnv}}{:=}\langle\lceil\underline{\mathbf{rv}}\rceil\cap\lceil\underline{\mathbf{nv}}\rceil\rangle$. Auf Basis dieser Strukturen wird die ursprüngliche Variableneliminierung nach folgendem Schema umgeformt:

$$
\begin{array}{ll}
 & \mathtt{DEF}\ (\lambda(\underline{\mathbf{lv}},\underline{\mathbf{rv}}).\underline{\mathbf{nv}}) \\
\Longrightarrow & (\mathtt{DEF\ FST\ SER\ DEF}\ (\lambda\underline{\mathbf{lv}}.\underline{\mathbf{lfnv}}))\ \mathtt{PAR}\ (\mathtt{DEF\ SND\ SER\ DEF}\ (\lambda\underline{\mathbf{rv}}.\underline{\mathbf{rfnv}})) \\
 & \mathtt{SER} \\
 & \mathtt{DEF}\ \lambda(\underline{\mathbf{lfnv}},\underline{\mathbf{rfnv}}).\underline{\mathbf{nv}}
\end{array}
$$

Daraufhin ist die Integration der beiden zweigspezifischen Variableneliminierungen ***lvs*** und ***rvs*** in den **PAR**-Ausdruck mit Hilfe des folgenden Theorems möglich:

$$
\begin{array}{l}
\vdash \forall\, \boldsymbol{L}\ \boldsymbol{lvs}\ \boldsymbol{R}\ \boldsymbol{rvs}. \\
\quad (\boldsymbol{L}\ \mathtt{PAR}\ \boldsymbol{R}) \\
\quad \mathtt{SER} \\
\quad ((\mathtt{DEF\ FST\ SER\ DEF}\ \boldsymbol{lvs})\ \mathtt{PAR}\ (\mathtt{DEF\ SND\ SER\ DEF}\ \boldsymbol{rvs})) \\
= (\boldsymbol{L}\ \mathtt{SER\ DEF}\ \boldsymbol{lvs})\ \mathtt{PAR}\ (\boldsymbol{R}\ \mathtt{SER\ DEF}\ \boldsymbol{rvs})
\end{array}
\tag{5.3}
$$

Die folgende Vorgehensweise weist einige Parallelen zu der Eliminierung toten Codes aus **IF**-Termen auf. Nach der Integration der Variableneliminierung wird das Verfahren zur Eliminierung toten Codes auf beide Zweige des **PAR**-Ausdrucks angewandt. Falls beide Zweige unnötige Variablen enthalten, werden als Folge dieses Schritts beide Zweige von einer Variableneliminierung angeführt:

$$(\mathtt{DEF}\ (\lambda\underline{\mathbf{ev}}.\underline{\mathbf{lnv}})\ \mathtt{SER}\ \boldsymbol{LR})\ \mathtt{PAR}\ (\mathtt{DEF}\ (\lambda\underline{\mathbf{ev}}.\underline{\mathbf{rnv}})\ \mathtt{SER}\ \boldsymbol{RR})$$

Anschließend werden den Eingangsvariablen der beiden **PAR**-Zweige die gleichen Namen zugeordnet. So ergibt sich die Menge der für den **PAR**-Ausdruck insgesamt notwendigen Variablen zu $\mathsf{nv}{:=}\lceil\underline{\mathbf{lnv}}\rceil\cup\lceil\underline{\mathbf{rnv}}\rceil$. Eine Variableneliminierung ist möglich, falls nv eine echte Untermenge der Menge eingehender Variablen $\mathsf{ev}{:=}\lceil\underline{\mathbf{ev}}\rceil$ ist. Im folgenden Beispiel entspricht die Menge der für den **PAR**-Ausdruck insgesamt notwendigen Variablen $\mathsf{nv}{:=}\lceil y\rceil\cup\lceil(z,y)\rceil{=}\{y,z\}$:

$$(\mathtt{DEF}\ (\lambda(x,y,z).y)\ \mathtt{SER}\ \boldsymbol{LR})\ \mathtt{PAR}\ (\mathtt{DEF}\ (\lambda(x,y,z).(z,y))\ \mathtt{SER}\ \boldsymbol{RR})$$

Nach der Ermittlung der notwendigen Variablen wird eine einheitliche Variableneliminierung ***uvs*** erzeugt, welche die eingehenden Variablen des **PAR**-Ausdrucks auf eine Struktur $\langle\mathsf{nv}\rangle$ der notwendigen Variablen abbildet. Im Beispiel wird für $\langle\mathsf{nv}\rangle$ die Variablenstruktur (y,z) gewählt. Demzufolge ergibt sich ***uvs*** zu $(\lambda(x,y,z).(y,z))$. Nach dem Herausziehen dieser einheitlichen Variableneliminierung aus den die beiden **PAR**-Zweige anführenden Variableneliminierungen erhält man:

$$
\begin{array}{l}
(\mathtt{DEF}\ (\lambda(x,y,z).(y,z))\ \mathtt{SER\ DEF}\ (\lambda(y,z).y)\ \mathtt{SER}\ \boldsymbol{LR}) \\
\mathtt{PAR} \\
(\mathtt{DEF}\ (\lambda(x,y,z).(y,z))\ \mathtt{SER\ DEF}\ (\lambda(y,z).(z,y))\ \mathtt{SER}\ \boldsymbol{RR})
\end{array}
$$

Daraufhin kann das folgende Theorem zur Extraktion der Variableneliminierung aus dem **PAR**-Term zum Einsatz kommen. Das Theorem bewirkt das Verschieben einer beliebigen Anweisung, welche die beiden Zweige eines **PAR**-Ausdrucks anführt, vor die entsprechende **PAR**-Anweisung:

$$\begin{aligned} &\vdash \forall\, \boldsymbol{A}\ \boldsymbol{LR}\ \boldsymbol{RR}. \\ &\quad (\boldsymbol{A}\ \mathtt{SER}\ \boldsymbol{LR})\ \mathtt{PAR}\ (\boldsymbol{A}\ \mathtt{SER}\ \boldsymbol{RR}) \\ &= \boldsymbol{A}\ \mathtt{SER}\ (\boldsymbol{LR}\ \mathtt{PAR}\ \boldsymbol{RR}) \end{aligned} \tag{5.4}$$

Nach Anwendung des Theorems 5.4 ergibt sich im Beispiel:

DEF (λ(x,y,z).(y,z))
SER
(**DEF** (λ(y,z).y) **SER** ***LR***) **PAR** (**DEF** (λ(y,z).(z,y)) **SER** ***RR***)

Jeglicher toter Code ist damit aus dem **PAR**-Term eliminiert worden und unnötige Eingangsvariablen der ursprünglichen **PAR**-Anweisung werden explizit durch die vorangestellte Variableneliminierung dargestellt.

Eliminierung toten Codes in WHILE-Termen

Folgt einem **WHILE**-Ausdruck keine Variableneliminierung, so sind alle Ausgangsvariablen notwendig. Da bei einem **WHILE**-Ausdruck die eingehenden Variablen den Ausgangsvariablen entsprechen, sind in diesem Fall auch implizit alle Eingangsvariablen notwendig, so dass sich die Eliminierung toten Codes darauf beschränkt, das entsprechende Verfahren auf den Rumpf der Schleife anzuwenden. Auch wenn alle Ein- und Ausgangsvariablen einer Schleife sich als notwendig erweisen, kann deren Rumpf dennoch toten Code enthalten. Zum Beispiel sind im folgenden Term im Rumpf des **WHILE**-Ausdrucks die Variable u sowie der zugehörige Ausdruck b$*$c unnötig und würden demzufolge eliminiert werden:

WHILE (λ(a,b,c).a>b)
  DO (**DEF** (λ(a,b,c).**let** a = b+1 **in**
    let u = b$*$c **in** (a,b,c,u))
    SER
    DEF (λ(a,b,c,u).**let** b = b$*$2 **in** (a,b,c)))

Bei **WHILE**-Ausdrücken gestaltet sich die Integration einer nachfolgenden Variableneliminierung aufwändiger als bei den übrigen Konstrukten. Der Ausgangsterm habe die folgende Form:

WHILE (λ**wvs**.***cnd***) **DO** ***BODY***
SER
DEF λ**wvs**.**fnv**

Um herauszufinden, welche der eingehenden Variablen des **WHILE**-Ausdrucks notwendig sind und welche nicht, muss eine spezielle Analyse des Datenflusses der Schaltung vorgenommen werden. Diese wird in der vorliegenden Arbeit den Beschreibungen in Kapitel 4 entsprechend außerhalb des Theorembeweisers durchgeführt. Sei nv bzw. uv:=$\lceil$**wvs**$\rceil \setminus$nv die Menge der notwendigen bzw. unnötigen Variablen.

Da im Allgemeinen nicht entscheidbar ist, ob **WHILE**-Ausdrücke terminieren, werden sie im Zuge der Eliminierung toten Codes grundsätzlich nicht entfernt. Die Eliminierung einer potentiell endlos laufenden Schleife würde das Gesamtverhalten des Programms ändern und ist daher nicht zulässig. Infolgedessen sind stets u. a. all diejenigen Variablen notwendig, die zur Berechnung der Schleifenbedingung erforderlich sind. Wurde zuvor die Bedingungsberechnung, wie in Abschnitt 4.2.2 beschrieben, herausgezogen, so besteht die Schleifenbedingung nur noch aus einer Funktion zur Selektion einer einzigen der eingehenden Variablen.

Die Eliminierung unnötiger Eingangsvariablen eines **WHILE**-Terms wird letzten Endes auf Basis des folgenden Theorems durchgeführt:

$$\begin{aligned}&\vdash \forall\, \boldsymbol{U}\ \boldsymbol{R}\ \boldsymbol{c}.\\ &\quad (\boldsymbol{U}\ \mathbf{SER}\ \mathbf{DEF}\ \mathbf{FST} = \mathbf{DEF}\ \mathbf{FST}\ \mathbf{SER}\ \boldsymbol{R})\\ &\Rightarrow (\mathbf{WHILE}\ (\boldsymbol{c} \circ \mathbf{FST})\ \mathbf{DO}\ \boldsymbol{U}\ \mathbf{SER}\ \mathbf{DEF}\ \mathbf{FST} = \mathbf{DEF}\ \mathbf{FST}\ \mathbf{SER}\ \mathbf{WHILE}\ \boldsymbol{c}\ \mathbf{DO}\ \boldsymbol{R})\end{aligned} \tag{5.5}$$

Die Anwendung dieses Theorems setzt voraus, dass die Struktur der eingehenden Variablen des **WHILE**-Ausdrucks die Form (**nvs**,**uvs**) hat. Die Variablenstruktur **nvs**:=⟨nv⟩ enthält dabei die notwendigen Variablen und die Struktur **uvs**:=⟨uv⟩ die unnötigen Variablen. In der Regel ist eine Umsortierung der Variablen des **WHILE**-Terms erforderlich, um zu dieser Form zu gelangen.

Zur Durchführung einer entsprechenden Umsortierung werden zwei Hilfsfunktionen erzeugt: eine Funktion ***new_vs***, welche die Struktur der ursprünglichen Eingangsvariablen auf die gewünschte Zielform abbildet, sowie deren Umkehrfunktion ***inv_vs***. Im konkreten Fall entspricht die Funktion ***new_vs*** der Funktion (λ**wvs**.(**nvs**,**uvs**)) und demzufolge ***inv_vs*** der Funktion (λ(**nvs**,**uvs**).**wvs**). Es ist leicht zu beweisen, dass die Komposition (***inv_vs*** $\circ$ ***new_vs***) dieser beiden Funktionen die Identität (**I**) ergibt. Diese Tatsache entspricht der Vorbedingung des folgenden Theorems zur Umsortierung der Variablenordnung eines **WHILE**-Terms:

$$\begin{aligned}&\vdash \forall\, \boldsymbol{new\_vs}\ \boldsymbol{inv\_vs}\ \boldsymbol{cnd}\ \boldsymbol{B}.\\ &\quad (\boldsymbol{inv\_vs} \circ \boldsymbol{new\_vs} = \mathbf{I})\\ &\Rightarrow (\mathbf{WHILE}\ (\boldsymbol{cnd})\ \mathbf{DO}\ \boldsymbol{B} = \mathbf{DEF}\ \boldsymbol{new\_vs}\\ &\qquad\qquad \mathbf{SER}\\ &\qquad\qquad \mathbf{WHILE}\ (\boldsymbol{cnd} \circ \boldsymbol{inv\_vs})\\ &\qquad\qquad\qquad \mathbf{DO}\ (\mathbf{DEF}\ \boldsymbol{inv\_vs}\ \mathbf{SER}\ \boldsymbol{B}\ \mathbf{SER}\ \mathbf{DEF}\ \boldsymbol{new\_vs})\\ &\qquad\qquad \mathbf{SER}\\ &\qquad\qquad \mathbf{DEF}\ \boldsymbol{inv\_vs})\end{aligned} \tag{5.6}$$

Die gesamte Transformation zur Entfernung toten Codes aus **WHILE**-Schleifen soll anhand eines Beispiels veranschaulicht werden. Ausgangspunkt sei folgendes Codefragment:

```
WHILE (λ(a,b,c,d).c>b)
      DO ( DEF (λ(a,b,c,d).
                  let a = d+1 in
                  let b = b+1 in
                  let c = c−b in
                  let d = a+b in (a,b,c,d)) )
SER DEF (λ(a,b,c,d).(c,b))
```

Eine Analyse des Codes ergibt, dass die Variablen a und d unnötig sind. Gemäß dieser Information wird zunächst die Variablenstruktur der Eingangsvariablen des **WHILE**-Ausdrucks umsortiert.

Zur Umsortierung der Variablenordnung in dem **WHILE**-Term werden die Typanpassungsfunktionen ***new_vs***:=(λ(a,b,c,d).((b,c),(a,d))) sowie deren zugehörige Umkehrfunktion ***inv_vs***:=(λ((b,c),(a,d)).(a,b,c,d)) erzeugt. Nach dem Beweis der Vorbedingung des Theorems 5.6 (***inv_vs*** $\circ$ ***new_vs*** = **I**) kann das Theorem auf den **WHILE**-Term des Beispiels angewandt werden. Man erhält:

```
DEF (λ(a,b,c,d).((b,c),(a,d)))
SER
WHILE ( (λ(a,b,c,d).c>b) ∘ (λ((b,c),(a,d)).(a,b,c,d)) )
       DO ( DEF (λ((b,c),(a,d)).(a,b,c,d))
            SER
            DEF (λ(a,b,c,d).
                   let a = d+1 in
                   let b = b+1 in
                   let c = c−b in
                   let d = a+b in (a,b,c,d))
            SER
            DEF (λ(a,b,c,d).((b,c),(a,d))) )
SER
DEF (λ((b,c),(a,d)).(a,b,c,d))
SER
DEF (λ(a,b,c,d).(c,b))
```

Nach diesem Schritt werden alle aufeinander folgende **DEF**-Terme entsprechend der Beschreibung in Abschnitt 4.2.3 zusammengefasst. Zusätzlich wird die Bedingungsfunktion der **WHILE**-Schleife vereinfacht:

```
DEF (λ(a,b,c,d).((b,c),(a,d)))
SER
WHILE (λ((b,c),(a,d)).c>b)
       DO ( DEF (λ((b,c),(a,d)).
                   let a = d+1 in
                   let b = b+1 in
                   let c = c−b in
                   let d = a+b in ((b,c),(a,d))) )
SER
DEF (λ((b,c),(a,d)).(c,b))
```

Nun muss der Code auf die Anwendung des Theorems 5.5 vorbereitet werden, dessen Einsatz am Ende die Entfernung der unnötigen Eingangsvariablen bewirken wird. Im Zuge dessen werden sowohl aus der Bedingungsberechnung der **WHILE**-Schleife als auch aus der der **WHILE**-Schleife folgenden Variableneliminierung die Funktion **FST** herausgezogen.

Die Umformung ergibt:

```
DEF (λ(a,b,c,d).((b,c),(a,d)))
SER
WHILE (λ(b,c).c>b) ∘ FST)
        DO ( DEF (λ((b,c),(a,d)).
                    let a = d+1 in
                    let b = b+1 in
                    let c = c−b in
                    let d = a+b in ((b,c),(a,d))) )
SER
DEF FST
SER
DEF (λ(b,c).(c,b))
```

Es verbleibt vor der Anwendung des Theorems 5.5 der Beweis dessen Vorbedingung (***U*** **SER DEF FST** = **DEF FST SER** ***R***). Im Term ***U*** **SER DEF FST** entspricht ***U*** dem Rumpf der Schleife. Nach der Konstruktion dieses Terms wird das allgemeine Verfahren zur Eliminierung toten Codes darauf angewandt. Da zuvor bereits alle notwendigen Variablen der Schleife in die linke Hälfte der Eingangsvariablenstruktur verschoben wurden, wird der Term nach der Eliminierung dessen toten Codes, soweit die Auswahl der notwendigen Variablen korrekt war, stets die Form **DEF FST SER** ***R*** haben. Dabei entspricht ***R*** der TC-reduzierten Form von ***U***. Damit ist die Voraussetzung für die Anwendung des Theorems 5.5 erfüllt.

Im Beispiel erhält man:

```
DEF (λ((b,c),(a,d)).                          =     DEF FST
    let a = d+1 in                                  SER
    let b = b+1 in                                  DEF (λ(b,c).
    let c = c−b in                                      let b = b+1 in
    let d = a+b in ((b,c),(a,d)))                       let c = c−b in (b,c))
SER
DEF FST
```

Auf Basis des Theorems 5.5 können nun alle unnötigen Variablen aus der **WHILE**-Schleife entfernt werden:

```
DEF (λ(a,b,c,d).((b,c),(a,d)))
SER
DEF FST
SER
WHILE (λ(b,c).c>b)
        ( DEF (λ(b,c).
                    let b = b+1 in
                    let c = c−b in (b,c)) )
SER
DEF (λ(b,c).(c,b))
```

Abschließend werden die beiden den **WHILE**-Term anführenden **DEF**-Terme zu der Variableneliminierung **DEF** (λ(a,b,c,d).(b,c)) zusammengefasst.

Nach Durchführung des Verfahrens zur Eliminierung toten Codes ergibt sich auch beim **WHILE**-Konstrukt, falls dieses unnötige Eingangsvariablen hat, die Form einer Variableneliminierung, welche explizit die unnötigen Variablen ausdrückt, gefolgt von der TC-reduzierten Form des ursprünglichen **WHILE**-Ausdrucks.

Die Ausführungen in diesem Abschnitt machen deutlich, wie eine Anwendung von Theoremen oft erst nach vorbereitenden Umformungen des zu transformierenden Terms möglich ist. Besonders aufwändig ist in dem Verfahren zur Eliminierung toten Codes die Vorbereitung von **WHILE**-Anweisungen, deren Eingangs- bzw. Ausgangsvariablenstruktur vor der Eliminierung des toten Codes umsortiert werden müssen. Erst nach dieser Umformung kann die Vorbedingung des Theorems 5.5 auf die vorgestellte Weise automatisch bewiesen und daraufhin das entsprechende Theorem angewandt werden. Auf ähnliche Weise werden auch die folgenden Transformationen zur Durchführung der formalen algorithmischen Synthese vorgenommen: Terme werden, falls erforderlich, in eine wohldefinierte Form überführt, um anschließend gegebenenfalls existierende Vorbedingungen des anzuwendenden Theorems automatisch beweisen und somit die Transformation vollautomatisch durchführen zu können.

5.2 Übergang zur steuerflussorientierten Darstellung

Vor der Durchführung der formalen algorithmischen Synthese wird die gegebene Schaltungsspezifikation als Erstes in die steuerflussorientierte Darstellung (siehe Abschnitt 3.3.4) überführt. Alle weiteren Schritte der algorithmischen Synthese werden auf Basis dieser Darstellungsform durchgeführt. Auf Grund der Ähnlichkeit dieser Darstellung zu Graphen ist sie besonders zur Durchführung der algorithmischen Synthese geeignet. Insbesondere Ablaufplanungsverfahren erwarten zumeist einen Graphen als Eingabe und stellen ihre Ergebnisse ebenso in Graphenform dar. So kann auf Basis der steuerflussorientierten Darstellung die Verhaltensspezifikation der Schaltung in eine Form gebracht werden, die dem Eingabegraphen des anzuwendenden Syntheseverfahrens entspricht, und anschließend konstruktiv derart transformiert werden, dass sie exakt dessen resultierenden Ausgabegraphen nachbildet.

Trennung zwischen Aktion und Transition

Im Zuge der Automatisierung der formalen Synthese wurde für die einzelnen Zustände in einer steuerflussorientierten Darstellung eine spezielle Form festgelegt, welche eine strikte Trennung zwischen der Durchführung von Berechnungen und der Bestimmung des Folgezustands verkörpert. So sind in dieser Arbeit die Rümpfe der Zustände grundsätzlich in zwei Funktionen aufgeteilt: in eine sogenannte *Aktion* und eine *Transitionsfunktion*. Während in der Aktion eines Zustands ein Teil der Schaltungspezifikation ausgeführt wird, führt die nachfolgende Transitionsfunktion allein die Bestimmung des Folgezustands durch. Diese Aufteilung erleichtert nicht nur die Automatisierung der Transformationen, welche auf der steuerflussorientierten Darstellung operieren, sondern erhöht auch deren Effizienz. Alle Beweise, die einen Bezug zu

den Nachfolgern eines Zustands haben, können in der Regel allein auf Basis der einfachen Transitionsfunktionen durchgeführt werden. Durch die Aufteilung ergibt sich für die Zustände der steuerflussorientierten Darstellung eine Syntax gemäß Abbildung 5.1[‡].

Bezeichner	= Konstante
VarStrukt	= Variable | "(" VarStrukt "," VarStrukt ")"
MuxStrukt	= Bezeichner| "**MUX (**" Variable "**,**" MuxStrukt "**,**" MuxStrukt "**)**"
Transitionsfunktion	= **DEF** "**(** λ**(**" VarStrukt$_1$ ["**,**" VarStrukt$_2$] "**).(**" VarStrukt$_1$ "**,**" MuxStrukt "**))**"
Zustand	= "**(**" Bezeichner "**,**" P-Term "**SER**" Transitionsfunktion "**)**"

Abbildung 5.1: Syntax der Zustände in der steuerflussorientierten Darstellung

Ergänzend muss zu dieser Syntaxdefinition angemerkt werden, dass bei der Spezifikation einer Transitionsfunktion die beiden Variablenstrukturen VarStrukt, welche mit dem Index 1 gekennzeichnet sind, identisch sein müssen. Zusätzlich darf die Mux-Struktur MuxStrukt der Transitionsfunktion nur von den Variablen der optionalen Eingabestruktur VarStrukt mit dem Index 2 abhängen.

Abbildung 5.2 gibt schematisch den Aufbau eines Zustands wieder. Auf Basis einer Eingabe $\underline{\mathbf{x}}$ berechnet die Aktion einen neuen Wert $\underline{\mathbf{x}}'$ und gegebenenfalls einige Statusdaten $\underline{\mathbf{stat}}$. Die Statusdaten bestehen aus einer beliebigen Anzahl boolescher Werte und steuern in der Transitionsfunktion die Bestimmung des Folgezustandsbezeichners s'. Den von der Aktion berechneten Wert $\underline{\mathbf{x}}'$ gibt die Transitionsfunktion zusätzlich zum Bezeichner des Folgezustands unverändert weiter.

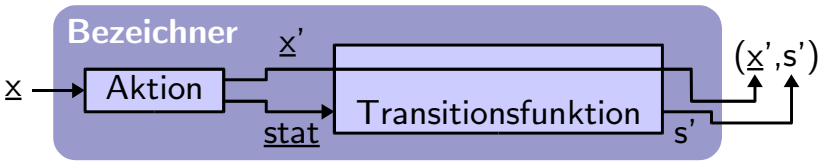


Abbildung 5.2: Aufbau eines Zustands

Hat ein Zustand nur einen möglichen Nachfolger, so erübrigt sich die Berechnung von Statusdaten in dessen Aktion, und seine Transitionsfunktion hat die einfache Form **DEF** ($\lambda\underline{\mathbf{x}}'.(\underline{\mathbf{x}}',s')$). Anderenfalls wird in der Transitionsfunktion des Zustands durch eine Struktur von Multiplexern der Bezeichner des Folgezustands anhand der Statuswerte ermittelt, welche sich bei der Ausführung seiner Aktion ergeben.

Abbildung 5.3 stellt eine typische Transitionsfunktion eines Zustands mit mehreren potentiellen Folgezuständen dar. Als Eingabe erhält die Transitionsfunktion in diesem Fall die von der Aktion berechneten Werte $\underline{\mathbf{x}}'$ sowie zwei boolesche Statussignale ***c1*** und ***c2***. Von der Transitionsfunktion wird der Wert $\underline{\mathbf{x}}'$ unverändert ausgegeben. Zusätzlich wird durch eine Multiplexerstruktur auf Basis der zwei Statuswerte der Bezeichner des nachfolgenden Zustands ausgewählt.

[‡]Unveränderte Definitionen von Nichtterminalen sind wiederum durch graue Schrift gekennzeichnet.

```
DEF ( λ(x',(c1,c2)).(x', MUX(c1, MUX(c2,1,4),
                                 MUX(c2,3,5))) )
```

Abbildung 5.3: Beispiel einer Transitionsfunktion eines Zustands mit mehreren Folgezuständen

Enthalten beispielsweise die beiden Statusvariablen ***c1*** und ***c2*** den Wert *wahr*, so wird der Zustand $\langle 1 \rangle$ als Nachfolger des Zustands ausgewählt.

5.2.1 Einführung der steuerflussorientierten Darstellung

Die Überführung einer Verhaltensspezifikation ***SPEC*** in eine steuerflussorientierte Darstellung beginnt mit der Anwendung des Theorems 5.7:

$$\begin{aligned} &\vdash (\textbf{start} \neq \textbf{exit}) \Rightarrow \\ &\quad \textit{SPEC} = \texttt{LOOP}\ \textbf{start}\ [(\textbf{start}, \textit{SPEC}\ \texttt{SER}\ \texttt{DEF}\ (\lambda \underline{x}'.(\underline{x}', \textbf{exit}))] \end{aligned} \tag{5.7}$$

Das Ergebnis von dessen Anwendung ist eine steuerflussorientierte Darstellung mit einem einzigen Zustand, dem Initialzustand $\langle$start$\rangle$ (siehe Abbildung 5.4). Die Aktion im Rumpf des Zustands $\langle$start$\rangle$ entspricht der ursprünglichen Verhaltensspezifikation ***SPEC***, d. h. in diesem einen Zustand wird die vollständige Spezifikation ausgeführt. Die Transitionsfunktion des Zustands wählt als Folgezustand unbedingt den Zustand $\langle$exit$\rangle$. Da der Bezeichner **exit** – entsprechend der Voraussetzung des Theorems 5.7 – nie dem Bezeichner **start** entsprechen darf und in der Zustandsliste nur ein einziger Zustand mit dem Bezeichner **start** existiert, ist der von der Transitionsfunktion gewählte Folgezustand $\langle$exit$\rangle$ implizit ein Endzustand. Das bedeutet, dass sofort nach Ausführung des Initialzustands $\langle$start$\rangle$ die **LOOP**-Schleife terminiert und somit mit Ausführung des Zustands $\langle$start$\rangle$ die ursprüngliche Spezifikation ***SPEC*** in der steuerflussorientierten Darstellung genau einmal ausgeführt wird.

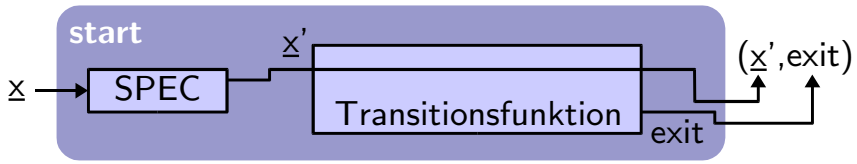


Abbildung 5.4: Erster Zustand nach Einführung der steuerflussorientierten Darstellung

Um eine Verhaltensspezifikation mit Hilfe des Theorems 5.7 in eine steuerflussorientierte Darstellung überführen zu können, muss diese einen Typ $\beta \rightarrow \beta\, partial$ haben. Im Allgemeinen hat eine Spezifikation allerdings einen Typ $\alpha \rightarrow \beta\, partial$, in welchem sich die Typen α und β unterscheiden. Infolgedessen ist in der Regel vor der Durchführung der Transformation eine Typanpassung der Spezifikation erforderlich. Dabei wird die Spezifikation derart transformiert, dass sie einen Typ $\beta \rightarrow \beta\, partial$ aufweist, und gleichzeitig in zwei **DEF**-Terme eingefasst, welche deren ursprünglichen Eingabetyp auf den neuen Eingabetyp bzw. den neuen Ausgangstyp auf deren ursprünglichen Ausgangstyp abbilden. Die transformierte Spezifikation kann daraufhin in die steuerflussorientierte Darstellung übertragen werden.

Typanpassung der Spezifikation: Der Typanpassung geht eine Analyse des Ein- und Ausgabetyps der Verhaltensspezifikation voraus. In der Analyse wird die maximale Anzahl der in diesen beiden Typen jeweils auftretenden atomaren Typen (siehe Abschnitt 3.1) ermittelt. Anschließend wird ein neuer einheitlicher Datentyp γ erzeugt, der alle atomaren Typen in der ermittelten Anzahl enthält, und so die Überführung des Eingabetyps α einer Spezifikation in γ und des Ausgabetyps $\beta\,partial$ in $\gamma\,partial$ erlaubt.

Zur Unterstützung der Typanpassung werden vier Hilfsfunktionen erzeugt: die Funktionen anachc:$\alpha \rightarrow \gamma$ und bnachc:$\beta \rightarrow \gamma$, welche die Typen α und β auf den Einheitstyp γ abbilden, sowie deren jeweilige Umkehrfunktion cnacha:$\gamma \rightarrow \alpha$ und cnachb:$\gamma \rightarrow \beta$. Als nächster Schritt wird die Spezifikation nach folgendem Schema in diese Typumwandlungen eingefasst:

		DEF anachc **SER DEF** cnacha		**DEF** anachc
		SER		**SER**
SPEC	$\rightarrow$	***SPEC***	$\rightarrow$	***SPEC′***
		SER		**SER**
		DEF bnachc **SER DEF** cnachb		**DEF** cnachb

Die Ausführung der Funktionen anachc bzw. bnachc mit der anschließenden Ausführung ihrer zugehörigen Umkehrfunktion ergeben jeweils die Identität, so dass das Verhalten der Spezifikation durch die erste Transformation nicht verändert wird. Nach dem Verschmelzen der Terme **DEF** cnacha und **DEF** bnachc mit der ursprünglichen Spezifikation ***SPEC*** weist die resultierende Spezifikation ***SPEC′*** den gewünschten Typ $\gamma \rightarrow \gamma\,partial$ auf und kann nun auf Basis des Theorems 5.7 in die steuerflussorientierte Darstellung überführt werden.

Die Vorgehensweise soll in einem Beispiel anhand der folgenden Spezifikation verdeutlicht werden:

```
DEF ( λ(i1, i2, i3).
        let a1 = i1 + i2 in
        let a2 = i2 ∗ a1 in (a2, T) )
```

Der Eingabetyp dieser Spezifikation ist $num \times num \times num$. Ihr Ausgabetyp ist $(num \times bool)\,partial$. Der atomare Typ num kommt in den beiden Typen maximal dreimal vor, während der Typ $bool$ höchstens einmal enthalten ist. Auf Basis dieser Elementartypen in der gegebenen Anzahl wird der neue einheitliche Datentyp erzeugt. Die Anordnung der Elementartypen in dem resultierenden Einheitstypen ist frei wählbar. Im Beispiel sei der Einheitstyp $num \times num \times num \times bool$.

Zur Anpassung des Ausgabetyps wird die Funktion **DEF** (λ(a2,b1).(a2,hvn1,hvn2,b1)) erstellt. Die neu eingeführten Hilfsvariablen hvn1 und hvn2 haben dabei den Typ num. Zusätzlich wird die zugehörige Umkehrfunktion **DEF** (λ(a2,hvn1,hvn2,b1).(a2,b1)) aufgebaut, welche die zur Anpassung des Ausgabetyps eingeführten Hilfsvariablen wieder fallen lässt und das ursprüngliche Ergebnis ausgibt. Diese beiden Funktionen werden als zusätzliche Anweisungen an die Spezifikation angehängt. Gleichzeitig werden die Funktionen **DEF** (λ(i1,i2,i3).(i1,i2,i3,hvb1)) sowie deren Umkehrfunktion zur Anpassung des Eingabetyps erzeugt und vor die ursprüngliche Spezifikation gestellt. Die Hilfsvariable hvb1 hat in diesem Fall den Typ $bool$.

