Modellgenerierung für die SAT-basierte Eigenschaftsprüfung

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von
Dipl. Math. Markus Wedler geb. Losch
geb. in Wuppertal, Deutschland

D 386

Dekan: Prof. Dr.-Ing. Wolfgang Kunz

Gutachter: Prof. Dr.-Ing. Wolfgang Kunz, Technische Universität Kaiserslautern
Prof. Dr. Hans-Joachim Wunderlich, Universität Stuttgart

Datum der mündlichen Prüfung: 08.08.2006

Inhaltsverzeichnis

Danksagung

Diese Arbeit ist das Ergebnis von fünf intensiven Jahren als wissenschaftlicher Mitarbeiter am Lehrstuhl von Prof. Wolfgang Kunz. Ich empfinde tiefe Dankbarkeit für die vielen Möglichkeiten, die mir Prof. Kunz als Betreuer meiner Forschungsarbeit eröffnet hat. Seine stetige Ermutigung und Richtungsweisung hat maßgeblich zur Entstehung dieser Dissertation beigetragen.

Für die Übernahme des Koreferates und das damit gezeigte Interesse an meiner Arbeit, sowie die wohlwollende Kommentierung danke ich Herrn Prof. Wunderlich.

Außerdem möchte ich mich bei meinen Kollegen Dominik Stoffel, Ingmar Neumann, Kolja Sulimma, Evgeny Karibaev, Minh Duc Nguyen, Max Thalmeier und Sacha Loitz bedanken. Viele der hier beschriebenen Ideen sind in unseren Diskussionsrunden gereift. Jörn Störk danke ich für die große Unterstützung beim Korrekturlesen.

Den Firmen OneSpin Solutions GmbH, Infineon und Siemens verdanke ich den Zugang zu industriellen Verifikationsproblemen und Werkzeugen. Mein Dank gebührt hier besonders Raik Brinkmann für seine Unterstützung bei der Anbindung der Verfahren aus dieser Arbeit an die Werkzeuge der OneSpin Solutions GmbH.

In tiefer Dankbarkeit fühle ich mich meiner Frau Andrea und unseren Kindern Maike und Kathi verbunden. Ohne ihr Verständnis und ihre Unterstützung wäre diese Arbeit nicht entstanden.

Kaiserslautern, den 21.8.2006

Markus Wedler

Kapitel 1

Einleitung

1.1 Verifikation von SoC-Entwürfen

Zu den spannendsten Entwicklungen im Bereich EDA (Electronic Design Automation) der letzten Jahre gehören zweifellos die großen Fortschritte durch den industriellen Einsatz formaler Methoden zur Verifikation mikroelektronischer Systeme. Die weltweit großen Anstrengungen im Bereich der formalen Verifikation sind durch das sog. verification gap motiviert: 60 - 80 % der Entwurfskosten für den System-on-Chip-Entwurf entfallen bereits heute auf die Verifikation. Die International Technology Roadmap for Semiconductors [ITR03] stellt fest, dass die Anzahl an Verifikationsingeneuren die Anzahl der Designer häufig im Verhältnis 3:1 oder 4:1 übersteigt. Nur wenn die Verifikationsmethodik drastisch verbessert wird, können die Fortschritte in der Halbleitertechnologie in einen entsprechenden Produktivitätszuwachs umgesetzt werden. Es besteht breiter Konsens, dass eine Weiterentwicklung der konventionellen Simulationstechnik alleine nicht in der Lage sein wird, diese Lücke zu schließen. Große Erfolge in der industriellen Praxis konnten in der jüngeren Vergangenheit mit formalen Methoden erzielt werden. Hierbei wurden grundlegende wissenschaftliche Ergebnisse der vergangenen Jahre konsequent umgesetzt.
Formale Verifikationsmethoden werden heute vorwiegend für die beiden folgenden Aufgaben eingesetzt:

- formaler Äquivalenzvergleich
- formale Eigenschaftsprüfung

Beide Methoden sollen im Folgenden kurz erläutert werden.

1.1.1 Formaler Äquivalenzvergleich

Aufgabe des formalen Äquivalenzvergleichs ist es, die funktionale Äquivalenz einer Spezifikation zu einer Implementierung zu überprüfen. Die Spezifikation ist in der Praxis auf Register-Transfer (RT)-Ebene oder auf Gatterebene gegeben. Die Implementierung liegt in der Regel als Gatternetzliste vor. Der formale Äquivalenzvergleich hat in den letzten Jahren industriell große Verbreitung gefunden und wird zur Verifikation von Entwurfsverfeinerungen (Synthese) eingesetzt. Er hat die früher übliche Logiksimulation auf Gatterebene weitgehend ersetzt.

Der formale Äquivalenzvergleich verdankt seinen großen industriellen Erfolg der Tatsache, dass in den meisten praktischen Fällen die folgenden Annahmen gemacht werden können:

a) Implementierung und Spezifikation besitzen die gleiche Zustandskodierung.

b) Es existieren strukturelle Ähnlichkeiten zwischen Spezifikation und Implementierung.

Kommerzielle Werkzeuge für den Äquivalenzvergleich beschränken sich deshalb in der Regel auf den rein kombinatorischen Fall, für den ausgefeilte Techniken zur Ausnutzung struktureller Ähnlichkeiten entwickelt worden sind [JMF95, Kun93, Mat96, KK97]. Die Ausnutzung von Struktureigenschaften war der Schlüssel zum Erfolg des formalen Äquivalenzvergleichs, der aus diesem Grund manchmal auch als ”struktureller Äquivalenzvergleich” bezeichnet wird. Moderne Werkzeuge sind in der Lage Entwürfe mit mehreren Millionen Gattern zu verifizieren.

1.1.2 Formale Eigenschaftsprüfung

Aufgabe der formalen Eigenschaftsprüfung ist es, die Spezifikation eines Entwurfs auf RT-Ebene zu überprüfen. Der Designer gewinnt Vertrauen in die Korrektheit und Vollständigkeit seiner Spezifikation, indem er Eigenschaften formuliert, die für seinen Entwurf gelten müssen. Ein Werkzeug zur formalen Eigenschaftsprüfung beweist oder widerlegt diese Eigenschaften und liefert dem Designer somit verlässliche Information, die eine wertvolle Ergänzung zu den Simulationsergebnissen auf höheren Entwurfsebenen darstellt. Mit der Designvalidierung kann bereits begonnen werden, bevor das Design vollständig implementiert ist und bevor eine Testumgebung aufgebaut ist, was zu einer wesentlich höheren Qualität der entworfenen Blöcke führt.

Neben der Designvalidierung in frühen Phasen des Hardwareentwurfs wird die Eigenschaftsprüfung auch für andere Zwecke eingesetzt. Soll ein Design für einen anderen Einsatzzweck adaptiert werden, ohne dass der ursprüngliche Designer greifbar ist, könnte sich herausstellen, dass die Dokumentation an bestimmten Stellen nicht detailliert genug ist. In einem solchen Fall kann die Eigenschaftsprüfung zum Reverse-Engineering des Designs eingesetzt werden. Ein Satz von Eigenschaften, der das Verhalten des Blocks eindeutig dokumentiert, trägt dann häufig zum Verständnis des Designs bei.

Ein weiteres Szenario ergibt sich, wenn man die Eigenschaftsprüfung zur Validierung solcher Designkomponenten einsetzt, bei denen die klassische Simulation keine ausreichende Sicherheit garantiert, weil beispielsweise viele wichtige Funktionalitäten von der korrekten Implementierung eines sehr komplexen Kontrollpfades abhängen.

Als letzte Anwendungsdomäne sei noch die Verwendung der Eigenschaftsprüfung beim Debugging erwähnt. Hier geht es darum, einen Fehler im Design zu lokalisieren. Gerade Fehler, die spät im Entwurfsprozess aufgedeckt werden, sind in der Regel sehr schwer zu lokalisieren. Hier hilft die Eigenschaftsprüfung, Hypothesen über die Fehlerursache schnell zu überprüfen, ohne dass riesige Simulationen entwickelt werden müssen. Eine ausführliche Darstellung industrieller Erfahrung zu allen gerade genannten Anwendungsbereichen der Eigenschaftsprüfung wird in [BJW04] gegeben.

Die formale Eigenschaftsprüfung wird in der Regel auf Block-Ebene eingesetzt. Sie ist bislang nicht so weit verbreitet wie der Äquivalenzvergleich, aber gerade in der formalen Eigenschaftsprüfung werden zur Zeit enorme Fortschritte erzielt. Es zeichnet sich für die kommenden Jahre eine ähnlich erfolgreiche Entwicklung ab, wie sie für den Äquivalenzvergleich in der jüngsten Vergangenheit stattgefunden hat. Diese praktischen Erfolge beruhen u. a. auf Fortschritten in den Booleschen Beweistechniken, die den algorithmischen Kern der formalen Verifikationswerkzeuge darstellen.

1.2 Beweismethoden der formalen Eigenschaftsprüfung

Wichtige Meilensteine bei der Entwicklung Boolescher Beweistechniken sind graphenbasierte Darstellungen Boolescher Funktionen und Methoden zur Lösung des Booleschen Erfüllbarkeitsproblems. Leistungsfähige Algorithmen zur Lösung Boolescher Probleme sind jedoch für den Erfolg der formalen Verifikation nur zum Teil ausschlaggebend. Genauso wichtig ist die Aufbereitung und richtige Formulierung der Verifikationsaufgabe unter Ausnutzung spezieller Charakteristika des CAD-gestützten Hardware-Entwurfs bei modernen Design-Flows. Viele wichtige Eigenschaften eines Hardware-Entwurfs lassen sich beispielsweise in beschränkten Zeitintervallen formulieren. Damit wird das iterative Schaltungsmodell zu einem geeigneten Ausgangspunkt für die Eigenschaftsprüfung.

1.2.1 Boolesche Beweistechniken

Darstellung Boolescher Funktionen durch BDDs

Binäre Entscheidungsdiagramme (engl. Binary Decision Diagrams, BDDs) stellen in heutigen Verifikationswerkzeugen nach wie vor ein wichtiges Werkzeug für viele Berechnungsaufgaben dar [Bry86]. BDDs werden in den meisten Equivalence Checkern [KK97, Mat96] eingesetzt und sind die Grundlage für das Verfahren des Symbolic Model Checkings [McM93], einer bekannten Methode für die formale Eigenschaftsprüfung.