Man erhält:

```
DEF ( λ(i1, i2, i3).(i1, i2, i3, hvb1) )
SER
DEF ( λ(i1, i2, i3, hvb1).(i1, i2, i3) )
SER
DEF ( λ(i1, i2, i3).
        let a1 = i1 + i2 in
        let a2 = i2 ∗ a1 in (a2, T) )
SER
DEF ( λ(a2, b1).(a2, hvn1, hvn2, b1) )
SER
DEF ( λ(a2, hvn1, hvn2, b1).(a2, b1) )
```

Ein Verschmelzen der ursprünglichen Spezifikation mit den beiden direkt angrenzenden Typanpassungen ergibt:

```
DEF ( λ(i1, i2, i3).(i1, i2, i3, hvb1) )
SER
DEF ( λ(i1, i2, i3, hvb1).
        let a1 = i1 + i2 in
        let a2 = i2 ∗ a1 in (a2, hvn1, hvn2, T) )
SER
DEF ( λ(a2, hvn1, hvn2, b1).(a2, b1) )
```

Die resultierende Spezifikation, umrahmt von der anfänglichen und abschließenden Typanpassung, hat nun einen Typ $\gamma \rightarrow \gamma\, partial$, wobei γ dem gewählten Einheitstyp entspricht. Damit erfüllt die Spezifikation nunmehr die Voraussetzung für eine Überführung in die steuerflussorientierte Darstellung. Wählt man `1` als Bezeichner des Startzustands und `0` für den Ausgangszustand, so erhält man:

```
DEF ( λ(i1, i2, i3).(i1, i2, i3, hvb1) )
SER
LOOP 1
 [(1, DEF ( λ(i1, i2, i3, hvb1).
            let a1 = i1 + i2 in
            let a2 = i2 ∗ a1 in (a2, hvn1, hvn2, T) )
    SER DEF (λx'.(x',0)))]
SER
DEF ( λ(a2, hvn1, hvn2, b1).(a2, b1) )
```

In diesem Beispiel besteht die Spezifikation nur aus einer einzigen **DEF**-Anweisung. Ansonsten können natürlich auch Anweisungen, wie **IF**-Verzweigungen und **WHILE**-Schleifen, auftreten. Deren Typen werden in ähnlicher Weise mit entsprechenden Hilfsfunktionen angepasst. Die Integration der Funktionen zur Typänderung erfolgt bei **IF**-Termen mit Hilfe der in Abschnitt 5.1.1 vorgestellten Theoreme 5.1 und 5.2. Eine Änderung des Typs von **WHILE**-Schleifen wird auf Basis des Theorems 5.6 durchgeführt. Im Falle von **PAR**-Konstrukten bietet

es sich im Allgemeinen an, diese vor einem Übergang zur steuerflussorientierten Darstellung aus der Spezifikation zu eliminieren. Die folgenden Verfahren gehen davon aus, dass dieser Schritt im Zuge der Vorbereitung der algorithmischen Synthese entsprechend der Beschreibung in Abschnitt 4.4.1 bereits vor dem Darstellungswechsel vollzogen wurde.

Abbildung 5.5 gibt die Spezifikation aus Abbildung 4.2 auf Seite 76 nach deren Überführung in die steuerflussorientierte Darstellung wieder.

```
DEF (λ(u,n).(u,hvn1,n,hvb1))
SER
LOOP 1
 [(1,( DEF (λ(u,hvn1,n,hvb1). ((u,1,n),T))
       SER
       (WHILE (λ((u,v,n),GT1).GT1) DO
               ( DEF (λ((u,v,n),GT1).
                       let GT1' = n> 0 in ((u,v,n),GT1'))
                 SER
                 IF (λ((u,v,n),GT1).GT1)
                    THEN (DEF (λ((u,v,n),GT1).
                                 let ODD1 = ODD n in ((u,v,n),ODD1))
                          SER
                          IF (λ((u,v,n),ODD1).ODD1)
                             THEN (DEF (λ((u,v,n),ODD1).
                                          let MULT1 = u∗v in
                                          let SUB1  = n−1 in
                                          ((u,MULT1,SUB1),T)))
                             ELSE (DEF (λ((u,v,n),ODD1).
                                          let MULT2 = u∗u in
                                          let DIV1  = n DIV 2 in
                                          ((MULT2,v,DIV1),T))))
                    ELSE (DEF (λ((u,v,n),GT1).((u,v,n),F))) ))
       SER
       DEF (λ((u,v,n),GT1).(u,v,n,GT1)) )
     SER DEF (λx'.(x',0))]
SER
DEF (λ(u,v,n,GT1).v)
```

Abbildung 5.5: Ergebnis der Überführung in die steuerflussorientierte Darstellung

Zuerst wurden der Ein- und Ausgabetyp der ursprünglichen Verhaltensspezifikation modifiziert, um eine Überführung der Spezifikation in die steuerflussorientierte Darstellung zu ermöglichen. In diesem Zusammenhang wurden für die Hilfsfunktion zur Anpassung des Eingabetyps der Spezifikation die numerische Hilfsvariable hvn1 sowie die boolesche Hilfsvariable hvb1 ein-

geführt. Anschließend wurde die Spezifikation auf Basis des Theorems 5.7 in die steuerflussorientierte Darstellung überführt, welche daraufhin die gesamte Spezifikation in der Aktion des einzigen Zustands enthält.

In den folgenden Beschreibungen werden die anfängliche und abschließende Typanpassung außerhalb eines **LOOP**-Terms meistens der Übersichtlichkeit halber weggelassen. Sie müssen erst bei der Anwendung des Theorems zur Synthese des Steuerwerks wieder in Betracht gezogen werden (siehe Abschnitt 5.6).

5.2.2 Ersetzen der Steuerkonstrukte durch Zustandsübergänge

Nach der Einführung der steuerflussorientierten Darstellung werden die Steuerkonstrukte der Spezifikation eliminiert und ihr Steuerfluss durch explizite Zustandsübergänge nachgebildet. Beispielsweise wird ein Zustand mit einer Aktion der Form ***A*** **SER** ***B*** in zwei Zustände aufgespalten: Im ersten resultierenden Zustand wird als Aktion ***A*** ausgeführt und unbedingt der zweite Zustand als Nachfolger ausgewählt. In dessen Aktion kommt schließlich der Term ***B*** zur Ausführung. Das Konstrukt **SER** wird im Zuge der Transformation eliminiert und sein entsprechender Steuerfluss durch einen unbedingten Zustandsübergang zwischen den beiden resultierenden Zuständen ersetzt.

Die in dieser Arbeit entwickelten Transformationen, welche die Zustände steuerflussorientierter Darstellungen modifizieren, sind zur Vereinfachung meistens darauf ausgelegt, dass die betroffenen Zustände am Anfang der Zustandsliste stehen. Somit ist vor deren Anwendung oft eine Umsortierung der Zustände erforderlich.

Umsortieren von Zuständen: Das folgende Theorem ermöglicht das Verschieben eines Zustands an den Kopf der Zustandsliste:

$$
\begin{array}{ll}
 & \vdash \forall\ \textbf{start}\ \boldsymbol{L}\ \textbf{s}. \\
1 & \quad (\texttt{StateExists}\ \boldsymbol{L}\ \textbf{s}) \\
2 & \quad \Longrightarrow (\textbf{LOOP start}\ \boldsymbol{L} = \\
3 & \qquad \textbf{LOOP start}\ ((\textbf{s},\texttt{StateCase}\ \boldsymbol{L}\ \textbf{s})\texttt{::DeleteState}\ \boldsymbol{L}\ \textbf{s}))
\end{array}
\tag{5.8}
$$

Ausgangspunkt dieser Transformation ist ein **LOOP**-Ausdruck mit einem Anfangszustand $\langle$start$\rangle$ und einer Zustandsliste ***L*** (Zeile 2). Voraussetzung für die Anwendbarkeit des Theorems ist, dass der zu verschiebende Zustand des Bezeichners ***s*** auch tatsächlich in der Zustandsliste ***L*** vorkommt (Zeile 1). Ergebnis der Transformation ist ein **LOOP**-Ausdruck mit einer Liste, an deren Anfang nun der Zustand $\langle$s$\rangle$ steht (Zeile 3). Der Rumpf dieses Zustands wird dabei mit Hilfe der Funktion **StateCase** aus der ursprünglichen Liste ***L*** ermittelt. Der Rest der neuen Liste wird durch den Ausdruck **DeleteState** ***L*** **s** gebildet. Dieser Ausdruck entspricht der ursprünglichen Liste ***L***, aus welcher der Zustand $\langle$s$\rangle$ entfernt wurde.

Nach der Überführung einer Verhaltensspezifikation in die steuerflussorientierte Darstellung enthält diese zunächst nur einen einzigen Zustand, dessen Aktion der gesamten ursprünglichen Spezifikation entspricht (siehe Abschnitt 5.2.1). Im Folgenden werden die Transformationen vorgestellt, mit deren Hilfe die Steuerkonstrukte in den Aktionen der Zustände durch eine Aufspaltung der Zustände und eine adäquate Anpassung von deren Transitionsfunktionen ersetzt werden können. Diese Transformationen werden schließlich durchgeführt, um als Vorbereitung der formalen algorithmischen Synthese zu einer steuerflussorientierten Darstellung der Spezifikation zu gelangen, in welcher die Aktionen der Zustände keinerlei Steuerkonstrukte mehr enthalten, sondern nur noch aus steuerflussfreien **DEF**-Anweisungen bestehen.

Aufspaltung von Zuständen mit Aktionen der Form *A* SER *B*

Ein Zustand mit einer Aktion der Form ***A*** **SER** ***B*** wird in zwei Zustände aufgespalten. Die Aktionen der beiden resultierenden Zustände entsprechen den Operanden des ursprünglichen **SER**-Ausdrucks und die Transitionsfunktion des Zustands mit der Aktion ***A*** bestimmt unbedingt den Zustand mit der Aktion ***B*** als Nachfolger. Die Aufspaltung ist schematisch in Abbildung 5.6 dargestellt.

$$(\mathbf{s}, \boldsymbol{A}\ \mathtt{SER}\ \boldsymbol{B}\ \mathtt{SER\ DEF}\ \boldsymbol{trf}) \overset{Aufspaltung}{\Longrightarrow} \begin{array}{l}(\mathbf{s},\ \boldsymbol{A}\ \mathtt{SER\ DEF}\ (\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\mathbf{n})))\\ (\mathbf{n},\ \boldsymbol{B}\ \mathtt{SER\ DEF}\ \boldsymbol{trf})\end{array}$$

Abbildung 5.6: Aufspaltung von **SER**-Aktionen

Die Schemata, welche u. a. die Transformationen zur Aufspaltung von Zuständen begleiten, stellen die Auswirkungen der einzelnen Transformationen vereinfacht dar. Der Übersichtlichkeit halber zeigen sie immer nur die direkt von der Transformation betroffenen Zustände. Tatsächlich operieren die meisten Transformationen auf der gesamten steuerflussorientierten Darstellung. Die genauen Funktionsweisen der Transformationen werden jeweils anschließend anhand der eingesetzten Theoreme ausführlich erläutert.

Die in Abbildung 5.6 schematisch dargestellte Transformation zur Aufspaltung einer **SER**-Aktion basiert auf dem Theorem 5.9. Ausgangspunkt der Transformation ist eine steuerflussorientierte Darstellung, deren Kopf der Zustandsliste die Form (**s**,***A*** **SER** ***B*** **SER DEF** ***trf***) hat (Zeile 6).

$$\vdash \forall\ \mathbf{n\ s\ start}\ \boldsymbol{A}\ \boldsymbol{B}\ \boldsymbol{trf}\ \boldsymbol{L}. \tag{5.9}$$

```
1  (n ≠ start) ∧
2  Disjoint (s::n::FST (UNZIP L)) ∧
3  (∀ x (p1,p2). ((trf x = (p1,p2)) ⇒ (p2 ≠ n)) ∧
4                (EXISTS (λU. U x = Defined (p1,p2)) (SND (UNZIP L))
5                 ⇒ (p2 ≠ n)))
6  ⟹ (LOOP start ((s,A SER B SER DEF trf)::L) =
7        LOOP start ((s,A SER DEF (λx. (x,n)))::
8                    (n,B SER DEF trf)::L))
```

Im Zuge der Transformation wird dieser Zustand ⟨s⟩ durch zwei neue Zustände ersetzt (Zeile 7-8). Dabei übernimmt einer der neuen Zustände den Bezeichner **s** des ursprünglichen Zustands. In der Aktion dieses Zustands wird der P-Term ***A*** ausgeführt (Zeile 7), der dem linken Operanden des ursprünglichen **SER**-Ausdrucks entspricht. Die Transitionsfunktion **DEF** (λ**x**. (**x**,**n**)) des Zustands bestimmt den zweiten neuen Zustand, welcher den neu eingeführten Bezeichner **n** hat, als unbedingten Nachfolger. In der Aktion des zweiten Zustands kommt der P-Term ***B*** zum Einsatz, der rechte Operand des ursprünglichen **SER**-Ausdrucks. Seinen Nachfolger bestimmt der Zustands ⟨n⟩ auf Basis der gleichen Transitionsfunktion **DEF** ***trf*** wie der ursprüngliche Zustand ⟨s⟩.

Vor der Anwendung des Theorems 5.9 müssen zahlreiche Vorbedingungen erfüllt werden. Zunächst darf der neue Bezeichner **n** nicht dem Bezeichner **start** des Startzustands entsprechen (Zeile 1). Zusätzlich darf die Eindeutigkeit der Zustandsbezeichner mit Einführung des neuen Bezeichners nicht verletzt werden (Zeile 2). Der Ausdruck **FST** (**UNZIP** ***L***) ergibt in diesem Zusammenhang eine Liste der Bezeichner aller Zustände in ***L***, so dass (**s::n::FST** (**UNZIP** ***L***)) alle Bezeichner der geplanten Zieldarstellung auflistet.

Weiter muss sichergestellt werden, dass der Bezeichner **n** ebenfalls noch nicht als Bezeichner eines Endzustands (siehe Abschnitt 3.3.4, Seite 58) eingesetzt wird. Dies wird durch die Bedingungen in den Zeilen 3-5 ausgeschlossen. Sie fordern, dass die Transitionsfunktionen der existierenden Zustände niemals einen Zustand mit dem Bezeichner **n** als Nachfolger auswählen. Die Bedingung in Zeile 3 bezieht sich dabei auf die Transitionsfunktion des zu ersetzenden Zustands ⟨s⟩, während sich der Ausdruck in den Zeilen 4-5 auf die Zustände im Rest ***L*** der Zustandsliste bezieht.

Eine Voraussetzung für die Anwendbarkeit des Theorems 5.9 zur Aufspaltung eines Zustands mit einer Aktion der Form ***A*** **SER** ***B*** ist, dass der Term ***B*** den gleichen Eingangstyp wie alle anderen Zustände der steuerflussorientierten Darstellung hat. Ähnlich wie bei Theorem 5.7 in Abschnitt 5.2.1 zur Einführung der steuerflussorientierten Darstellung ist auch in diesem Fall zumeist eine Typanpassung vor Durchführung der Transformation erforderlich.

Typanpassung vor Aufspaltung von Zuständen mit Aktionen der Form *A* SER *B*: Der Eingangstyp des Terms ***B*** sei γ und der Eingangstyp des Terms ***A*** (und damit der Eingangstyp aller Zustände) sei β. Es werden wie in Abschnitt 5.2.1 beschrieben zwei Hilfsfunktionen eingeführt, welche den Typ γ auf β abbilden und umgekehrt. Diese Funktionen werden zwischen den Termen ***A*** und ***B*** eingefügt, und die jeweils an die Terme ***A*** und ***B*** angrenzende Hilfsfunktion mit dem entsprechenden Term verschmolzen. Die Vorgehensweise soll an folgendem Beispiel verdeutlicht werden.

Gegeben sei die folgende Aktion ***A*** **SER** ***B*** eines Zustands:

```
DEF ( λ(i1, i2).
        let a1 = i1 + i2 in a1 )
SER
DEF ( λa1.
        let b1 = a1 + 7 in
        let b2 = a1 ∗ 2 in
        let c = b1 > b2 in ((b1, b2), c) )
```

Der Eingangstyp des Terms ***A*** ist $num \times num$. Der Eingangstyp des Terms ***B*** ist num und muss somit vor der Aufspaltung des Zustands angepasst werden. Zur Anpassung des Typs werden die Hilfsfunktionen **DEF** (λa1.(a1, hvn1)) und die zugehörige Umkehrfunktion **DEF** (λ(a1, hvn1).a1) erstellt. Die neu eingeführte Hilfsvariable hvn1 hat dabei den Typ num.

Die beiden Funktionen werden zwischen den Termen ***A*** und ***B*** wie folgt eingefügt:

```
DEF ( λ(i1, i2).
        let a1 = i1 + i2 in a1 )
SER
DEF ( λa1.(a1, hvn1) )
SER
DEF ( λ(a1, hvn1).a1 )
SER
DEF ( λa1.
        let b1 = a1 + 7 in
        let b2 = a1 * 2 in
        let c = b1 > b2 in ((b1, b2), c) )
```

Da die beiden zusätzlichen Hilfsfunktionen sich gegenseitig aufheben, also deren aufeinanderfolgende Ausführung der Identität entspricht, ändert diese Erweiterung die Funktion der Spezifikation nicht. Ein entsprechender Beweis wird automatisch im Theorembeweiser erbracht.

Die ursprünglichen Terme ***A*** und ***B*** werden nun mit der jeweils angrenzenden Typanpassung verschmolzen. Man erhält:

```
DEF ( λ(i1, i2).
        let a1 = i1 + i2 in (a1, hvn1) )
SER
DEF ( λ(a1, hvn1).
        let b1 = a1 + 7 in
        let b2 = a1 * 2 in
        let c = b1 > b2 in ((b1, b2), c) )
```

Infolge dieser Transformation haben die resultierenden **DEF**-Terme nun beide den gleichen Eingangstyp β wie der ursprüngliche Term ***A***, so dass einer Aufspaltung des Zustands auf Basis des Theorems 5.9 nichts mehr im Wege steht.

Aufwändiger wird die Typanpassung, wenn wie im folgendem Beispiel der Eingangstyp des Terms ***B*** mehr atomare Typen enthält als der Eingangstyp der Zustände der steuerflussorientierten Darstellung bereitstellt:

```
DEF ( λ(i1, i2).
        let a1 = i1 + i2 in (i1, i2, a1) )
SER
DEF ( λ(i1, i2, a1).
        let b1 = i1 − i2 in
        let b2 = a1 * 2 in
        let c = b1 > b2 in ((b1, b2), c) )
```

Der Eingangstyp des Terms ***A*** (und somit der Eingangstyp aller Zustände) ist wieder $num \times num$. Der Eingangstyp des Terms ***B*** ist in diesem Fall allerdings $num \times num \times num$. Er enthält also einmal mehr den atomaren Typ num als der Eingangstyp der Zustände bereitstellt. In diesem Fall ist eine Typänderung aller Zustände erforderlich. Der Eingangstyp der Zustände muss um einen atomaren Typ num erweitert werden. Eine entsprechende Typanpassung der Zustände einer steuerflussorientierten Darstellung erfolgt auf Basis des Theorems 5.25, dessen Anwendung ausführlich in Abschnitt 5.4 beschrieben wird.

Aufspaltung von Zuständen mit Aktionen der Form `IF` *cnd* `THEN` *A* `ELSE` *B*

Ein Zustand mit einem **`IF`**-Ausdruck als Aktion wird in drei Zustände aufgespalten. Im Ersten der drei resultierenden Zustände wird die Bedingung des **`IF`**-Ausdrucks berechnet. In Abhängigkeit von dem Ergebnis dieser Bedingungsberechnung wird einer der beiden anderen Zustände als Nachfolger ausgewählt. Deren Aktionen entsprechen den beiden Zweigen des ursprünglichen **`IF`**-Ausdrucks. Ein Schema der Transformation ist in Abbildung 5.7 dargestellt.

(**s**, **`IF`** ***cnd*** **`THEN`** ***A*** **`ELSE`** ***B*** **`SER DEF`** ***trf***)
$\overset{Aufspaltung}{\Longrightarrow}$
(**s**, **`DEF`** (λ**x**.(**x**,***cnd*** **x**)) **`SER DEF`** (λ(**x**,**c**).(**x**,**`MUX`**(**c**,**a**,**b**))))
(**a**, ***A*** **`SER DEF`** ***trf***)
(**b**, ***B*** **`SER DEF`** ***trf***)

Abbildung 5.7: Aufspaltung von **`IF`**-Aktionen

Das Fundament dieser Transformation bildet das Theorem 5.10. Der zu transformierende Zustand hat die Form (**s**, **`IF`** ***cnd*** **`THEN`** ***A*** **`ELSE`** ***B*** **`SER DEF`** ***trf***) und steht am Anfang der Zustandsliste (Zeile 6). Im Zuge der Transformation wird dieser Zustand durch drei neue Zustände ersetzt (Zeile 7-10).

$\vdash \forall$***cnd B trf A*** **start** ***L*** **s b a**. (5.10)

```
1   Disjoint [start; a; b] ∧
2   Disjoint (a::b::s::FST (UNZIP L)) ∧
3   (∀q. (SND (trf q) ≠ a) ∧ (SND (trf q) ≠ b)) ∧
4   (∀x (p1,p2). EXISTS (λU. U x = Defined (p1,p2)) (SND (UNZIP L))
5                ⇒ (p2 ≠ a) ∧ (p2 ≠ b))
6   ⟹ (LOOP start ((s,IF cnd THEN A ELSE B SER DEF trf)::L) =
7       LOOP start ((s,DEF (λx.(x,cnd x)) SER
8                          DEF (λ(x,c).(x,MUX(c,a,b)))))::
9                  (a,A SER DEF trf)::
10                 (b,B SER DEF trf)::L))
```

Der erste dieser Zustände (Zeile 7-8) hat den gleichen Bezeichner **s** wie der zu ersetzende Zustand. In der Aktion dieses Zustands wird die Bedingung ***cnd*** $\underline{\mathbf{x}}$ des ursprünglichen **IF**-Ausdrucks berechnet und zusätzlich zur Eingabe $\underline{\mathbf{x}}$ des Zustands an die Transitionsfunktion weitergereicht. In der Transitionsfunktion steuert das Ergebnis der Bedingungsberechnung die Auswahl des Nachfolgers. Ergab die Bedingungsberechnung den Wert *wahr*, so wird als Nachfolger derjenige Zustand ausgewählt, dessen Aktion dem Inhalt des ursprünglichen **THEN**-Zweiges entspricht. Im Theorem ist dies der Zustand ⟨a⟩ (Zeile 9). Ansonsten wird der Zustand ⟨b⟩ ausgewählt, dessen Aktion dem P-Term im **ELSE**-Zweig des ursprünglichen Zustands entspricht. Die Transitionsfunktionen der Zustände ⟨a⟩ und ⟨b⟩ entsprechen der Transitionsfunktion **DEF** ***trf*** des ursprünglichen Zustands ⟨s⟩.

Die Vorbedingungen des Theorems 5.10 stellen wie schon diejenigen des Theorems 5.9 die Eindeutigkeit aller verwendeten Zustandsbezeichner sicher. Die Bedingung in Zeile 1 schreibt vor, dass die Bezeichner **start**, **a** und **b** wechselseitig verschieden sein müssen. In Zeile 2 wird gefordert, dass alle Bezeichner der Zustände in der neu aufzubauenden Zustandsliste (Zeile 7-10) eindeutig zu sein haben.

Eine weitere Bedingung für die Anwendbarkeit des Theorems ist, dass die neuen Bezeichner **a** und **b** noch nicht als Bezeichner eines Endzustands verwendet werden: Die Transitionsfunktion des ursprünglichen Zustands ⟨s⟩ darf niemals **a** oder **b** als Bezeichner des Nachfolgers wählen (Zeile 3). Dies gilt auch für die Zustände im Rest ***L*** der ursprünglichen Zustandsliste (Zeile 4-5). Der Ausdruck **SND** (**UNZIP** ***L***) ergibt hierbei eine Liste der Rümpfe aller Zustände in ***L***. Unter Verwendung der Funktion **EXISTS** wird gefordert, dass diese Rümpfe für alle möglichen Eingaben $\underline{\mathbf{x}}$ niemals **a** oder **b** als Bezeichner ihres Nachfolgers ergeben.

Aufspaltung von Zuständen mit Aktionen der Form WHILE *cnd* DO *R*

Aus der Aufspaltung eines Zustands, dessen Aktion ein **WHILE**-Ausdruck ist, gehen zwei neue Zustände hervor (siehe Abbildung 5.8). In demjenigen der beiden Zustände, welcher den ursprünglichen Zustand ersetzt, wird die Bedingung des **WHILE**-Ausdrucks ausgewertet. Ist die Bedingung *wahr*, so wird als Nachfolger der andere der beiden Zustände bestimmt. Dessen Aktion entspricht dem Rumpf der **WHILE**-Schleife. Nach einmaliger Ausführung des Schleifenrumpfes wählt dieser Zustand wieder unbedingt den ersten Zustand als Nachfolger, in welchem erneut die Berechnung der Schleifenbedingung erfolgt. Ergibt die Schleifenbedingung den Wert *falsch*, wird der gleiche Nachfolger wie durch den ursprünglichen Zustand bestimmt.

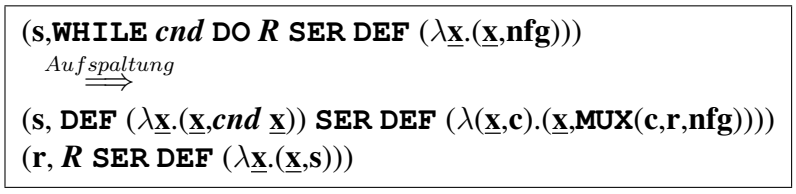


Abbildung 5.8: Aufspaltung von **WHILE**-Aktionen

Zum Aufspalten von **WHILE**-Zuständen wird das Theorem 5.11 verwendet. Ausgangspunkt ist ein Zustand mit einem Bezeichner **s** und einem Rumpf der Form **WHILE** ***cnd*** **DO** ***R*** **SER DEF** (λ**x**.(**x**,**nfg**)) (Zeile 6). Da **WHILE**-Ausdrücke stets einen Typ $\alpha \rightarrow \alpha\, partial$ haben (siehe Abschnitt 3.3.3), haben die Transitionsfunktionen entsprechender Zustände immer den Eingangstyp α, also den gleichen Eingangstyp wie der Zustand selbst. Es werden somit nie Statussignale zusätzlich zum Ergebnis des **WHILE**-Ausdrucks an die Transitionsfunktion übergeben. Daraus folgt, dass der Nachfolger eines Zustands mit einem **WHILE**-Ausdruck als Aktion grundsätzlich eindeutig ist.

$$\vdash \forall \textit{cnd}\ \textbf{nfg}\ \textit{R}\ \textbf{start}\ \textit{L}\ \textbf{s}\ \textbf{r}. \tag{5.11}$$

```
1   Disjoint (r::s::FST (UNZIP L)) ∧
2   (start ≠ r) ∧
3   (nfg ≠ r) ∧
4   (∀x (p1,p2). EXISTS (λU. U x = Defined (p1,p2)) (SND (UNZIP L))
5                ⇒ (p2 ≠ r))
6   ⟹ (LOOP start ((s, WHILE cnd DO R SER DEF (λx.(x,nfg))::L) =
7       LOOP start ((s, DEF (λx.(x,cnd x)) SER
8                        DEF (λ(x,c).(x,MUX(c,r,nfg)))))::
9                   (r,R SER DEF (λx.(x,s)))::L))
```

Der Ausgangszustand im Theorem 5.11 wird im Zuge der Aufspaltung durch zwei neue Zustände ersetzt. Der erste Zustand behält den Bezeichner **s** des zu ersetzenden Zustands bei (Zeile 7-8). In seiner Aktion wird die Schleifenbedingung ***cnd*** ausgeführt. Das Ergebnis dieser Bedingungsberechnung wird als Statussignal zusätzlich zur Eingabe **x** des Zustands an dessen Transitionsfunktion (Zeile 8) weitergereicht. Ergab die Bedingungsberechnung den Wert *wahr*, so wählt die Transitionsfunktion den zweiten neuen Zustand $\langle$r$\rangle$ als Nachfolger (Zeile 9). Ansonsten wird der gleiche Nachfolger $\langle$nfg$\rangle$ wie durch den ursprünglichen Zustand $\langle$s$\rangle$ ausgewählt. Im Zustand $\langle$r$\rangle$ wird der Rumpf ***R*** der Schleife einmalig ausgeführt. Anschließend bestimmt dessen Transitionsfunktion unbedingt wieder den neuen Zustand $\langle$s$\rangle$ als Nachfolger.

Die Vorbedingungen des Theorems 5.11 sind denen des Theorems 5.10 sehr ähnlich. Auch in diesem Fall beziehen sie sich auf die Eindeutigkeit der Zustandsbezeichner. Der Bezeichner **r** muss sich vom Bezeichner **start** des Initialszustands (Zeile 2) wie auch vom Bezeichner **nfg** des Nachfolgers des ursprünglichen Zustands (Zeile 3) unterscheiden. Er darf ebenfalls noch nicht als Bezeichner eines Endzustands in irgendeiner Transitionsfunktion der übrigen existierenden Zustände (Zeile 4-5) verwendet werden. Weiter müssen alle Zustandsbezeichner der resultierenden Zustandsliste eindeutig sein (Zeile 1).

Zum besseren Verständnis wurden im Zusammenhang mit der Aufspaltung von Zuständen Theoreme präsentiert, die Vereinfachungen der tatsächlich in dem formalen Synthesesystem eingesetzten Theoreme darstellen, welche in Hinsicht auf die Effizienz der sie nutzenden Aufspaltungstransformationen optimiert wurden. Bei der Durchführung der Aufspaltungstransformationen auf Basis dieser optimierten Theoreme wird stets das Wissen, dass nach erfolgreicher Durchführung einer Aufspaltung alle Bezeichner der Zustände in der resultierenden Darstellung eindeutig sind, an die nächste Transformation weitergereicht. Dadurch muss eine nachfolgende

Transformation zur Einführung eines weiteren neuen Zustands nicht nochmals die Eindeutigkeit dieser bereits bestehenden Zustände beweisen, sondern nur, dass der neu hinzuzufügende Zustandsbezeichner sich von den bisher existierenden unterscheidet. Der Aufwand für die Durchführung der Aufspaltungstransformationen konnte durch diese Maßnahme beträchtlich reduziert werden.

Abbildung 5.9 zeigt die Spezifikation aus Abbildung 5.5 nach der Aufspaltung aller Zustände, deren Aktionen Steuerkonstrukte enthielten, gemäß der in diesem Abschnitt aufgeführten Transformationen. Als Ergebnis entsprechen alle Aktionen nun steuerflusslosen **DEF**-Anweisungen.

```
DEF (λ(u,n).(u,hvn1,n,hvb1)) SER
LOOP 1
 [(1, DEF (λ(u,hvn1,n,hvb1).(u,1,n,T)) SER DEF (λx.(x,2)));
  (2, DEF (λ(u,v,n,GT1).((u,v,n,GT1),GT1)) SER DEF (λ(x,GT1).(x,MUX(GT1,3,0))));
  (3, DEF (λ(u,v,n,GT1).let GT1 = n> 0 in (u,v,n,GT1)) SER DEF (λx.(x,4)));
  (4, DEF (λ(u,v,n,GT1).((u,v,n,GT1),GT1)) SER DEF (λ(x,GT1).(x,MUX(GT1,5,6))));
  (5, DEF (λ(u,v,n,GT1).let ODD1 = ODD n in (u,v,n,ODD1)) SER DEF (λx.(x,7)));
  (6, DEF (λ(u,v,n,GT1).(u,v,n,F)) SER DEF (λx.(x,2)));
  (7, DEF (λ(u,v,n,ODD1).((u,v,n,ODD1),ODD1))
      SER DEF (λ(x,ODD1).(x,MUX(ODD1,8,9))));
  (8, DEF (λ(u,v,n,ODD1).let MULT1 = u∗v in
                         let SUB1 = n−1 in (u,MULT1,SUB1,T))
      SER DEF (λx.(x,2)));
  (9, DEF (λ(u,v,n,ODD1).let MULT2 = u∗u in
                         let DIV1 = n DIV 2 in (MULT2,v,DIV1,T))
      SER DEF (λx.(x,2)))]
SER
DEF (λ(u,v,n,GT1).v)
```

Abbildung 5.9: Beispielspezifikation nach Aufspaltung der steuerflussbehafteten Zustände

Unter bestimmten Umständen können nach der Aufspaltungsphase Zustände auftreten, deren Aktionen keine Operationen enthalten. Derartige Zustände bewirken ausschließlich eine Transition zu weiteren Zuständen und werden als *operationslos* bezeichnet. Am häufigsten treten entsprechende Zustände nach der Aufspaltung von Zuständen mit **IF**- oder **WHILE**-Konstrukten als Aktionen auf. Enthält zum Beispiel einer der Zweige einer **IF**-Aktion keine Operationen, so ist der Zustand, der nach der Aufspaltung den entsprechenden Zweig repräsentiert, ebenfalls operationslos. Wurde außerdem, wie allgemein üblich, als Vorbereitung der formalen algorithmischen Synthese ein Herausziehen der Bedingungsberechnungen aus den Bedingungsblöcken der **IF**- und **WHILE**-Anweisungen vorgenommen (siehe Abschnitt 4.2.2), so sind die Zustände, die nach einer Aufspaltung der entsprechenden **IF**- bzw. **WHILE**-Zustände deren ursprüngliche Bedingungsblöcke enthalten, ebenfalls operationslos. In der Spezifikation in Abbildung 5.9 sind die Zustände $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 4 \rangle$, $\langle 6 \rangle$ und $\langle 7 \rangle$ operationslos.

Eliminierung operationsloser Zustände

Operationslose Zustände können eliminiert werden, indem sie mit ihren Vorgängern verschmolzen werden. Enthält zum Beispiel ein Zustand ⟨sa⟩ keine Operationen, sondern bewirkt nur eine Transition zu einem Zustand ⟨sb⟩, so resultiert ein Verschmelzen dieses Zustands ⟨sa⟩ mit seinen Vorgängern darin, dass diese, anstatt den Umweg über Zustand ⟨sa⟩ zu nehmen, direkt den Zustand ⟨sb⟩ als Nachfolger wählen.

Hat ein operationsloser Zustand mehrere Vorgänger, so muss dieser Zustand beim Verschmelzen mit einem seiner Vorgänger zunächst erhalten bleiben, da er noch Nachfolger der übrigen Vorgänger ist. Hat der entsprechende Zustand dagegen nur (noch) einen Vorgänger, so wird er beim Verschmelzen mit seinem Vorgänger eliminiert. Somit gibt es zwei Arten des Verschmelzens von Zuständen: eine, bei welcher der operationslose Zustand erhalten bleibt, und eine, bei der er eliminiert wird.