Werkzeuge zur Lösung Boolescher Erfüllbarkeitsprobleme (engl. SAT-Solver)

Die Algorithmen des SAT-Solvings sind stark mit den Algorithmen der automatischen Testmustergenerierung (ATPG) verwandt und stellen neben den BDDs die zweite wichtige Klasse von Booleschen Beweisern dar, die in der formalen Verifikation von Hardware zum Einsatz gelangen. Sie sind fester Bestandteil in Equivalence Checkern und werden zunehmend für die Eigenschaftsprüfung eingesetzt. Fortschritte wurden in jüngerer Zeit vor allem durch Verbesserungen der Datenstrukturen erzielt [MMZ$^+$01]. Ähnlich wie es bereits seit geraumer Zeit für BDDs der Fall ist, nehmen nun auch bei den SAT-Algorithmen Implementierungsaspekte einen größeren Raum in der aktuellen Forschung ein.

1.2.2 Formale Eigenschaftsprüfung basierend auf dem iterativen Schaltungsmodell (engl. Bounded Model Checking)

Die großen aktuellen Fortschritte im Bereich der formalen Eigenschaftsprüfung beruhen auf einer Modellierung der Schaltung durch ein iteratives Schaltungsmodell. Die Schaltung wird in einem Zeitfenster von k Taktintervallen (engl. time frames) betrachtet und die Gültigkeit von Eigenschaften lediglich innerhalb dieses Zeitrahmens bewiesen. Dies bedeutet zwar eine deutliche Einschränkung gegenüber den prinzipiellen Möglichkeiten des klassischen Model Checkings, die Praxis zeigt aber, dass sich auch innerhalb eines begrenzten Zeitfensters viele wichtige Eigenschaften eines Entwurfs erfassen lassen. Mit Bounded Model Checking [BCC$^+$99] (BMC) können wesentlich größere Blöcke gehandhabt werden als mit den herkömmlichen Methoden des Model Checkings. Als Beweismaschine dient in der Regel ein SAT-Solver.

1.3 Beitrag dieser Arbeit

Formale Verifikationswerkzeuge benötigen ein formales Modell des zu verifizierenden Objekts. Heutige Werkzeuge generieren dieses Modell aus dem Schaltungsentwurf, der in der Regel in einer Hardwarebeschreibungssprache (engl. hardware description language (HDL)) vorliegt. Dazu stellen die Werkzeuge sogenannte Front-Ends für die verschiedenen HDLs (Verilog, VHDL, System C, System Verilog) zur Verfügung. Die so erzeugten Modelle können dann mit unterschiedlichen Back-Ends zur Lösung der Verifikationsaufgabe weiterverarbeitet werden.

Bisher werden für die Modellgenerierung in der Eigenschaftsprüfung im Allgemeinen die gleichen Syntheseverfahren benutzt wie bei der späteren Implementierung. Diese Verfahren zielen aber in erster Linie auf eine kosteneffiziente Implementierung ab. Optimierungskriterien sind dabei z.B. Signallaufzeiten, Chipfläche, etc.. In der Eigenschaftsprüfung müssen diese Optimierungskriterien aber keinesfalls zu optimalen Ergebnissen führen. Aus Implementierungssicht kann es z.B. sinnvoll sein, bestimmte Logikteile zu duplizieren und an verschiedenen Stellen des Chips die gleiche Funktion zu berechnen, um dadurch die Signallaufzeit zu reduzieren. Der Verifikationsaufwand wird durch solche redundante Strukturen aber eher erhöht. Das Optimierungskriterium sollte in der Eigenschaftsprüfung also die Schwierigkeit des zu lösenden Booleschen Problems sein.

Diese Arbeit stellt deshalb Methoden zur Verfügung, um im Front-End eines Eigenschaftsprüfers Modelle zu generieren, die zu leichter lösbaren SAT-Instanzen für das Back-End führen. Es werden exemplarisch zwei Anwendungsfälle der SAT-basierten Eigenschaftsprüfung untersucht, die mit bisherigen Methoden nur unzureichend behandelt werden können. Dabei handelt es sich zum einen um die Verifikation von sequentiellem Verhalten auf unbeschränkten Zeitintervallen und zum anderen um die Verifikation komplexer arithmetischer Schaltungsblöcke. In beiden Fällen wird aufgezeigt, wie eine abgestimmte Kombination von Modellgenerierung im Front-End und Lösungsalgorithmus im Back-End des Eigenschaftsprüfers zu enormer Steigerung der Performanz des Verifikationswerkzeugs und damit zur Lösung der Verifikationsaufgabe beiträgt [WSKer].

1.3.1 Sequentielles Verhalten mit langen Zeitfenstern

Eigenschaften, die das Verhalten einer Schaltung über große oder ggf. unbeschränkte Zeitfenster spezifizieren, können mit SAT-basierten Methoden bisher nicht behandelt werden.

Wir beschränken uns auf die Untersuchung von Sicherheitseigenschaften als einem Vertreter von Eigenschaften, die nicht in einem beschränkten Zeitfenster bewiesen werden können. Eine solche Eigenschaft legt fest, dass ein bestimmtes Fehlverhalten p der Schaltung niemals auftritt.

Im Gegensatz dazu fordern Lebendigkeitseigenschaften, dass ein bestimmtes gewünschtes Verhalten der Schaltung immer wieder auftritt. Lebendigkeitseigenschaften können nach [BAS02] auf Sicherheitseigenschaften abgebildet werden. Damit ist die o.g. Einschränkung lediglich technischer Natur.

Die bisherigen Verfahren zur Verifikation von Sicherheitseigenschaften lassen sich in zwei Klassen einteilen. Zunächst einmal gibt es Verfahren [BCL$^+$94,McM93], die auf der Berechnung der Menge aller erreichbaren Zustände eines Automaten basieren. Aufgrund der sogenannten Zustandsexplosion sind diese Verfahren allerdings nur für sehr kleine Schaltungen praktisch einsetzbar. Einen vielversprechenden Ansatz zur Überwindung dieser Schranke stellen die induktionsbasierten Eigenschaftsprüfer [ABE00, SSS00] dar. Diese Verfahren sind insbesondere dann effizient, wenn die zu verifizierende Eigenschaft möglichst viel Erreichbarkeitsinformation enthält. Es liegt deshalb nahe, Invarianten zu generieren und mit der Eigenschaft gemeinsam zu verifizieren.

Die Bestimmung von Erreichbarkeitsinformationen ist bekanntlich ein hoch komplexes Problem und selbst approximative Verfahren scheitern häufig für Designs von realistischen Größenordnungen. Ein Freiheitsgrad bei der Modellgenerierung, der in diesem Zusammenhang noch weitgehend ungenutztes Potential bietet, liegt in der Zustandskodierung für die Komponenten eines Hardwaredesigns. Die Zustandskodierung des Designs kann für die Eigenschaftsprüfung prinzipiell unabhängig von der späteren Design-Implementierung gewählt werden.

Im Kontext der induktionsbasierten Eigenschaftsprüfung wird diese Kodierung nun darauf abzielen, dass sich Erreichbarkeitsinformation für das entstandene Modell leichter gewinnen lässt. Es zeigt sich, dass eine geeignete Zustandskodierung die Generierung von Invarianten unterstützt.

1.3.2 Arithmetische Blöcke

Wie bereits in 1.2.2 erläutert, bildet das iterative Schaltungsmodell die Grundlage für die SAT-basierte Eigenschaftsprüfung. Moderne SAT-Solver sind aber oftmals nicht in der Lage, Instanzen zu lösen, die der Verifikation arithmetischer Schaltungen dienen. Typischerweise scheitert die SAT-basierte Eigenschaftsprüfung bei Datenpfaden mit großen oder vielen arithmetischen Blöcken. Diesem Problem stehen große Erfolge bei der Verifikation der Kontrolllogik gegenüber.

In den Datenpfaden begnügt man sich aus Komplexitätsgründen häufig mit unvollständigen Beweisen, die mit Hilfe sogenannter Bit-Slicing Techniken gewonnen werden. Durch die Beschränkung des Beweises auf einzelne Bits eines Datenpfades wird dabei die Komplexität ausreichend reduziert. Dies genügt in vielen Fällen, um Fehler im Design aufzudecken, kann jedoch nie die Abwesenheit solcher Fehler garantieren. Besonders Fehler in Spezialfällen werden dabei häufig übersehen. Motiviert durch dieses Vorgehen wurden deshalb in [Joh01, JD01] automatische Verfahren entwickelt, die die Bitbreite eines Datenpfades reduzieren. Die dabei entstehenden Modelle sind oftmals immer noch zu groß und es gibt Fälle, in denen gar keine Reduzierung möglich ist. Schwierig wird dies insbesondere dann, wenn arithmetische Blöcke aus Performanzgründen auf der Bitebene entworfen werden.

Die enge Verflechtung von Bitebene und Wortebene stellt auch die sogenannten Wortebenen-Solver, welche entweder auf Integer Linear Programming (ILP) [BD02, FDK01, ZKC01] oder auf Constraint Logic Programming (CLP) [ZCR01] basieren, vor schwer überwindbare Probleme. Dies liegt darin begründet, dass Wortebenen-Solver im Booleschen Teil der Probleme nicht wesentlich mehr Informationen zur Beschränkung des Suchraums deduzieren können, als dies bei SAT-Solvern der Fall ist. Dies führt zu erheblichem Mehraufwand, da diese Solver für jeden Entscheidungsschritt wesentlich mehr Rechenzeit benötigen.

Ein weiteres Problem für Solver der Wortebene besteht darin, dass sich die Zwischenergebnisse in den Pipeline-Stufen eines Designs oftmals gar nicht auf der Wortebene modellieren lassen. In diesen Fällen degeneriert das Wortebenen-Problem zu einem SAT-Problem mit nur sehr geringem Wortebenenanteil. Bei Verwendung von ILP-Techniken kommt noch hinzu, dass Multiplikationen hier erst noch linearisiert werden müssen.