Der Zustand ⟨2⟩ in Abbildung 5.9 ist operationslos. Einer der Vorgänger dieses Zustands ist der Startzustand ⟨1⟩. Im Zustand ⟨2⟩ wird in der Transitionsfunktion eine Bedingung überprüft, welche im Zustand ⟨1⟩ konstant auf *wahr* gesetzt wird. Wird der operationslose Zustand ⟨2⟩ mit seinem Vorgänger ⟨1⟩ verschmolzen, so kann die konstante Verzweigungsbedingung direkt ausgewertet werden. Als Folge wählt der aus der Verschmelzung resultierende neue Zustand ⟨1⟩ nun direkt den Zustand ⟨3⟩ als Nachfolger:

$$(1, \mathbf{DEF}\ (\lambda(u,hvn1,n,hvb1).(u,1,n,\mathbf{T}))\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},3)))$$

Auf diese Weise werden während der Verschmelzung operationsloser Zustände mit ihren Vorgängern konstante Sprungbedingungen jeweils ausgewertet, und so automatisch überflüssige Steuerstrukturen eliminiert. Dieses Vorgehen entspricht der Eliminierung überflüssiger Steuerstrukturen, wie sie auch bei dem externen Steuer-/Datenflussgraphen (siehe Abschnitt 4.4.1, Seite 85) durchgeführt wird.

Eine Sonderstellung nimmt ein operationsloser Zustand ein, wenn er der Startzustand einer steuerflussorientierten Darstellung ist. Soweit er das Ziel anderer Zustände ist, wird er mit diesen verschmolzen. Auf Grund der Tatsache, dass es sich bei dem Zustand um den Startzustand handelt, kann dieser im Zuge der Verschmelzung mit seinen Vorgängern nie eliminiert werden. Für diesen Fall gibt es eine spezielle Transformation, welche den ersten Zustand einer steuerflussorientierten Darstellung unter bestimmten Voraussetzungen vor die **LOOP**-Schleife ziehen kann. Bevor diese Transformation im Detail beschrieben wird, sollen jedoch zuerst die beiden Transformationen zur Verschmelzung zweier Zustände vorgestellt werden.

Verschmelzen von Zuständen

Abbildung 5.10 gibt schematisch das Verschmelzen zweier Zustände wieder, bei dessen Durchführung der zweite der beiden Zustände erhalten bleibt. In dem resultierenden ersten Zustand ⟨sa⟩ werden sowohl die Aktion ***a*** und die Transitionsfunktion ***trfa*** des ursprünglichen Zustands ⟨sa⟩ als auch die entsprechenden Funktionen des zweiten Zustands ⟨sb⟩ ausgeführt. Anschließend wird durch einen Multiplexer überprüft, ob der Zustand ⟨sb⟩ durch die Transitionsfunktion ***trfa*** als Nachfolger des Zustands ⟨sa⟩ ausgewählt wurde oder nicht. Falls ja, so wird (**ob**,**nfb**) selektiert und somit ein Ergebnis ausgegeben, welches der sequentiellen Ausführung

der ursprünglichen Zustände ⟨sa⟩ und ⟨sb⟩ entspricht. Ansonsten wird (**oa**,**nfa**) ausgewählt, so dass das Ergebnis des Zustands der alleinigen Ausführung des ursprünglichen Zustands ⟨sa⟩ entspricht.

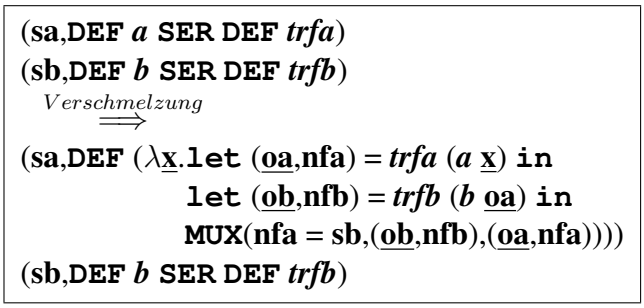


Abbildung 5.10: Verschmelzen zweier Zustände ohne Eliminierung des Zielzustands

Das Verschmelzen von Zuständen ohne Eliminierung des Nachfolgers wird auf Basis des Theorems 5.12 durchgeführt. Ausgangspunkt der Verschmelzung sind die beiden ersten Zustände ⟨sa⟩ und ⟨sb⟩ der Zustandsliste (Zeile 4-5).

$$
\begin{array}{rl}
 & \vdash \forall\ \mathbf{sa\ sb\ start}\ \boldsymbol{a\ b\ trfa\ trfb\ L}. \\
1 & \quad (\mathbf{sb} \neq \mathbf{sa}) \wedge \\
2 & \quad \neg(\texttt{StateExists}\ \boldsymbol{L}\ \mathbf{sb}) \wedge \\
3 & \quad \neg(\texttt{StateExists}\ \boldsymbol{L}\ \mathbf{sa}) \\
4 & \quad \Longrightarrow (\texttt{LOOP}\ \mathbf{start}\ ((\mathbf{sa},\texttt{DEF}\ \boldsymbol{a}\ \texttt{SER DEF}\ \boldsymbol{trfa}) :: \\
5 & \qquad\qquad (\mathbf{sb},\texttt{DEF}\ \boldsymbol{b}\ \texttt{SER DEF}\ \boldsymbol{trfb}) :: \boldsymbol{L}) \\
6 & \qquad = \\
7 & \qquad \texttt{LOOP}\ \mathbf{start}\ ((\mathbf{sa},\texttt{DEF}\ (\lambda\underline{\mathbf{x}}.\texttt{let}\ (\underline{\mathbf{oa}},\mathbf{nfa}) = \boldsymbol{trfa}\ (\boldsymbol{a}\ \underline{\mathbf{x}})\ \texttt{in} \\
8 & \qquad\qquad\qquad\qquad \texttt{let}\ (\underline{\mathbf{ob}},\mathbf{nfb}) = \boldsymbol{trfb}\ (\boldsymbol{b}\ \underline{\mathbf{oa}})\ \texttt{in} \\
9 & \qquad\qquad\qquad\qquad \texttt{MUX}(\mathbf{nfa} = \mathbf{sb},(\underline{\mathbf{ob}},\mathbf{nfb}),(\underline{\mathbf{oa}},\mathbf{nfa})))) :: \\
10 & \qquad\qquad (\mathbf{sb},\texttt{DEF}\ \boldsymbol{b}\ \texttt{SER DEF}\ \boldsymbol{trfb}) :: \boldsymbol{L}))
\end{array}
\tag{5.12}
$$

Der Rumpf des neuen Zustands ⟨sa⟩, welcher das Ergebnis der Verschmelzung der beiden Zustände repräsentiert und den ursprünglichen Zustand ⟨sa⟩ ersetzt, besteht zunächst aus einem einzigen **DEF**-Term. In diesem werden als Erstes auf Basis der Eingabe **x** des Zustands die Aktion ***a*** des ursprünglichen Zustands ⟨sa⟩ ausgeführt und auf das Ergebnis dieser Berechnung gleich im Anschluss daran die Transitionsfunktion ***trfa*** angewandt. Deren Ausgabe wird in der Variablenstruktur (**oa**,**nfa**) abgelegt (Zeile 7). In dem **let**-Ausdruck in Zeile 8 werden dann entsprechend die Aktion ***b*** und Transitionsfunktion ***trfb*** des Zustands ⟨sb⟩ auf die Ausgabe **oa** angewandt. Das Ergebnis dieser Berechnung wird der Variablenstruktur (**ob**,**nfb**) zugewiesen. Daraufhin enthält (**oa**,**nfa**) die Ausgabewerte des ursprünglichen Zustands ⟨sa⟩ und (**ob**,**nfb**) die Ausgabewerte, die sich aus der sequentiellen Ausführung des ursprünglichen Zustands ⟨sa⟩ *und* des Zustands ⟨sb⟩ ergeben. Welches dieser beiden Ausgabenpaare der Multiplexer in Zeile 9 schließlich auswählt, hängt davon ab, ob die Transitionsfunktion ***trfa*** des ursprünglichen Zustands ⟨sa⟩ den Zustand ⟨sb⟩ als Nachfolger bestimmt hat. Falls dies der Fall ist, wenn also

(**nfa = sb**) gilt, so bestimmt der Multiplexer die Variablenstruktur (**ob**,**nfb**) als Ausgabewerte des neuen Zustands ⟨sa⟩, anderenfalls die Variablenstruktur (**oa**,**nfa**).

Die Vorbedingungen des Theorems 5.12 stellen die Eindeutigkeit der Zustandsbezeichner **sa** und **sb** in der Zustandsliste sicher. Sie dürfen sich nicht gleichen (Zeile 1) und müssen sich auch jeweils von allen anderen Zustandsbezeichnern im Rest ***L*** der Zustandsliste unterscheiden (Zeile 2-3).

Soll im Verlauf der Verschmelzung zweier Zustände der Nachfolgezustand eliminiert werden, kommt statt des Theorems 5.12 das Theorem 5.13 zum Einsatz:

$$\vdash \forall\ \mathbf{sa\ sb\ start}\ \textit{a\ b\ trfa\ trfb}\ \boldsymbol{L}. \tag{5.13}$$

```
1   (sb ≠ sa) ∧
2   ¬(StateExists L sb) ∧
3   ¬(StateExists L sa) ∧
4   (sb ≠ start) ∧
5   (∀ y. SND (trfb y) ≠ sb) ∧
6   (∀ x (p1,p2). (EXISTS (λU. U x = Defined (p1,p2)) (SND (UNZIP L))
7                 ⇒ (p2 ≠ sb)))
8   ⟹ (LOOP start ((sa,DEF a SER DEF trfa)::
9                  (sb,DEF b SER DEF trfb)::L)
10      =
11      LOOP start ((sa,DEF (λx.let (oa,nfa) = trfa (a x) in
12                               let (ob,nfb) = trfb (b oa) in
13                               MUX(nfa = sb,(ob,nfb),(oa,nfa))))::L))
```

Zusätzlich zu den Vorbedingungen des Theorems 5.12 (Zeile 1-3) wird im Theorem 5.13 gefordert, dass der zu eliminierende Zustand ⟨sb⟩ nicht Nachfolger weiterer Zustände als des Zustands ⟨sa⟩ ist. So darf ⟨sb⟩ weder Startzustand sein (Zeile 4) noch sich selbst als Nachfolger bestimmen können (Zeile 5). Auch darf keiner der Zustände im Rest ***L*** der Zustandsliste den Zustand ⟨sb⟩ als Nachfolger haben (Zeile 6-7). Ansonsten unterscheidet sich das Theorem 5.13 vom Theorem 5.12 nur dadurch, dass in der resultierenden Zustandsliste der Zustand ⟨sb⟩ nicht mehr auftritt (Zeile 11-13).

Unmittelbar nach der Verschmelzung zweier Zustände wird stets der Rumpf des Zustands, welcher das Ergebnis der Verschmelzung enthält, wieder in eine Aktion und eine Transitionsfunktion aufgeteilt, so dass wieder alle Zustandsrümpfe der steuerflussorientierten Darstellung die Form **DEF** ***aktion*** **SER DEF** ***transitionsfunktion*** haben.

Extraktion des ersten Zustands

Wie bereits erläutert, kann der Startzustand im Zuge des Verschmelzens mit seinen Vorgängern niemals eliminiert werden. Enthält allerdings der Startzustand keine Operation, sondern beispielsweise nur die Initialisierung einiger Werte vor der Ausführung einer anschließenden Schleife, so führte er nach dem in dieser Arbeit entwickelten Konzept zur formalen algorithmischen Synthese zu einer Implementierung auf der RT-Ebene, welche in ihrem ersten Takt nur die

entsprechenden Initialisierungen, aber keinerlei Berechnungen, durchführen würde. Da dies natürlich unerwünscht ist, wurde eine Transformation entwickelt, welche entsprechende Startzustände mit Hilfe des Theorems 5.14 vor deren zugehörige **LOOP**-Schleife verschieben kann.

$$
\begin{array}{ll}
 & \vdash \forall \boldsymbol{L}\ \boldsymbol{U}\ \textbf{zst2}\ \textbf{start}. \qquad (5.14)\\
1 & \quad (\textbf{start} \neq \textbf{zst2}) \wedge \\
2 & \quad \neg(\texttt{StateExists}\ \boldsymbol{L}\ \textbf{start}) \wedge \\
3 & \quad (\forall \underline{\mathbf{x}}\ (\underline{\mathbf{p1}},\mathbf{p2}).\ \texttt{EXISTS}\ (\lambda \boldsymbol{U}.\ \boldsymbol{U}\ \underline{\mathbf{x}} = \mathsf{Defined}\ (\underline{\mathbf{p1}},\mathbf{p2}))\ (\texttt{SND}\ (\texttt{UNZIP}\ \boldsymbol{L})) \\
4 & \qquad\qquad \Rightarrow (\mathbf{p2} \neq \mathbf{start})) \\
5 & \quad \Longrightarrow (\texttt{LOOP}\ \mathbf{start}\ ((\mathbf{start}, \boldsymbol{U}\ \texttt{SER}\ \texttt{DEF}\ (\lambda \underline{\mathbf{x}}.(\underline{\mathbf{x}},\mathbf{zst2}))) :: \boldsymbol{L}) = \\
6 & \qquad \mathrm{U} \\
7 & \qquad \texttt{SER} \\
8 & \qquad \texttt{LOOP}\ \mathbf{zst2}\ \boldsymbol{L})
\end{array}
$$

Ausgangspunkt dieser Transformation ist eine steuerflussorientierte Darstellung mit einem Startzustand, dessen Rumpf aus einer Aktion ***U*** und einer Transitionsfunktion **DEF** ($\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\mathbf{zst2})$) besteht (Zeile 5). Die Transitionsfunktion bestimmt unbedingt einen Zustand $\langle$zst2$\rangle$, den zweiten auszuführenden Zustand, als Nachfolger. Nach Durchführung der Transformation wird die Aktion des ursprünglichen Startzustands vor der **LOOP**-Schleife ausgeführt (Zeile 6). Der neue Startzustand der resultierenden **LOOP**-Schleife ist der Zustand $\langle$zst2$\rangle$, der direkte Nachfolger des ursprünglichen Startzustands.

In den Vorbedingungen des Theorems 5.14 wird gefordert, dass der Zustand $\langle$**start**$\rangle$ weder sich selbst (Zeile 1) noch einen anderen Zustand der restlichen Liste ***L*** (Zeile 3-4) als Vorgänger hat. Weiter muss der Bezeichner **start** (wie üblich) eindeutig sein. Kein anderer Zustand der Liste darf diesen Bezeichner haben (Zeile 2).

Auf Basis der Transformationen zur Verschmelzung der Zustände und zur Extraktion des ersten Zustands werden als Vorbereitung der weiteren Syntheseschritte die gegebenenfalls in der Schaltungsbeschreibung auftretenden operationslosen Zustände eliminiert. Die resultierende Darstellung entspricht dem Steuerflussgraphen der Spezifikation. Dieser gleicht dem aus der in Abschnitt 4.4 vorgestellten Steuerflussanalyse hervorgehenden Steuerflussgraphen der Verhaltensspezifikation bis auf die Tatsache, dass die Zustände in der steuerflussorientierten Darstellung nicht die einzelnen Operationen der Spezifikation, sondern jeweils einen kompletten Grundblock repräsentieren. Die Steuerstruktur der beiden Graphen ist jedoch völlig identisch.

Abbildung 5.11 zeigt die Spezifikation aus Abbildung 5.9 auf Seite 115, nachdem alle operationslosen Zustände durch Verschmelzung mit ihren jeweiligen Vorgängern und durch Extraktion des Startzustands der Spezifikation, der in diesem Beispiel ebenfalls keine Operationen enthält, eliminiert wurden. In dieser Darstellung wurde die Aktion des extrahierten Startzustands bereits mit der der **LOOP**-Schleife vorangehenden Typanpassung zu dem Term **DEF** (λ(u,n).(u,1,n,**T**)) verschmolzen. Ihr entsprechender Graph ist in Abbildung 5.12 dargestellt.

```
DEF (λ(u,n).(u,1,n,T))
SER
LOOP 3
 [(3, DEF (λ(u,v,n,GT1).
            let GT1 = n> 0 in ((u,v,n,GT1),GT1))
     SER DEF (λ(x,GT1).(x,MUX(GT1,5,0))));
  (5, DEF (λ(u,v,n,GT1).
            let ODD1 = ODD n in ((u,v,n,ODD1),ODD1))
     SER DEF (λ(x,ODD1).(x,MUX(ODD1,8,9))));
  (8, DEF (λ(u,v,n,ODD1).
            let MULT1 = u∗v in
            let SUB1 = n−1 in (u,MULT1,SUB1,T))
     SER DEF (λx.(x,3)));
  (9, DEF (λ(u,v,n,ODD1).
            let MULT2 = u∗u in
            let DIV1 = n DIV 2 in (MULT2,v,DIV1,T))
     SER DEF (λx.(x,3)))]
SER
DEF (λ(u,v,n,GT1).v)
```

Abbildung 5.11: Beispielspezifikation nach der Eliminierung operationsloser Zustände

Nachdem die Verhaltensspezifikation entsprechend der in diesem Abschnitt beschriebenen Maßnahmen vorbereitet wurde, liegt sie in einer steuerflussorientierten Darstellung vor, welche sich für eine vollautomatische Umsetzung der gewünschten Entwurfsziele innerhalb des formalen Synthesesystems eignet. Wie bereits in Abschnitt 4.5 erwähnt, können die Synthesealgorithmen, deren Ergebnisse zur Steuerung der algorithmischen Synthese in der formalen Umgebung hinzugezogen werden, in einer beliebigen Reihenfolge oder auch iterativ ausgeführt werden. Die

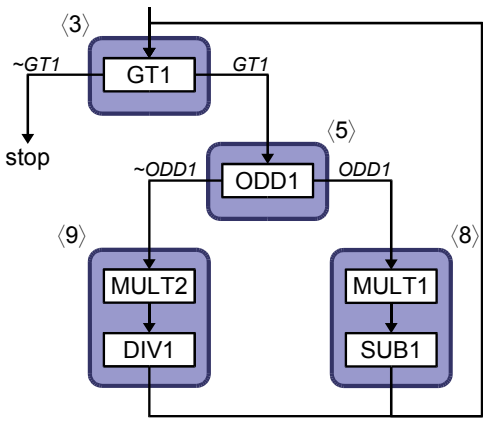


Abbildung 5.12: Steuerflussgraph der steuerflussorientierten Darstellung aus Abb. 5.11

Umsetzung der erzielten Ergebnisse innerhalb des formalen Systems wird allerdings in einer fest vorgegebenen Reihenfolge durchgeführt. Nachteile entstehen hieraus nicht, da die formale Umsetzung erst beginnt, wenn bereits alle Entwurfsziele auf Basis der Synthesealgorithmen ermittelt worden sind.

So werden die Ergebnisse der Synthesealgorithmen innerhalb des Theorembeweisers in folgender Reihenfolge umgesetzt: Zunächst werden die Ergebnisse der Ablaufplanung (Abschnitt 5.3) nachvollzogen. Anschließend erfolgt die Bereitstellung und Zuweisung von Registern (Abschnitt 5.4) sowie die Synthese des Operationswerks (Abschnitt 5.5). Die formale algorithmische Synthese endet mit der Synthese des Steuerwerks (Abschnitt 5.6).

5.3 Ablaufplanung

Die Umsetzung der Ablaufplanung wird in der formalen Synthese durch die Überführung der Schaltungsspezifikation in eine steuerflussorientierte Darstellung realisiert, die exakt den Vorgaben eines für diese Spezifikation zuvor ermittelten Ablaufplans entspricht [KaSa04]. Der Ablaufplan gibt vor, in welchem Takt der resultierenden RT-Ebenen-Implementierung die einzelnen Operationen der Spezifikation zur Ausführung kommen sollen. Sein genauer Aufbau wurde bereits in Abschnitt 4.6 vorgestellt.

Da die Umsetzung der Ablaufplanung bei reinen Datenflussspezifikationen deutlich einfacher ist als bei steuerflussbehafteten Schaltungsspezifikationen, können einige der Schritte, welche bei der Ablaufplanung steuerflussbehafteter Schaltungsspezifikationen anfallen, eingespart werden. Aus Effizienzgründen wird daher bei Datenflussspezifikationen eine vereinfachte Vorgehensweise angewandt. Diese wird ausführlich in Abschnitt 5.3.5 beschrieben. Die prinzipielle Vorgehensweise der formalen Umsetzung ist jedoch bei beiden Arten von Schaltungsspezifikationen identisch.

Im Folgenden wird zunächst das Verfahren beschrieben, welches im Zuge dieser Arbeit zur vollautomatischen Umsetzung eines Ablaufplans für steuerflussorientierte Schaltungsspezifikationen innerhalb des formalen Synthesesystems entwickelt wurde. Das Verfahren ist für die Umsetzung von Ablaufplänen ausgelegt, die aus dem Einsatz von Ablaufplanungsalgorithmen hervorgehen, welche zur Erzielung hochwertiger Ergebnisse u. a. Techniken wie das Entrollen von Schleifen [GoVM89], *Chaining* sowie *Loop Pipelining* [SaCo95] einsetzen und dabei stets die vom Entwerfer spezifizierte Ausführungsreihenfolge der einzelnen Operationen exakt aufrechterhalten [RaJe95b]. Damit können eine Vielzahl hochqualitativer Ablaufplanungsalgorithmen in den formalen Syntheseprozess integriert werden. Bekannte Beispiele derartiger Algorithmen sind der *Path-Based-Scheduling*-Algorithmus [Camp91], der *Pipeline-Path-Based-Scheduling*-Algorithmus [RaJe95] sowie der *Loop-Directed-Scheduling*-Algorithmus [BhDB94]. Letzterer wurde exemplarisch in dieser Arbeit implementiert.

In der formalen Synthese erfolgt die Ablaufplanung steuerflussbehafteter Schaltungsspezifikationen nach den Vorgaben eines zuvor ermittelten Ablaufplans in vier Schritten:

1. Aufspalten der Zustände der steuerflussorientierten Darstellung gemäß der Aufteilung der jeweils entsprechenden Grundblöcke im Ablaufplan (siehe Abschnitt 5.3.1)

2. Kopieren der Zustände, so dass deren Anzahl mit der Anzahl der entsprechenden Grundblockteile im Ablaufplan übereinstimmt (siehe Abschnitt 5.3.2)

3. Nachbildung der Übergänge zwischen den Grundblockteilen im Ablaufplan durch Modifikation der Transitionsfunktionen der Zustände (siehe Abschnitt 5.3.3)

4. Verschmelzen der Zustände der steuerflussorientierten Darstellung zur Nachbildung der Zustände des Ablaufplans (siehe Abschnitt 5.3.4)

Ziel dieser Schritte ist die Überführung der Spezifikation in eine steuerflussorientierte Darstellung, die exakt den vorgegebenen Ablaufplan widerspiegelt. Als begleitendes Beispiel dieser Vorgehensweise soll im Folgenden die Spezifikation in Abbildung 5.11 den Vorgaben des Ablaufplans in Abbildung 5.13 entsprechend umgeformt werden.

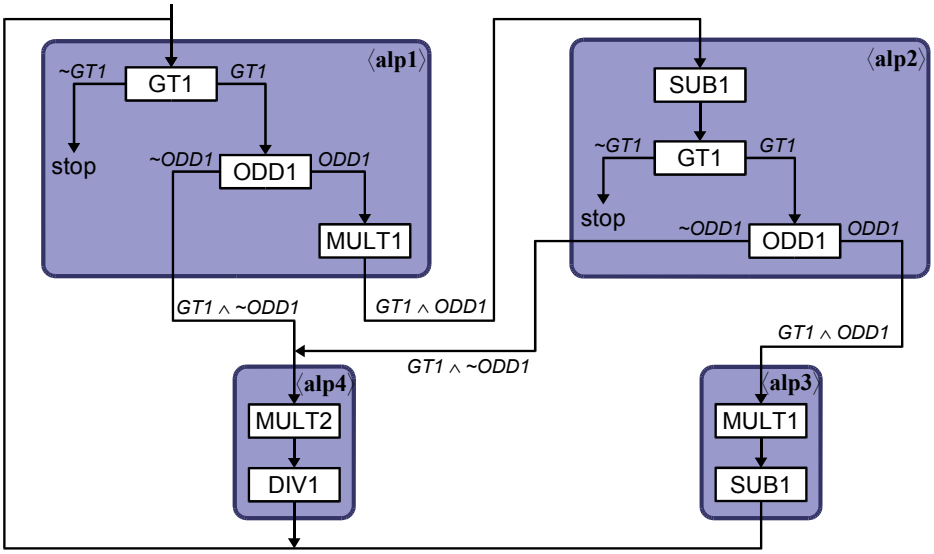


Abbildung 5.13: Ein Ablaufplan für die Spezifikation in Abbildung 5.11

5.3.1 Aufteilung der Grundblöcke

Der erste Schritt zur Umsetzung eines Ablaufplans in der formalen Synthese beginnt mit der Analyse des Ablaufplans. Nach der in Abschnitt 5.2 dargestellten Überführung der Verhaltensspezifikation in eine steuerflussorientierte Darstellung entsprechen die Aktionen der resultierenden Zustände jeweils einem Grundblock der Spezifikation. Nun wird untersucht, wie die Operationen dieser Grundblöcke im Ablaufplan verteilt sind. Für jeden Grundblock werden dessen kleinste Anteile ermittelt, die als Bestandteil eines Zustands im Ablaufplan auftreten. Aus der Analyse ergibt sich eine Menge der kleinsten Grundblockteile, die zur Umsetzung der Ablaufplanung in der formalen Synthese erforderlich sind.

Abbildung 5.12 auf Seite 120 stellt den Steuerflussgraphen der Beispielspezifikation aus Abbildung 5.11 dar. Während die Zustände $\langle 3 \rangle$ und $\langle 5 \rangle$ jeweils nur aus einer einzigen Operation

bestehen, entsprechen die Zustände ⟨8⟩ und ⟨9⟩ Grundblöcken mit jeweils zwei Operationen. In dem Ablaufplan in Abbildung 5.13 kommen die Operationen des Zustands ⟨9⟩ der steuerflussorientierten Darstellung ausschließlich gemeinsam vor, nämlich im Zustand ⟨alp4⟩. Die Operationen des Zustands ⟨8⟩ kommen dagegen im Ablaufplan sowohl gemeinsam im Zustand ⟨alp3⟩ als auch aufgeteilt auf die Zustände ⟨alp1⟩ und ⟨alp2⟩ vor. Die kleinsten Anteile dieses Grundblocks, welche in den Zuständen des Ablaufplans verwendet werden, enthalten somit entweder die Operation MULT1 oder die Operation SUB1.

Nach Durchführung der Analyse werden die Zustände der steuerflussorientierten Darstellung gemäß der Aufteilung der Operationen im Ablaufplan aufgespalten. Dabei wird jede Aktion, deren Operationen im Ablaufplan auf verschiedene Zustände verteilt zur Ausführung kommen, derart in mehrere mit dem Konstrukt **SER** verknüpfte **DEF**-Terme überführt, dass jeder dieser **DEF**-Terme einem der auftretenden Grundblockteile entspricht. Anschließend werden die Zustände, deren Aktionen aufgeteilt wurden, wie in Abschnitt 5.2.2 beschrieben in mehrere Zustände aufgespalten (siehe Seite 109). Die Zustände der resultierenden steuerflussorientierten Darstellung enthalten daraufhin jeweils einen der für die Durchführung der folgenden Schritte erforderlichen Grundblockteile.

Da im Beispiel die Operationen des Zustands ⟨8⟩ der steuerflussorientierten Darstellung in der Ablaufplanung u. a. auf zwei Zustände verteilt zum Einsatz kommen, muss dieser in zwei Zustände aufgespalten werden, von denen jeder eine der beiden Operationen enthält. Zu diesem Zweck wird zunächst die Aktion des Zustands nach folgendem Schema in zwei **DEF**-Terme aufgeteilt:

DEF (λ(u,v,n,ODD1).
let MULT1 = u∗v **in**
let SUB1 = n−1 **in**
(u,MULT1,SUB1,**T**))

$\overset{Aufteilung}{\Longrightarrow}$

DEF (λ(u,v,n,ODD1).
let MULT1 = u∗v **in**
(u,MULT1,n,ODD1))
SER
DEF (λ(u,MULT1,n,ODD1).
let SUB1 = n−1 **in**
(u,MULT1,SUB1,**T**))

Daraufhin hat die Aktion des Zustands die Form ***A*** **SER** ***B*** und kann wie in Abschnitt 5.2.2 beschrieben aufgespalten werden.

Anstatt die Grundblöcke erst in der steuerflussorientierten Darstellung aufzuteilen, können diese auch bereits in der Verhaltensbeschreibung aufgespalten werden. Dies hat den Vorteil, dass schon vor der Überführung der Verhaltensspezifikation in die steuerflussorientierte Darstellung die Schnittstellen bzw. Typen zwischen den einzelnen GROPIUS-Anweisungen analysiert und vereinheitlicht werden können. Nach der Vereinheitlichung erhält man eine Verhaltensspezifikation, deren Anweisungen, bis auf die erste und letzte Anweisung, alle den gleichen Ein-/Ausgabetyp haben. Die erste und letzte Anweisung sind **DEF**-Terme, welche den Ein- bzw. Ausgabetyp wie in Abschnitt 5.2.1 beschrieben an den Einheitstyp anpassen. Die Vereinheitlichung der Typen der internen GROPIUS-Anweisungen hat den Vorteil, dass beim Aufspalten der Zustände, welche im Zuge des Übergangs zur steuerflussorientierten Darstellung durchgeführt wird, ansonsten gegebenenfalls erforderliche Typanpassungen der Zustände entfallen. Da die Grundblöcke bereits

in der Verhaltensbeschreibung wunschgemäß aufgeteilt wurden, enthalten die Zustände der resultierenden steuerflussorientierten Darstellung automatisch jeweils einen der benötigten Grundblockteile. Diese alternative Vorgehensweise zur Aufteilung der Grundblockteile wird auf Grund ihrer höheren Effizienz bevorzugt verwendet. Das Ergebnis ist bei beiden Verfahrensweisen das gleiche.

Abbildung 5.14 stellt den Steuerflussgraph der steuerflussorientierten Darstellung aus Abbildung 5.11 nach der Aufteilung des Grundblocks in Zustand ⟨8⟩ dar. Der ursprüngliche Zustand ⟨8⟩ wurde in einen modifizierten Zustand ⟨8⟩, der nur noch die erste Operation der ursprünglichen Aktion enthält, und einen neuen Zustand ⟨10⟩, welcher den zweiten Grundblockteil enthält, aufgeteilt. Damit liegt die Schaltungsspezifikation in einer steuerflussorientierten Darstellung vor, in welcher jeder Zustand jeweils einen der zur Umsetzung der Ablaufplanung erforderlichen Grundblockteile enthält.

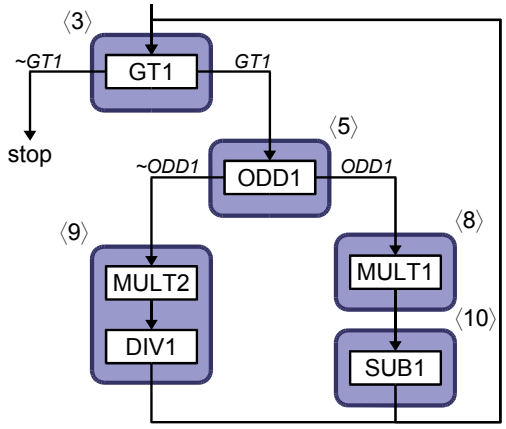


Abbildung 5.14: Steuerflussgraph der Schaltungsspezifikation nach Grundblockaufteilung

5.3.2 Kopieren der Grundblockteile

Nach der Aufspaltung der Zustände der steuerflussorientierten Darstellung gemäß der Verteilung der Operationen im Ablaufplan wird untersucht, wie oft jeder Grundblockteil im Ablaufplan auftritt. Daraufhin werden in der steuerflussorientierten Darstellung durch das Kopieren der Zustände, welche die jeweiligen Grundblockteile repräsentieren, eine entsprechende Anzahl an Instanzen der Grundblockteile angefertigt.

Für das laufende Beispiel listet Tabelle 5.1 zu jedem Grundblockteil die Anzahl seiner Vorkommen im Ablaufplan in Abbildung 5.13 auf. Jeder Zustand der steuerflussorientierten Darstel-

Grundblockteil	GT1	ODD1	MULT1	MULT2, DIV1	SUB1
Zustand	⟨3⟩	⟨5⟩	⟨8⟩	⟨9⟩	⟨10⟩
Anzahl	2	2	2	1	2

Tabelle 5.1: Anzahl der Grundblockteile

lung bis auf den Zustand $\langle 9 \rangle$ wird demnach zweimal zur Nachbildung des Ablaufplans benötigt und muss somit jeweils einmal kopiert werden.

Beim Kopieren von Zuständen wird die *Ähnlichkeit* zwischen der Kopie und deren jeweiligem Quellzustand in einer speziellen Liste vermerkt. Die Eigenschaft der Ähnlichkeit wird für die Durchführung des im nächsten Abschnitt beschriebenen Schritts, der Nachbildung der Übergangsstruktur des Ablaufplans, benötigt. Unter Ausnutzung dieser Eigenschaft ist es möglich, den Übergang von einem Zustand zu einem seiner Nachfolger auf einen anderen Zustand umzuleiten, falls der vorgesehene Ersatzzustand dem ursprünglichen Nachfolger ähnlich ist.