Gefolgert wird aus dieser Problematik, dass nur die Entwicklung hybrider Solver einen Ausweg darstellt. Die Kombination von Bitebenen- und Wortebenen-Solvern stellt sich allerdings als schwierig heraus. So besitzen die Solver in der Regel unterschiedliche Problemdarstellungen, was den Austausch von Informationen über die jeweiligen Nichtlösungsgebiete erschwert. Erste Ansätze, die diese Probleme angehen, findet man in [ABC$^+$02] und [CK03]. Dabei verallgemeinert [ABC$^+$02] den Davis-Putnam-Logeman-Loveland-Algorithmus (DPLL) für pseudo-Boolesche Constraints, die Anstelle von Klauseln zur Problemdarstellung verwendet werden. Des Ansatz aus [CK03] verknüpft lineare Gleichungen als elementare Aussagen zu einer Booleschen Formel. In beiden Fällen bleibt unklar, wie man die kritischen BMC-Probleme auf Instanzen der betrachteten Problemklassen abbilden will.

Ein weiterer Impuls für die Verifikation arithmetischer Schaltungen wurde von [SK01] ausgelöst. Darin wurde eine Methode für den Äquivalenzvergleich für Multiplizierer entwickelt, die auf einer Beschreibung der Schaltungen auf der arithmetischen Bitebene beruht. Diese Beschreibung enthält partielle Produkte und ein Netzwerk aus 1-Bit-Additionseinheiten (Halbaddierer, Volladdierer, XOR). Um zu beweisen, dass eine Gatternetzliste einen Multiplizierer implementiert, wird diese Beschreibung aus der Netzliste extrahiert. Die Extraktion ist in der Eigenschaftsprüfung allerdings überflüssig, da man die benötigte Information auch im Front-End generieren kann. Es stellt sich vielmehr die Frage, wie man die Beschreibung auf der arithmetischen Bitebene nutzen kann, um SAT-Instanzen aus der Eigenschaftsprüfung zu lösen. Einen ersten Ansatz dazu liefert [WSK04a], indem die Information über die Additionsnetzwerke in der SAT-Instanz genutzt wird, um den Suchraum eines SAT-Solvers zu beschneiden. Dies geschieht, indem die Vorwärtspropagierung innerhalb der Additionsnetzwerke verbessert wird. Der Ansatz hat sich lokal, d.h. für die einzelnen Additionsnetzwerke, als gewinnbringend herausgestellt. Allerdings reichen diese Gewinne nicht aus, um dem exponentiellen Blowup zu begegnen, der in stark optimierten Datenpfaden mit mehreren arithmetischen Blöcken beobachtet wird.

In dieser Arbeit wird deshalb eine neue Normalisierungstechnik auf der arithmetischen Bitebene vorgeschlagen. Als Basis dieser Technik wird eine Beschreibung auf der arithmetische Bitebene (ABL) für arithmetische Problembestandteile entwickelt, die neben den partiellen Produkten und den Additionsnetzwerken auch Vergleiche aus der Probleminstanz berücksichtigt. Letztere dienen als Bezugspunkte für die eigentliche Normalisierung.

Im Rahmen dieser Arbeit werden die normalisierten Instanzen an einen SAT-Solver [ES03] weitergereicht. Es ist allerdings genauso möglich, hybride Solver wie in [ABC$^+$02, CK03] zu benutzen. Der vorgeschlagene Ansatz sollte auch hier nützlich sein, indem Abstraktionen auf der Wortebene bereitgestellt werden, welche diese Solver ausnutzen können.

1.4 Aufbau dieser Arbeit

In Kapitel 2 werden die fundamentalen Notationen und Begriffe aus Schaltungstheorie und formaler Logik eingeführt, soweit sie für diese Arbeit von Bedeutung sind.

Aufbauend darauf wird in Kapitel 3 der Stand der Technik in der formalen Eigenschaftsprüfung für synchrone digitale Schaltungen dargestellt. Hierbei werden die zwei Problemfelder identifiziert, für die bisher keine geeigneten Verfahren existieren.

Zur Lösung der Probleme bei Eigenschaften in unbeschränkten Zeitfenstern werden dann in Kapitel 4 Freiheitsgrade bei der Modellgenerierung im Front-End des Eigenschaftsprüfers ausgenutzt. Die erzeugten Modelle erlauben die effiziente Generierung von Invarianten für die induktionsbasierte Eigenschaftsprüfung.

Um das Verfahren zur Normalisierung einer Beschreibung auf der arithmetischen Bitebene (ABL) von Schaltung und Eigenschaft anzuwenden, welches im Kapitel 5 vorgestellt wird, muss diese Beschreibung bei der Modellgenerierung erzeugt werden. Durch die Normalisierung wird die zu lösende SAT-Instanz dramatisch vereinfacht.

Abschließend wird in Kapitel 6 auf die zukünftigen Perspektiven dieser Arbeit eingegangen und geplante Erweiterungen der vorgestellten Verfahren skizziert.

Kapitel 2

Grundlagen

Dieses Kapitel führt in aller Kürze die fundamentalen mathematischen Konzepte und Notationen ein, die im Rahmen dieser Arbeit von Bedeutung sind. Dem erfahrenen Leser werden diese wohlvertraut sein. Deshalb empfiehlt es sich, dieses Kapitel ggf. zu überspringen und später bei Bedarf darauf zurückzukommen. Es liegt in der Natur der Sache, dass eine solche Einführung niemals erschöpfend sein kann. Aus diesem Grund sei auf die vielen guten Textbücher zur diskreten Mathematik (z.B. [GKP94, GT96, Kal86]), Schaltungstheorie (z.B. [Kat94]), formalen Logik (z.B. [Sch00]) und zu Verifikationsalgorithmen (z.B. [HS96]) verwiesen, sofern der Bedarf zur Vertiefung besteht.

2.1 Relationen

Unter dem kartesischen Produkt $A \times B$ zweier Mengen A und B versteht man die Menge aller geordneten Paare (a, b) mit $a \in A$ und $b \in B$. Jede Teilmenge $R \subseteq A \times B$ des kartesischen Produkts wird (binäre) Relation zwischen A und B genannt. Für $(a, b) \in R$ schreiben wir auch aRb. Im Fall $A = B$ sprechen wir von einer Relation auf A. Die folgende Definition benennt einige interessante Eigenschaften von Relationen.

Definition 2.1. ***(Eigenschaften von Relationen)***
Sei R eine Relation auf A.

- ***Reflexivität:*** *R heißt reflexiv genau dann, wenn $\forall x \in A : xRx$ gilt.*
- ***Symmetrie:*** *R heißt symmetrisch genau dann, wenn $\forall x, y \in A : (xRy \Rightarrow yRx)$ gilt.*
- ***Antisymmetrie:*** *R heißt antisymmetrisch genau dann, wenn $\forall x, y \in A : ((xRy \wedge yRx) \Rightarrow x = y)$ gilt.*
- ***Transitivität:*** *R heißt transitiv genau dann, wenn $\forall x, y, z \in A : ((xRy \wedge yRz) \Rightarrow xRz)$ gilt.*
- ***Totale Relation:*** *R heißt (links) total genau dann, wenn $\forall x \in A\ \exists y \in A : xRy$ gilt.*

- ***Eindeutigkeit:*** *R heißt (rechts) eindeutig genau dann, wenn $\forall x, y, z \in A : xRy \wedge xRz \Rightarrow y = z$ gilt.*

Auf Basis dieser Eigenschaften können die wichtigen Relationstypen der Äquivalenz-, Ordnungsrelation und Funktion definiert werden.
Eine links totale, rechts eindeutige Relation F wird Funktion genannt. Statt xFy schreibt man bei Funktionen $F(x) = y$.
Unter einer *Äquivalenzrelation* versteht man eine reflexive, symmetrische und transitive Relation. Eine solche Äquivalenzrelation auf A induziert eine Zerlegung der Grundmenge A in Äquivalenzklassen. Diese sind aufgrund der Transitivität paarweise disjunkt. Umgekehrt definiert jede Zerlegung der Grundmenge in paarweise disjunkte Teilmengen eine Äquivalenzrelation.
Ist eine Relation reflexiv, antisymmetrisch und transitiv, wird sie eine *Halbordnung* oder *Ordnungsrelation* genannt. Gilt darüber hinaus für je zwei Elemente $a, b \in A$: aRb oder bRa, so heißt die Relation *totale Ordnung*. Unter einer (total) geordneten Menge A verstehen wir eine Menge, die durch eine (totale) Ordnungsrelation R geordnet wird. Entfernt man aus einer (totalen) Ordnung die Diagonale $\{(x, x)|x \in A\}$ so spricht man von einer strikten Ordnung. Strikte Ordnungen werden in dieser Arbeit durch Hinzufügen der Diagonale mit der entsprechenden Ordnung identifiziert. In einer geordneten Menge lassen sich bestimmte Elemente identifizieren.

Definition 2.2. ***(Besondere Elemente einer Halbordnung)***
Sei $\leq$ Halbordnung auf A und $B \subseteq A$.

- ***Obere Schranke von B:*** *Eine obere Schranke von B ist ein Element $a \in A$ mit $\forall b \in B : b \leq a$. Gilt ferner $a \leq a'$ für alle anderen oberen Schranken a' von B, so heißt a kleinste obere Schranke von B.*

- ***Untere Schranke von B:*** *Eine untere Schranke von B ist ein Element $a \in A$ mit $\forall b \in B : a \leq b$. Gilt ferner $a' \leq a$ für alle anderen unteren Schranken a' von B, so heißt a größte untere Schranke von B.*

- ***Maximum:*** *Ist die kleinste obere Schranke x von B in B enthalten, so heißt x Maximum.*

- ***Minimum:*** *Ist die größte untere Schranke x von B in B enthalten, so heißt x Minimum.*

Eine geordnete Menge A heißt *Verband*, wenn jede Menge $\{a, b\} \subseteq A$ eine kleinste obere Schranke *sup*$(\{a, b\})$und eine größte untere Schranke *inf*$(\{a, b\})$ besitzt. Die Operationen *sup* und *inf* eines Verbandes werden häufig auch mit $+$ und $\cdot$ bezeichnet. Wir schließen uns im Folgenden dieser Notation an. Aus der Definition des Verbands folgt unmittelbar, dass ein Verband jeweils ein eindeutiges Maximum und eindeutiges Minimum enthält. Das Maximum bezeichnen wir mit 1, das Minimum mit 0. Mit Hilfe des Verbands wird in Abschnitt 2.3 die Boolesche Algebra definiert. An dieser Stelle seien bereits einige Rechenregeln für den Verband angegeben.