Die Ähnlichkeit zweier Zustände ist auf Basis ihrer Rümpfe definiert. Zwei Zustände sind genau dann ähnlich, wenn entweder ihre Rümpfe identisch sind oder ihre Rümpfe für jede Eingabe das gleiche Ergebnis bis auf die Folgezustände liefern, welche nur ähnlich, nicht aber identisch, zu sein haben. Daraus folgt, dass die Kopie eines Zustands grundsätzlich dem Originalzustand ähnelt, da die Kopie den gleichen Rumpf wie der Originalzustand hat.

Vor der Definition der Ähnlichkeit selbst ist die Definition einer sogenannten *Ähnlichkeitsliste* erforderlich, in welcher die Ähnlichkeiten zwischen den Zuständen vermerkt wird. Es handelt sich dabei um eine Liste, deren Elemente Paare von Zustandsbezeichnern sind. Jedes Paar ordnet einem Zustand, dessen Bezeichner den linken Teil des Paares bildet, eine Zustandsklasse zu, welche durch den Bezeichner im rechten Teil des Paars repräsentiert wird. Zum Auslesen der Ähnlichkeitsklasse eines Zustands aus einer Ähnlichkeitsliste dient die Funktion **`SimClass`**:

$$\begin{aligned} &(\ \texttt{SimClass}\ ((q,c)::R)\ s := (\text{if } s = q \text{ then } c \text{ else } \texttt{SimClass}\ R\ s)\ ) \wedge \\ &(\ \texttt{SimClass}\ [\,]\ s \qquad\quad := s\ ) \end{aligned} \tag{5.15}$$

Die Funktion **`SimClass`** überprüft, ob der Bezeichner s des Zustands, dessen Ähnlichkeitsklasse bestimmt werden soll, dem linken Teil des Kopfs (q,c) der Ähnlichkeitsliste entspricht. Falls ja, so wird der rechte Teil c als Ähnlichkeitsklasse ausgegeben. Ansonsten wird die Ähnlichkeitsliste weiter rekursiv durchsucht. Kommt der Bezeichner s niemals als linker Teil eines der Paare der Liste vor, so wird der Zustandsbezeichner selbst als Ähnlichkeitsklasse ausgegeben. Solange also ein Zustand nicht einer anderen Ähnlichkeitsklasse zugeordnet ist, bildet er unter seinem eigenen Bezeichner seine eigene Ähnlichkeitsklasse.

Die Ähnlichkeit zweier Zustände ist nun auf Basis des Prädikats **`StateSim`** wie folgt definiert. Zwei Zustände mit den Rümpfen ***R1*** und ***R2*** sind in Bezug auf eine Ähnlichkeitsliste ***SimList*** genau dann ähnlich, wenn sie das Prädikat **`StateSim`** erfüllen:

$$\begin{aligned} &\texttt{StateSim}\ \mathit{SimList}\ \mathit{R1}\ \mathit{R2} := \\ 1\quad &\forall x.\ ((\mathit{R1}\ x = \mathsf{Undefined}) \wedge (\mathit{R2}\ x = \mathsf{Undefined})) \vee \\ 2\quad &\quad \big(\exists\, \mathit{x1}\ \mathit{n1}\ \mathit{x2}\ \mathit{n2}. \\ 3\quad &\quad\quad (\mathit{R1}\ x = \mathsf{Defined}\ (\mathit{x1},\mathit{n1})) \wedge (\mathit{R2}\ x = \mathsf{Defined}\ (\mathit{x2},\mathit{n2})) \wedge \\ 4\quad &\quad\quad (\mathit{x1} = \mathit{x2}) \wedge \\ 5\quad &\quad\quad (\texttt{SimClass}\ \mathit{SimList}\ \mathit{n1} = \texttt{SimClass}\ \mathit{SimList}\ \mathit{n2})\ \big) \end{aligned} \tag{5.16}$$

Für alle Eingaben x terminiert die Ausführung der beiden Rümpfe ***R1*** sowie ***R2*** entweder nicht (Zeile 1) oder beide Berechnungen terminieren mit Werten $(x1,n1)$ bzw. $(x2,n2)$ (Zeile 2-3), so dass die Werte ***x1*** und ***x2*** gleich sind (Zeile 4) und die resultierenden Bezeichner ***n1*** und ***n2*** der

Nachfolger der beiden Zustände auf ähnliche Zustände verweisen, also auf Zustände, welche der gleichen Ähnlichkeitsklasse angehören (Zeile 5).

Damit eine Liste für die Zustände einer steuerflussorientierten Darstellung eine gültige Ähnlichkeitsrelation darstellt, muss sie einige Eigenschaften aufweisen. Diese wurden in dem Prädikat **SimRel** zusammengefasst:

$$
\begin{array}{ll}
 & \textbf{SimRel}\ \boldsymbol{L}\ \boldsymbol{SimList} := \\
1 & \big(\ \textbf{ALL\_EL}\ \big(\lambda \boldsymbol{el}.\ (\neg\ \textbf{MEM}\ (\textbf{FST}\ \boldsymbol{el})\ (\textbf{SND}\ (\textbf{UNZIP}\ \boldsymbol{SimList}))) \wedge \\
2 & \qquad\qquad (\neg\ \textbf{MEM}\ (\textbf{SND}\ \boldsymbol{el})\ (\textbf{FST}\ (\textbf{UNZIP}\ \boldsymbol{SimList}))) \wedge \\
3 & \qquad\qquad (\textbf{StateExists}\ \boldsymbol{L}\ (\textbf{FST}\ \boldsymbol{el})) \wedge \\
4 & \qquad\qquad (\textbf{StateExists}\ \boldsymbol{L}\ (\textbf{SND}\ \boldsymbol{el}))\big)\ \boldsymbol{SimList}\ \big) \wedge \\
5 & (\textbf{Disjoint}\ (\textbf{FST}\ (\textbf{UNZIP}\ \boldsymbol{SimList}))) \wedge \\
6 & \big(\ \forall\ \boldsymbol{s1}\ \boldsymbol{s2}. \\
7 & \quad \neg\ (\boldsymbol{s1} = \boldsymbol{s2}) \wedge (\textbf{SimClass}\ \boldsymbol{SimList}\ \boldsymbol{s1} = \textbf{SimClass}\ \boldsymbol{SimList}\ \boldsymbol{s2}) \\
8 & \quad \Longrightarrow \textbf{StateSim}\ \boldsymbol{SimList}\ (\textbf{StateCase}\ \boldsymbol{L}\ \boldsymbol{s1})\ (\textbf{StateCase}\ \boldsymbol{L}\ \boldsymbol{s2})\ \big)
\end{array}
\tag{5.17}
$$

Eine Liste ***SimList*** repräsentiert demnach eine Ähnlichkeitsliste einer steuerflussorientierten Darstellung mit einer Zustandsliste ***L***, wenn keiner der Zustandsbezeichner sowohl den linken Teil eines der Paare der Ähnlichkeitsliste bildet als auch als rechter Teil eines der Paare auftritt (Zeile 1-2). Weiter muss zu jedem Bezeichner, der in der Ähnlichkeitsliste vorkommt, der entsprechende Zustand in der Zustandsliste ***L*** der steuerflussorientierten Darstellung existieren (Zeile 3-4). Jedem Zustand darf in der Ähnlichkeitsliste nur einmalig eine Ähnlichkeitsklasse zugeordnet werden, d. h. die Bezeichner im linken Teil der Paare der Liste müssen eindeutig sein (Zeile 5). Letztendlich muss für alle unterschiedlichen Zustandsbezeichner, deren Zustände der gleichen Ähnlichkeitsklasse angehören (Zeile 7), gelten, dass die entsprechenden Zustände $\langle$s1$\rangle$ und $\langle$s2$\rangle$ auch tatsächlich ähnlich zueinander sind (Zeile 8).

Trivialerweise ist die leere Liste eine gültige Ähnlichkeitsliste für die Zustandslisten beliebiger steuerflussorientierter Darstellungen:

$$\vdash \forall\ \boldsymbol{L}.\ \textbf{SimRel}\ \boldsymbol{L}\ [\,] \tag{5.18}$$

Die leere Liste bildet den Ausgangspunkt für den Aufbau einer Ähnlichkeitsliste, wie er während des Kopierens der Zustände vorgenommen wird. Eine Änderung oder Erweiterung einer Ähnlichkeitsliste erfolgt während des Kopiervorgangs auf Basis der Funktion **ExpSimList**:

$$
\begin{array}{l}
(\ \textbf{ExpSimList}\ \boldsymbol{class}\ \boldsymbol{s}\ [\,] \qquad := [(\boldsymbol{s}, \boldsymbol{class})]\ ) \wedge \\
(\ \textbf{ExpSimList}\ \boldsymbol{class}\ \boldsymbol{s}\ (\boldsymbol{h}::\boldsymbol{R}) := (\text{if}\ \boldsymbol{s} = \textbf{FST}\ \boldsymbol{h}\ \text{then}\ (\boldsymbol{s}, \boldsymbol{class})::\boldsymbol{R} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else}\ \boldsymbol{h}::\textbf{ExpSimList}\ \boldsymbol{class}\ \boldsymbol{s}\ \boldsymbol{R})\ )
\end{array}
\tag{5.19}
$$

Die Funktion **ExpSimList** überprüft rekursiv die existierende Ähnlichkeitsliste, ob einem Zustand mit dem Bezeichner ***s***, dem die neue Ähnlichkeitsklasse ***class*** zugewiesen werden soll, bereits eine andere Ähnlichkeitsklasse zugeordnet wird. Falls ja, so wird dessen Ähnlichkeitsklasse auf die neue Klasse ***class*** abgeändert. Ansonsten wird der Liste ein Element (***s***, ***class***) angehängt, in welchem dem Zustand $\langle \boldsymbol{s} \rangle$ die Ähnlichkeitsklasse ***class*** zugeordnet wird.

Auf Basis der soeben definierten Funktionen und Prädikate kann nun das Theorem 5.20 formuliert werden, welches die Grundlage für die Transformation zum Kopieren der Zustände einer steuerflussorientierten Darstellung bildet.

$$\vdash \forall\ \textit{SimList}\ \mathbf{start}\ \textit{L}\ \mathbf{q}\ \mathbf{z}. \tag{5.20}$$

1 $\quad$ `SimRel` $\textit{L}\ \textit{SimList} \wedge$
2 $\quad$ `StateExists` $\textit{L}\ \mathbf{q} \wedge$
3 $\quad \neg$ `StateExists` $\textit{L}\ \mathbf{z} \wedge$
4 $\quad (\mathbf{start} \neq \mathbf{z}) \wedge$
5 $\quad (\forall \underline{\mathbf{x}}\ (\underline{\mathbf{p1}},\mathbf{p2}).$ `EXISTS` $(\lambda \textit{U}.\ \textit{U}\ \underline{\mathbf{x}} = \mathsf{Defined}\ (\underline{\mathbf{p1}},\mathbf{p2}))$ (`SND` (`UNZIP` $\textit{L}$))
6 $\quad\quad\quad \Rightarrow (\mathbf{p2} \neq \mathbf{z}))$
7 $\quad \Longrightarrow \big($`let` $\textit{L}' = ((\mathbf{z},$ `StateCase` $\textit{L}\ \mathbf{q})::\textit{L})$ `in`
8 $\quad\quad$ `let` $\textit{SimList}'$ = `ExpSimList` (`SimClass` $\textit{SimList}\ \mathbf{q})\ \mathbf{z}\ \textit{SimList}$ `in`
9 $\quad\quad$ (`LOOP` $\mathbf{start}\ \textit{L}$ = `LOOP` $\mathbf{start}\ \textit{L}') \wedge$
10 $\quad\quad$ (`SimRel` $\textit{L}'\ \textit{SimList}'$)$\big)$

In dem Theorem wird der bestehenden Zustandsliste $\textit{L}$ eine Kopie des Quellzustands $\langle\mathbf{q}\rangle$ vorangestellt (Zeile 7). Die Kopie hat den gleichen Rumpf (`StateCase` $\textit{L}\ \mathbf{q}$) wie der Quellzustand und einen neuen eindeutigen Bezeichner **z**. Gleichzeitig wird die Ähnlichkeitsliste ***SimList*** um einen neuen Eintrag erweitert, in welchem dem neuen Zustand $\langle\mathbf{z}\rangle$ die gleiche Ähnlichkeitsklasse (`SimClass` ***SimList*** **q**) wie dem Quellzustand $\langle\mathbf{q}\rangle$ zugeordnet wird (Zeile 8). Nach Durchführung der Transformation besagt das resultierende Theorem, dass das Ersetzen der Liste $\textit{L}$ durch die erweiterte Liste $\textit{L}'$ in der steuerflussorientierten Darstellung korrekt ist (Zeile 9) und die modifizierte Ähnlichkeitsliste $\textit{SimList}'$ eine gültige Ähnlichkeitsrelation für die neue Zustandsliste $\textit{L}'$ bildet (Zeile 10).

Voraussetzung für die Anwendung des Theorems 5.20 ist, dass die Ähnlichkeitsliste ***SimList*** eine gültige Ähnlichkeitsrelation für die gegebene steuerflussorientierte Darstellung $\textit{L}$ repräsentiert (Zeile 1). Als Ausgangspunkt für die Ähnlichkeitsliste wird dabei bei der Kopie des ersten Zustands die leere Liste genommen, die nach Theorem 5.18 eine gültige Ähnlichkeitsliste für alle möglichen Zustandslisten repräsentiert. Die aus der Kopie resultierende Ähnlichkeitsliste $\textit{SimList}'$ wird daraufhin als Eingabe des nächsten Kopiervorgangs verwendet.

Weitere Vorbedingungen des Theorems 5.20 sind, dass der zu kopierende Zustand $\langle\mathbf{q}\rangle$ bereits existiert (Zeile 2), während der Bezeichner **z** des Zielzustands sich von allen Bezeichnern der Zustände in der Zustandsliste (Zeile 3) und ebenso vom Bezeichner des Startzustands unterscheiden muss (Zeile 4) und ebensowenig bereits als Nachfolger eines der Zustände in der Liste verwendet werden darf (Zeile 5-6).

Unter Einsatz dieses Theorems werden nun die Zustände der steuerflussorientierten Darstellung entsprechend der Anzahl der jeweiligen Grundblockteile im Ablaufplan, die sie repräsentieren, vervielfältigt. Die Kopie des Zustands $\langle 3\rangle$ erhält den Bezeichner **12**, des Zustands $\langle 5\rangle$ den Bezeichner **13**, des Zustands $\langle 8\rangle$ den Bezeichner **14** und des Zustands $\langle 10\rangle$ den Bezeichner **11**. Alle diese Kopien haben den gleichen Rumpf wie ihre Originale und verweisen somit auch auf die gleichen Nachfolger wie die Originalzustände. Sie haben zu diesem Zeitpunkt noch keine Vorgänger. In der Ähnlichkeitsliste, die sich aus den Kopiervorgängen er-

gibt, werden die Ähnlichkeiten zwischen den resultierenden Zuständen verzeichnet. Im gegebenen Beispiel lautet die Ähnlichkeitsliste nach der Durchführung der oben genannten Kopien: [(**12**,**3**);(**13**,**5**);(**14**,**8**);(**11**,**10**)].

```
LOOP 3
 [( 3, DEF (λ(u,v,n,GT1).
            let GT1 = n> 0 in ((u,v,n,GT1),GT1))
       SER DEF (λ(x,GT1).(x,MUX(GT1,5,0))));
  ( 5, DEF (λ(u,v,n,GT1).
            let ODD1 = ODD n in ((u,v,n,ODD1),ODD1))
       SER DEF (λ(x,ODD1).(x,MUX(ODD1,8,9))));
  ( 8, DEF (λ(u,v,n,ODD1).
            let MULT1 = u∗v in (u,MULT1,n,ODD1))
       SER DEF (λx.(x,10)));
  ( 9, DEF (λ(u,v,n,ODD1).
            let MULT2 = u∗u in
            let DIV1 = n DIV 2 in (MULT2,v,DIV1,T))
       SER DEF (λx.(x,3)));
  (10, DEF (λ(u,MULT1,n,ODD1).
            let SUB1 = n−1 in (u,MULT1,SUB1,T))
       SER DEF (λx.(x,3)));
  (11, DEF (λ(u,MULT1,n,ODD1).
            let SUB1 = n−1 in (u,MULT1,SUB1,T))
       SER DEF (λx.(x,3)));
  (12, DEF (λ(u,v,n,GT1).
            let GT1 = n> 0 in ((u,v,n,GT1),GT1))
       SER DEF (λ(x,GT1).(x,MUX(GT1,5,0))));
  (13, DEF (λ(u,v,n,GT1).
            let ODD1 = ODD n in ((u,v,n,ODD1),ODD1))
       SER DEF (λ(x,ODD1).(x,MUX(ODD1,8,9))));
  (14, DEF (λ(u,v,n,ODD1).
            let MULT1 = u∗v in (u,MULT1,n,ODD1))
       SER DEF (λx.(x,3)))]
```

Abbildung 5.15: **LOOP**-Term der Beispielspezifikation nach dem Kopieren der Grundblockteile

Abbildung 5.15 enthält die Beispielspezifikation nach dem Aufteilen der Grundblöcke und der Anfertigung der erforderlichen Kopien der Grundblockteile. Es wird nur das **LOOP**-Konstrukt dargestellt. Der führende und der abschließende **DEF**-Term wurden der Übersichtlichkeit halber weggelassen.

5.3.3 Nachbildung der Übergangsstruktur des Ablaufplans

Nachdem alle Operationen bzw. Grundblockteile in der erforderlichen Zahl bereitstehen, kann in einem weiteren Schritt die Übergangsstruktur zwischen den einzelnen Grundblockteilen nach Vorbild des Ablaufplans in der steuerflussorientierten Darstellung nachgebildet werden. Abbildung 5.16 gibt den strukturellen Aufbau des Ablaufplans des laufenden Beispiels wieder. Auf die besondere Bedeutung der gestrichelten Übergänge wird später eingegangen.

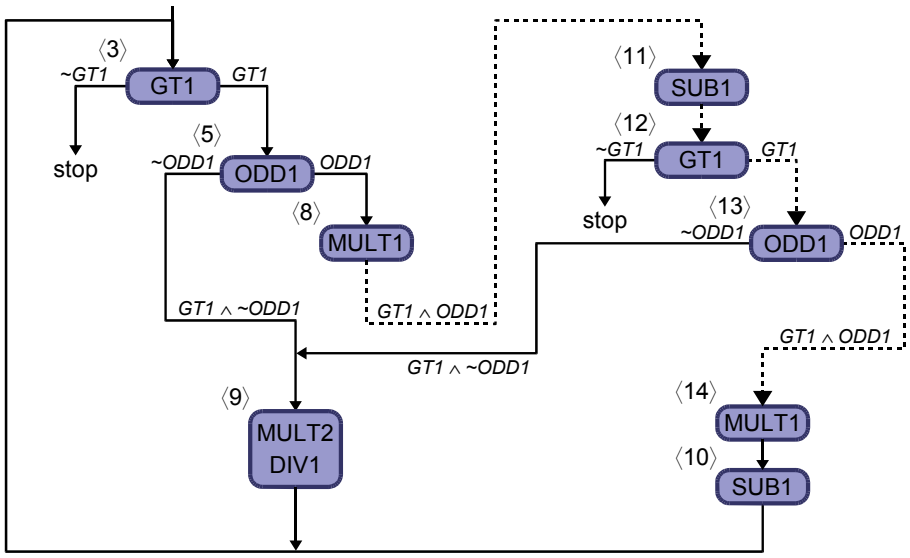


Abbildung 5.16: Struktureller Aufbau des Ablaufplans

Um die Struktur des Ablaufplans innerhalb des formalen Synthesesystems nachzubilden, wird zunächst jedem Grundblockteil des Ablaufplans einer der entsprechenden Zustände der steuerflussorientierten Darstellung zugeordnet.

So wird beispielsweise der in Abbildung 5.16 links aufgeführten Operation GT1 des Ablaufplans der Zustand $\langle \mathbf{3} \rangle$ und der rechten der Zustand $\langle \mathbf{12} \rangle$ der steuerflussorientierten Darstellung zugeordnet. Eine umgekehrte Zuordnung ist ebenso denkbar. In Abbildung 5.16 sind alle für dieses Beispiel gewählten Zuordnungen bereits dargestellt.

Nach erfolgter Zuordnung werden die Transitionsfunktionen der Zustände der steuerflussorientierten Darstellung derart modifiziert, dass sie die Übergänge zwischen den Grundblockteilen im Ablaufplan exakt nachbilden. Die neuen Übergänge, die zur Erreichung dieses Ziels geschaffen werden müssen, sind in Abbildung 5.16 durch gestrichelte Kanten hervorgehoben.

Die Umleitung des Übergangs eines Zustands zu seinem Nachfolger auf einen dem Nachfolger ähnlichen Zustand erfolgt auf Basis des Theorems 5.21:

$$
\begin{array}{ll}
 & \vdash \forall\ \textit{SimList}\ \textbf{start}\ L\ \textbf{s}\ \textit{Rumpf}'. \qquad (5.21)\\
1 & \quad \texttt{StateExists}\ L\ \textbf{s} \wedge \\
2 & \quad \texttt{SimRel}\ L\ \textit{SimList} \wedge \\
3 & \quad \texttt{StateSim}\ \textit{SimList}\ (\texttt{StateCase}\ L\ \textbf{s})\ \textit{Rumpf}' \\
4 & \quad \Longrightarrow \big(\ \texttt{let}\ L' = ((\textbf{s}, \textit{Rumpf}')\texttt{::}(\texttt{DeleteState}\ L\ \textbf{s}))\ \texttt{in} \\
5 & \qquad (\texttt{LOOP}\ \textbf{start}\ L = \texttt{LOOP}\ \textbf{start}\ L') \wedge \\
6 & \qquad (\texttt{SimRel}\ L'\ \textit{SimList})\ \big)
\end{array}
$$

Das Theorem ist derart formuliert, dass der gesamte Rumpf eines existierenden Zustands $\langle \mathrm{s} \rangle$ durch einen neuen Rumpf ***Rumpf′*** ersetzt wird. Im konkreten Anwendungsfall unterscheiden sich die beiden Rümpfe allerdings nur in den Nachfolgern, die innerhalb der beiden Transitionsfunktionen bestimmt werden. Voraussetzungen für den Einsatz des Theorems sind, dass der zu modifizierende Zustand $\langle \mathrm{s} \rangle$ existiert (Zeile 1) und ***SimList*** eine gültige Ähnlichkeitsrelation für die gegebene steuerflussorientierte Darstellung mit der Zustandsliste ***L*** repräsentiert (Zeile 2). Letztere Bedingung wird in der Regel nicht speziell bewiesen, sondern ist Ergebnis der im letzten Abschnitt beschriebenen Kopiervorgänge.

Beim Einsatz des Theorems 5.21 im Zuge der entsprechenden Transformation wird zunächst auf Basis des Rumpfes des zu modifizierenden Zustands (**`StateCase`** ***L*** **s**) ein neuer Rumpf ***Rumpf′*** konstruiert, in welchem die Zielzustände der Struktur des Ablaufplans angepasst sind. Anschließend wird bewiesen, dass der Zustand mit dem neuen Rumpf ***Rumpf′*** ähnlich zu dem zu ersetzenden Zustand ist (Zeile 3). In der Praxis kommen dabei zwei Hilfstheoreme zum Einsatz, welche die automatische Beweisführung für den vorliegenden Fall stark vereinfachen und damit auch beschleunigen. Vor der ausführlichen Diskussion der Durchführung von Ähnlichkeitsbeweisen unter Verwendung dieser beiden Hilfstheoreme soll jedoch zunächst die Beschreibung des Theorems 5.21 zu Ende geführt werden.

In Zeile 4 des Theorems wird eine neue Zustandsliste $\boldsymbol{L'}$ erzeugt, in welcher der Rumpf des ursprünglichen Zustands $\langle \mathrm{s} \rangle$ durch den neuen Rumpf ***Rumpf′*** ersetzt wird. Der Aufbau von $\boldsymbol{L'}$ erfolgt, indem aus der ursprünglichen Zustandsliste ***L*** der Zustand $\langle \mathrm{s} \rangle$ entfernt wird und die resultierende Liste um einen neuen Zustand mit demselben Bezeichner **s** und dem neuen Rumpf ***Rumpf′*** erweitert wird. Sind alle Voraussetzungen des Theorems erfüllt, so kann die Zustandsliste der steuerflussorientierten Darstellung durch die neue Zustandsliste $\boldsymbol{L'}$ ersetzt werden (Zeile 5). Gleichzeitig erhält man den Beweis, dass die Ähnlichkeitsliste ***SimList*** auch für die modifizierte Zustandsliste $\boldsymbol{L'}$ gilt (Zeile 6).

Ähnlichkeit eines Zustands nach Änderung seiner Nachfolger

Wie bereits erwähnt, wird der neue Zustand auf Basis des ursprünglichen Zustands erzeugt, indem einige Bezeichner der Nachfolger in der Transitionsfunktion ausgetauscht werden. Die Aktion des Zustands bleibt unverändert. Beim Austausch der Nachfolger eines Zustands bzw. bei dem Beweis der Ähnlichkeit zwischen dem resultierenden Zustand und seinem Ursprung kommt

der Vorteil der Trennung der Zustandsrümpfe in Aktion und Transitionsfunktion zur vollen Auswirkung. Eine separate Transitionsfunktion erleichtert nicht nur erheblich das Austauschen der Nachfolger eines Zustands, sondern auch den in Zeile 3 des Theorems 5.21 geforderten Ähnlichkeitsbeweis.

Hat der zu modifizierende Zustand nur einen möglichen Nachfolger, so hat seine Transitionsfunktion die simple Form **DEF** ($\lambda x.(x,\mathbf{z1})$). In diesem Fall kann der Bezeichner des ursprünglichen Folgezustands $\langle$z1$\rangle$ einfach durch den Bezeichner des gewünschten Nachfolgers ersetzt werden. Der Beweis, dass Zustände auf Basis des alten und neuen Rumpfs ähnlich sind, wird durch das Hilfstheorem 5.22 vereinfacht. In diesem Fall muss zum Beweis der Ähnlichkeit der beiden Zustände nur gezeigt werden, dass die Ähnlichkeitsklassen der beiden Nachfolger, welche mit Hilfe der Funktion **SimClass** aus der Ähnlichkeitsliste ***SimList*** ermittelt werden, identisch sind:

$$\begin{aligned}&\vdash \forall\ \textit{SimList}\ A\ \mathbf{z1}\ \mathbf{z2}. \\ &\quad (\mathbf{SimClass}\ \textit{SimList}\ \mathbf{z1} = \mathbf{SimClass}\ \textit{SimList}\ \mathbf{z2}) \Rightarrow \\ &\quad \mathbf{StateSim}\ \textit{SimList}\ (A\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda x.(x,\mathbf{z1})))\ (A\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda x.(x,\mathbf{z2})))\end{aligned} \tag{5.22}$$

Im Beispiel stellt Zustand $\langle$11$\rangle$ eine Kopie des Zustands $\langle$10$\rangle$ dar. Sie haben beide exakt den gleichen Rumpf und verweisen somit beide auf den Zustand $\langle$3$\rangle$ als Nachfolger. Die eben beschriebene Vorgehensweise wird nun eingesetzt, um den Nachfolger $\langle$3$\rangle$ des Zustands $\langle$11$\rangle$ auf den Zustand $\langle$12$\rangle$ abzuändern (siehe gestrichelte Kante in Abbildung 5.16). Diese Änderung ist zulässig, da laut der Ähnlichkeitsliste [(**12**,**3**);(**13**,**5**);(**14**,**8**);(**11**,**10**)] sowohl der Zustand $\langle$3$\rangle$ als auch der Zustand $\langle$12$\rangle$ die Ähnlichkeitsklasse **3** haben. Zustand $\langle$3$\rangle$ liegt in der Ähnlichkeitsklasse **3**, da laut der Definition 5.15 der Funktion **SimClass** die Ähnlichkeitsklasse eines jeden Zustands, dessen Bezeichner nicht als linker Teil eines der Paare der Zustandsliste enthalten ist, seinem eigenen Bezeichner entspricht. Der Nachfolger $\langle$10$\rangle$ des Zustands $\langle$8$\rangle$ wird entsprechend auf den Zustand $\langle$11$\rangle$ abgeändert.

Etwas komplizierter gestaltet es sich, wenn der zu modifizierende Zustand mehrere potentielle Nachfolger hat. In diesem Fall müssen die Bedingungen, unter denen der auszutauschende Nachfolger ausgewählt wird, berücksichtigt werden.

Im Beispiel aus Abbildung 5.15 hat der Zustand $\langle$13$\rangle$ zum gegenwärtigen Zeitpunkt die Transitionsfunktion **DEF** ($\lambda(\underline{\mathbf{x}}$,ODD1).($\underline{\mathbf{x}}$,**MUX**(ODD1,8,9))). Zur Nachbildung der Struktur des Ablaufplans muss dieser Zustand für den Fall, dass die Bedingung ODD1 *wahr* ist, den Zustand $\langle$14$\rangle$ anstatt des Zustands $\langle$8$\rangle$ als Nachfolger wählen. Somit ergibt sich die neue Transitionsfunktion zu **DEF** ($\lambda(\underline{\mathbf{x}}$,ODD1).($\underline{\mathbf{x}}$,**MUX**(ODD1,14,9))). Weiter muss im Beispiel der Zustand $\langle$12$\rangle$, ebenfalls wenn die Bedingung ODD1 *wahr* ist, den Zustand $\langle$13$\rangle$ statt $\langle$5$\rangle$ als Nachfolger selektieren.

Zur Vereinfachung des Ähnlichkeitsbeweises von Zuständen mit mehreren potentiellen Nachfolgern dient das Hilfstheorem 5.23:

$$\begin{aligned}&\vdash \forall\ \textit{SimList}\ A\ \mathbf{trf}\ \mathbf{trf}'. \\ &\quad (\ \forall\ \underline{\mathbf{c}}\ .\ \mathbf{SimClass}\ \textit{SimList}\ (\mathbf{trf}\ \underline{\mathbf{c}}) = \mathbf{SimClass}\ \textit{SimList}\ (\mathbf{trf}'\ \mathbf{c})\ ) \Rightarrow \\ &\quad \mathbf{StateSim}\ \textit{SimList}\ (A\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda(\underline{\mathbf{x}},\underline{\mathbf{c}}).(\underline{\mathbf{x}},\mathbf{trf}\ \underline{\mathbf{c}})))\ (A\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda(\underline{\mathbf{x}},\underline{\mathbf{c}}).(\underline{\mathbf{x}},\mathbf{trf}'\ \underline{\mathbf{c}})))\end{aligned} \tag{5.23}$$

Um die Ähnlichkeit zwischen dem ursprünglichen Zustand und dem resultierenden Zustand zu beweisen, müssen auch in diesem Fall nicht die vollständigen Rümpfe der Zustände untersucht

werden. Nach einer Umformung der Transitionsfunktionen in die Form **DEF** (λ(**x**,**c**).(**x**,**trf c**)) reicht es aus, die sich ergebenden *Bestandteile* **trf** sowie **trf′** der beiden Transitionsfunktionen zu untersuchen, die konkret in Abhängigkeit der Bedingungen, welche in der Aktion der Zustände berechnet werden, den Bezeichner des Nachfolgers bestimmen. Wenn sich für alle möglichen Kombinationen der Bedingungen in der Variablenstruktur **c** die Ähnlichkeitsklassen der aus der alten und neuen Transitionsfunktion resultierenden Nachfolger als identisch erweisen, so sind zwei Zustände, welche sich nur in diesen beiden Transitionsfunktionsteilen unterscheiden, ähnlich und somit austauschbar.

Vor Einsatz des Hilfstheorems 5.23 müssen die zu überprüfenden Transitionsfunktionen zunächst in die geforderte Form **DEF** (λ(**x**,**c**).(**x**,**trf c**)) gebracht werden. So wird die Transitionsfunktion des Zustands ⟨13⟩ in **DEF** (λ(**x**,ODD1).(**x**,(λ ODD1.**MUX**(ODD1,8,9)) ODD1)) überführt. Die Funktion **trf** im Theorem 5.23 entspricht somit in diesem Fall der Abstraktion (λ ODD1.**MUX**(ODD1,8,9)). Nach dem Austausch des Zielzustands ⟨8⟩ ergibt sich die Funktion **trf′** entsprechend zu (λ ODD1.**MUX**(ODD1,14,9)). Ist ODD1 *wahr*, so bestimmen die beiden Funktionen die Zustände ⟨8⟩ und ⟨14⟩ als Nachfolger, welche sich laut der zu diesem Zeitpunkt gültigen Ähnlichkeitsliste ähneln. Ist ODD1 *falsch*, so ergeben beide Funktionen den gleichen Nachfolger ⟨9⟩. Unter allen möglichen Eingabebedingungen ergeben die alte und die neue Transitionsfunktion jeweils identische oder zumindest ähnliche Nachfolger. Die Vorbedingung des Theorems 5.23 ist somit erfüllt. Der ursprüngliche und der modifizierte Zustand sind ähnlich und können auf Basis des Theorems 5.21 ausgetauscht werden. Auf gleiche Weise werden anschließend die Nachfolger des Zustands ⟨12⟩ angepasst.

Nach der Anpassung der Transitionsfunktionen der steuerflussorientierten Darstellung spiegelt diese genau die Verbindungsstruktur zwischen den Grundblockteilen wider, wie sie vom Ablaufplan vorgegeben wird. Zur vollständigen Umsetzung der Ablaufplanung in dem formalen System verbleibt nur, die Zustände des Ablaufplans in der steuerflussorientierten Darstellung nachzubilden.

5.3.4 Nachbildung der Zustände des Ablaufplans

Die Zustände des Ablaufplans werden innerhalb des formalen Systems nachgebildet, indem jeweils die Zustände der steuerflussorientierten Darstellung, deren entsprechende Grundblockteile im Ablaufplan zu einem Zustand zusammengefasst sind, ebenfalls zu einem einzigen Zustand verschmolzen werden. Das Verschmelzen der Zustände erfolgt wie in Abschnitt 5.2.2 (Seite 116) beschrieben.

Um den Zustand ⟨alp1⟩ des Ablaufplans in Abbildung 5.17 nachzubilden, müssen die Zustände ⟨3⟩, ⟨5⟩ und ⟨8⟩ der steuerflussorientierten Darstellung zu einem Zustand verschmolzen werden. Die Reihenfolge der Verschmelzung ist nur insofern vorgegeben, als jeweils zwei aufeinanderfolgende Zustände verschmolzen werden müssen. So könnten in diesem Fall zuerst die Zustände ⟨3⟩ und ⟨5⟩ und anschließend der sich daraus ergebende Zustand mit Zustand ⟨8⟩ verschmolzen werden. Auf die gleiche Weise werden die übrigen Zustände des Ablaufplans nachgebildet.

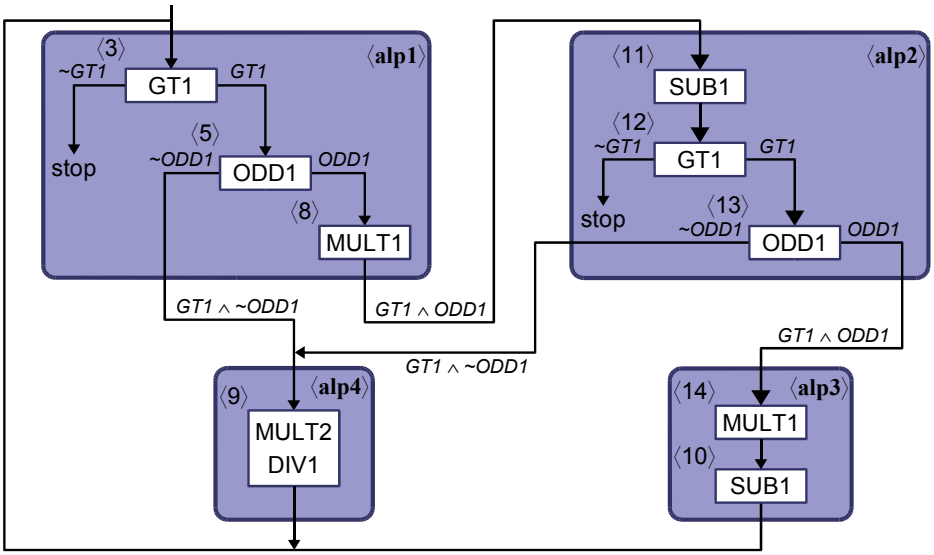


Abbildung 5.17: Verschmelzung der Grundblockteile

Abbildung 5.18 enthält die Beispielspezifikation in der steuerflussorientierten Darstellung nach der vollständigen Umsetzung der Ablaufplanung. Nach Durchführung der Verschmelzungen entspricht in der steuerflussorientierten Darstellung der Zustand ⟨3⟩ dem Zustand ⟨alp1⟩ des Ablaufplans, Zustand ⟨11⟩ dem Zustand ⟨alp2⟩, Zustand ⟨14⟩ dem Zustand ⟨alp3⟩ und Zustand ⟨9⟩ dem Zustand ⟨alp4⟩.

Mit der Nachbildung der Zustände des Ablaufplans endet bei steuerflussbehafteten Schaltungsbeschreibungen die formale Umsetzung der Ablaufplanung. Die steuerflussorientierte Darstellung, die ursprünglich dem Steuerflussgraphen der Verhaltensspezifikation entsprach, konnte automatisch auf die vorgestellte Weise umgeformt worden, so dass sie exakt den Aufbau des vorgegebenen Ablaufplans widerspiegelt. Alle Operationen der Spezifikation, welche innerhalb eines Taktes in der RT-Ebenen-Implementierung zur Ausführung kommen sollen, sind nun jeweils in einem Zustand der steuerflussorientierten Darstellung zusammengefasst.

Im Folgenden wird ein Verfahren vorgestellt, welches auf die Umsetzung der Ablaufplanung bei reinen Datenflussspezifikationen spezialisiert ist. Eine spezielle Behandlung steuerflussloser Schaltungsspezifikationen bietet sich in diesem Zusammenhang an, da deren Ablaufplanung prinzipiell einfacher und daher auch effizienter durchführbar ist.

5.3.5 Umsetzung der Ablaufplanung bei reinen Datenflussspezifikationen

Bei der formalen Realisierung der Ablaufplanung können bei Schaltungsbeschreibungen, die keinen Steuerfluss aufweisen, einige der in den letzten Abschnitten beschriebenen Schritte eingespart werden. In [Blum00] hat Blumenröhr bereits ausführlich eine Möglichkeit zur Umsetzung der Ablaufplanung bei steuerflusslosen Schaltungsbeschreibungen vorgestellt. Es ist gelungen, diesen Teil seiner Arbeit in die vorliegende Arbeit derart zu integrieren, dass im weiteren Verlauf die Synthese von Schaltungsspezifikationen sowohl mit als auch ohne Steuerfluss auf eine

```
DEF (λ(u,n).(u,1,n,T))
SER
LOOP 3
 [(  3, DEF (λ(u,v,n,GT1).
              let GT1 = n> 0 in
              let ODD1 = ODD n in
              let MULT1 = u∗v in
              let v' = MUX(ODD1,MULT1,v) in
              let (v'',GT1') = MUX(GT1,(v',ODD1),(v,GT1)) in
                  ((u,v'',n,GT1'),ODD1,GT1))
      SER DEF (λ(x,ODD1,GT1).(x,MUX(GT1,MUX(ODD1,11,9),0))));
  (  9, DEF (λ(u,v,n,ODD1).
              let MULT2 = u∗u in
              let DIV1 = n DIV 2 in (MULT2,v,DIV1,T))
      SER DEF (λx.(x,3)));
  (11, DEF (λ(u,MULT1,n,ODD1).
              let SUB1 = n−1 in
              let GT1 = SUB1> 0 in
              let ODD1 = ODD SUB1 in
              let C = MUX(GT1,ODD1,F) in ((u,MULT1,SUB1,C),ODD1,GT1))
      SER DEF (λ(x,ODD1,GT1).(x,MUX(GT1,MUX(ODD1,14,9),0))));
  (14, DEF (λ(u,v,n,ODD1).
              let MULT1 = u∗v in
              let SUB1 = n−1 in (u,MULT1,SUB1,T))
      SER DEF (λx.(x,3)))]
SER
DEF (λ(u,v,n,GT1).v)
```

Abbildung 5.18: Beispielspezifikation nach Umsetzung der Ablaufplanung

einheitliche Weise erfolgen kann. In [Blum00] war eine durchgehende Sonderbehandlung der beiden Spezifikationsarten während der gesamten algorithmischen Synthese erforderlich.

Blumenröhr hat in seiner Arbeit einige Algorithmen für die Ablaufplanung reiner Datenflussspezifikationen implementiert, darunter den *List-Based-Scheduling*-Algorithmus [DLSM81] und eine Variante des *Force-Directed-Scheduling*-Algorithmus [PaKn87]. Im Folgenden wird beschrieben, wie seine Implementierungen uneingeschränkt in der vorliegenden Arbeit weiterverwendet werden können.

Schaltungsspezifikationen ohne Steuerfluss haben die Form **DEF** f. Ein Beispiel für eine steuerflusslose Spezifikation gibt Abbildung 5.19.

Die Ablaufplanung beginnt mit der Ermittlung eines geeigneten Ablaufplans für die gegebene Spezifikation **DEF** f auf Basis eines der zur Verfügung stehenden Ablaufplanungsalgorithmen.

```
DEF (λ(a0,a1,a2,a3,b0,b1).
      let MULT1 = b1∗a3 in
      let SUB1  = a2−MULT1 in
      let MULT2 = b0∗SUB1 in
      let SUB2  = a0−MULT2 in
      let MULT3 = b0∗a3 in
      let MULT4 = b1∗SUB1 in
      let ADD1  = MULT3+MULT4 in
      let SUB3  = a1−ADD1 in
      (SUB1,a3,SUB2,SUB3))
```

Abbildung 5.19: Eine Schaltungsspezifikation ohne Steuerfluss

Die Umsetzung des Ablaufplans nach [Blum00] besteht aus der Überführung des DFG-Terms f in eine Komposition mehrerer DFG-Terme ($f_n \circ f_{n-1} \circ \cdots \circ f_1$), so dass jeder der resultierenden DFG-Terme genau diejenigen Operationen enthält, die später in einem bestimmten Takt ausgeführt werden sollen. Die Anzahl der DFG-Terme spiegelt die Anzahl der Takte wider, welche auf der RT-Ebene zur vollständigen Ausführung der Spezifikation erforderlich sein werden. Für den Beweis, dass die Aufteilung des ursprünglichen DFG-Terms der Spezifikation in die Komposition mehrerer DFG-Terme korrekt ist, also dass $f = (f_n \circ f_{n-1} \circ \cdots \circ f_1)$ gilt, wurde in [Blum00] bereits eine effiziente Vorgehensweise ausführlich vorgestellt. Diese wurde in die vorliegende Arbeit unverändert integriert.

Um eine identische Weiterverarbeitung wie bei steuerflussorientierten Schaltungsbeschreibungen zu gewährleisten, wird diese Schaltungsbeschreibung, welche nach der Umsetzung des Ablaufplans nach [Blum00] in der Form **DEF** ($f_n \circ f_{n-1} \circ \cdots \circ f_1$) vorliegt, auf Basis des folgenden Theorems umgeformt:

$$\vdash \forall \boldsymbol{A}\ \boldsymbol{B}.\ \texttt{DEF}\ (\boldsymbol{B} \circ \boldsymbol{A}) = \texttt{DEF}\ \boldsymbol{A}\ \texttt{SER}\ \texttt{DEF}\ \boldsymbol{B} \tag{5.24}$$

Durch mehrfache Anwendung des Theorems 5.24 werden alle Verknüpfungsoperatoren $\circ$ aus der Schaltungsbeschreibung entfernt, so dass die Schaltungsbeschreibung schließlich die Form **DEF** f_1 **SER DEF** f_2 **SER** ... **SER DEF** f_n hat.

Abbildung 5.20 stellt die Spezifikation aus Abbildung 5.19 nach dem Einsatz des ASAP-Ablaufplanungsalgorithmus (*as soon as possible*) dar. Jeder der resultierenden **DEF**-Terme enthält diejenigen Operationen, die später in einem bestimmten Takt auf der RT-Ebene zur Ausführung kommen sollen.

Im Anschluss an die Unterteilung der Spezifikation in mehrere **DEF**-Terme erfolgt deren Überführung in die steuerflussorientierte Darstellung (siehe Abschnitt 5.2.1). Nach dem Übergang in die neue Darstellungsform werden alle **SER**-Konstrukte in den Aktionen der Zustände, wie in Abschnitt 5.2.2 beschrieben, durch Aufspaltung der Zustände eliminiert. Ergebnis ist eine steuerflussorientierte Darstellung, in welcher jeder Zustand diejenigen Operationen enthält, die nach den Vorgaben des Ablaufplans innerhalb eines bestimmten Taktes in der resultierenden RT-Ebenen-Implementierung zur Ausführung kommen sollen. Damit ist die Ablaufplanung

```
DEF (λ(a0,a1,a2,a3,b0,b1).
      let MULT1 = b1∗a3 in
      let MULT3 = b0∗a3 in (MULT1,MULT3,a0,a1,a2,a3,b0,b1))
SER
DEF (λ(MULT1,MULT3,a0,a1,a2,a3,b0,b1).
      let SUB1  = a2−MULT1 in (MULT3,SUB1,a0,a1,a3,b0,b1))
SER
DEF (λ(MULT3,SUB1,a0,a1,a3,b0,b1).
      let MULT2 = b0∗SUB1 in
      let MULT4 = b1∗SUB1 in (MULT2,MULT3,MULT4,SUB1,a0,a1,a3))
SER
DEF (λ(MULT2,MULT3,MULT4,SUB1,a0,a1,a3).
      let SUB2  = a0−MULT2 in
      let ADD1  = MULT3+MULT4 in (ADD1,SUB1,SUB2,a1,a3))
SER
DEF (λ(ADD1,SUB1,SUB2,a1,a3).
      let SUB3  = a1−ADD1 in (SUB1,a3,SUB2,SUB3))
```

Abbildung 5.20: Datenflussspezifikation nach Aufteilung entsprechend dem ASAP-Algorithmus

vollständig formal umgesetzt. Alle weiteren Schritte, die bei der Umsetzung der Ablaufplanung von Schaltungsspezifikationen mit Steuerfluss anfallen, erübrigen sich hier.

Abbildung 5.21 zeigt die steuerflusslose Spezifikation aus Abbildung 5.19 nach der kompletten Umsetzung der Ablaufplanung.

Nach der Umsetzung der Ablaufplanung wird die formale algorithmische Synthese mit der Bereitstellung und Zuweisung der Register fortgesetzt. Die Bereitstellung und Zuweisung der Funktions- und Verbindungseinheiten findet anschließend während der Operationswerksynthese statt (siehe Abschnitt 5.5).

5.4 Bereitstellung und Zuweisung der Register

Im Zuge der Bereitstellung und Zuweisung der Register wird zunächst analysiert, wieviele Register welchen Typs zur Realisierung der Spezifikation erforderlich sind und in welchen Registern die verschiedenen Ausgabewerte der Zustände an deren jeweiligen Nachfolger übertragen werden sollen. Dabei gibt es unterschiedliche Strategien. In dieser Arbeit wird exemplarisch eine einfache Heuristik eingesetzt, welche das Ziel hat, die Schaltung mit einer möglichst geringen Anzahl an Registern zu realisieren, und danach strebt, Werte, die über mehrere Zustände hinweg unverändert weitergereicht werden, denselben Registern zuzuordnen[§]. Im Folgenden wird beschrieben, wie die aus der Analyse resultierenden Registerbelegungen vollautomatisch innerhalb des formalen Synthesesystems umgesetzt werden können.

[§]Ein Umspeichern entsprechender Werte soll vermieden werden, da es im Allgemeinen zu einer erhöhten Schaltaktivität in der Schaltung führt und dadurch einen größeren Leistungsverbrauch verursacht.

```
DEF (λ(a0,a1,a2,a3,b0,b1).(a0,a1,a2,a3,b0,b1,hvn1,hvn2))
SER
LOOP 1
 [(1, DEF (λ(a0,a1,a2,a3,b0,b1,hvn1,hvn2).
          let MULT1 = b1∗a3 in
          let MULT3 = b0∗a3 in (MULT1,MULT3,a0,a1,a2,a3,b0,b1))
     SER DEF (λx.(x,2)));
  (2, DEF (λ(MULT1,MULT3,a0,a1,a2,a3,b0,b1).
          let SUB1 = a2−MULT1 in (MULT3,SUB1,a0,a1,a3,b0,b1,hvn))
     SER DEF (λx.(x,3)));
  (3, DEF (λ(MULT3,SUB1,a0,a1,a3,b0,b1,hvn).
          let MULT2 = b0∗SUB1 in
          let MULT4 = b1∗SUB1 in (MULT2,MULT3,MULT4,SUB1,a0,a1,a3,hvn))
     SER DEF (λx.(x,4)));
  (4, DEF (λ(MULT2,MULT3,MULT4,SUB1,a0,a1,a3,hvn).
          let SUB2 = a0−MULT2 in
          let ADD1 = MULT3+MULT4 in (ADD1,SUB1,SUB2,a1,a3,
                                     hvn1,hvn2,hvn3))
     SER DEF (λx.(x,5)));
  (5, DEF (λ(ADD1,SUB1,SUB2,a1,a3,hvn1,hvn2,hvn3).
          let SUB3 = a1−ADD1 in (SUB1,a3,SUB2,SUB3,hvn1,hvn2,hvn3,hvn4))
     SER DEF (λx.(x,0)))]
SER
DEF (λ(SUB1,a3,SUB2,SUB3,hvn1,hvn2,hvn3,hvn4).(SUB1,a3,SUB2,SUB3))
```

Abbildung 5.21: Datenflussspezifikation nach Umsetzung der Ablaufplanung

Die Ausführung eines Zustands der steuerflussorientierten Darstellung entspricht auf der RT-Ebene genau der Ausführung eines Taktes. Die Datenmenge, welche von einem Zustand zum nächsten übertragen wird, spiegelt sich in der steuerflussorientierten Darstellung in dem Eingabetyp der Zustände wider. Eine aus einer entsprechenden Schaltungspezifikation abgeleitete RT-Ebenen-Implementierung muss schließlich eine Registerzahl aufweisen, die zur Speicherung der erforderlichen Datenmenge zwischen zwei Takten ausreicht. So wird für jeden der elementaren Eingabetypen der Zustände der steuerflussorientierten Darstellung genau ein Register des passenden Typs auf der RT-Ebene bereitgestellt.

In der Spezifikation in Abbildung 5.18 ist der Eingabetyp der Zustände beispielsweise $num \times num \times num \times bool$. Eine direkt aus dieser Spezifikation abgeleitete RT-Ebenen-Implementierung würde eine Registerbank aufweisen, welche aus drei Registern des Typs num sowie einem booleschen Register besteht.

In den bis zu diesem Zeitpunkt aufgeführten Theoremen zur Transformation steuerflussorientierter Darstellungen wird stets die Möglichkeit berücksichtigt, dass mehrere Zustandsbezeichner

den Ausgangszustand repräsentieren können. Nach der Definition der steuerflussorientierten Darstellung in Abschnitt 3.26 steht in der Tat jeder Bezeichner, welcher sich von allen Bezeichnern der Zustände in der Zustandsliste unterscheidet, für den Ausgangszustand, so dass theoretisch mehrere Bezeichner zur Repräsentation des Ausgangszustands denkbar sind. Wird allerdings wie in den letzten Abschnitten beschrieben die steuerflussorientierte Darstellung aus der Verhaltensspezifikation abgeleitet (siehe Abschnitt 5.2.1), so ist der Bezeichner des Ausgangszustands grundsätzlich eindeutig. Ansonsten besteht auch die Möglichkeit, die Bezeichner aller Ausgangszustände durch eine Umkodierung der Zustandsbezeichner auf einen einzigen Bezeichner abzubilden (siehe Abschnitt 5.6, Seite 155).

Um die Bereitstellung und Zuweisung der Register innerhalb des formalen Systems umzusetzen, müssen die gewünschten Registerbelegungen zunächst in einer formalen Darstellung ausgedrückt werden. Zu diesem Zweck wird für jeden Zustand der steuerflussorientierten Darstellung – auch für den Ausgangszustand – eine sogenannte *Zuweisungsfunktion* erstellt. Diese Zuweisungsfunktion gibt an, in welcher Weise die Eingabewerte eines Zustands umgeordnet werden müssen, so dass jeder Eingabewert in demjenigen Register an den Zustand übergeben wird, welches im Zuge der Analyse ausgewählt wurde. Implizit wird damit gleichzeitig festgelegt, wie die Ausgaben der Zustände auf die einzelnen Elementartypen verteilt zu sein haben.

Stellt sich heraus, dass nicht alle elementaren Eingabetypen der Zustände einer steuerflussorientierten Darstellung benötigt werden, so wird der Eingabetyp der Zustände derart umgeformt, dass sich ein Paar zweier Typstrukturen ergibt, in welchem die rechte Typstruktur die elementaren Eingabetypen enthält, welche eingespart werden können, während die linke Typstruktur die benötigten Elementartypen enthält. Diese Aufteilung ist eine Vorbereitung für die spätere Eliminierung der überflüssigen Eingabetypen.

Im Beispiel in Abbildung 5.18 ergibt die Ausführung eines Analysealgorithmus zur Bereitstellung und Zuweisung von Registern, dass von den drei Eingabetypen num und dem einen booleschen Eingabetyp der Zustände der letztere überflüssig ist. Ansonsten soll nach den Vorgaben des Algorithmus für die Eingabewerte der Zustände $\langle 9 \rangle$, $\langle 11 \rangle$ und $\langle 14 \rangle$ keine Umsortierung erforderlich sein. Im Zustand $\langle 3 \rangle$ sowie dem Ausgangszustand mit dem Bezeichner 0 sollen allerdings jeweils die beiden linken Eingaben, welche zum Beispiel innerhalb des Zustands $\langle 3 \rangle$ den Variablen ***u*** und ***v*** zugeordnet werden, vertauscht werden. Somit ergeben sich für den Zustand $\langle 3 \rangle$ sowie für den Ausgangszustand die Zuweisungsfunktion λ(i1,i2,i3,i4).((i2,i1,i3),i4) und für die Zustände $\langle 9 \rangle$, $\langle 11 \rangle$ und $\langle 14 \rangle$ jeweils die Zuweisungsfunktion λ(i1,i2,i3,i4).((i1,i2,i3),i4). Die Zuweisungsfunktionen werden eine Änderung des Eingabetyps der Zustände zur Folge haben. Er wird von $num \times num \times num \times bool$ in $(num \times num \times num) \times bool$ überführt. Dadurch wird erreicht, dass die neue Typstruktur ein Paar ist, dessen rechte Hälfte aus dem booleschen Eingabetyp besteht, der sich während der Analyse als überflüssig erwiesen hat. Diese Typänderung erlaubt später die Eliminierung des überflüssigen booleschen Werts.

Nach dem Aufbau der Zuweisungsfunktionen wird zu jeder dieser Funktionen deren Umkehrfunktion generiert. Anschließend wird eine sogenannte *Zuweisungsliste* erstellt, deren Elemente die Form (*z*,(**zwf**,**inv**)) haben. Dabei entspricht *z* dem Bezeichner eines Zustands, **zwf** der Zuweisungsfunktion des entsprechenden Zustands sowie **inv** der Umkehrfunktion der jeweiligen Zuweisungsfunktion. Die Struktur der Zuweisungsliste erlaubt es, mit Hilfe der bereits existierenden Funktion `StateCase` anhand des Bezeichners eines Zustands dessen zugehörige

```
[ (  0, ((λ(i1,i2,i3,i4).((i2,i1,i3),i4)), (λ((i2,i1,i3),i4).(i1,i2,i3,i4))) );
  (  3, ((λ(i1,i2,i3,i4).((i2,i1,i3),i4)), (λ((i2,i1,i3),i4).(i1,i2,i3,i4))) );
  (  9, ((λ(i1,i2,i3,i4).((i1,i2,i3),i4)), (λ((i1,i2,i3),i4).(i1,i2,i3,i4))) );
  ( 11, ((λ(i1,i2,i3,i4).((i1,i2,i3),i4)), (λ((i1,i2,i3),i4).(i1,i2,i3,i4))) );
  ( 14, ((λ(i1,i2,i3,i4).((i1,i2,i3),i4)), (λ((i1,i2,i3),i4).(i1,i2,i3,i4))) )]
```

Abbildung 5.22: Zuweisungsliste für die Beispielspezifikation in Abbildung 5.18

Zuweisungsfunktion sowie deren Umkehrfunktion zu ermitteln. Abbildung 5.22 stellt die Zuweisungsliste für das laufende Beispiel dar.

Nach Aufbau der Zuweisungsliste kann unter Einsatz des Theorems 5.25 die Umordnung der Eingaben in der steuerflussorientierten Darstellung erfolgen.

```
    ⊢ ∀ ZWL L start exit.                                                    (5.25)
1     ¬(StateExists L exit) ∧
2     ( ((StateExists L start) ∧
3       (∀x (p1,p2). EXISTS (λU. U x = Defined (p1,p2)) (SND (UNZIP L))
4                    ⇒ ( (StateExists L p2) ∨ (p2 = exit)))) ∨
5       (start = exit) ) ∧
6     ( ALL_EL (λz. (λtfs. SND tfs ∘ FST tfs = I) (StateCase ZWL z))
7               (exit ::FST (UNZIP L)) )
8     ⟹ ( LOOP start L =
9          DEF (FST (StateCase ZWL start))
10         SER
11         (LOOP start
12                ( MAP (λ(z,Rumpf).
13                       (z, DEF (SND (StateCase ZWL z))
14                          SER
15                          Rumpf
16                          SER
17                          DEF (λ(x,z′).
18                                 (FST (StateCase ZWL z′) x, z′))))
19                  L )
20         SER
21         DEF (SND (StateCase ZWL exit))) )
```

Voraussetzung für die Anwendung des Theorems 5.25 ist, dass nur ein Bezeichner für den Ausgangszustand, nämlich **exit**, verwendet wird (Zeile 1-5). Wie bereits erwähnt ist diese Tatsache grundsätzlich eine Folge der bisher beschriebenen Vorgehensweise. Zum Beweis der entsprechenden Vorbedingung muss zunächst sichergestellt werden, dass der Bezeichner **exit** überhaupt auf den Ausgangszustand verweist. Dies ist genau dann der Fall, wenn er sich von allen Bezeichnern der Zustände in der Zustandsliste unterscheidet (Zeile 1). Die Aussage in Zeile 5 deckt die

Möglichkeit ab, dass der Startzustand dem Ausgangszustand gleicht. Bei der bisher beschriebenen Vorgehensweise tritt dieser Fall niemals ein. Insofern existiert der Startzustand ⟨**start**⟩ in der Zustandsliste (Zeile 2), so dass verbleibt, für alle potentiellen Nachfolger der Zustände der steuerflussorientierten Darstellung zu fordern (Zeile 3), dass sie entweder in der Zustandsliste enthalten sind oder auf den (einzigen) Ausgangszustand mit dem Bezeichner **exit** verweisen (Zeile 4).

Eine weitere Vorbedingung des Theorems 5.25 bezieht sich auf den korrekten Aufbau der Zuweisungsliste (Zeile 6-7). Für alle Zustände der Zustandsliste sowie für den Ausgangszustand ⟨**exit**⟩ (Zeile 7) muss ein Eintrag in der Zuweisungsliste existieren. Für jeden dieser Einträge muss gelten, dass die zusätzlich zur Zuweisungsfunktion gespeicherte Funktion tatsächlich deren Umkehrfunktion darstellt. Dies folgt aus dem Beweis, dass die Verknüpfung jeder Zuweisungsfunktion mit ihrer entsprechenden Umkehrfunktion jeweils die Identität (**I**) ergibt (Zeile 6).

Abbildung 5.23 zeigt die Beispielspezifikation nach der Umsetzung der Registerzuweisung. Dabei sind die Zuweisungsfunktionen bereits in die Transitionsfunktionen und deren Umkehrfunktionen in die Aktionen der Zustände integriert worden. Die vor bzw. nach der **LOOP**-Schleife stehenden **DEF**-Terme wurden ebenfalls bereits miteinander verknüpft.

Nach der Integration der Zuweisungsfunktionen in die Transitionsfunktionen der Zustände verteilen diese die Ausgabewerte in Abhängigkeit des Folgezustands auf die gewünschten elementaren Ausgabetypen. Diese Verteilung wird vor der Durchführung der weiteren Syntheseschritte in die Aktionen der Zustände verlagert, so dass sich die Transitionsfunktionen wieder, wie ursprünglich vorgesehen (siehe Abschnitt 5.2), auf die Bestimmung des Folgezustands beschränken und die Ausgabe der Aktionen unverändert weitergeben. Mit diesem Schritt ist die Zuweisung der Register abgeschlossen.

Entfernen überflüssiger Register

Falls sich bei der Analyse der benötigten Register einige der Register als überflüssig erwiesen haben, wurden bereits im Zuge der letzten Transformation die entsprechenden elementaren Eingabetypen der Zustände der steuerflussorientierten Darstellung in die rechte Hälfte des Eingabetyps eingeordnet. Der Eingabetyp der Zustände hat in diesem Fall die Form $\beta_1 \times \beta_2$. Dabei steht β_1 für die Typstruktur, welche den benötigten Eingabetypen entspricht, und β_2 für eine Struktur der überflüssigen Eingabetypen. Ist der Typ der Zustandsbezeichner α, so hat der Rumpf jedes Zustands den Typ $(\beta_1 \times \beta_2) \rightarrow ((\beta_1 \times \beta_2) \times \alpha)\, partial$. Bevor der Eingabetyp auf die linke Hälfte β_1 reduziert werden kann, muss bewiesen werden, dass die Werte, die in der rechten Hälfte β_2 des Eingabetyps übertragen werden, und somit auch alle Operation, welche die entsprechenden Werte berechnen, auf die Gesamtberechnung der Spezifikation keinen Einfluss haben. In diesem Zusammenhang kommt die Funktion **SND_ARG_INSIGNIFICANT** zum Einsatz:

$$\textbf{SND\_ARG\_INSIGNIFICANT}\ f := (\ \forall\ x\ y\ z.\ f(x,y) = f(x,z)\ ) \tag{5.26}$$

Die Funktion **SND_ARG_INSIGNIFICANT** beschreibt für eine Funktion f die Eigenschaft, dass deren zweites Argument insignifikant ist, also auf das Ergebnis der Berechnung keinen Einfluss hat. Dies ist genau dann der Fall, wenn für alle Werte x des ersten Arguments, unabhängig von dem Wert des zweiten Arguments (y bzw. z), die Funktion f stets das gleiche Ergebnis liefert.

```
DEF (λ(u,n).((1,u,n),T)))
SER
LOOP 3
 [(  3, DEF (λ((v,u,n),GT1).
              let GT1 = n> 0 in
              let ODD1 = ODD n in
              let MULT1 = u∗v in
              let v' = MUX(ODD1,MULT1,v) in
              let (v",GT1') = MUX(GT1,(v',ODD1),(v,GT1)) in
                   ((u,v",n,GT1'),ODD1,GT1))
      SER DEF (λ((i1,i2,i3,i4),ODD1,GT1).
                   MUX(GT1,MUX(ODD1,(((i1,i2,i3),i4),11),(((i1,i2,i3),i4),9)),
                        (((i2,i1,i3),i4),0))));
  (  9, DEF (λ((u,v,n),ODD1).
              let MULT2 = u∗u in
              let DIV1 = n DIV 2 in (MULT2,v,DIV1,T))
      SER DEF (λ(i1,i2,i3,i4).(((i2,i1,i3),i4),3)));
  (11, DEF (λ((u,MULT1,n),ODD1).
              let SUB1 = n−1 in
              let GT1 = SUB1> 0 in
              let ODD1 = ODD SUB1 in
              let C = MUX(GT1,ODD1,F) in ((u,MULT1,SUB1,C),ODD1,GT1))
      SER DEF (λ((i1,i2,i3,i4),ODD1,GT1).
                   MUX(GT1,MUX(ODD1,(((i1,i2,i3),i4),14),(((i1,i2,i3),i4),9)),
                        (((i2,i1,i3),i4),0))));
  (14, DEF (λ((u,v,n),ODD1).
              let MULT1 = u∗v in
              let SUB1 = n−1 in (u,MULT1,SUB1,T))
      SER DEF (λ(i1,i2,i3,i4).(((i2,i1,i3),i4),3)))]
SER
DEF (λ((v,u,n),GT1).v)
```

Abbildung 5.23: Beispielspezifikation nach Umsetzung der Registerzuweisung

Mit Hilfe der Funktion **SND_ARG_INSIGNIFICANT** kann nun für jeden Zustand der steuerflussorientierten Darstellung folgende Eigenschaft formuliert werden:

SND_ARG_INSIGNIFICANT (***Rumpf*** **SER DEF** (λ((**o1**,**o2**),*z*).(**o1**,*z*)))

Der dem Zustandsrumpf ***Rumpf*** nachgestellte **DEF**-Term bewirkt, dass diejenigen Werte, welche auf den zu eliminierenden Typ ausgegeben würden, fallen gelassen werden. Der gesamte Ausdruck beschreibt demzufolge die Eigenschaft, dass die Ausgabe des Rumpfes, wenn die zu eliminierende Ausgabe außer Acht gelassen wird, von dem zweiten Argument unabhängig

ist. Als Vorbereitung für die Entfernung der überflüssigen Eingabetypen auf Basis des Theorems 5.27 muss diese Eigenschaft für den Rumpf eines jeden Zustands der steuerflussorientierten Darstellung nachgewiesen werden.

Zur Durchführung des Beweises werden der Rumpf und der **DEF**-Term zu einem einzigen **DEF**-Term f zusammengefasst und anschließend die Funktion **SND_ARG_INSIGNIFICANT** durch ihre Definition 5.26 ersetzt. Erweist sich das zweite Argument der Funktion f erwartungsgemäß als überflüssig, so ergibt die Normalisierung der beiden aus der Ersetzung von **SND_ARG_INSIGNIFICANT** resultierenden Terme $f(x,y)$ und $f(x,z)$ deren Äquivalenz. Aus dieser folgt die Korrektheit der untersuchten Eigenschaft.

Der Beweis, dass diese Eigenschaft für alle Zustandsrümpfe der steuerflussorientierten Darstellung gilt, ist eine der Voraussetzungen des Theorems 5.27 zur Entfernung überflüssiger Register (Zeile 1-3).

```
    ⊢ ∀ L start.                                                    (5.27)
1     ( ALL_EL (λU. SND_ARG_INSIGNIFICANT
2                   (U SER DEF (λ((o1,o2),z).(o1,z))))
3                   (SND (UNZIP L)) )
4     ⟹ ( LOOP start L
5          SER
6          DEF FST
7          =
8          DEF FST
9          SER
10         (LOOP start
11               ( MAP (λ(s,Rumpf).
12                     (s, DEF (λx.(x,@@))
13                        SER
14                        Rumpf
15                        SER
16                        DEF (λ((x,y),z).(x,z)))) L ))
```

Dass die rechte Hälfte der Ausgabewerte auch für dem **LOOP**-Konstrukt folgende Anweisungen keine Relevanz hat, wird sichergestellt, indem die Spezifikation vor Anwendung des Theorems in die Form **LOOP start** L **SER DEF FST** überführt wird (Zeile 4-6). Durch das **DEF FST** im Anschluss an die **LOOP**-Anweisung wird eindeutig nur die linke Hälfte der Ausgabe der **LOOP**-Anweisung selektiert. In der Spezifikation in Abbildung 5.23 wird diese Form beispielsweise durch die Aufspaltung des abschließenden **DEF**-Terms **DEF** (λ((v,u,n),GT1).v) in **DEF FST SER DEF** (λ(v,u,n).v) erzielt.