Satz 2.1. *In einem Verband gelten die folgenden Rechenregeln:*

- *Idempotenz:* $a \cdot a = a + a = a$
- *Kommutativität:* $a \cdot b = b \cdot a$ *und* $a + b = b + a$
- *Absorption:* $a + (a \cdot b) = a = a \cdot (a + b)$
- *Assoziativität:* $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ *und* $a + (b + c) = (a + b) + c$
- *0-1-Gesetze:* $a + 0 = a$, $a \cdot 0 = 0$, $a \cdot 1 = a$, $a + 1 = 1$

2.2 Graphen

Ein *(gerichteter) Graph* $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten in Form einer binären Relation E auf V. E wird dabei auch Adjazenz- oder Nachbarschaftsrelation genannt. Die Eigenschaften eines Graphen sind im wesentlichen durch seine Adjazenzrelation bestimmt. Sämtliche Eigenschaften für Relationen aus Abschnitt 2.1 gelten analog für Graphen. So ist ein Graph beispielsweise symmetrisch genau dann, wenn seine Adjazenzrelation symmetrisch ist. Im Rahmen dieser Arbeit betrachten wir stets gerichtete Graphen. Ungerichtete Graphen lassen sich leicht durch symmetrische Graphen modellieren. Durch Hinzunahme der invertierten Kanten $E' := \{(v, w) | (w, v) \in E\}$ geht man von einem gerichteten Graphen zu dem entsprechenden ungerichteten Graphen über. Die Teilmengenrelation $\subseteq$ induziert eine Untergraphenrelation $\ll$ wie folgt:

$$G' = (V', E') \ll G = (V, E) :\Leftrightarrow (V' \subseteq V) \wedge (E' \subseteq E).$$

Der Eingangsgrad eines Knoten $v \in V$ ist die Anzahl der eingehenden Kanten $(v', v) \in E$, der Ausgangsgrad die Anzahl der wegführenden Kanten $(v, v') \in E$. Die Summe aus Eingangsgrad und Ausgangsgrad heißt Grad des Knoten v.

Ein Knoten mit Eingangsgrad 0 wird *Quelle*, ein Knoten mit Ausgangsgrad 0 *Senke* genannt. Unter einer *Schlinge* verstehen wir eine Kante $(v, v) \in E$ mit identischem Anfangs- und Endknoten.

Eine endliche Folge adjazenter Kanten $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-2}, v_{n-1}), (v_{n-1}, v_n))$ ist eine *Kantenfolge*. Gilt $v_1 = v_n$ so heißt die Kantenfolge *geschlossen*, andernfalls heißt sie *offen*. Sind die Kanten einer Kantenfolge paarweise verschieden, so nennen wir diese Kantenfolge *Kantenzug*. Sind die Knoten, mit Ausnahme des Anfangsknotens v_1 und des Endknotens v_n, paarweise verschieden, so heißt eine Kantenfolge *Pfad*. Ein Pfad mit $v_1 = v_n$ heißt geschlossen und wird auch *Zyklus* genannt. Ist dies nicht der Fall, heißt der Pfad offen. Ein Graph ohne Zyklen wird azyklischer Graph genannt. Für jede Kante $(v, w) \in E$ heißt v direkter Vorgänger von w und umgekehrt w direkter Nachfolger von v. Sind beide Knoten durch einen Pfad verbunden, so spricht man vom transitiven Vorgänger bzw. transitiven Nachfolger. Die Menge der (transitiven) Vorgänger bzw. Nachfolger wird auch als *(transitiver) Fanin bzw. Fanout* bezeichnet.

Ein Graph, dessen Knoten paarweise durch Pfade verbunden sind, heißt stark zusammenhängend. Ist für einen gerichteten Graphen lediglich der entsprechende ungerichtete Graph zusammenhängend, so spricht man von schwachem Zusammenhang.
Ein *Baum* ist ein ungerichteter, azyklischer und zusammenhängender Graph. Oftmals wird in einem Baum ein Knoten als Wurzel markiert. Knoten, die nur einen benachbarten Knoten haben, heißen *Blätter* des Baums.

2.3 Boolesche Algebra

Die Boolesche Algebra ist das fundamentale mathematische Werkzeug zur Synthese, Verifikation und zum Test digitaler Schaltungen. Bereits in den 1930er Jahren hat Shannon in [Sha38] gezeigt, dass das Verhalten solcher Schaltungen durch die zweiwertige Boolesche Algebra beschrieben werden kann. Boolesche Algebren lassen sich sehr elegant als komplementärer, distributiver Verband definieren.

Definition 2.3. ***(Boolesche Algebra)***
Ein Verband auf der Menge B *heißt Boolesche Algebra, wenn für* $a, b, c \in B$ *gilt:*

- *Distributivität:* $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ *und* $a + (b \cdot c) = (a + b) \cdot (a + c)$
- *Komplement: Für jedes Element* $a \in B$ *existiert eine eindeutiges Element* $\overline{a} \in B$ *mit* $a \cdot \overline{a} = 0$ *und* $a + \overline{a} = 1$.

Die hier vorgestellte Definition der Booleschen Algebra ist natürlich nur eine mögliche Variante unter vielen. Eine äquivalente Definition erhält man, wenn man die so genannten Huntington-Axiome zugrunde legt. Dabei wird die Menge B mit zwei Operationen $+$ und $\cdot$ betrachtet, für die eine Reihe von Gesetzen gelten. Diese sind Kommutativität, Distributivität, Komplement und 0-1-Gesetze. Es ist sogar möglich, die Boolesche Algebra mit einer einzigen Gleichung zu definieren. Dies wurde in [MVF$^+$02] gezeigt.
Die folgenden Verbände sind Boolesche Algebren:

Beispiel 2.1. *(Schaltalgebra)*
$\mathbb{B} = \{0, 1\}$ *mit* $0 \leq 1$. *Diese Algebra wird auch Schaltalgebra genannt. Die Operationen* $+$ *und* $\cdot$ *und das Komplement werden dabei oftmals auch durch* $\vee$, $\wedge$ *und* $\neg$ *bezeichnet.*

Beispiel 2.2. *(Bitvektoralgebra)*
$\mathbb{B}^n$ *mit* $(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n) :\Leftrightarrow \forall_i a_i \leq b_i$. *Diese Algebra wird auch Bitvektoralgebra genannt.*

Beispiel 2.3. *(Boolesche Funktionen) Die Menge der Booleschen Funktionen* $f : \mathbb{B}^n \to \mathbb{B}$ *mit* $f \leq g :\Leftrightarrow \forall_x f(x) \leq g(x)$.

Beispiel 2.4. *(Bitvektorfunktionen)*
Die Menge der Bitvektorfunktionen $f : \mathbb{B}^n \to \mathbb{B}^n$ *mit* $f \leq g :\Leftrightarrow \forall_x f(x) \leq g(x)$.

Beispiel 2.5. *(Potenzmenge)*
Die Potenzmenge $P(M)$ mit der Teilmengenrelation $\subseteq$.

Im Folgenden verstehen wir unter einer Booleschen Algebra stets die Schaltalgebra. Zur Verbesserung der Lesbarkeit führen wir noch folgende Symbole ein:

- XOR: $a \oplus b := \overline{a}b + a\overline{b}$
- Äquivalenz: $a \equiv b := \overline{a \oplus b}$
- Implikation: $a \Rightarrow b := b \Leftarrow a := \overline{a} + b$
- If-Then-Else: $ite(a, b, c) := (ab) + (\overline{a}c)$.

2.4 Darstellungen Boolescher Funktionen

Zur Beschreibung des Verhaltens kombinatorischer digitaler Schaltungen eignen sich Boolesche Funktionen, wie sie in Abschnitt 2.3 in Beispiel 2.3 erwähnt wurden. Typischerweise werden digitale Schaltungen aus Gattern aufgebaut, die logische Grundfunktionen $(+, \cdot, \oplus, \neg, \ldots)$ implementieren. Gatter sind elektronische Bausteine, die einen oder mehrere Eingänge und einen Ausgang besitzen. Die Spannung am Ausgang eines Gatters hängt von den Eingangsspannungen ab. Man unterscheidet jeweils zwei Spannungspegel ”high” und ”low” und weist diesen die logischen Werte 1 und 0 zu. In Tabelle 2.1 sind einige Standard-Logikgatter dargestellt.
Eine Menge von Logikgattern heißt *funktionell* vollständig oder auch eine *Basis* für den Raum der Booleschen Funktionen, wenn sich jede Boolesche Funktion als Komposition der Schaltfunktionen der Gatter darstellen lässt. So ist z.B. $\{$UND, ODER, INVERTER$\}$ funktionell vollständig, wobei man sogar entweder UND oder ODER noch aus dieser Menge entfernen kann. Damit sind auch$\{$NAND$\}$ und $\{$NOR$\}$ funktionell vollständig. Im Rahmen dieser Arbeit wird das funktionale Verhalten von Schaltungen betrachtet. Deshalb können Gatter mit ihrer Schaltfunktion identifiziert werden.
Im Folgenden werden wir nun einige Darstellungen für Boolesche Funktionen einführen, die für die formale Hardware-Verifikation von Bedeutung sind.

2.4.1 Wahrheitstafel

Die Wahrheitstafel einer Booleschen Funktion $f : \mathbb{B}^n \to \mathbb{B}$ enthält für jede der 2^n Eingabekombinationen $(x_1, \ldots, x_n)$ den Funktionswert $f(x_1, \ldots, x_n)$. Da die Funktion eindeutig durch die Menge ihrer Funktionswerte definiert ist, handelt es sich bei der Wahrheitstafel um eine bis auf Permutation der Zeilen eindeutige Darstellung. Eine solche Darstellung wird auch kanonisch genannt. Allerdings ist die Wahrheitstafel nicht kompakt. Es werden 2^n Einträge zur Darstellung benötigt. Daher ist die Wahrheitstafel nur für sehr kleine Funktionen praktikabel. Tabelle 2.2 ist ein Beispiel für eine Wahrheitstafel einer Funktion $f : \mathbb{B}^3 \to \mathbb{B}$.