Die Anwendung des Theorems auf den Spezifikationsteil **LOOP start** L **SER DEF FST** (Zeile 4-6) resultiert in einer **LOOP**-Anweisung, die nun von dem Term **DEF FST** angeführt wird, und eine modifizierte Zustandsliste enthält (Zeile 8-16). Dieser Term **DEF FST** bewirkt, dass die überflüssigen Eingaben bereits vor der Ausführung der neuen **LOOP**-Anweisung fallen gelassen werden.

Die Rümpfe der Zustände werden dahingehend modifiziert, dass wie in der Vorbedingung in Zeile 2 ein **DEF**-Term angehängt wird, in welchem die überflüssigen Ausgaben fallen gelassen werden (Zeile 15-16). Vorangestellt wird den neuen Rümpfen jeweils der Term **DEF** (λx.(x,@@)) (Zeile 12-13), welcher die verbleibenden benötigten Eingaben des Zustands um einen beliebigen Wert (@@) des zu eliminierenden Eingabetyps erweitert. Auf diese Weise wird der Eingangstyp der Zustände für die anschließende Ausführung des ursprünglichen Zustandsrumpfes angepasst. Eine Verschmelzung aller **DEF**-Terme der neuen Zustandsrümpfe führt schließlich wieder zur Eliminierung der eben eingeführten Konstante @@. Die resultierenden Rümpfe werden wieder jeweils in eine Aktion und eine Transitionsfunktion unterteilt. Die Transformation endet mit der Verschmelzung der beiden **DEF**-Terme, welche nach Anwendung des Theorems das resultierende **LOOP**-Konstrukt anführen.

```
DEF (λ(u,n).(1,u,n)))
SER
LOOP 3
 [(  3, DEF (λ(v,u,n).
              let GT1 = n> 0 in
              let ODD1 = ODD n in
              let MULT1 = u∗v in
              let o1 = MUX(GT1,u,v) in
              let o2 = MUX(ODD1,MULT1,v) in
              let o2' = MUX(GT1,o2,u) in ((o1,o2',n),ODD1,GT1))
      SER DEF (λ(x,ODD1,GT1).(x,MUX(GT1,MUX(ODD1,11,9),0))));
  (  9, DEF (λ(u,v,n).
              let MULT2 = u∗u in
              let DIV1 = n DIV 2 in (v,MULT2,DIV1))
      SER DEF (λx.(x,3)));
  (11, DEF (λ(u,MULT1,n).
              let SUB1 = n−1 in
              let GT1 = SUB1> 0 in
              let ODD1 = ODD SUB1 in
              let o1 = MUX(GT1,u,MULT1) in
              let o2 = MUX(GT1,MULT1,u) in ((o1,o2,SUB1),ODD1,GT1))
      SER DEF (λ(x,ODD1,GT1).(x,MUX(GT1,MUX(ODD1,14,9),0))));
  (14, DEF (λ(u,v,n).
              let MULT1 = u∗v in
              let SUB1 = n−1 in (MULT1,u,SUB1))
      SER DEF (λx.(x,3)))]
SER
DEF (λ(v,u,n).v)
```

Abbildung 5.24: Beispielspezifikation nach der Registerzuweisung und -bereitstellung

Abbildung 5.24 zeigt die Beispielspezifikation aus Abbildung 5.23 nach der vollständigen Durchführung der Registerzuweisung und -bereitstellung.

Bei steuerflusslosen Schaltungspezifikationen kann die Bereitstellung und Zuweisung der Register auf Grund deren zyklenfreien Aufbaus auch bereits vor deren Überführung in die steuerflussorientierte Darstellung umgesetzt werden. Ausgangspunkt diesen Schritts ist hierbei die Schaltungsbeschreibung, die sich bei Datenflussspezifikationen aus der Umsetzung des Ablaufplans ergibt (siehe Abschnitt 5.3.5). Zu diesem Zeitpunkt hat die Spezifikation die Form **DEF** f_1 **SER DEF** f_2 **SER** ... **SER DEF** f_n. Im Falle steuerflussloser Schaltungspezifikationen ist es möglich, auf Basis der ermittelten Entwurfsziele Zuweisungsfunktionen zu erzeugen, die gleichzeitig sowohl die Bereitstellung als auch die korrekte Zuweisung der Register bewirken. Diese Zuweisungsfunktionen werden entsprechend der Transformation, die im Zusammenhang mit der Typanpassung von **DEF**-Termen vor der Aufspaltung von Zuständen mit Aktionen der Form ***A*** **SER** ***B*** beschrieben wurde (siehe Abschnitt 5.2.2), jeweils zwischen zwei aufeinanderfolgenden **DEF**-Termen eingebettet und mit den angrenzenden **DEF**-Termen verschmolzen. Als Folge dieser Transformation werden die Daten zwischen den **DEF**-Termen exakt den Entwurfsvorgaben entsprechend übertragen, womit die Bereitstellung und Zuweisung der Register bereits vollständig umgesetzt ist. Nach diesem Schritt wird die resultierende Spezifikation schließlich in die steuerflussorientierte Darstellung überführt. Diese alternative Vorgehensweise wird in der Regel auf Grund ihrer höheren Effizienz bevorzugt angewandt.

Ab diesem Punkt verlaufen die verbleibenden Schritte, die für die Durchführung einer vollständigen formalen algorithmischen Synthese ausgeführt werden müssen, für steuerflussbehaftete und steuerflusslose Schaltungsbeschreibungen grundsätzlich identisch.

5.5 Synthese des Operationswerks

Der letzte Schritt der algorithmischen Synthese, bevor die Spezifikation auf eine Implementierung auf der RT-Ebene abgebildet wird, umfasst die Bereitstellung und Zuweisung der Operationseinheiten innerhalb des formalen Synthesesystems. Im Zuge der Bereitstellung der Operationseinheiten wird ermittelt, welche Operationseinheiten in welcher Anzahl zur Realisierung der Implementierung eingesetzt werden sollen. Bei der Zuweisung der Operationseinheiten wird jeder in der Schaltungsbeschreibung spezifizierten Operation eine der bereitgestellten Operationseinheiten, welche eine der jeweiligen Operation entsprechende Funktion in Hardware realisiert, zugeordnet. Die eigentliche Bereitstellung und Zuweisung der Operationseinheiten läuft in einer gesonderten Analysephase üblicherweise, wie in Abschnitt 4.1 beschrieben, außerhalb der formalen Umgebung ab. Anschließend werden die Analyseergebnisse innerhalb des formalen Systems umgesetzt.

Als Ergebnis der Analysephase erwartet das formale System ein Operationswerk sowie eine Menge zustandsabhängiger Steuerdaten. Das Operationswerk, das aus den bereitzustellenden Operationseinheiten besteht, spiegelt dabei bereits die gewünschte Zuweisung der einzelnen Operationseinheiten wider. Die Steuerdaten geben für jeden Zustand der Schaltung eine passende Konfiguration des Operationswerks vor.

Das Operationswerk wird in dem formalen System als DFG-Term aufgebaut und hat im Allgemeinen den Typ $(\beta_1 \times ctrl) \rightarrow (\beta_1 \times stats)$. Der Typ β_1 entspricht dem Eingabetyp der Zustände, also auf RT-Ebene den Registern, und $ctrl$ den Steuereingängen des Operationswerks. Über die Steuereingänge erfolgt die zustandsabhängige Konfiguration des Operationswerks. Die Konfiguration steuert die Belegung der Eingänge der Operationseinheiten in dem Operationswerk, gegebenenfalls den Arbeitsmodus von Multifunktionseinheiten sowie die Verteilung der berechneten Werte auf die Ausgänge des Operationswerks. Der Typ β_1 der Ausgabewerte entspricht demjenigen der Eingabewerte der Zustände. Zusätzlich gibt das Operationswerk in der Regel Statussignale aus, die hier durch den zusätzlichen Ausgabetyp $stats$ repräsentiert werden. In Abhängigkeit der Statussignale ermittelt die Transitionsfunktion nach Ausführung des Operationswerks den Folgezustand. Bei steuerflusslosen Schaltungen entfällt die Ausgabe der Statussignale, da jeder Zustand nur einen eindeutigen Nachfolger hat.

```
OW := (λ((r1,r2,r3),(c1,c2)).
        let (x,y)   = MUX(¬(c1∨c2),(r2,r1),(r1,r2))in
        let y'      = MUX(¬c1∧c2,x,y)in
        let MULTc   = x∗y' in
        let SUBc    = r3−1 in
        let n'      = MUX(c1∧ ¬c2,SUBc,r3)in
        let ODDc    = ODD n' in
        let GTc     = n'>0 in
        let DIVc    = r3 DIV 2 in
        let r1'     = MUX(c2,MUX(c1,MULTc,y),MUX(GTc,x,y)) in
        let r2'     = MUX(c1,MUX(c2,x,MUX(GTc,y,x)),
                              MUX(c2,MULTc,MUX(GTc,MUX(ODDc,MULTc,y),x))) in
        let r3'     = MUX(c1,SUBc,MUX(c2,DIVc,r3)) in
                      ((r1',r2',r3'),(GTc,ODDc)))
```

Abbildung 5.25: Ein mögliches Operationswerk für die Spezifikation aus Abbildung 5.24

Abbildung 5.25 stellt ein mögliches Operationswerk für die Spezifikation aus Abbildung 5.24 dar. Der Typ der Ein-/Ausgabewerte β_1 ist in diesem Fall $num \times num \times num$. Die Steuereingänge bestehen aus den beiden booleschen Signalen c1 und c2. Der Typ $ctrl$ der Steuereingänge ist somit in diesem Fall $bool \times bool$. Die Statusausgabe erfolgt ebenfalls mittels zweier boolescher Signale, hat also ebenfalls den Typ $bool \times bool$. Zur Realisierung der Schaltung wurden fünf Operationseinheiten bereitgestellt: ein Multiplizierer $*$, ein Subtrahierer $-$, ein Komparator $>$, eine Einheit **ODD**, die überprüft, ob ein Wert gerade oder ungerade ist, sowie eine Einheit **DIV**, die eine Division bewirkt. Die Tabelle 5.2 listet die Belegung der Steuereingänge des Operationswerks ***OW*** in Abhängigkeit der Zustände der Schaltung auf.

Die Multiplexer in dem Operationswerk sorgen in Abhängigkeit der Steuersignale für die korrekte Beschaltung der Eingänge der Operationseinheiten und unter zusätzlicher Berücksichtigung der Ergebnisse der Operationseinheiten für die korrekte Belegung der Ausgänge des Operationswerks. Die angegebene Multiplexerstruktur kann natürlich jederzeit durch gegebenenfalls

Zustand	⟨3⟩	⟨9⟩	⟨11⟩	⟨14⟩
Steuersignale (c1,c2)	(**F**,**F**)	(**F**,**T**)	(**T**,**F**)	(**T**,**T**)

Tabelle 5.2: Konfigurationsdaten des Operationswerks in Abbildung 5.25

geeignetere kombinatorische Schaltnetze ersetzt werden. Derartige Optimierungen sind jedoch nicht Aufgabe der algorithmischen Synthese, sondern später durchzuführender Logikoptimierungen.

Die Einbettung des von dem externen Synthesealgorithmus gelieferten Operationswerks wird in dem formalen System umgesetzt, indem der Rumpf eines jeden Zustands der steuerflussorientierten Darstellung in die folgende Form überführt wird:

$$\texttt{DEF}\ (\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\underline{\mathbf{conf}}))\ \texttt{SER DEF}\ \boldsymbol{OW}\ \texttt{SER DEF}\ (\lambda(\underline{\mathbf{x}}',\underline{\mathbf{stats}}).(\underline{\mathbf{x}}',\boldsymbol{trf}\ \underline{\mathbf{stats}})) \tag{5.28}$$

Die Struktur **conf** ist eine Konstante und enthält die Konfigurationsdaten für das Operationswerk ***OW***. Nach der Tabelle 5.2 entspricht **conf** im Zustand ⟨3⟩ beispielsweise dem Wert (**F**,**F**). Die Funktion **DEF** ($\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\underline{\mathbf{conf}})$), die sogenannte *Konfigurationsfunktion*, gibt den Eingabewert **x** des Zustands unverändert an das Operationswerk weiter und ergänzt ihn um die in dem jeweiligen Zustand gültige Konfiguration **conf**. Im Anschluss an die Ausführung des Operationswerks bestimmt die Transitionsfunktion **DEF** ($\lambda(\underline{\mathbf{x}}',\underline{\mathbf{stats}}).(\underline{\mathbf{x}}',\boldsymbol{trf}\ \underline{\mathbf{stats}})$) in Abhängigkeit der Statussignale **stats** des Operationswerks den Folgezustand und gibt den Ausgabewert $\underline{\mathbf{x}}'$ des Operationswerks unverändert an den nächsten Zustand weiter. Während die Konfigurationsdaten und die Teilfunktion ***trf*** der Transitionsfunktion, welche die eigentliche Berechnung des Folgezustands durchführt, von dem jeweiligen Zustand abhängen, ist das Operationswerk selbst in allen Zuständen identisch.

Das Operationswerk ***OW*** und dessen zustandsabhängige Konfigurationen **conf** können direkt aus den Ergebnissen der Analysephase abgeleitet werden. Bevor eine Überführung der Zustandsrümpfe in die in dem Term 5.28 dargestellte Form durchgeführt werden kann, müssen noch aus den ursprünglichen Transitionsfunktionen neue, an das Operationswerk angepasste Transitionsfunktionen **DEF** ($\lambda(\underline{\mathbf{x}}',\underline{\mathbf{stats}}).(\underline{\mathbf{x}}',\boldsymbol{trf}\ \underline{\mathbf{stats}})$) abgeleitet werden. Dabei ist sicherzustellen, dass jede der neuen Transitionsfunktionen die Statusdaten zur Ermittlung des Folgezustands exakt in der gleichen Weise heranzieht wie die ursprüngliche Transitionsfunktion. Es kann zum Beispiel vorkommen, dass eine Transitionsbedingung, welche von der ursprünglichen Aktion eines Zustands auf einem bestimmten Statussignal übergeben wird, vom Operationswerk auf einem anderen Statussignal ausgegeben wird.

Im Falle des Zustands ⟨3⟩ in Abbildung 5.24 wird der Folgezustand auf Basis zweier Statussignale berechnet. Es ist nicht garantiert, dass das Operationswerk die Statussignale in der gleichen Reihenfolge wie die ursprüngliche Aktion ausgibt. Ist die Reihenfolge unterschiedlich, so muss dies beim Aufbau der neuen Transitionsfunktion berücksichtigt werden. Oft werden manche Signale, die das Operationswerk erzeugt, von einigen der Transitionsfunktionen gar nicht verwendet. Die Transitionsfunktionen der Zustände ⟨9⟩ und ⟨14⟩ sind beispielsweise von den Statussignalen unabhängig, da der Nachfolger dieser Zustände eindeutig ist.

Zur Feststellung, welche der Statussignale des Operationswerks den ursprünglichen Statussignalen der Aktion eines Zustands entsprechen, wird ein Term (**SND** ∘ ***OW*** ∘ (λ**x**.(**x**,**conf**))) aufgebaut, in welchem das Operationswerk dem betreffenden Zustand entsprechend konfiguriert wird und mit Hilfe der Funktion **SND** nur die rechte Hälfte seiner Ausgabe, also dessen Statusausgabe, selektiert wird. Anschließend wird der resultierende Term normalisiert (siehe Abschnitt 3.3.1, Seite 36).

Für den Zustand ⟨3⟩ des Beispiels ergibt die Normalisierung des entsprechenden Terms:

$$(\mathbf{SND} \circ \textit{OW} \circ (\lambda x.(x,(\mathbf{F},\mathbf{F})))) \overset{Normalisierung}{\Longrightarrow} (\lambda(r1,r2,r3).(r3{>}0,\mathbf{ODD}\ r3))$$

Der Übersichtlichkeit halber wurde das Operationswerk mit ***OW*** abgekürzt. Die Variable repräsentiert das Operationswerk aus Abbildung 5.25.

Zur Ermittlung der Statusausgabe der Aktion **DEF** ***aktion*** des entsprechenden Zustands wird deren DFG-Term ***aktion*** ebenfalls mit Hilfe der Funktion **SND** zu einem Term (**SND** ∘ ***aktion***) verknüpft und anschließend normalisiert.

Im Beispiel erhält man:

SND ∘ (λ(v,u,n).
let GT1 = n> 0 **in**
let ODD1 = **ODD** n **in**
let MULT1 = u∗v **in**
let o1 = **MUX**(GT1,u,v) **in**
let o2 = **MUX**(ODD1,MULT1,v) **in**
let o2' = **MUX**(GT1,o2,u) **in**
((o1,o2',n),ODD1,GT1))

$\overset{Normalisierung}{\Longrightarrow}$ (λ(v,u,n).(**ODD** n,n>0))

Eine Gegenüberstellung der beiden normalisierten Terme ergibt, dass die beiden Statusausgaben des Operationswerks gegenüber den Statusausgaben der ursprünglichen Aktion vertauscht sind. Entsprechend werden zur Erstellung der neuen Transitionsfunktion, welche im Anschluss an das Operationswerk ausgeführt werden soll, die beiden eingehenden Statussignale ODD1 und GT1 der ursprünglichen Transitionsfunktionen vertauscht. Der Körper der Abstraktion bleibt unverändert. Mit dieser neuen Transitionsfunktion ergibt sich schließlich der neue Rumpf des Zustands ⟨3⟩ zu folgendem Term:

DEF (λx.(x,(**F**,**F**)))
SER DEF ***OW***
SER DEF (λ(**x**,GT1,ODD1).(**x**,**MUX**(GT1,**MUX**(ODD1,11,9),0)))

Es verbleibt, die neue Transitionsfunktion als Vorbereitung für die Abbildung der Schaltungsbeschreibung auf die RT-Ebene in die Form **DEF** (λ(**x**′,**stats**).(**x**′,***trf*** **stats**)) zu überführen:

DEF (λ(**x**,GT1,ODD1).(**x**,**MUX**(GT1,**MUX**(ODD1,11,9),0)))
⟹ **DEF** (λ(**x**,**stats**).(**x**,(λ(GT1,ODD1).(**x**,**MUX**(GT1,**MUX**(ODD1,11,9),0))) **stats**))

Durch diesen Schritt wird explizit hervorgehoben, dass die Funktion ***trf***, die innerhalb der Transitionsfunktion letztendlich den Folgezustand bestimmt, allein von den Statussignalen des Operationswerks abhängt. Der Rumpf weist damit die in dem Term 5.28 vorgestellte Form auf.

Nach dem Aufbau eines neuen Rumpfes (***Konfigfkt*** **SER DEF** ***OW*** **SER** ***Transitionsfkt′***) wird jeweils bewiesen, dass er exakt die gleiche Funktionalität wie der ursprüngliche Rumpf (***Aktion*** **SER** ***Transitionsfkt***) aufweist. Ist dies der Fall, so kann der alte Rumpf endgültig durch den neu generierten Rumpf ersetzt werden. Für den Beweis muss gelten, dass für alle Eingaben $\underline{\mathbf{x}}$ der neue und der alte Rumpf das gleiche Ergebnis liefern, dass also gilt:

$$\forall \underline{\mathbf{x}}.\ (\textit{Aktion}\ \mathbf{SER}\ \textit{Transitionsfkt})\ \underline{\mathbf{x}} = (\textit{Konfigfkt}\ \mathbf{SER\ DEF}\ \textit{OW}\ \mathbf{SER}\ \textit{Transitionsfkt'})\ \underline{\mathbf{x}} \quad (5.29)$$

Zur Durchführung dieses Beweises wird die Konfigurationsfunktion mit dem Operationswerk und der neuen Transitionsfunktion verschmolzen. Ebenso werden die Aktion und die Transitionsfunktion des ursprünglichen Rumpfes zusammengefasst. Anschließend werden beide Terme normalisiert. Ein Vergleich der resultierenden Terme, in Zuge dessen gegebenenfalls eine Fallunterscheidung über die Bedingungen der verbleibenden Multiplexer durchgeführt wird und bei Bedarf noch weitere Eigenschaften der Operationen, wie zum Beispiel deren Kommutativität, in Betracht gezogen werden, ergibt die funktionale Äquivalenz der beiden Rümpfe. Der ursprüngliche Rumpf kann daraufhin durch die neue Darstellung mit dem Operationswerk ersetzt werden.

Mit der Überführung aller Rümpfe der steuerflussorientierten Darstellung in die neue Form mit einem einheitlichen Operationswerk wird die Bereitstellung und Zuweisung der Operationseinheiten vollständig umgesetzt.

Abbildung 5.26 zeigt das Beispiel aus Abbildung 5.24 nach der Umsetzung der Bereitstellung und Zuweisung der Operationseinheiten gemäß dem Operationswerk ***OW*** aus Abbildung 5.25 und dessen Konfiguration in Tabelle 5.2.

```
DEF (λ(u,n).(1,u,n)))
SER
LOOP 3
 [( 3, DEF (λx.(x,(F,F))) SER DEF OW
       SER DEF (λ(x,stats).(x,(λ(GT1,ODD1).MUX(GT1,MUX(ODD1,11,9),0)) stats)));
  ( 9, DEF (λx.(x,(F,T))) SER DEF OW
       SER DEF (λ(x,stats).(x,(λ(GT1,ODD1).3) stats)));
  (11, DEF (λx.(x,(T,F))) SER DEF OW
       SER DEF (λ(x,stats).(x,(λ(GT1,ODD1).MUX(GT1,MUX(ODD1,14,9),0)) stats)));
  (14, DEF (λx.(x,(T,T))) SER DEF OW
       SER DEF (λ(x,stats).(x,(λ(GT1,ODD1).3) stats)))]
SER
DEF (λ(v,u,n).v)
```

Abbildung 5.26: Ergebnis der Bereitstellung und Zuweisung der Operationseinheiten

5.6 Synthese des Steuerwerks

Die Spezifikation einer Schaltung besteht neben der Beschreibung ihres funktionalen Verhaltens aus der Auswahl eines geeigneten Schnittstellenverhaltensmusters, welches die Kommunikation der Schaltung mit der Außenwelt definiert (siehe Abschnitt 3.3.3). Das gewählte Schnittstellenverhaltensmuster hat direkten Einfluss auf das Steuerwerk der resultierenden RT-Ebenen-Implementierung.

Für jedes der in Abschnitt 3.3.3 vorgestellten Schnittstellenverhaltensmuster wurde exemplarisch ein spezielles *Implementierungsmuster* auf der Register-Transfer-Ebene entwickelt und jeweils der Beweis erbracht, dass jede RT-Ebenen-Implementierung, die auf Basis eines solchen Implementierungsmusters erzeugt wird, ein Schnittstellenverhalten aufweist, welches den Vorgaben des zugehörigen Schnittstellenverhaltensmusters entspricht. Die aus diesen Beweisen resultierenden Theoreme werden als *Implementierungstheoreme* bezeichnet. Kern eines jeden Implementierungsmusters ist dabei ein vorgegebenes Steuerwerk, welches die Steuerung der Schaltung und deren Kommunikation mit der Außenwelt exakt nach den Vorgaben des zugehörigen Schnittstellenverhaltensmusters realisiert. Durch die Spezialisierung der freien Variablen eines Implementierungsmusters wird schließlich die vom Entwerfer spezifizierte Funktionalität der Schaltung in das jeweilige Muster eingebracht und auf diese Weise eine konkrete Implementierung der Schaltung auf der Register-Transfer-Ebene erzeugt.

Gleichung 5.30 gibt als Beispiel die Definition des Implementierungsmusters **GEN_IMP_P** wieder, welches ein Verhalten realisiert, das dem auf Seite 51 definierten Schnittstellenverhaltensmuster **GEN_SSVM** entspricht.

```
GEN_IMP_P(inputsq,resetsq,startsq,outputsq,readysq,                    (5.30)
          init,s0,CFG,OW,TRF,outsel) :=
  ∃ register0 statereg0 outputreg0.
    (λt.(outputsq t,readysq t)) =
    automaton
      ((λ((inputsig,resetsig,startsig),(register,statereg,outputreg,readyreg)).
        let resorrdy     = resetsig ∨ readyreg in
        let begin        = startsig ∧ resorrdy in
        let state        = MUX(begin,s0,statereg) in
        let initx        = init inputsig in
        let x            = MUX(begin,initx,register) in
        let idle         = ¬startsig ∧ resorrdy in
        let ctrl         = CFG state in
        let (x',stats)   = OW (x,ctrl) in
        let (state',cont) = TRF state stats in
        let readysig     = idle ∨ ¬cont in
        let finoutput    = outsel x' in
        let outputsig    = MUX(idle,outputreg,finoutput) in
            ((outputsig,readysig),(x',state',outputsig,readysig))),
            (register0,statereg0,outputreg0,T))
      (λt. (inputsq t,resetsq t,startsq t))
```

Zur Definition wird ein Zusammenhang zwischen den Eingabesignalen und den Ausgabesignalen der Musterimplementierung hergestellt. Die Ausgabesequenzen ***outputsq*** und ***readysq*** ergeben sich aus der Ausführung des Automaten auf den Eingabesequenzen ***inputsq***, ***resetsq*** und ***startsq***. Die Sequenzen beschreiben den Werteverlauf der zugehörigen Signale in Abhängigkeit von der Zeit t, welche mit Hilfe des Typs num modelliert wird (siehe Abschnitt 3.3.2, Seite 42). Die Werte **register0**, **statereg0** und **outputreg0** bezeichnen die Initialwerte der Register der Schaltung und können frei gewählt werden. Die Funktionalität der Schaltung wird durch die freien Variablen ***init***, **s0**, ***CFG***, ***OW***, ***TRF*** und ***outsel*** vorgegeben, welche im Zuge der Abbildung der Schaltung auf die RT-Ebene mit konkreten Werten belegt werden. Von diesen Variablen stehen beispielsweise **s0** für den Anfangszustand und ***OW*** für das Operationswerk der Schaltung.

In Abbildung 5.27 ist das Implementierungsmuster **GEN_IMP_P** zusätzlich schematisch dargestellt.

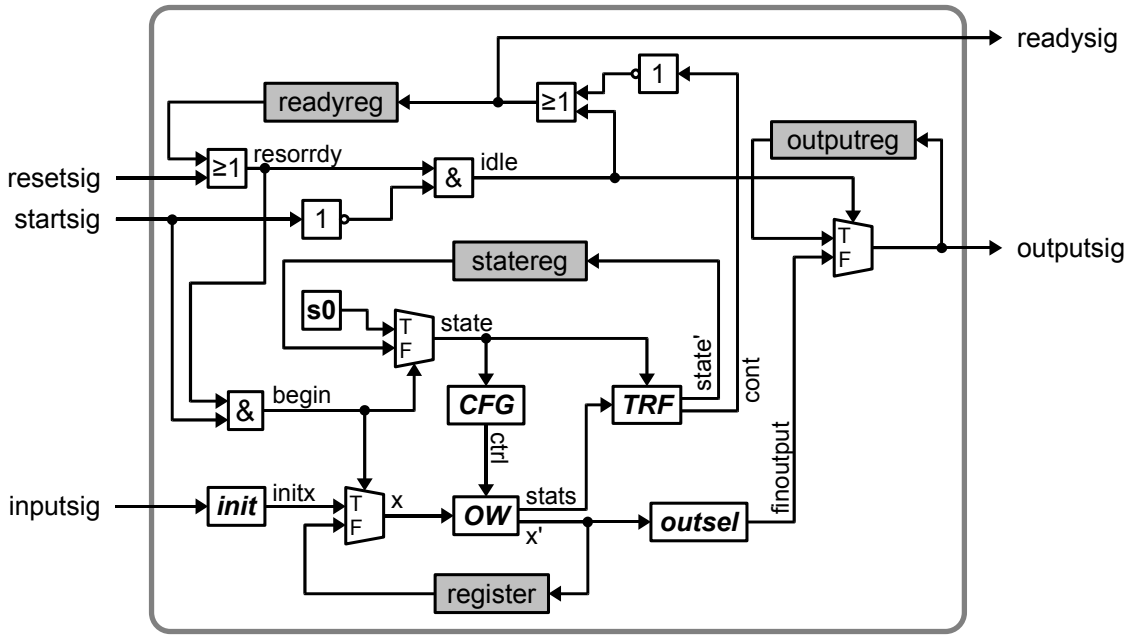


Abbildung 5.27: Implementierungsmuster **GEN_IMP_P**

Die bis zu diesem Zeitpunkt vorgestellten Transformationen operierten stets innerhalb der algorithmischen Ebene und ergaben somit immer Theoreme, welche die Äquivalenz zwischen dem jeweiligen Ergebnis der Transformation und deren Ausgangspunkt ausdrückten (siehe Abschnitt 3.2.1). Im Gegensatz dazu ist im Fall der Implementierungstheoreme die resultierende Relation eine Implikation (siehe Abschnitt 3.2.2).

In dem Implementierungstheorem 5.31, welches die Relation zwischen dem Schnittstellenverhaltensmuster **GEN_SSVM** und dem Implementierungsmuster **GEN_IMP_P** beschreibt, ist die entsprechende Implikation, welche das Ergebnis der Transformation darstellen wird, in den Zeilen 13-17 enthalten.

Im Falle der Implementierungstheoreme sind die Spezifikation und die Implementierung nicht mehr äquivalent, da beim Wechsel von der algorithmischen Ebene auf die Register-Transfer-Ebene eine Verfeinerung der Spezifikation vorgenommen wird. Teil dieser Verfeinerung ist beispielsweise, dass für die Durchführung einer Berechnung nach dem Schnittstellenverhaltensmuster **GEN_SSVM** nur vorgegeben ist, dass nach dem Anstoßen der Berechnung deren Durchführung eine nicht näher bestimmte Zeit in Anspruch nimmt und, falls sie nicht abgebrochen wird und schließlich terminiert, das Ergebnis der Berechnung am Ausgang der Schaltung anliegt und zeitgleich der Wert des Ausgabesignals readysig von *falsch* auf *wahr* wechselt. In der RT-Ebenen-Implementierung ist zusätzlich genau definiert, wann welche Operationen der Berechnung ausgeführt werden und welche Zwischenwerte von der Implementierung während der Berechnungsphase über outputsig ausgegeben werden. Es handelt sich dabei also um eine wesentlich detailliertere Beschreibung.

$$
\begin{array}{ll}
 & \vdash \forall\ \textit{inputsq resetsq startsq outputsq readysq init}\ \mathbf{s0}\ \textit{OW outsel CTList} \\
 & \quad \textit{CFG TRF}. \\
1 & \quad \texttt{let}\ \textit{L} = \texttt{MAP}\ (\lambda(\mathbf{s},\underline{\mathbf{cnf}},\textit{trf}). \\
2 & \qquad\qquad (\mathbf{s},\ \texttt{DEF}\ (\lambda\underline{\mathbf{x}}.\ (\underline{\mathbf{x}},\underline{\mathbf{cnf}}))\ \texttt{SER DEF}\ \mathrm{OW} \\
3 & \qquad\qquad\quad \texttt{SER DEF}\ (\lambda(\underline{\mathbf{x}'},\underline{\mathbf{stats}}).\ (\underline{\mathbf{x}'},\textit{trf}\ \underline{\mathbf{stats}})))))\ \textit{CTList}\ \texttt{in} \\
4 & \quad \texttt{let}\ \textit{cnfBlk} = \texttt{StateCase} \\
5 & \qquad\qquad (\texttt{MAP}\ \big(\lambda(\mathbf{s},\underline{\mathbf{cnf}},\textit{trf}).\ (\mathbf{s},\underline{\mathbf{cnf}})\big)\ \textit{CTList})\ \texttt{in} \\
6 & \quad \texttt{let}\ \textit{trfBlk} = \texttt{StateCase} \\
7 & \qquad\qquad (\texttt{MAP}\ \big(\lambda(\mathbf{s},\underline{\mathbf{cnf}},\textit{trf}).(\mathbf{s},\ (\lambda\mathbf{s}'.(\mathbf{s}',\texttt{StateExists}\ \textit{L}\ \mathbf{s}'))\circ\textit{trf})\big) \\
8 & \qquad\qquad\quad \textit{CTList})\ \texttt{in} \\
9 & \quad \texttt{StateExists}\ \textit{L}\ \mathbf{s0} \wedge \\
10 & \quad \texttt{ALL\_EL}\ (\lambda\mathbf{s}.\ (\textit{CFG}\ \mathbf{s} = \textit{cnfBlk}\ \mathbf{s}) \wedge (\textit{TRF}\ \mathbf{s} = \textit{trfBlk}\ \mathbf{s})) \\
11 & \qquad\quad (\texttt{FST}\ (\texttt{UNZIP}\ \textit{CTList})) \wedge \\
12 & \quad \Longrightarrow \\
13 & \quad \big(\ \texttt{GEN\_IMP\_P}\ (\textit{inputsq},\textit{resetsq},\textit{startsq},\textit{outputsq},\textit{readysq}, \\
14 & \qquad\quad \textit{init},\mathbf{s0},\textit{CFG},\textit{OW},\textit{TRF},\textit{outsel}) \\
15 & \qquad \Longrightarrow \\
16 & \qquad \texttt{GEN\_SSVM}\ (\texttt{DEF}\ \textit{init}\ \texttt{SER LOOP}\ \mathbf{s0}\ \textit{L}\ \texttt{SER DEF}\ \textit{outsel}) \\
17 & \qquad\qquad\quad (\textit{startsq},\textit{resetsq},\textit{readysq},\textit{inputsq},\textit{outputsq})\quad \big)
\end{array}
\tag{5.31}
$$

Die Konsistenz der Implementierungsmuster wurde in dieser Arbeit durch den Beweis sichergestellt, dass es für alle möglichen Eingabesequenzen unter beliebiger Belegung der Variablen ***init***, **s0**, ***CFG***, ***OW***, ***TRF*** und ***outsel***, welche die Funktionalität der Schaltung widerspiegeln, gültige Ausgabesequenzen gibt. Entsprechend wurde diese Eigenschaft auch für das Implementierungsmuster **GEN_IMP_P** nachgewiesen:

$$
\begin{array}{l}
\vdash \forall\ \textit{inputsq resetsq startsq init}\ \mathbf{s0}\ \textit{CFG OW TRF outsel}. \\
\quad \exists\ \textit{outputsq readysq}. \\
\qquad \texttt{GEN\_IMP\_P}\ (\textit{inputsq},\textit{resetsq},\textit{startsq},\textit{outputsq},\textit{readysq},\textit{init},\mathbf{s0},\textit{CFG},\textit{OW},\textit{TRF}, \\
\qquad\qquad \textit{outsel})
\end{array}
\tag{5.32}
$$

Wird für ein Implementierungsmuster kein entsprechender Beweis durchgeführt, so ist nicht ausgeschlossen, dass sich die auf Basis dieses Musters erzeugten Implementierungen als inkonsistent erweisen (siehe [Krop99]) und der im Implementierungsmuster vorgegebene Zusammenhang zwischen den Ein- und Ausgabesignalen der resultierenden Implementierungen niemals erfüllt ist. In diesem Fall wäre der Beweis eines Implementierungstheorems trivial, da der linke Teil der Implikation (beispielsweise die Zeilen 13-14 in Theorem 5.31) stets *falsch*, die Implikation aber dadurch grundsätzlich *wahr* wäre (ex falso quodlibet). Der Beweis eines entsprechenden Implementierungstheorems wäre somit nur ein Scheinerfolg.