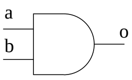
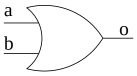
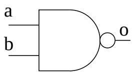
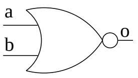
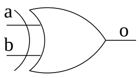
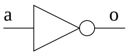
Gattersymbol	Name	Schaltfunktion
	AND	$o = a \cdot b$
	OR	$o = a + b$
	NAND	$o = \overline{a \cdot b}$
	NOR	$o = \overline{a + b}$
	XOR	$o = a \oplus b$
	INVERTER	$o = \overline{a}$

Tabelle 2.1: Beispiele für Grundgatter

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Tabelle 2.2: Wahrheitstafel der Funktion $f(x_1, x_2, x_3) = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$

2.4.2 Normalformen

In diesem Abschnitt wird gezeigt, dass sich jede Boolesche Funktion in eine äquivalente zweistufige Darstellung überführen lässt. Dies Darstellung setzt sich aus Variablen bzw. negierten Variablen zusammen. Ein *Literal* ist eine Variable x oder seine Negation $\overline{x}$. Ein *Produktterm* ist die Konstante 1 oder ein Produkt von Literalen $l_1 \cdot l_2 \cdot \cdots \cdot l_{k-1} \cdot l_k$, wobei keine Variable mehrfach vorkommt. Ein Produktterm heißt *Minterm* für $\mathbb{B}^n$, wenn er alle Variablen $x_1, \ldots, x_n$ enthält. Ein *Summenterm* ist die Konstante 0 oder eine Summe von Literalen $l_1 + l_2 + \cdots + l_k$, wobei keine Variable mehrfach vorkommt. Ein Summenterm heißt *Maxterm* für $\mathbb{B}^n$, wenn er alle Variablen $x_1, \ldots, x_n$ enthält.
Eine Summe von Produkttermen heißt *disjunktive Normalform (DNF)*. Ein Produkt von Summentermen heißt *konjunktive Normalform (KNF)*. Die Summenterme einer KNF werden auch Klauseln genannt. Enthält eine DNF nur Minterme, so ist sie *kanonisch*. Eine KNF, die nur Maxterme enthält, ist ebenfalls *kanonisch*. Die beiden letzten Aussagen sieht man leicht ein, wenn man sich verdeutlicht, dass jeder Min- bzw. Maxterm genau einer Zeile der Wahrheitstabelle entspricht. Die kanonische DNF enthält alle Minterme, für welche die Funktion den Wert 1 annimmt. Analog enthält die kanonische KNF alle Maxterme, für welche die Funktion verschwindet.
Die Funktion aus unserem Beispiel in Tabelle 2.2 ist bereits als DNF gegeben. Allerdings ist diese noch nicht kanonisch. Die kanonische DNF lautet:

$$f(x_1, x_2, x_3) = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3.$$

Aus der Wahrheitstabelle liest man ferner die kanonische KNF ab:

$$f(x_1, x_2, x_3) = (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x_2} + x_3) \cdot (\overline{x_1} + x_2 + x_3) \cdot (\overline{x_1} + x_2 + \overline{x_3}).$$

Eine weitere KNF für f ist

$$f(x_1, x_2, x_3) = (x_1 + x_3) \cdot (x_2 + x_3) \cdot (\overline{x_1} + x_2).$$

Im Allgemeinen kann eine Boolesche Funktion mit Hilfe des Shannon'schen Entwicklungssatzes in eine der beiden Normalformen überführt werden. Um diesen Satz aufstellen zu können, soll zunächst der Begriff des Kofaktors einer Booleschen Funktion eingeführt werden.

Definition 2.4. ***(Kofaktoren Boolescher Funktionen)***
Sei $f : \mathbb{B}^n \rightarrow \mathbb{B}$ eine Boolesche Funktion und sei $v_i \in \mathbb{B}$. Die Funktion $f|_{x_i=v_i} : \mathbb{B}^{n-1} \rightarrow \mathbb{B}$ mit $f|_{x_i=v_i}(x_1, \ldots, x_{n-1}) := f(x_1, \ldots, x_{i-1}, v_i, x_i, \ldots, x_{n-1})$ heißt Kofaktor von f zum Wert v_i für x_i.

Man erhält einen Kofaktor der Funktion also durch Einsetzen eines Wertes für x_i. Damit kann nun der Shannon'sche Entwicklungssatz formuliert werden.

Satz 2.2. ***(Shannon'scher Entwicklungssatz)***
Für jede Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ *und jede Variable* x_i *gilt:*

a) $f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}$ *und*

b) $f = (x_i + f|_{x_i=0}) \cdot (\overline{x_i} + f|_{x_i=1})$.

Das Beispiel in diesem Abschnitt zeigt schon, dass die Eigenschaft der Kanonizität zu Lasten der Kompaktheit der Darstellung einer Booleschen Funktion gehen kann. Die kanonische DNF und KNF einer Funktion wachsen im Allgemeinen exponentiell mit der Variablenanzahl, was ja schon bei der Wahrheitstafel als Nachteil angeführt wurde.
Neben der Shannon'schen Dekomposition sind eine ganze Reihe weiterer Dekompositionen untersucht worden. Beispielhaft sei hier noch die positive Davio-Dekomposition genannt.

Satz 2.3. ***(Positive Davio-Dekomposition)*** *Für jede Boolesche Funktion* $f : \mathbb{B}^n \rightarrow \mathbb{B}$ *und jede Variable* x_i *gilt:*

$$f = f|_{x_i=0} \oplus x_i f|_{x_i=1} \oplus x_i f|_{x_i=0}$$

Iterierte Anwendung dieser Dekomposition liefert eine zweistufige Darstellung der Funktion aus XOR-und UND-Gattern. Für eine detailliertere Einführung in die Theorie der Dekompositionen sei der Leser auf die vielfältige Literatur (z.B. [DB98]) verwiesen.

2.4.3 Binäre Entscheidungsdiagramme

Binäre Entscheidungsdiagramme sind ursprünglich von Lee [Lee59] und Akers [Ake78] vorgeschlagen worden, um Boolesche Funktionen zu repräsentieren. Die weitverbreitete Anwendung ist allerdings hauptsächlich durch Bryant's [Bry86] Idee, auf den Variablen einer Funktion eine totale Ordnung einzuführen, ausgelöst worden. Dazu wird mit jeder Variablen ein Index assoziiert. Die so eingeschränkten Diagramme werden *Ordered Binary Decision Diagram (OBDD)* genannt.

Definition 2.5. ***(OBDD)***
Ein OBDD ist ein gerichteter, azyklischer Graph $F = (V, E)$ *mit genau einer Quelle (Wurzel), wobei jeder Knoten* $v \in V$*, der keine Senke (Blatt) ist, mit einem Index index*(v) *markiert wird, der auf eine Variable der Menge* $\{x_1, \ldots x_n\}$ *verweist. Jeder Knoten* $v \in V$*, der kein Blatt ist, hat genau zwei Nachfolger high*(v) *und low*(v)*. Jedes Blatt* v *ist mit einem Wert value*$(v) \in \mathbb{B}$ *markiert. Für jede Kante* $(v, w) \in E$*, bei der* w *kein Blatt ist, gilt: index*$(v) <$ *index*(w)*.*

Mit dieser Definition ist allerdings nur die Struktur eines OBDDs festgelegt. Es bleibt, die Semantik eines OBDDs festzulegen. Dazu ist für jeden OBDD eine Boolesche Funktion anzugeben, die von diesem OBDD repräsentiert wird. Dies geschieht in Definition 2.6.

Definition 2.6. ***(Semantik von OBDDs)***
Jeder Knoten v eines OBDD definiert eine Boolesche Funktion
$f^v : \mathbb{B}^n \rightarrow \mathbb{B}$ *mit:*

- $f^v(x_1, \ldots, x_n) = \mathit{value}(v)$*, falls v ein Blatt ist und*
- $f^v(x_1, \ldots, x_n) = \overline{x_i} \cdot f^{\mathit{low}(v)} + x_i \cdot f^{\mathit{high}(v)}$*, falls v kein Blatt und index$(v) = i$.*

Ein OBDD mit Wurzel v stellt also die Funktion f^v dar. Die Funktionen $f^{\mathit{high}(v)}$ bzw. $f^{\mathit{low}(v)}$ entsprechen den Kofaktoren bezüglich $x_{\mathit{index}(v)}$.
Ein OBDD kann isomorphe Untergraphen enthalten. Fordert man für OBDDs, dass alle isomorphen Untergraphen durch einen festgelegten Vertreter ihrer Isomorphieklasse substituiert werden, so erhält man die sogenannten reduzierten OBDDs (ROBDDs). Ein ROBDD ist insbesondere eine kanonische Darstellung für eine Boolesche Funktion. Die Basisoperationen der Booleschen Algebra können auf ROBDDs in polynomieller Zeit in der Anzahl der Knoten durchgeführt werden. ROBDDs sind natürlich kein Allheilmittel zur Behandlung unlösbarer Probleme. Einige Boolesche Funktionen benötigen nämlich ROBDDs von exponentieller Größe. Exemplarisch sei hier die Schaltfunktion einer Multiplikation genannt. Die ROBDDs für diese Funktion wachsen unter jeder Variablenordnung exponentiell mit der Anzahl der Variablen. Häufig hängt die Größe des ROBDDs jedoch sehr stark von der vorgegebenen Variablenordnung ab und kann zwischen exponentieller Größe im schlimmsten Fall und linearer Größe im besten Fall variieren.
Da das Problem, die optimale Variablenordnung zu finden, NP-vollständig ist, wurde an dieser Stelle sehr viel Forschungsaufwand betrieben, um gute Heuristiken zu entwickeln. Allerdings konnte für einige Funktionen, wie die arithmetische Multiplikation, nachgewiesen werden, dass der ROBDD bei jeder Variablenordnung exponentiell wächst [Bry86].

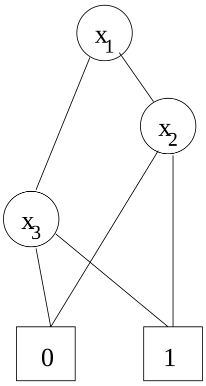


Abbildung 2.1: BDD von $f(x_1, x_2, x_3) = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$ zur Variablenordnung $x_1 < x_2 < x_3$

In Abbildung 2.1 ist abschließend noch ein ROBDD für die Beispielfunktion aus den vorherigen Abschnitten dargestellt. Im Folgenden benutzen wir den Begriff BDD als Synonym für den Begriff ROBDD.

2.4.4 Boolesche Netzwerke

Ein Boolesches Netzwerk ist eine technologieunabhängige Beschreibung einer kombinatorischen Schaltung. Die Implementierung einer Schaltung auf Basis der in Abschnitt 2.4.2 eingeführten Normalformen Boolescher Funktionen erzeugt zweistufige Schaltungen, d.h. jeder Pfad von einem Eingang zu einem Ausgang der Schaltung enthält maximal zwei Gatter, wobei die Inverter nicht mitgezählt werden. Häufig erhält man wesentlich kleinere Schaltungen mit der gleichen Funktion, wenn man mehrstufige Darstellungen Boolescher Funktionen zulässt. Der Aufbau einer solchen Darstellung wird durch den Begriff des Booleschen Netzwerks präzisiert.

Definition 2.7. ***(Boolesches Netzwerk)***
Ein Boolesches Netzwerk ist ein Tripel (V, E, F), wobei $G := (V, E)$ ein gerichteter azyklischer Graph und F eine Menge Boolescher Funktionen ist, so dass gilt:

a) *Die Menge der Knoten ist wie folgt partitioniert: I ist die Menge der Quellen und wird als Menge der Eingänge bezeichnet. O ist die Menge der Senken und wird als Menge der Ausgänge bezeichnet. $S = V \setminus (I \cup O)$ wird als Menge der internen Signale bezeichnet. Die Knoten aus O haben genau einen Vorgänger.*

b) *Jedem Knoten $v \in S$ mit direkten Vorgängern $v_1, \ldots, v_n \in V$ wird eine n-stellige Funktion $f_v : \mathbb{B}^n \rightarrow \mathbb{B} \in F$ zugeordnet.*

Bemerkungen:
Wir betrachten ein Boolesches Netzwerk (V, E, F) mit einer Menge von Eingängen $I = \{i_1, \ldots, i_k\}$. Jedem Knoten $v \in V$ kann eine Boolesche Funktion $g_v : \mathbb{B}^k \rightarrow \mathbb{B}$, wie folgt, zugeordnet werden:

- Für jeden Eingang i_l $(l = 1 \ldots k)$ sei $g_{i_l}(x_1, \ldots, x_k) := x_l$.
- Für jeden internen Knoten v mit direkten Vorgängern $v_1, \ldots, v_n \in V$ sei $g_v := f_v(g_{v_1}, \ldots, g_{v_n})$.
- Für jeden Ausgang o mit direktem Vorgänger v_o sei $g_o := g_{v_o}$.

Ein Boolesches Netzwerk mit der Eingangsmenge $I = \{i_1, \ldots, i_k\}$ und Ausgangsmenge $O = \{o_1, ..., o_m\}$ ist also eine Darstellung der Booleschen Funktionen $g_{o_1}, \ldots, g_{o_m}$ oder der Bitvektorfunktion $g : \mathbb{B}^k \rightarrow \mathbb{B}^m$ mit
$g(x_1, \ldots, x_k) = (g_{o_1}(x_1, \ldots, x_k), \ldots, g_{o_m}(x_1, \ldots, x_k))$.
Beschränkt man sich bei der Menge F auf die Schaltfunktionen von Gattern, so spricht man auch von einer Gatternetzliste. In Abbildung 2.2 ist ein Beispiel für eine Gatternetzliste dargestellt.
Desweiteren lässt sich die Definition des Booleschen Netzwerks leicht auf die sogenannte Wortebene übertragen, indem man die Knoten $v \in V$ des Graphen mit Gewichten $w(v) \in \mathbb{N}$ versieht, die der Bitbreite eines internen Signals entsprechen. Die Knoten werden dann mit Bitvektorfunktionen markiert, die zu den Bitbreiten des jeweiligen Knotens und seiner Vorgänger passen.

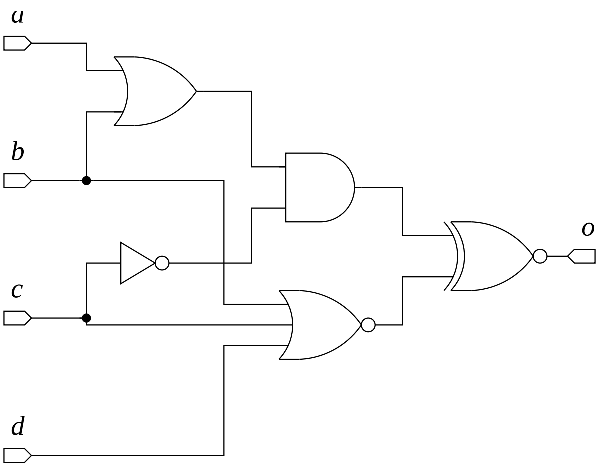


Abbildung 2.2: Gatternetzliste für $f(a,b,c,d) = ((a+b) \cdot \overline{b}) \oplus (\overline{a+b+d})$

Definition 2.8. ***(Bitvektornetzliste)***
Eine Netzliste aus Bitvektorfunktionen (Bitvektornetzliste) ist ein Tripel (V, E, F), wobei $G := (V, E)$ ein gerichteter azyklischer Graph und F eine Menge von Bitvektorfunktionen ist, so dass gilt:

a) Jedem Knoten $v \in V$ ist eine Bitbreite $w(v)$ zugeordnet.

b) Die Menge der Knoten ist wie folgt partitioniert: I ist die Menge der Quellen und wird als Menge der Eingänge bezeichnet. O ist die Menge der Senken und wird als Menge der Ausgänge bezeichnet. $S = V \setminus (I \cup O)$ wird als Menge der internen Signale bezeichnet. Die Knoten aus O haben genau einen Vorgänger.

c) Jedem Knoten $v \in S$ mit direkten Vorgängern $v_1, \ldots, v_n \in V$ wird eine Bitvektorfunktion

$$f_v : \mathbb{B}^k \to \mathbb{B}^{w(v)} \in F, k = \sum_{i=1\ldots n} w(v_i)$$

zugeordnet.

Bemerkung: Die Globalfunktion g_v eines Knotens einer Bitvektornetzliste erhält man völlig analog zur Globalfunktion eines booleschen Netzwerks.
Abschließend wollen wir ohne Anspruch auf Vollständigkeit eine Liste typischer in Bitvektornetzlisten verwendeter Funktionen angeben. Eine exakte Semantik dieser Funktionen findet man in [Bri03].

Bitvektorfunktion	Typ	Beschreibung
not	unär	bitweise Negation
minus	unär	Berechnung von $-x$
uand	unär	Konjunktion aller Bits des Vektors
uor	unär	Disjunktion aller Bits des Vektors
uxor	unär	Parität der Bits des Vektors
zeroextend	unär	Anhängen von führenden Nullen
signextend	unär	Vorzeichen-Erweiterung
and	binär	bitweise Konjunktion
or	binär	bitweise Disjunktion
xor	binär	bitweise Antivalenz
add	binär	Addition
mult	binär	Multiplikation
div	binär	Division (mit Rest)
mod	binär	Rest bei Division
ror	binär	Rechtsrotation
rol	binär	Linksrotation
shr	binär	logisches Rechtsschieben
shl	binär	logisches Linksschieben
eq	binär	Test auf Gleichheit
lt	binär	Vergleich bzgl. ”kleiner”Relation
concat	binär	Konkatenation zweier Bitvektoren
slice	binär	Ausschneiden eines Teilvektors ab einer bestimmten Position
ite	tertiar	If-Then-Else-Operator

Tabelle 2.3: Typische Bitvektorfunktionen

2.5 Erfüllbarkeitsprüfung (SAT)

Viele Probleme in der Entwurfsautomatisierung lassen sich auf das Erfüllbarkeitsproblem zurückführen, welches wie folgt definiert ist:

Definition 2.9. ***(Erfüllbarkeitsproblem)***
Sei $f : \mathbb{B}^n \to \mathbb{B}$ *eine Boolesche Funktion. Gibt es eine Belegung* $x \in \mathbb{B}^n$ *mit* $f(x) = 1$*?*

Ist die Funktion in KNF-Darstellung gegeben handelt es sich hierbei um eines der klassischen NP-vollständigen Probleme. Alle bekannten exakten Lösungsverfahren für diese Probleme haben exponentielles Worst-Case-Laufzeitverhalten. Allerdings hat es in den letzten Jahren enorme Performanzsteigerung bei SAT-Solvern gegeben [MMZ$^+$01, GN02, ES03]. Während die wesentlichen algorithmischen Ideen, wie nicht-chronologisches Backtracking durch intensive Konfliktanalyse, bereits in [MSS99] zu finden sind, hat sich gezeigt, dass die effiziente Implementierung dieser Algorithmen ebenfalls ein Forschungsgebiet ist, welches intensiv bearbeitet wurde und zu dramatischen Performanzsteigerungen geführt hat.

Für die Lösung des Erfüllbarkeitsproblems bietet sich die KNF-Darstellung der Booleschen Funktion an. Eine Belegung erfüllt eine Funktion in KNF genau dann, wenn sie jede ihrer Klauseln erfüllt. Damit kann lokal entschieden werden, ob eine unvollständige Belegung der Variablen der Funktion bereits im Widerspruch zu einer dieser Klauseln steht und damit nicht mehr zu einer Lösung vervollständigt werden kann. Moderne SAT-Solver arbeiten deshalb fast alle auf einer Funktionsdarstellung in konjunktiver Normalform. Aus einer Gatternetzliste kann eine solche KNF effizient erzeugt werden, wenn man die Einführung zusätzlicher Variablen für die internen Signale der Gatternetzliste erlaubt. Die KNF der gesamten Netzliste ergibt sich dann aus dem Produkt der KNFs für die einzelnen Gatter. Als Beispiel betrachten wir ein UND-Gatter mit der Schaltfunktion $o = a \cdot b$. Die KNF dieses Gatters besteht aus den Klauseln $(o + \overline{a} + \overline{b})$, $(\overline{o} + a)$ und $(\overline{o} + b)$. Andere Gatter können analog umgewandelt werden.