Die Implementierungstheoreme beschreiben generisch die korrekte Abbildung beliebiger Verhaltensspezifikationen auf die Register-Transfer-Ebene. Voraussetzung für den Einsatz der im Zuge dieser Arbeit entwickelten Implementierungstheoreme ist, dass die Spezifikationen, wie in den letzten Abschnitten beschrieben, in eine bestimmte Form überführt worden sind. Die geforderte Form entspricht dem Ausgabeformat der im letzten Abschnitt vorgestellten Operationswerksynthese, nach welcher die Zustände der steuerflussorientierten Darstellung folgende Struktur aufweisen:

$$(s, \mathbf{DEF}\ (\lambda\underline{\mathbf{x}}.(\underline{\mathbf{x}},\underline{\mathbf{conf}}))\ \mathbf{SER}\ \mathbf{DEF}\ \boldsymbol{OW}\ \mathbf{SER}\ \mathbf{DEF}\ (\lambda(\underline{\mathbf{x}'},\underline{\mathbf{stats}}).(\underline{\mathbf{x}'},\boldsymbol{trf}\ \underline{\mathbf{stats}})))$$

Als Vorbereitung der Anwendung des gewählten Implementierungstheorems wird auf Basis der steuerflussorientierten Darstellung eine Liste ***CTList*** aufgebaut, in welcher für jeden Zustand **s** der Zustandsliste die zugehörige Konfiguration **conf** des Operationswerks sowie der zustandsabhängige Anteil ***trf*** der betreffenden Transitionsfunktion enthalten sind. In Abbildung 5.28 ist diese sogenannte *Konfigurations-/Transitionsliste* dargestellt, wie sie sich aus der Spezifikation in Abbildung 5.26 ergibt.

```
[(  3, (F,F), (λ(GT1,ODD1).MUX(GT1,MUX(ODD1,11,9),0)));
 (  9, (F,T), (λ(GT1,ODD1).3));
 (11, (T,F), (λ(GT1,ODD1).MUX(GT1,MUX(ODD1,14,9),0)));
 (14, (T,T), (λ(GT1,ODD1).3))]
```

Abbildung 5.28: Konfigurations-/Transitionsliste des laufenden Beispiels

Nach der Spezialisierung des Implementierungstheorems 5.31 mit der Konfigurations-/Transitionsliste ***CTList*** und dem Operationswerk ***OW***, welches im Zuge der Operationswerksynthese erstellt wurde (siehe Abbildung 5.25), wird in den Zeilen 1-3 des Theorems die Zustandsliste ***L*** der steuerflussorientierten Darstellung nachgebaut. Der explizite Nachbau der Schaltungsspezifikation innerhalb des Implementierungstheorems wird durchgeführt, um sicherzustellen, dass die Schaltungsspezifikation in der geforderten Form vorliegt und die Konfigurations-/Transitionsliste ***CTList*** korrekt aus der Schaltungsspezifikation extrahiert wurde: Nach der Belegung der Variablen ***init*** und ***outsel*** mit den DFG-Termen, welche in den **DEF**-Anweisungen vor bzw. nach der steuerflussorientierten Darstellung auftreten (im Beispiel in Abbildung 5.26: (λ(u,n).(1,u,n)) und (λ(v,u,n).v)), und der Belegung von **s0** mit dem Startzustand der steuerflussorientierten Darstellung ergibt sich schließlich in Zeile 16 des Theorems als Parameter des Schnittstellenverhaltens-

musters **GEN_SSVM** ein exaktes Abbild der zu diesem Zeitpunkt vorliegenden Schaltungsspezifikation.

Man beachte, dass die Schaltungsspezifikation, welche aus der Operationswerksynthese hervorging, allein auf Basis von Äquivalenzumformungen aus der ursprünglichen Verhaltensspezifikation abgeleitet wurde. Somit liegt innerhalb des formalen Systems ein Theorem vor, welches die Äquivalenz dieser Spezifikation und der ursprünglichen Verhaltensspezifikation ausdrückt (siehe Abschnitt 3.2).

Dass es sich bei der innerhalb des Implementierungstheorems 5.31 nachgebildeten Schaltungsbeschreibung um ein exaktes Abbild der vorliegenden Schaltungsspezifikation handelt, wird letztendlich sichergestellt, indem diese Schaltungsbeschreibung mittels einfacher Termersetzung im Theorembeweiser durch den Ausgangspunkt der algorithmischen Synthese, der ursprünglichen Verhaltensspezifikation (siehe Abbildung 3.16 auf Seite 47), ersetzt wird. Während der Termersetzung führt der Theorembeweiser HOL eine Mustererkennung durch und nur, wenn die im Implementierungstheorem nachgebildete Spezifikation exakt derjenigen entspricht, welche das Ergebnis der Operationswerksynthese darstellt, findet die Termersetzung statt. Ansonsten ist offensichtlich ein Fehler beim Erstellen der Liste ***CTList*** bzw. bei der Analyse der konkreten Werte für ***init***, ***outsel*** bzw. **s0** aufgetreten.

Im Anschluss an die Reproduktion der Ausgangsspezifikation wird in den Zeilen 4-5 des Implementierungstheorems 5.31 auf Basis der Liste ***CTList*** automatisch die Spezifikation eines Funktionsblocks generiert, welcher in Abhängigkeit des Zustands die korrekte Steuerung des Operationswerks übernimmt. Im Beispiel ergibt sich der Konfigurationsblock ***cnfBlk*** zu:

StateCase [(3,(**F**,**F**)); (9,(**F**,**T**)); (11,(**T**,**F**)); (14,(**T**,**T**))]

Auf ähnliche Weise wird in den Zeilen 6-8 des Implementierungstheorems auf Basis der Konfigurations-/Transitionsliste ***CTList*** ein Funktionsblock generiert, welcher in Abhängigkeit des Zustands und der Statussignale des Operationswerks den Folgezustand bestimmt. Mit einem zusätzlichen booleschen Wert gibt der Transitionsblock außerdem an, ob die Berechnung bereits abgeschlossen ist oder nicht. Bei dem in dieser Arbeit vorgestellten Ansatz ist die Ermittlung der Abbruchbedingung bereits am Ende jedes Taktes möglich. In [Blum00] war zur Berechnung der Abbruchbedingung noch ein zusätzlicher Takt erforderlich.

Der in dem laufenden Beispiel resultierende Transitionsblock ***trfBlk*** lautet:

StateCase [(3,(λ(GT1,ODD1).**MUX**(GT1,**MUX**(ODD1,(11,**T**),(9,**T**)),(0,**F**))));
(9,(λ(GT1,ODD1).(3,**T**)));
(11,(λ(GT1,ODD1).**MUX**(GT1,**MUX**(ODD1,(14,**T**),(9,**T**)),(0,**F**))));
(14,(λ(GT1,ODD1).(3,**T**)))]

Der Konfigurations- und der Transitionsblock nutzen beide das Konstrukt **StateCase** und definieren somit nur die Ausgaben für die wirklich existierenden Zustände. Die Ausgabe der beiden Blöcke für andere Zustände werden nicht unnötig vorbelegt, was das Optimierungspotential der Schaltung auf der RT-Ebene erhöht. Freiheitsgrade bei der weiteren Synthese bleiben so erhalten. Stehen die endgültigen Implementierungen der beiden Funktionsblöcke einmal fest,

so wird bewiesen, dass diese Implementierungen (nur) für alle existierenden Zustände der Zustandliste die gleiche Ausgabe erzeugen wie die Spezifikation der Funktionsblöcke. Wurde dieser Nachweis erbracht (entsprechend der Zeilen 10-11 im Implementierungstheorem 5.31), so können die tatsächlichen Implementierungen ***CFG*** und ***TRF*** der beiden Funktionsblöcke in die RT-Ebenen-Implementierung der Schaltung übernommen werden (Zeile 13-14).

Abschließend muss im Zuge der Transformation noch nachgewiesen werden, dass der Startzustand **s0** der steuerflussorientierten Darstellung existiert (Zeile 9). Alle weiteren Syntheseschritte, wie zum Beispiel Retiming und diverse Logikoptimierungen, gehen von dem resultierenden Automaten auf der RT-Ebene aus und werden ausführlich in [Eise99] behandelt.

Steuerwerke für reine Datenflussbeschreibungen

Da bei der Synthese reiner Datenflussspezifikationen die Übergänge zwischen den einzelnen Zuständen stets eindeutig sind, wurde für jedes Implementierungstheorem eine zusätzliche Variante entwickelt, in welcher der Transitionsblock nicht von Statussignalen des Operationswerks abhängt. Weiter kann der Fall auftreten, dass das Operationswerk keiner zustandsabhängigen Steuerung bedarf. Somit ergibt sich eine weitere Variationsmöglichkeit, so dass letztendlich für jedes Schnittstellenverhaltensmuster zumindest vier verschiedene Implementierungsmuster bzw. Implementierungstheoreme entwickelt werden müssen.

Eine Variante zur Implementierung des Schnittstellenverhaltensmusters **GEN_SSVM** für reine Datenflussbeschreibungen und ohne zustandsabhängige Steuerung des Operationswerks hat beispielsweise folgende vereinfachte Form:

$$\vdash \forall\ \textit{inputsq resetsq startsq outputsq readysq init}\ \mathbf{s0}\ \textit{OW outsel CTList TRF}. \qquad (5.33)$$

```
1   let L = MAP (λ(s,s'). (s, DEF OW SER DEF (λx'. (x',s')))) CTList in
2   let trfBlk = StateCase
3                  (MAP (λ(s,s').(s,(s',StateExists L s'))) CTList) in
4   StateExists L s0 ∧
5   ALL_EL (λs. (TRF s = trfBlk s)) (FST (UNZIP CTList)) ∧
6   ⟹
7   ( GEN_IMP_DFG_S (inputsq,resetsq,startsq,outputsq,readysq,
8                    init,s0,OW,TRF,outsel)
9     ⟹
10    GEN_SSVM (DEF init SER LOOP s0 L SER DEF outsel)
11             (startsq,resetsq,readysq,inputsq,outputsq)  )
```

Entsprechend gestaltet sich die zugehörige Implementierung **GEN_IMP_DFG_S** ebenfalls etwas einfacher. Der Funktionsblock zur Übernahme der Konfiguration des Operationswerks

entfällt und der Transitionsblock hängt nur noch vom aktuellen Zustand ab:

```
GEN_IMP_DFG_S (inputsq,resetsq,startsq,outputsq,readysq,init,s0,OW,TRF,outsel) :=
  ∃ register0 statereg0 outputreg0.
    (λt.(outputsq t,readysq t))
    =
    automaton
      ((λ((inputsig,resetsig,startsig),(register,statereg,outputreg,readyreg)).
        let resorrdy     = resetsig ∨ readyreg in
        let begin        = startsig ∧ resorrdy in
        let state        = MUX(begin,s0,statereg) in
        let initx        = init inputsig in
        let x            = MUX(begin,initx,register) in
        let idle         = ¬startsig ∧ resorrdy in
        let x'           = OW x in
        let (state',cont) = TRF state in
        let readysig     = idle ∨ ¬cont in
        let finoutput    = outsel x' in
        let outputsig    = MUX(idle,outputreg,finoutput) in
            ((outputsig,readysig),(x',state',outputsig,readysig))),
            (register0,statereg0,outputreg0,T))
    (λt. (inputsq t,resetsq t,startsq t))
```

(5.34)

Umkodierung der Zustandsbezeichner

Oft bietet es sich an, vor der Abbildung der Spezifikation auf die RT-Ebene die Zustandsbezeichner umzukodieren. Beispielsweise hat die Zustandskodierung Einfluss auf die Implementierungen der Konfigurations- und der Transitionsblöcke. Durch eine geschickte Wahl der Zustandsbezeichner können diese gegebenenfalls optimiert werden. Auch können aufeinanderfolgende Zustände mit einer möglichst kleinen Hamming-Distanz kodiert werden, um die Schaltaktivität in der resultierenden Schaltung und damit deren Energiebedarf zu reduzieren.

Zur Umbenennung der Zustände wird eine sogenannte *Umbenennungsliste* aufgebaut. Diese Liste bestehend aus Paaren, deren linke Hälfte den Bezeichnern der Zustände in der steuerflussorientierten Darstellung entsprechen. In der rechten Hälfte der Paare steht jeweils der neu zu vergebende Name. Für alle Zustände, die nicht in der Zustandsliste der steuerflussorientierten Darstellung vorkommen, also für die Ausgangszustände, wird zusätzlich außerhalb der Liste ein eindeutiger Name spezifiziert.

Für das Beispiel in Abbildung 5.26 könnte eine Umbenennung auf Basis folgender Liste erfolgen.

[(3,(**F**,**F**,**T**)); (9,(**F**,**T**,**T**)); (11,(**T**,**F**,**T**)); (14,(**T**,**T**,**T**))]

Zusätzlich muss ein Bezeichner für den Ausgangszustand spezifiziert werden. In diesem Beispiel wäre der Bezeichner (**F**,**F**,**F**) denkbar. Würde in dem laufenden Beispiel eine derartige

Anpassung der Bezeichner vor der Abbildung der Spezifikation auf die RT-Ebene auf Basis des Implementierungstheorems 5.30 durchgeführt, so ließe sich der in diesem Fall resultierende Konfigurationsblock **StateCase** [((**F**,**F**,**T**),(**F**,**F**)); ((**F**,**T**,**T**),(**F**,**T**)); ((**T**,**F**,**T**),(**T**,**F**)); ((**T**,**T**,**T**),(**T**,**T**))] durch die einfache Funktion (λ(s1,s2,s3).(s1,s2)) realisieren. Der folgende Term stellt die Implementierung eines für diesen Fall geeigneten Transitionsblocks dar:

TRF := (λ(s1,s2,s3).
λ(ODD1,GT1).
let op1 = ¬s2 ∧ s3 **in**
let op2 = ¬s1 ∧ op1 **in**
let op3 = s1 ∧ op1 **in**
let op4 = GT1 ∨ ¬op2 ∧ ¬op3 **in**
((ODD1 ∧ GT1 ∧ (op2 ∨ op3),
GT1 ∧ **MUX**(op2,¬ODD1,op3),op4),op4))

Im Theorem 5.36 zur Umkodierung der Zustandsbezeichner wird die Funktion **NewStateName** eingesetzt, welche jedem alten Zustandsbezeichner auf Basis einer gegebenen Umbenennungsliste einen neuen Bezeichner zuordnet:

(**NewStateName** ***exit*** [ ] ***s*** := ***exit***) ∧ (5.35)
(**NewStateName** ***exit*** (***h*** : : ***R***) ***s*** := (if ***s*** = **FST** ***h*** then **SND** ***h***
else **NewStateName** ***exit R s***))

Die Funktion durchsucht rekursiv die vorgegebene Umbenennungsliste (***h*** : : ***R***). Wird ein Paar ***h*** gefunden, dessen linke Hälfte **FST** ***h*** dem gesuchten Zustandsbezeichner ***s*** entspricht, so wird der neue Bezeichner, der in der rechten Hälfte des entsprechenden Paares enthalten ist, ausgegeben. Ansonsten wird der Rest der Liste weiter durchsucht. Enthält die Liste keinen Eintrag für den gesuchten Zustandsbezeichner ***s***, so wird der außerhalb der Liste zusätzlich definierte Bezeichner ***exit*** ausgegeben.

Die Umbenennung der Zustandsbezeichner erfolgt schließlich auf Basis des folgenden Theorems:

$\vdash \forall$ ***L UL*** **start** ***exit***. (5.36)
1 (**FST** (**UNZIP** ***UL***) = **FST** (**UNZIP** ***L***)) ∧
2 **Disjoint** (***exit*** : : (**SND** (**UNZIP** ***UL***)))
3 $\Longrightarrow$ (**LOOP start** ***L***
4 =
5 (**LOOP** (**RenameState** ***exit UL*** **start**)
6 (**MAP** (λ(**s**,***Rumpf***).
7 (**RenameState** ***exit UL*** **s**,
8 ***Rumpf***
9 **SER**
10 **DEF** (λ($\underline{\mathbf{x}}'$,**s**′).($\underline{\mathbf{x}}'$,**RenameState** ***exit UL*** **s**′)))) ***L***))

Vorbedingungen für die Anwendbarkeit des Theorems sind, dass die alten Zustandsbezeichner in der Umbenennungsliste ***UL*** den Bezeichnern der Zustände der aktuellen Zustandsliste ***L*** entsprechen (Zeile 1), und dass die in der Umbenennungsliste spezifizierten neuen Bezeichner sowie der neue für den Ausgangszustand vorgegebene Bezeichner ***exit*** wechselseitig verschieden sind (Zeile 2). Sind diese Voraussetzungen erfüllt, so kann die gegebene **`LOOP`**-Anweisung (Zeile 3) in eine neue steuerflussorientierte Darstellung mit den modifizierten Zustandsnamen (Zeile 5-10) überführt werden.

In der neuen Darstellung wird jeder Zustandsbezeichner auf Basis der Funktion **`RenameState`** umbenannt: In Zeile 5 wird dem Startzustand ein neuer Bezeichner zugewiesen und in den Zeilen 6-10 werden alle Zustandsbezeichner angepasst, welche innerhalb der Zustandsliste ***L*** auftreten. Jeder Zustand wird dahingehend modifiziert, dass deren ursprünglicher Bezeichner **s** durch den zugehörigen neuen Bezeichner **`RenameState`** ***exit UL*** **s** ersetzt wird (Zeile 7) und dem ursprünglichen Rumpf ***Rumpf*** (Zeile 8) eine Funktion zur Anpassung der Bezeichner der Nachfolger (Zeile 10) angehängt wird. Nach Anwendung des Theorems werden die an die Rümpfe angehängten Funktionen schließlich mit den jeweiligen Transitionsfunktionen verschmolzen.

Endergebnis der algorithmischen Synthese

Werden die Zustände des laufenden Beispiels wie im letzten Abschnitt beschrieben umkodiert und entsprechend die vorgeschlagenen Implementierungen des Konfigurations- und Transitionsblocks eingesetzt, so erhält man in dem Beispiel als Endergebnis der formalen algorithmischen Synthese folgendes Theorem:

$$
\begin{array}{l}
\vdash \forall\ \textit{inputsq resetsq startsq outputsq readysq}. \\
\quad \big(\ \texttt{GEN\_IMP\_P}\ (\textit{inputsq},\textit{resetsq},\textit{startsq},\textit{outputsq},\textit{readysq}, \\
\qquad\qquad (\lambda(\mathrm{u,n}).(1,\mathrm{u,n}))),(\texttt{F},\texttt{F},\texttt{T}),(\lambda(\mathrm{s1,s2,s3}).(\mathrm{s1,s2})),\ \textit{OW},\textit{TRF},(\lambda(\mathrm{v,u,n}).\mathrm{v})) \\
\quad\ \Longrightarrow \\
\quad\ \texttt{GEN\_SSVM}\ \textit{SPEC}\ (\textit{startsq},\textit{resetsq},\textit{readysq},\textit{inputsq},\textit{outputsq})\ \big)
\end{array}
$$

SPEC steht für die Verhaltensspezifikation aus Abbildung 3.16 auf Seite 47, welche Ausgangspunkt der Synthese war. ***TRF*** repräsentiert die auf der Seite 156 vorgeschlagene Implementierung des Transitionsblocks und ***OW*** das Operationswerk aus Abbildung 5.25 auf Seite 145. Das Endergebnis ist also der formale Beweis, dass die sich ergebende Implementierung auf Basis des Musters **`GEN_IMP_P`** mit den im Zuge der algorithmischen Synthese abgeleiteten Parametern exakt die ursprüngliche Verhaltensbeschreibung ***SPEC*** mit einem dem Muster **`GEN_SSVM`** entsprechenden Schnittstellenverhalten realisiert. Diese Implementierung ist nun Ausgangspunkt für die weiteren Syntheseschritte auf den unteren Abstraktionsebenen (siehe [Eise99]).

Kapitel 6

Experimentelle Ergebnisse

Um die Leistungsfähigkeit des in dieser Arbeit entwickelten Konzepts zur formalen algorithmischen Synthese digitaler Schaltungen zu demonstrieren, wurde ein Synthesesystem implementiert, das eine Verhaltensbeschreibung in GROPIUS diesem Konzept entsprechend vollautomatisch auf eine RT-Ebenen-Implementierung abbildet. Das Synthesesystem wurde auf zahlreiche Beispiele angewandt, welche zu diesem Zweck in die formale Hardwarebeschreibungssprache GROPIUS übersetzt wurden. In Tabelle 6.1 sind zu jedem Beispiel die Anzahl der enthaltenen Operationen sowie der jeweils auftretenden **`WHILE`**-, **`IF`**- und **`SER`**-Konstrukte aufgeführt.

Programm	Operationen	**`WHILE`**	**`IF`**	**`SER`**	**`DEF`**
`dct(2,2)`	26	-	-	-	1
`dct(3,3)`	95	-	-	-	1
`horner(2,2)`	12	-	-	-	1
`horner(4,4)`	40	-	-	-	1
`horner(6,6)`	84	-	-	-	1
`binpot`	6	1	1	2	4
`fibonacci`	22	1	2	2	5
`diffeq`	11	1	0	2	3
`gcd`	16	3	1	5	7
`bubblesort`	19	2	1	6	8
`fp_add`	61	1	11	12	24
`kalman`	53	6	4	14	19

Tabelle 6.1: Aufbau und Umfang der Beispiele

Die Beispielspezifikationen

Die Beispiele `dct`, `horner` und `fibonacci` wurden von [Blum00] übernommen. Bei `dct` und `horner` handelt es sich um reine Datenflussbeschreibungen. Alle übrigen Spezifikationen sind steuerflussbehaftet. Das Programm `dct` führt eine diskrete Cosinustransformation durch,

welche bei der Kompression von Bildern Verwendung findet. Die Größe eines konkreten `dct`-Terms hängt dabei von den Ausmaßen der zu verarbeitenden Bildblöcke ab. Das Programm `horner` realisiert eine Polynomdivision nach dem Horner-Schema. Dieses Beispiel ist in Bezug auf den Grad des Zähler- und des Nennerpolynoms skalierbar. Die konkrete Parametrisierung dieser beiden Beispiele ist jeweils zusätzlich zum Programmnamen in Klammern angegeben. Das dritte aus [Blum00] übernommene Programm `fibonacci` implementiert einen effizienten Algorithmus zur Berechnung von Fibonacci-Zahlen.

Die Programme `diffeq` und `kalman` entstammen der *High-Level-Synthesis-Benchmark*-Sammlung, die von Nikil Dutt und Preeti Ranjan Panda Mitte der Neunziger zusammengetragen wurde. Das Programm `diffeq` dient zur Lösung von Differentialgleichungen [PaKG86] und `kalman` implementiert einen Kalman-Filter. Letzteres Beispiel enthält besonders viele verschachtelte Verzweigungen und Schleifen.

Das Programm `gcd` berechnet den größten gemeinsamen Teiler (*ggT*) zweier natürlicher Zahlen nach dem binären euklidischen Algorithmus. Im Gegensatz zum klassischen Algorithmus zur ggT-Berechnung werden dabei Divisionen ausschließlich durch bitweise Verschiebungen realisiert. Daraus resultiert eine höhere Effizienz bei der Verarbeitung des Algorithmus auf Computersystemen.

Das Beispiel `binpot` implementiert einen binären Algorithmus zur Berechnung von Potenzen und ist, wie `gcd`, auf eine Ausführung auf Computersystemen optimiert. Es handelt sich bei diesem Beispiel um die Spezifikation aus Abbildung 3.16 auf Seite 47, welche bereits in den Kapiteln 4 und 5 zur Veranschaulichung des in dieser Arbeit entwickelten Konzepts mehrfach herangezogen wurde.

Bei `bubblesort` handelt es sich um eine Implementierung des gleichnamigen Sortieralgorithmus und `fp_add` führt die Addition zweier Gleitkommazahlen durch. Vorlage für `fp_add` war die Funktion `__addsf3` in der Datei `floatlib.c` des GNU C-Compilers Version 3.3. Alle in dieser Funktion auftretenden C-Makros wurden dabei expandiert.

Experimentelle Durchführung der formalen algorithmischen Synthese

Im Folgenden wird der Aufwand für die Umsetzung der Entwurfsziele diskutiert, die wie in Kapitel 4 beschrieben zumeist außerhalb des Theorembeweisers ermittelt werden. Eine Beurteilung der Leistungsfähigkeit der Optimierungs- und Syntheseverfahren, welche in dieser Arbeit zur Bestimmung der Entwurfsziele eingesetzt werden, findet nicht statt. Ziel der Arbeit war die Sicherstellung der Korrektheit des Syntheseergebnisses und nicht die Entwicklung möglichst hochwertiger Optimierungs- und Synthesealgorithmen. Der Schwerpunkt lag auf der Fähigkeit des formalen Synthesesystems, vorliegende Entwurfsziele möglichst uneingeschränkt umsetzen zu können. Die Verfahren zur Ermittlung der Entwurfsziele sollten weitestgehend austauschbar und so der jeweiligen Situation anzupassen sein. Entsprechend sollte auch die manuelle Vorgabe von Entwurfszielen möglich sein. Die Entwurfsziele des Programms `binpot` wurden beispielsweise vollständig manuell entworfen.

Während die Umsetzbarkeit von Ablaufplänen in dem vorgestellten Konzept noch gewissen Einschränkungen unterliegt (siehe Abschnitt 5.3), sind die Implementierungen für das Operationswerk (siehe Abschnitt 5.5) sowie für den Konfigurations- und Transitionsblock (siehe Ab-

schnitt 5.6) weitestgehend frei wählbar. Auch können die Zustandsbezeichner praktisch beliebig kodiert werden. Hierbei muss nur in dem formalen Synthesesystem eine Funktion zum Beweis der Gleichheit bzw. Ungleichheit der neuen Zustandsbezeichner bereitgestellt werden.

In den aufgeführten Experimenten wurden die Ablaufpläne der steuerflussbehafteten Schaltungsbeschreibungen mittels des *Loop-Directed-Scheduling*-Algorithmus [BhDB94] (siehe Abschnitt 5.3) und die der Datenflussbeschreibungen auf Basis des *Force-Directed-Scheduling*-Algorithmus [PaKn87] ermittelt. Für alle Beispiele galten jeweils die gleichen Ressourcen- und Zeitbeschränkungen.

Um den Implementierungsaufwand in Grenzen zu halten, wurden zur Ermittlung der weiteren Entwurfsziele verhältnismäßig einfache Verfahren eingesetzt, deren Ergebnisse teilweise noch ein signifikantes Optimierungspotential bieten. Wie bereits erwähnt, ist deren Ersatz durch anspruchsvollere Verfahren jedoch jederzeit möglich.

Neben dem Ablaufplan wird für jedes der Beispiele, wie in Abschnitt 5.4 beschrieben, die Anzahl der erforderlichen Register sowie deren Zuteilung bestimmt. Zusätzlich wird ein passendes Operationswerk erzeugt, welches die Bereitstellung und Zuweisung der zur Verfügung stehenden Operationseinheiten widerspiegelt (siehe Abschnitt 5.5). Schließlich wird für jeden Zustandsbezeichner eine binäre Kodierung ermittelt.

Als Schnittstellenverhaltensmuster wurde für alle Beispiele das in Abschnitt 3.19 beschriebene Muster **GEN_SSVM** gewählt und die Abbildung auf die Register-Transfer-Ebene auf Basis eines jeweils passenden Steuerwerks bzw. Implementierungsmusters durchgeführt (siehe Abschnitt 5.6).

In den folgenden Betrachtungen wird die Umsetzung der Ablaufplanung, der Bereitstellung und Zuweisung der Register sowie die damit verbundene Überführung der Spezifikation in die steuerflussorientierte Darstellung getrennt von der Umsetzung der verbleibenden Syntheseschritte diskutiert. Diese Aufteilung wurde vorgenommen, da die aufgezählten Schritte bei steuerflussbehafteten und steuerflusslosen Schaltungen auf eine unterschiedliche Weise durchgeführt werden. Die Umsetzung aller weiteren Syntheseschritte läuft dagegen einheitlich ab.

Umsetzung der Ablaufplanung und der Registerbereitstellung- und zuweisung

Steuerflusslose Spezifikationen werden vor deren Überführung in die steuerflussorientierte Darstellung in eine Form gebracht, welche bereits den vorgegebenen Ablaufplan und in der Regel auch die beabsichtige Bereitstellung und Zuweisung der Register widerspiegelt (siehe Abschnitt 5.3.5). Die Dauer dieser Umformung ist in Abbildung 6.1 dargestellt. Alle Berechnungen wurden auf einem Athlon XP1700+ mit 512 MB Arbeitsspeicher unter Linux 2.4-21 durchgeführt.

Zunächst wird die Spezifikation nach der in [Blum00] beschriebenen Methode in eine Komposition mehrerer DFG-Terme transformiert, so dass jeder DFG-Term die in einem bestimmten Takt auszuführenden Operationen enthält. Diese Darstellung wird anschließend in die Form **DEF** f_1 **SER DEF** f_2 **SER** ... **SER DEF** f_n (siehe Abschnitt 5.3.5) überführt, um die Schaltung dem in dieser Arbeit vorgestellten Konzept entsprechend weiterverarbeiten zu können. Nach dieser Integrationstransformation kann die Umsetzung der Ablaufplanung abgeschlossen wer-

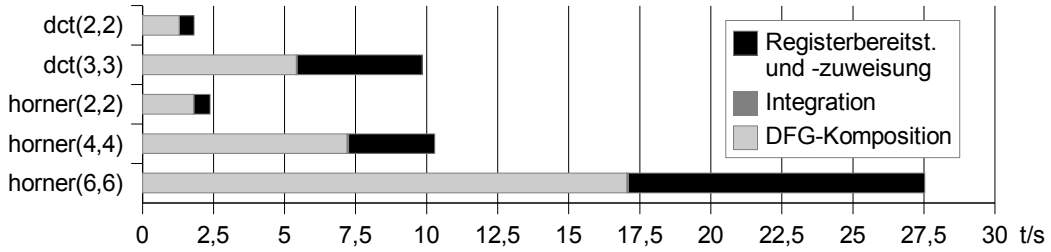


Abbildung 6.1: Zeiten der für steuerflusslose Schaltungen spezifischen Transformationen

den, indem die Schaltungsbeschreibung in die steuerflussorientierte Darstellung überführt wird. Für gewöhnlich wird jedoch bei steuerflusslosen Spezifikationen zuvor noch die Bereitstellung und Zuweisung der Register umgesetzt (siehe Abschnitt 5.4). Die Laufzeiten dieser weiteren Transformationen sind ebenfalls in Abbildung 6.1 dargestellt.

Die Laufzeitkomplexität der Integrationstransformation und der für steuerflusslose Schaltungen spezifischen Transformation zur Registerbereitstellung und -zuweisung sind linear von der Anzahl der komponierten DFG-Terme abhängig. Die Dauer der Integrationstransformation liegt dabei in allen Beispielen unter einer Zehntelsekunde. Für eine Diskussion der Laufzeitkomplexität der Transformation, welche den Aufbau dieser Komposition durchführt, sei auf [Blum00] verwiesen.

Bei steuerflussbehafteten Schaltungsbeschreibungen kann die Umsetzung der Ablaufplanung erst nach deren Überführung in die steuerflussorientierte Darstellung erfolgen. Tabelle 6.2 gibt an, wie viele Zustände aufgespalten bzw. verschmolzen wurden, um sowohl die vorverarbeiteten steuerflusslosen als auch die steuerflussbehafteten Beispiele in die neue Darstellungsform zu überführen (siehe Abschnitt 5.2). Die rechte Spalte gibt die Anzahl der Zustände in der resultierenden Darstellung an.