Um eine erfüllende Belegung der Variablen mit Booleschen Werten zu finden, führen moderne SAT-Solver eine Tiefensuche nach dem Davis-Putnam-Logeman-Loveland (DPLL) Algorithmus [DLL60] durch, wie er in Tabelle 2.4 dargestellt wird. Der SAT-Solver trifft dabei sukzessive Entscheidungen (*decide_next_branch()*) zur Belegung von Variablen und ermittelt nach jeder dieser Entscheidungen durch *Boolean Constraint Propagation (BCP),* also durch Implikationen, ob diese in Konflikt zu einer der vorhandenen Klauseln steht (*status* = *deduce()*). Entstehende Konflikte werden in einer Konfliktanalyse (*analyze_conflict()*) ausgewertet und es wird eine sogenannte Konfliktklausel berechnet. Dabei wird auch festgestellt, ob der gerade aufgetretene Konflikt tatsächlich von den zuletzt getroffenen Entscheidungen abhängt. Häufig ist dies nicht der Fall, und es können alle Entscheidungen bis zu einer bestimmten Ebene (*blevel*) des Entscheidungsbaums zurückgenommen werden. Diese Vorgehensweise wird auch *nicht-chronologisches Backtracking* genannt. Die generierte Konfliktklausel verhindert dann, dass dieser Teil des Suchraums erneut betreten werden kann.

```
dpll() {
  preprocess();
  while (true) {
    decide_next_branch();
    status = deduce();
    if (status == conflict) {
      blevel= analyze_conflict();
      if (blevel == 0)
        return UNSAT;
      else backtrack (blevel);
    }elseif (status == SAT)
      return SAT;
    else break;
  }
}
```

Tabelle 2.4: Pseudocode des DPLL Algorithmus

Entscheidend für die Performanz heutiger Solver sind Datenstrukturen und Algorithmen, die eine sehr schnelle Konstantenpropagierung erlauben. Hier haben sich so genannte *Lazy-Datastructures* als gewinnbringend herausgestellt. Moderne Solver verwenden ausnahmslos das *two-literals-watching* Schema. Dahinter steht die Überlegung, dass eine Klausel erst dann eine Konstantenpropagierung auslösen kann, wenn nur noch eine Variable der Klausel unbelegt ist. Deshalb genügt es, sich auf zwei unbelegte Literale pro Klausel zu konzentrieren und auf Wertezuweisungen für diese Literale zu reagieren. Wird eines dieser beiden Literale l im Verlauf der Suche erfüllt, so ist die ganze Klausel erfüllt und braucht damit für das aktuell untersuchte Teilproblem nicht mehr betrachtet werden. Nimmt l dagegen den Wert 0 an, so muss ein neues Literal zur Beobachtung ausgewählt werden. Sollte dies nicht mehr möglich sein, weil alle anderen Literale der Klausel bereits den Wert 0 haben, so kann daraus für das andere beobachtete Literal l' die Propagierung $l' = 1$ gefolgert werden.

Darüber hinaus sind sehr leistungsstarke Heuristiken zur Wahl der Entscheidungsvariablen und zur Berechnung einer guten Konfliktklausel entwickelt worden. Einen Überblick über diese Heuristiken, die in modernen SAT-Solvern verwendet werden, findet der interessierte Leser in [HS02].

Um sich nicht in einem schwierigen Teilproblem zu verlieren, hat sich der periodische Neustart des Solvers bewährt. Startet man die Suche von Zeit zu Zeit von neuem, ergibt sich die Möglichkeit, in ein völlig anderes Gebiet des Suchraums einzutauchen und dadurch das Problem eher zu lösen. Dabei profitiert man von den gelernten Konfliktklauseln der früheren Suchdurchgänge. Um Vollständigkeit zu garantieren, wird die Neustartperiode mit jedem Neustart vergrößert.

Als letzte Maßnahme zur Performanzsteigerung sei an dieser Stelle das Löschen gelernter Klauseln genannt. Empirische Untersuchungen zeigen, dass gelernte Klauseln in der Regel nur sehr lokal zur Steuerung der Suche nützlich sind. Die Performanz der Konstantenpropagierung nimmt aber mit der Anzahl der zu bearbeitenden Klauseln ab. Um hier eine Entlastung zu schaffen, werden, heuristisch gesteuert, periodisch einige Konfliktklauseln gelöscht, von denen man sich erhofft, dass sie für die weitere Suche irrelevant sind. Dies birgt natürlich das Risiko, das ein bestimmter Teil des Suchraums mehrfach besucht wird. Vollständigkeit wird wie bei den Neustarts durch Inkrementierung der Periode erreicht.

2.6 Wortebenen-Entscheidungsverfahren

In der Eigenschaftsprüfung liegen die Erfüllbarkeitsprobleme häufig in Form einer Bitvektornetzliste vor, für welche die Konstanz bestimmter Ausgänge bewiesen werden soll. Diese Darstellung ist in der Regel wesentlich kompakter als die entsprechende KNF. Es liegt also nahe, die Information auf der Wortebene einer solchen Darstellung auch für das Entscheidungsverfahren zu verwenden. So wurden z.B. eine Reihe von Entscheidungsdiagrammen auf der Wort-Ebene, wie *Binary Moment Diagrams (BMDs)* oder *Taylor Expansion Diagrams (TEDs)* vorgeschlagen [SBW98, HD99, CZKR02]. Man stößt jedoch bei der Generierung dieser Diagramme auf ähnliche Probleme wie bei ROBDDs. Insbesondere ist es trotz intensiver Forschung nicht gelungen, ein zuverlässiges und robustes Verfahren zur Synthese solcher Diagramme aus industriellen Schaltungsbeschreibungen zu entwickeln [SKKK04]. Aus diesem Grund haben sich diese Entscheidungsdiagramme in der Praxis nicht durchsetzen können.

Als Alternative wurde weiterhin vorgeschlagen, das Erfüllbarkeitsproblem mit Techniken des Integer Linear Programming (ILP) [BD02, FDK01, ZKC01] oder des Constraint Logic Programming (CLP) [ZCR01] zu bearbeiten. Eine angemessene Darstellung aller in diesem Zusammenhang wichtigen Techniken füllt ganze Bücher [BR85, Sch90, AMTW99, FA97, MTP94]. Diese Arbeit beschränkt sich deshalb auf die Darstellung der Modellierung eines SAT-Problems als ILP bzw. CLP.

2.6.1 Integer Linear Programming (ILP)

Das *Integer Linear Programming* Problem ist wie folgt definiert:

Definition 2.10. ***(ILP-Problem)***
Für eine rationale Matrix A und rationale Vektoren b und c
ist $max\{cx|Ax \leq b; x \in \mathbb{Z}^n\}$ zu bestimmen.

Zur Lösung solcher Probleme gibt es kommerzielle Werkzeuge wie z.B. CPLEX [CPL]. Aktuelle ILP-Solver führen im Kern eine LP-Relaxation durch. Dabei wird die Ganzzahligkeitsbedingung für die Variablen des IP vernachlässigt. Allerdings kann die LP-Relaxation durch sogenannte *cutting planes* verstärkt werden, die diese Bedingungen ausnutzen. Häufig reicht die LP-Relaxation aus, um ein Problem als unlösbar auszuweisen.

Bei der Modellierung eines Erfüllbarkeitsproblems auf der Wortebene als ILP reicht es aus, die Bitvektorfunktionen aus Tabelle 2.3 einzeln zu betrachten und durch eine Menge von lineare Constraints zu modellieren. Das Modell für das gesamte SAT-Problem ergibt sich dann als Konjunktion der Constraints für die einzelnen Bitvektorfunktionen. Allerdings hat man keine natürliche Zielfunktion, da alle Lösungen zum Beweis der Erfüllbarkeit gleichermaßen geeignet sind. Die Zielfunktion kann also beliebig gewählt werden.

In der folgenden Tabelle ist exemplarisch für einige Bitvektorfunktionen die Linearisierung angegeben, wie sie in [BD02, FDK01, ZKC01] vorgeschlagen wurde:

Bitvektorfunktion	Linearisierung
$r = add(a, b)$	$r + 2^n o = a + b$
$r = ite(s, a, b)$	$r - a - l * (1 - s) \leq 0$ $a - r - l * (1 - s) \leq 0$ $r - b - l * s \leq 0$ $b - r - l * s \leq 0$ $l \geq a$ $l \geq b$
$r = lt(a, b)$	$a - b - l * (1 - r) \leq -1$ $a - b + l * r \geq 0$ $l \geq a$ $l \geq b$
$r = and(a, b)$	$r_i - a_i \leq 0$ $r_i - b_i \leq 0$ $r_i - b_i - a_i + 1 \geq 0$

Tabelle 2.5: Linearisierung von Bitvektorfunktionen

Eine solche einfache Linearisierung reicht in der Praxis jedoch nicht aus. Moderne ILP-Solver wie CPLEX [CPL] arbeiten intern mit einer 32-Bit Datenrepräsentation. Numerische Eigenschaften der verwendeten Lösungsverfahren schränken allerdings den tatsächlich nutzbaren Bereich weiter ein. In [ZKC01] wurden 24 Bit als obere Schranke für den nutzbaren Wertebereich angegeben. Daher müssen Worte größerer Bitbreite zerlegt werden. Dies geschieht für eine Variable der Bitbreite $m + n$ durch einen Constraint der Form:

$$a = 2^n a_1 + a_2$$

mit den Randbedingungen

$$0 \leq a_1 < 2^m, 0 \leq a_2 < 2^n.$$

Die gerade beschriebene Modellierung von Bitvektorfunktionen im ILP muss dann auf die Teilworte der Variablen angewendet werden. Schwierig wird dies, wenn, wie z.B. bei der Addition, ein Teilwort des Ergebnisses von allen Teilworten der Operanden abhängt.

In diesem Fall müssen dann Überträge zwischen den Teilwortadditionen betrachtet und durch zusätzliche Variablen abgebildet werden.
Ein weiteres Problem stellt die Multiplikation dar, da diese keine lineare Operation ist. Die Multiplikation von Wortebenenvariablen ist deshalb im ILP nicht erlaubt. Bei der ILP-Modellierung muss daher eine Linearisierung durchgeführt werden, indem ein Operand in einzelne Bits zerlegt und die Multiplikation durch eine Folge von Additionen implementiert wird. Der ite-Operator wird dabei benutzt, um die einzelnen Summanden dieser Addition zu bestimmen. Für die Bitvektorfunktion $r = mult(a, b)$ entstehen dann die folgenden ILP-Constraints, wobei für die Variablen r, a, b jeweils Bitbreite n angenommen wird:

- $a = \sum_{i=0...n-1} 2^i a_i$
- $0 \leq a_i \leq 1 \forall_{i=0...n-1}$
- $p_i = ite(a_i, 2^i b, 0) \forall_{i=0...n-1}$
- $r = \sum_{i=0...n-1} p_i + 2^n s$

Damit ergeben sich für die Linearisierung einer Multiplikation $2+7n$ ILP-Constraints und $2n+1$ zusätzliche Variablen, die zur Hälfte nur Boolesche Werte annehmen dürfen. Haben beide Operanden die gleiche Bitbreite, so ist a priori auch nicht festgelegt, welcher der Operanden in seine Bits zerlegt werden soll. Damit ist klar, dass mit reinem ILP das Ziel nicht erreicht wird, das Verifikationsproblem für Arithmetik auf der reinen Wortebene zu lösen.