Programm	Aufspaltungen	Verschmelzungen	Zustände
`dct(2,2)`	4	-	5
`dct(3,3)`	6	-	7
`horner(2,2)`	7	-	8
`horner(4,4)`	13	-	14
`horner(6,6)`	19	-	20
`binpot`	7	8	5
`fibonacci`	20	11	17
`diffeq`	11	6	9
`gcd`	20	23	10
`bubblesort`	14	14	9
`fp_add`	39	28	28
`kalman`	44	45	28

Tabelle 6.2: Transformationen zur Überführung in die steuerflussorientierte Darstellung

Die Laufzeiten für den Übergang zur steuerflussorientierten Darstellung sind in Abbildung 6.2 dargestellt. Die Überführung einer Verhaltensbeschreibung in die steuerflussorientierte Darstellung hat eine Laufzeitkomplexität von $O(a^2 + av)$. Dabei bezeichnen a bzw. v die Anzahl der Aufspaltungstransformationen bzw. der Verschmelzungen.

Eine einzelne Aufspaltungstransformation ist linear von der Anzahl der Zustände in der gegebenen steuerflussorientierten Darstellung abhängig. Der quadratische Aufwand für die Durchführung aller a Aufspaltungstransformationen folgt aus der Tatsache, dass nach jeder Aufspaltung ein Zustand mehr existiert. Bei der Überführung einer Spezifikation in die steuerflussorientierte Darstellung ist der Ausgangspunkt der Aufspaltungstransformationen eine Spezifikation mit einem einzigen Zustand. Im Allgemeinen ist der Aufwand für a Aufspaltungen ausgehend von einer Darstellung mit n Zuständen von der Ordnung $O(a^2 + an)$.

Der Aufwand für das Verschmelzen von Zuständen ist ebenfalls linear von der Anzahl der Zustände abhängig. Das v-fache Ausführen der Verschmelzungstransformation auf einer Darstellung mit n Zuständen hat somit die Laufzeitkomplexität $O(nv)$.

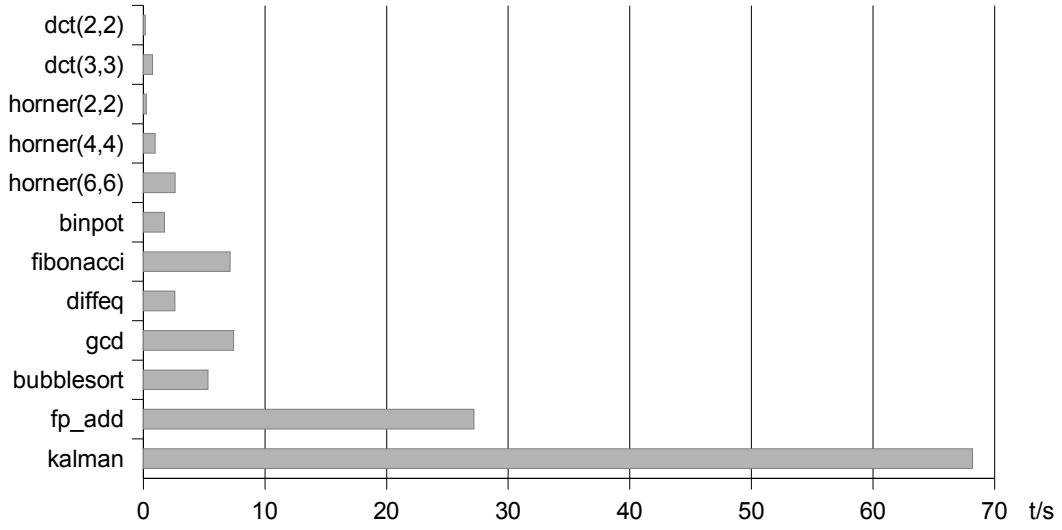


Abbildung 6.2: Zeiten für die Überführung der Beispiele in die steuerflussorientierte Darstellung

Eine steuerflussbehaftete Verhaltensbeschreibung entspricht nach deren Überführung in die steuerflussorientierte Darstellung dem Steuerflussgraphen der Spezifikation. Hier wird die Ablaufplanung umgesetzt, indem diese Darstellung durch das Kopieren, Umleiten und Verschmelzen von Zuständen in eine neue steuerflussorientierte Darstellung überführt wird, welche dem jeweils vorgegebenen Ablaufplan entspricht (siehe Abschnitt 5.3). Jeder der Zustände in der resultierenden Darstellung definiert eine Menge von Operationen, die nach Abschluss der Synthese in der RT-Ebenen-Implementierung in einem Takt zur Ausführung kommen werden.

Tabelle 6.3 gibt an, wie viele der verschiedenen Transformationsschritte im Falle der steuerflussbehafteten Beispiele erforderlich waren, um deren Ablaufplanung in dem formalen Synthesesystem vollständig umzusetzen. Zusätzlich ist die Anzahl der jeweils resultierenden Zustände aufgelistet.

Programm	Kopien	Umleitungen	Verschmelzungen	Zustände
binpot	4	4	5	4
fibonacci	12	12	17	12
diffeq	2	2	3	8
gcd	28	38	31	7
bubblesort	7	9	9	7
fp_add	24	32	24	28
kalman	43	58	57	14

Tabelle 6.3: Umsetzung der Ablaufplanung bei den steuerflussbehafteten Beispielen

Der zeitliche Aufwand für die Umsetzung der Ablaufplanung bei den gegebenen Beispielen ist in Abbildung 6.3 dargestellt. Sie hat eine Laufzeitkomplexität der Ordnung $O(n^2 + k^2)$. Dabei repräsentiert n die Anzahl der Zustände nach der Überführung des jeweiligen Beispiels in die steuerflussorientierte Darstellung und k die Anzahl der Kopie-Transformationen.

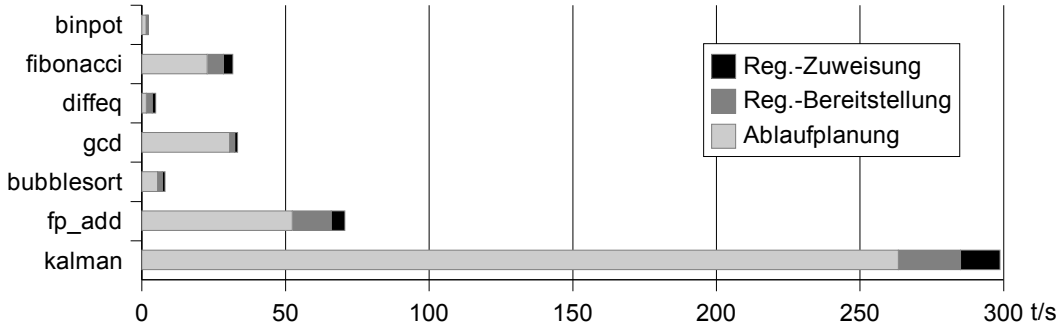


Abbildung 6.3: Zeiten der für steuerflussbehaftete Schaltungen spezifischen Transformationen

Die Kopie-Transformationen haben, analog zu den Aufspaltungstransformationen, in der Summe die Laufzeitkomplexität $O(k^2+kn)$. Der Aufwand der einzelnen Umleitungstransformationen ist linear von der Anzahl der Zustände abhängig. Da für jeden Zustand nur die einmalige Durchführung einer Umleitungstransformation sinnvoll ist, haben die Umleitungstransformationen bei einer steuerflussorientierten Darstellung mit n Zuständen insgesamt eine Laufzeitkomplexität der Ordnung $O(n^2)$.

Bei der Umsetzung der Ablaufplanung werden ausschließlich Verschmelzungstransformationen eingesetzt, welche die Eliminierung einer der beiden zu verschmelzenden Zuständen bewirken (siehe Abschnitt 5.2.2). Daher ist die Häufigkeit von deren Anwendung durch die Anzahl n der in der gegebenen Darstellung existierenden Zustände begrenzt. Somit hat die Summe aller Verschmelzungstransformationen in diesem Fall die Laufzeitkomplexität $O(n^2)$.

Der zeitliche Aufwand, der für die Umsetzung der Bereitstellung und Zuweisung der Register im Falle der steuerflussbehafteten Beispiele aufzubringen war, ist ebenfalls in Abbildung 6.3 dargestellt. Die Laufzeitkomplexität der Registerzuweisung hängt quadratisch von der Anzahl der Zustände einer gegebenen Spezifikation ab, da für jeden Zustand eine Liste durchsucht werden muss, in welcher diesem eine bestimmte Registerbelegung zugewiesen wird (siehe Abschnitt 5.4). Der Aufwand für die Umsetzung der Registerbereitstellung ist linear von der Zustandszahl abhängig.

Durchführung der verbleibenden Syntheseschritte

Abbildung 6.4 gibt die Laufzeiten für die übrigen Schritte an, die im Zuge der formalen algorithmischen Synthese durchzuführen sind. Der Aufwand für die Umsetzung der Operations- und Steuerwerksynthese sowie der Einbettung konkreter Implementierungen für die Funktionsblöcke (dem Konfigurations- und dem Transitionsblock) ist jeweils linear von der Anzahl s der Zustände abhängig, welche die zu synthetisierende Schaltungsspezifikation nach der Umsetzung der Ablaufplanung enthält. Die Laufzeitkomplexität der Transformation zur Umkodierung der Zustandsbezeichner ist, analog zur Komplexität der Registerbereitstellung, von der Ordnung $O(s^2)$.

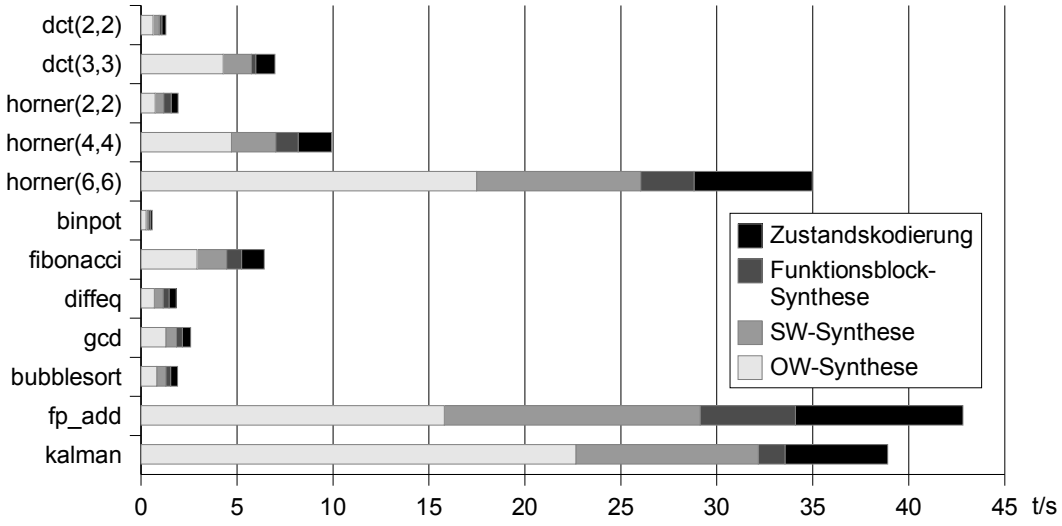


Abbildung 6.4: Zeiten für die Umsetzung der verbleibenden Syntheseschritte

Neben der Anzahl der Zustände fällt bei der Laufzeit der Transformationen vor allem auch die Größe der jeweils zu verarbeitenden Terme ins Gewicht. Die Laufzeit für die Integration des Rechenwerks ist beispielsweise stark von dessen Größe abhängig.

Wie bereits erwähnt, sind die Verfahren, die in der gegenwärtigen Implementierung des Synthesesystems zum Entwurf der Operationswerke bzw. der Konfigurations- und Transitionsblöcke eingesetzt werden, verhältnismäßig einfach. Durch deren Ersatz durch fortgeschrittenere Syntheseverfahren können effizientere und kompaktere Implementierungen erzeugt werden. Eine

entsprechende Optimierung der Implementierungen führt nicht nur zu einer höherwertigen Gesamtimplementierung der zu synthetisierenden Schaltung, sondern auch zu deren beschleunigten Einbettung während des formalen Syntheseprozesses.

Insgesamt zeigt sich, dass die Durchführung der formalen algorithmischen Synthese in allen Beispielen im Bereich weniger Minuten liegt. Dabei ist besonders hervorzuheben, dass die Laufzeit des Verfahrens von der Bitbreite der in der Spezifikation auftretenden Signale völlig unabhängig ist. Unendliche Datentypen können gleichermaßen in der Spezifikation eingesetzt werden, so dass beispielsweise auch die formale algorithmische Synthese generischer Schaltungsbeschreibungen variabler Bitbreite möglich ist.

Als Ergebnis einer formalen Synthese erhält man automatisch neben der resultierenden Implementierung den in einem Theorembeweiser geführten Beweis, dass diese Implementierung exakt den funktionalen Vorgaben der zu synthetisierenden Schaltungsspezifikation entspricht.

Da im Anschluss an eine konventionelle Synthese noch zusätzlich eine Überprüfung der funktionalen Korrektheit der Implementierung anfällt, ist der Mehraufwand, der sich für die Durchführung einer formalen Synthese ergibt, als sehr gering einzustufen. Der Aufwand der Verfahren, die nachträglich die korrekte Funktionsweise der Implementierung nachzuweisen versuchen, wächst in der Regel exponentiell mit der Bitbreite der Eingabesignale bzw. der Speicherelemente der Schaltung. Entsprechende Verfahren sind daher sehr aufwändig und können überdies meist nur bei kleinen Schaltungen eine erschöpfende Überprüfung aller möglichen Zustände vornehmen.

Kapitel 7

Zusammenfassung

Zur Herstellung fehlerfreier Hardware muss, abgesehen von der Korrektheit der Spezifikation, vor allem auch die exakte Einhaltung der Vorgaben bei der Synthese einer Implementierung sichergestellt werden. Während die Einhaltung physikalischer Randbedingungen für gewöhnlich verhältnismäßig einfach mit analytischen Mitteln überprüft werden kann, ist der Nachweis, dass die Implementierung exakt die in der Spezifikation definierte Funktionalität aufweist, weitaus schwieriger zu erbringen. Verfahren, die eine entsprechende Überprüfung nach Abschluss der Synthese durchführen, kommen zumeist nur für kleinere Schaltungen in Frage, da deren Aufwand im Allgemeinen exponentiell mit der Bitbreite der Eingabesignale bzw. der in der Schaltung enthaltenen Speicherelemente steigt. Eine Softwareverifikation der eingesetzten Syntesewerkzeuge ist auf Grund deren Größe und Komplexität in der Regel ebenfalls kaum praktikabel, so dass es sich anbietet, die Korrektheit der Synthese während deren Durchführung sicherzustellen.

Einen entsprechenden Ansatz verfolgt die formale Synthese. Die Implementierung wird hierbei aus einer Schaltungsspezifikation allein auf Basis verhaltenserhaltender Transformationen innerhalb eines Theorembeweisers abgeleitet. Infolge der Einbettung des kompletten Syntheseprozesses in einen Theorembeweiser liefert die Synthese zusätzlich zu der Implementierung der spezifizierten Schaltung automatisch den formalen Beweis, dass die Implementierung genau das in der Spezifikation definierte Verhalten aufweist.

Während auf den unteren Abstraktionsebenen bereits große Fortschritte im Bereich der formalen Synthese erzielt werden konnten, wies bislang die formale algorithmische Synthese vor allem in Bezug auf die Synthese steuerflussbehafteter Schaltungsbeschreibungen gravierende Unzulänglichkeiten auf. Zum einen schränkte die bisherige Vorgehensweise die Optimierungsmöglichkeiten, die sich im Laufe einer algorithmischen Synthese ergeben, stark ein. Zum anderen war eine Einflussnahme auf die Qualität der resultierenden Implementierungen nur in sehr geringem Umfang und darüber hinaus nur interaktiv möglich, so dass vom Entwerfer weitreichende Kenntnisse im Bereich der formalen Methoden gefordert waren. Des Weiteren wich das Konzept so stark von der konventionellen algorithmischen Synthese ab, dass der Einsatz existierender Optimierungs- und Syntheseverfahren kaum möglich und somit eine entsprechend hohe Implementierungsqualität nicht zu erreichen war.

Zur Überwindung der aufgeführten Einschränkungen wurde ein neues Konzept für die formale algorithmische Synthese entwickelt und in dieser Arbeit vorgestellt. Es erlaubt über die Synthese reiner Datenflussbeschreibungen hinaus die vollautomatische formale Synthese steuerflussbehafteter Schaltungsspezifikationen. Durch den Einsatz einer neuen formalen Schaltungsrepräsentation können die Steuerinformationen, die der Entwerfer in der Spezifikation vorgegeben hat, während des gesamten Syntheseverlaufs nach Belieben aufrechterhalten werden. Dies hat eine beträchtliche Erhöhung des während der Synthese zur Verfügung stehenden Optimierungspotentials zur Folge.

Da im Bereich der konventionellen Synthese bereits unzählige Verfahren zur Optimierung und Synthese digitaler Schaltungen entwickelt wurden, bietet es sich an, aus diesem enormen Fundus ausgereifter Techniken zu schöpfen, um auch in der formalen Synthese hochwertige Implementierungen zu erzielen. Anstatt also neue, spezielle Optimierungs- und Syntheseverfahren für die formale Synthese zu entwickeln, wurde ein Rahmenwerk geschaffen, das die einfache Integration existierender Verfahren unterstützt und überdies deren effiziente Ausführung außerhalb des Theorembeweisers ermöglicht.

Um ein möglichst breites Spektrum der konventionellen Verfahren zur Steuerung des formalen Entwurfsprozesses hinzuziehen zu können, wurde das neue Synthesekonzept stark an die klassische algorithmische Synthese angelehnt, bei deren Durchführung im Allgemeinen die Ablaufplanung, die Bereitstellung und Zuweisung der Ressourcen sowie die Steuerwerksynthese unterschieden werden.

In der vorliegenden Arbeit wurden die Darstellungsformen, Theoreme und Transformationen vorgestellt, welche die Grundlage für eine automatische Durchführung der formalen algorithmischen Synthese bilden, und gezeigt, wie durch deren gezielten Einsatz aus einer Verhaltensspezifikation auf der algorithmischen Ebene effizient eine Implementierung auf der Register-Transfer-Ebene unter Berücksichtigung vorgegebener Entwurfsvorgaben abgeleitet werden kann. Es ist gelungen, den Ableitungsprozess vollständig zu automatisieren. Eine interaktive Vorgabe der Entwurfsziele durch den Benutzer ist zwar nach wie vor möglich, aber nicht erforderlich. Durch die vollständige Automatisierung der formalen Synthese ist es nun auch Entwerfern ohne Kenntnisse in formalen Methoden möglich, hochwertige Implementierungen zu entwickeln, die bewiesenermaßen funktional korrekt bezüglich ihrer Spezifikation sind. Die Effizienz der entwickelten Vorgehensweise eröffnet dabei Möglichkeiten für deren Einsatz in der Praxis.

Literaturverzeichnis

[AaLe91] M. Aagaard and M. Leeser. A formally verified system for logic synthesis. In *IEEE International Conference on Computer Design on VLSI in Computer and Processors*, pages 346–350. IEEE, October 1991.

[AaLe94] 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*, 21(10):822–833, October 1995.

[AhSU86] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, 1986.

[BBFL01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[BELR01] C. Blank, H. Eveking, J. Levihn, and G. Ritter. Symbolic simulation techniques - state-of-the-art and applications. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, page 45, Monterey, California, November 2001. IEEE Computer Society.

[BhDB94] S. Bhattacharya, S. Dey, and F. Brglez. Performance analysis and optimization of schedules for conditional and loop-intensive specifications. In Michael Lorenzetti, editor, *Proceedings of the 31st Conference on Design Automation*, pages 491–496. ACM Press, June 1994.

[BlEi98] C. Blumenröhr and D. Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In *Digital System Design Workshop at the 24th EUROMICRO 98 Conference*, pages 34–37, Västerås, Sweden, August 1998. IEEE Computer Society Press.

[BlEK96] C.Blumenröhr, D. Eisenbiegler, and R.Kumar. Applicability of formal synthesis illustrated via scheduling. In *Workshop on Logic and Architecture Synthesis*, pages 345–352, Grenoble, France, December 1996. Institut National Polytechnique de Grenoble.

[Blum00] C. Blumenröhr. *Formale Spezifikation und Synthese digitaler Schaltungen auf höheren Abstraktionsebenen*. Dissertation, Universität Karlsruhe, Deutschland, 2000.

[Blum99] C. Blumenröhr. A Formal Approach to Specify and Synthesize at the System Level. In *GI/ITG/GME Workshop on Modellierung und Verifikation von Schaltungen und Systemen*, pages 11–20, Braunschweig, Germany, February 1999. Shaker-Verlag.

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

[BuDi94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994.

[CABC86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., 1986.

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

[Chur40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[ClGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT, London, England, 1999.

[DeMi94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[DLSM81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[EiBK96] D. Eisenbiegler, C. Blumenröhr, and R. Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in HOL. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 157–172. Springer-Verlag, 1996.

[EiKu95] Dirk Eisenbiegler and Ramayya Kumar. Formally embedding existing high level synthesis algorithms. In *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 71–83. Springer-Verlag, 1995.

[Eise97b] D. Eisenbiegler. Automata – A theory dedicated towards formal circuit synthesis. Technical Report 14/97, Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, 1997.

[Eise99] D. Eisenbiegler. *Ein Kalkül für die formale Schaltungssynthese*. Dissertation, Universität Karlsruhe, Deutschland, 1999.

[GaRa94] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[GeRo99] J. Gerlach and W. Rosenstiel. Transformationale Optimierung von Entwurfsdarstellungen der algorithmischen Ebene. Technical Report WSI-99-19, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Tübingen, Germany, November 99.

[GMMN78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A Metalanguage for interactive proof in LCF. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages (POPL '78)*, pages 119–130, Tucson, Arizona, 1978. ACM Press.

[GMWa79] M. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GoMe93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gord85] M. Gordon. HOL : A machine oriented formulation of higher order logic. Technical Report UCAM-CL-TR-68, University of Cambridge, Computer Laboratory, July 1985.

[GoVM89] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling for DSP-systems. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation*, pages 826–831. ACM Press, June 1989.

[Gupt92] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992.

[HiER99] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *Proceedings of DMTCS and CATS*, Auckland, January 1999. Springer.

[Hinr98] H. Hinrichsen. Formally correct construction of a pipelined DLX architecture. Technical report, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, May 1998.

[HiRE99] H. Hinrichsen, G. Ritter, and H. Eveking. Automatische Synthese und Verifikation von RISC-Prozessoren. In *GI/ITG/GMM Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 1–10, Braunschweig, Germany, February 1999.

[KaSa03] K. Kapp and V. Sabelfeld. Dead code elimination in formal synthesis. In Rolf Drechsler, editor, *GI/ITG/GMM Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 121–130, Bremen, Germany, February 2003. Shaker Verlag.

[KaSa04] K. Kapp and V. Sabelfeld. Automatic correct scheduling of control flow intensive behavioral descriptions in formal synthesis. In *DAC '04: Proceedings of the 41st Conference on Design Automation*, pages 61–66. ACM Press, June 2004.

[KBES96] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design - A classification and survey. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 294–309. Springer-Verlag, November 1996.

[KMNS92] T. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken. The Sprite Input Language - an intermediate format for high level synthesis. In *3rd European Conference on Design Automation*, pages 186–192, Brussels, March 1992.

[Krop99] T. Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag New York, Inc., 1999.

[LaRJ99] G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Wavesched: a novel scheduling technique for control-flow intensive designs. *IEEE Transactions on CAD*, May 1999.

[Lars94] M. Larsson. An engineering approach to formal system design. In Thomas F. Melham and Juanito Camileri, editors, *Higher Order Logic Theorem Proving and its Applications*, number 859 in Lecture Notes in Computer Science, pages 300–315, Valletta, Malta, September 1994. Springer-Verlag.

[Lars95] M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, 1995.

[Lars96] M. Larsson. Improving the result of high-level synthesis using interactive transformational design. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 299–314. Springer-Verlag, 1996.

[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 Systems*, 6(2):191–214, August 1993.

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

[Lees92] M. Leeser. Using nuprl for the verification and synthesis of hardware. *Philosophical Transactions Royal Society London*, 339:49–68, 1992.

[Lin97] Y.-L. Lin. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2(1):2–21, January 1997.

[McFa93] M. C. McFarland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2(3):231–257, 1993.

[McPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[MeHe98] J. M. Mendías and R. Hermida. Automatic Formal Derivation applied to High-Level Synthesis. In Juan Carlos López, Román Hermida, and Walter Geisselhardt, editors, *Advanced Techniques for Embedded Systems Design and Test*, chapter 5, pages 103–123. Kluwer Academic Publishers, 1998.

[MeHF96] J. M. Mendías, R. Hermida, and M. Fernández. Algebraic support for transformational hardware allocation. In *EDTC '96: Proceedings of the 1996 European Conference on Design and Test*, page 601. IEEE, 1996.

[MeHF97] J. M. Mendías, R. Hermida, and M. Fernández. Formal techniques for hardware allocation. In *Tenth International Conference on VLSI Design*, pages 161–165. IEEE, January 1997.

[MeHF98] J. M. Mendías, R. Hermida, and M. Fernández. Correct high-level synthesis: a formal perspective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 977–978. IEEE Computer Society, February 1998.

[MeHP02] J. M. Mendías, R. Hermida, and O. Penalba. A study about the efficiency of formal high-level synthesis applied to verification. *Integration, the VLSI Journal*, 31(2):101–131, May 2002.

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

[MHMP02] J. M. Mendías, R. Hermida, M. C. Molina, and O. Peñalba. Efficient verification of scheduling, allocation and binding in high-level synthesis. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 308–315. IEEE Computer Society, 2002.

[MHMP96] P. Middelhoek, C. Huijs, G. Mekenkamp, E. Prangsma, E. Engels, J. Hofstede, and T. Krol. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE International High Level Design Validation and Test Workshop*, Oakland, California, November 1996.

[Midd97] P.F.A. Middelhoek. *Transformational Design: An Architecture Independent Interactive Design Methodology for the Synthesis of Correct and Efficient Digital Systems*. PhD thesis, Universiteit Twente, NL, 1997.

[MiRa96] P. F. A. Middelhoek and S. P. Rajan. From VHDL to efficient and first-time-right designs: a formal approach. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(2):205–250, April 1996.

[MiTM97] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[Much97] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[NaVe98] N. Narasimhan and R. Vemuri. On the effectiveness of theorem proving guided discovery of formal assertions for a register allocator in a high-level synthesis system. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 367–386. Springer-Verlag, 1998.

[NiBr97a] T. P. K. Nijhar and A. D. Brown. Source level optimisation of VHDL for behavioural synthesis. *IEEE Proceedings on Computers and Digital Techniques*, 144(1):1–6, January 1997.

[NiBr97b] T. P. K. Nijhar and A. D. Brown. HDL-specific source level behavioural optimisation. *IEEE Proceedings on Computers and Digital Techniques*, 144(2):138–144, March 1997.

[NTRG01] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Formal Methods in System Design*, 19(3):237–273, 2001.

[OwRS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992.

[PaKG86] P. G. Paulin, J. P. Knight, and E. F. Girczyc. HAL: A multi-paradigm approach to automatic data path synthesis. In ACM/IEEE, editor, *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 263–270, Las Vegas, NV, June 1986. IEEE Computer Society Press.

[PaKn87] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *Proceedings of the 24th ACM/IEEE Conference on Design Automation*, pages 195–202. ACM Press, 1987.

[PaKn89] P. G. Paulin and J. P. Knight. Scheduling and binding algorithms for high-level synthesis. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation*, pages 1–6. ACM Press, 1989.

[PeKu94] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):150–166, February 1994.

[Raja95a] S. P. Rajan. Formal verification of transformations on dependency graphs in optimizing compilers. In *Proceedings of California Software Engineering Symposium*, Irvine, CA, March 1995.

[Raja95b] S. P. Rajan. Correctness of transformations in high level synthesis. In Steven D. Johnson, editor, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, pages 597–603, Chiba, Japan, August 1995.

[Raja95c] S. P. Rajan. *Transformations on dependency graphs: formal specification and efficient mechanical verification*. PhD thesis, University of British Columbia, March 1995.

[RaJe95] M. Rahmouni and A. A. Jerraya. PPS: A pipeline path-based scheduler. In *Proceedings of the European Conference on Design and Test*, pages 557–561. IEEE Computer Society, March 1995.

[RaJe95b] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proceedings of the Conference on European Design Automation*, pages 386–391. IEEE, December 1995.

[RaTV00] R. Radhakrishnan, E. Teica, and R. Vermuri. An approach to high-level synthesis system validation using formally verified transformations. In *Proceedings of High-Level Design Validation and Test Workshop*, pages 80–85, November 2000.

[RaTv01] R. Radhakrishnan, E. Teica, and R. Vemuri. Verification of basic block schedules using RTL transformations. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 173–178. Springer-Verlag, September 2001.

[RiEH99] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 234–249. Springer-Verlag, 1999.

[RiHE99] G. Ritter, H. Hinrichsen, and H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 308–321. Springer-Verlag, 1999.

[RKDV92] J. Roy, N. Kumar, R. Dutta, and R. Vemuri. DSS: A distributed high-level synthesis system. *IEEE Des. Test*, 9(2):18–32, 1992.

[SaBK01] V. Sabelfeld, C. Blumenröhr, and K. Kapp. Semantics and transformations in formal synthesis at system level. In *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 149–156. Springer-Verlag, July 2001.

[SaCo95] F. Sanchez and J. Cortadella. Time-constrained loop pipelining. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD-95*, pages 592–596, November 1995.

[SaKa03] V. Sabelfeld and K. Kapp. Arithmetics in formal synthesis. In Rolf Drechsler, editor, *GI/ITG/GMM Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 112–120, Bremen, Germany, February 2003. Shaker Verlag.

[SaKa03b] V. Sabelfeld and K. Kapp. Numeric types in formal synthesis. In M. Broy and A. Zamulin, editors, *Perspectives of System Informatics, 5th International Andrei Ershov Memorial Conference*, volume 2890 of *LNCS*, pages 79–90, Novosibirsk, Russia, July 2003. Springer.

[Schn03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[TeRV01] E. Teica, R. Radhakrishnan, and R. Vemuri. On the verification of synthesized designs using automatically generated transformational witnesses. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 798. IEEE Press, March 2001.

[TeVe01] E. Teica and R. Vemuri. Mechanizing in higher-order logic proofs of correctness and completeness for a set of RTL transformations. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, Informatics Report Series, pages 368–383, September 2001.

[TeVe01b] E. Teica and R. Vemuri. *Formal correctness and completeness for a set of uninterpreted rtl transformations*. PhD thesis, University of Cincinnati, 2001.

[ThMo02] D. E. Thomas and P. R. Moorby. *The verilog hardware description language*. Kluwer Academic Publishers, 2002.

[VHDL02] IEEE. *IEEE Standard VHDL Language Reference Manual Std 1076-2002*, 2002.

[WaAs85] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.

[Wirt85] N. Wirth. *Programming in MODULA-2 (3rd corrected ed.)*. Springer-Verlag New York, Inc., 1985.

Universität Karlsruhe (TH)
Institut für Technische Informatik

Grundvoraussetzung für die Herstellung fehlerfreier Hardware ist neben der korrekten Spezifikation der gewünschten Schaltung vor allem auch die fehlerfreie Umsetzung der spezifizierten Eigenschaften in der resultierenden Implementierung der Schaltung.

Während die Einhaltung physikalischer Randbedingungen für gewöhnlich verhältnismäßig einfach mit analytischen Mitteln überprüft werden kann, ist der Nachweis, dass die Implementierung exakt die in der Spezifikation definierte Funktionalität aufweist, weitaus schwieriger zu erbringen. Verfahren, die eine entsprechende Überprüfung nach Abschluss der Synthese durchführen, kommen aus Komplexitätsgründen meist nur für kleinere Schaltungen in Frage. Eine Softwareverifikation der eingesetzten Synthesewerkzeuge ist auf Grund deren Größe und Komplexität in der Regel ebenfalls kaum praktikabel, so dass es sich anbietet, die Korrektheit der Synthese bereits während deren Durchführung sicherzustellen.

Einen entsprechenden Ansatz verfolgt die formale Synthese. Die Implementierung wird hierbei aus einer Schaltungsspezifikation allein auf Basis verhaltenserhaltender Transformationen innerhalb eines Theorembeweisers abgeleitet. Mit der erfolgreichen Durchführung einer Transformation liefert der Theorembeweiser automatisch den Beweis für deren Korrektheit. Durch eine entsprechende Einbettung des gesamten Syntheseprozesses in einen Theorembeweiser wird mit der Implementierung der spezifizierten Schaltung gleichzeitig der formale Beweis generiert, dass die Implementierung genau das in der Spezifikation definierte Verhalten aufweist.

Während auf den unteren Abstraktionsebenen bereits große Fortschritte im Bereich der formalen Synthese erzielt werden konnten, wies bislang die formale algorithmische Synthese, vor allem in Bezug auf die Synthese steuerflussbehafteter Schaltungsbeschreibungen, gravierende Unzulänglichkeiten auf. Ein neues Konzept zur Überwindung vieler der existierenden Einschränkungen in der formalen algorithmischen Synthese wird in diesem Buch vorgestellt. Es erlaubt über die Synthese reiner Datenflussbeschreibungen hinaus auch die vollautomatische formale Synthese steuerflussbehafteter Schaltungsspezifikationen.

ISBN 3-937300-90-2