2.6.2 Constraint Logic Programming

Bei Constraint Logic Programming (CLP) handelt es sich um ein Lösungsverfahren basierend auf Methoden der logischen Programmierung.
Das wichtigste Arbeitspferd dieser Techniken ist die Propagierung von Wertebereichen für die beteiligten Variablen eines Problems. Lässt sich der Wertebereich einer Variablen explizit darstellen, so kann diese Propagierung exakt durchgeführt und damit der Suchraum effektiv beschnitten werden.
In der jüngeren Vergangenheit wurde viel Aufwand investiert, um CLP für praktische Probleme anwendbar zu machen [JM94]. Als Ergebnis stehen nun eine ganze Reihe CLP-Solver zur Verfügung, wobei GNU Prolog [DC00] zu den leistungsfähigsten zählt. Dieser Solver unterstützt eine ganze Palette von eingebetteten Prädikaten für das Schließen in endlichen Domänen. Die folgende Tabelle zeigt die Modellierung einiger Bitvektorfunktionen mit GNU-Prolog-Prädikaten.
Im Gegensatz zum ILP erlaubt CLP die direkte Modellierung der Multiplikation. Allerdings ist auch hier die nutzbare Bitbreite beschränkt, für GNU Prolog sind beispielsweise 28 Bit als maximale Bitbreite angegeben. Es müssen also wie beim ILP große Constraints zerlegt werden.

Bitvektorfunktion	Prädikat
$r = and(a, b)$	$a\# \wedge b\# <=> r$
$r = ite(s, a, b)$	$r\# = s * a + (1 - s) * b$
$r = lt(a, b)$	$r\# <=> a\# < b$
$r = add(a, b)$	$r\# = a + b$
$r = mult(a, b)$	$r\# = a * b$

Tabelle 2.6: CLP-Prädikate für Bitvektorfunktionen

Darüber hinaus werden in der Regel nur kleine Domänen mit bis zu 256 Elementen explizit dargestellt. Für solche Domänen greifen die Methoden von CLP, um den Suchraum eines Problems zu beschneiden. Für größere Domänen arbeiten die Solver mit Abschätzungen durch obere und untere Schranken. Dies führt in der Praxis dazu, dass der Solver große Teile der Wertebereiche enumerieren muss, um die Unerfüllbarkeit einer Instanz nachzuweisen. Auf einigen SAT-Instanzen [ZCR01] schneiden CLP-Solver dennoch erheblich besser ab als klassische SAT-Solver. Dieses positive Verhalten skaliert jedoch nicht bezüglich der Bitbreite der betrachteten arithmetischen Operationen. Tabelle 2.7 gibt die Laufzeiten von GNU-Prolog V.1.2.16 zum Beweis der folgenden Gleichungen an.

a) $xy = yx$

b) $x(y + z) = xy + yz$

c) $x * y = x_l * y_l + 2^{n/2} x_l * y_u + 2^{n/2} x_u * y_l + 2^n x_u * y_u$

Hierbei bezeichnen x_l und y_l das untere und x_u sowie y_u das obere Halbwort der jeweiligen Variable.

	CPU-Zeit(ms)		
Bit Breite	a)	b)	c)
8	38	16423	87
9	148	131551	
10	592	1052842	1554
11	2361	>3500000	
12	9435	>3500000	28988
13	37699	>3500000	
14	150221		463808

Tabelle 2.7: CLP-Laufzeiten für einige Wortebenen-Probleme

Es zeigt sich deutlich das exponentielles Wachstum der Laufzeiten, was auf die Enumeration aller Eingabewerte schließen lässt. Dieses Verhalten schließt eine Verwendung von reinem CLP zur Datenpfadverifikation von realistischen Designs mit Bitbreiten von bis zu 64 Bit aus.

2.7 Hybride Solver

Um sowohl von den Stärken von SAT auf der Bitebene, als auch von der Mächtigkeit der gerade beschriebenen Verfahren der Wortebene profitieren zu können, wurden in den letzten Jahren verstärkt Ansätze zur Integration von Konzepten aus SAT, ILP und CLP in sogenannte hybride Solver entwickelt. Einige dieser Ansätze sollen im Folgenden kurz betrachtet werden.

2.7.1 Pseudo-Boolesche Constraint Solver

Ein Ansatz, um Konzepte aus SAT und ILP zu kombinieren besteht, darin, das zu lösende Entscheidungsproblem durch Pseudo-Boolesche Constraints (PBC) zu kodieren. Effiziente Pseudo-Boolesche Constraint Solver wurden in [CK03, CK05, SS05] vorgestellt. Darüber hinaus wurde im Rahmen der SAT 2005-Konferenz eine erste Evaluierung für Pseudo-Boolesche Constraint Solver durchgeführt [MR].

Definition 2.11. ***(Pseudo-Boolesche-Constraints)***
Ein (linearer) Pseudo-Boolescher-Constraint ist eine Ungleichung der Form:

$$\sum a_i l_i \geq k, a_i, k \in \mathbb{Z}^+, l_i \in \{0, 1\}.$$

Das Beispiel von *Minisat+*, einer PB-Erweiterung von *Minisat* [ES03], zeigt die enge Verwandtschaft von SAT- und PB-Solvern. Dies liegt daran, dass Klauseln spezielle PB-Constraints mit $a_i = 1$ und $k = 1$ sind. Andererseits kann man auch PB-Constraints durch Klauseln darstellen. Dazu wird allerdings im schlimmsten Fall eine exponentielle Anzahl Klauseln benötigt, wie in [ARMS02] gezeigt wurde. Durch die Einführung zusätzlicher Variablen in der SAT-Instanz lässt sich dieses Verhalten allerdings vermeiden.

Bei der Implementierung von PB-Solvern werden verschiedene Strategien verfolgt. So gibt es Solver wie *Minisat+*, die das PBC-Problem durch Einführung zusätzlicher Variablen auf SAT abbilden. Auf der anderen Seite haben Solver wie *Galena* oder *Pueblo* spezielle Implementierungen der DPLL-Basisfunktionen zur Handhabung von PB-Constraints. Entscheidend für die Performanz dieser Solver ist die Strategie zur Steuerung des Lernens zusätzlicher Constraints.

Während allgemeine PB-Constraints die größte Ausdruckskraft haben und damit für die Beschneidung des Suchraums zunächst am effektivsten erscheinen, geht dieser Gewinn durch die erhöhten Verwaltungskosten für diese Constraints wieder verloren. Würde man sich beim Lernen hingegen auf Klauseln beschränken, so liegt der Gewinn lediglich in der kompakteren Darstellung der Probleminstanz. Als guter Kompromiss zwischen beiden Extremen wurden in [CK05] Kardinalitäts-Constraints, das sind Constraints der Form $\sum l_i \geq k$, identifiziert.

Eine andere Strategie wird in [SS05] verfolgt. Hier werden Klauseln und sogenannte starke PB-Constraints gelernt. Letztere sind dadurch charakterisiert, dass sie keiner Klausel entsprechen, zu der aktuellen Wertebelegung in Konflikt stehen und eine begrenzte Anzahl von Literalen mit Koeffizienten $a_i > 2$ haben.

Im Rahmen dieser Arbeit ist natürlich die Performanz dieser Solver auf nicht erfüllbaren Instanzen von besonderem Interesse. Bei der oben genannten Evaluierung [MR] waren die meisten nicht erfüllbaren Instanzen aus reinen Klauseln aufgebaut. Damit ist es auch nicht verwunderlich, dass *Minisat+* auf diesen Benchmarks am besten abgeschnitten hat. Es bleibt also abzuwarten, wie sich PB-Solver mit speziellen PB-Mechanismen gegenüber erweiterten SAT-Solvern entwickeln.

Ein weiteres ungeklärtes Problem für den Einsatz von PB-Solvern in der Eigenschaftsprüfung liegt in der Modellgenerierung. Es gibt derzeit keinen Ansatz, wie man Probleme der Eigenschaftsprüfung so auf PB-Constraints abbildet, dass das Potential der kompakteren Darstellung des Problems wirklich ausgenutzt wird.

2.7.2 Hybride Wortebenen-Solver

Abschließend werden in diesem Abschnitt einige in der Literatur vorgeschlagene Strategien zur Integration der in diesem Kapitel beschriebenen Entscheidungsverfahren diskutiert. Zunächst wären an dieser Stelle die Ansätze [HC01, ABC$^+$02] zu nennen, die verschiedene Entscheidungsverfahren sehr lose kombinieren. In [ABC$^+$02] ergeben sich ILPs in den Blättern einer DPLL-Suche, die dann mit einer Reihe verschieden mächtiger Solver bearbeitet werden. Dahinter steht die Hoffnung, die Unlösbarkeit eines Teilproblems bereits auf der Booleschen Ebene oder durch einen Solver niedriger Komplexität zu erkennen. In der Konfliktanalyse werden außer der Unlösbarkeit des Teilproblems und der beteiligten Constraints keine Informationen des ILPs verarbeitet.

Die Autoren von [HC01] kombinieren mit SAT verwandte Testgenerierungsmethoden (ATPG) mit CLP-Techniken. Dabei wird das ATPG-Werkzeug benutzt, um konsistente Belegungen für die Kontrolllogik eines Designs zu generieren. Dies führt zu Bedingungen an den Datenpfad, die dann mit Hilfe des Constraint-Solvers aufgelöst werden. Ist dies für ein Teilproblem nicht möglich, wird mit der nächsten Belegung des ATPG-Werkzeugs fortgefahren. Hier findet demnach ebenfalls kein detaillierter Informationsaustausch über Nichtlösungsgebiete beider Solver statt. Bei gültigen Eigenschaften führt dies zur Enumeration aller möglichen Belegungen für den Kontrollpfad des Designs und zur Lösung sehr vieler ähnlicher CLP-Probleme. Schwierig ist außerdem die geforderte klare Aufteilung des Hardwaredesigns in Datenpfad und Kontrollpfad, die in praktischen Designs häufig nicht ganz scharf vorzunehmen ist.

Die Erfahrung mit den gerade geschilderten Ansätzen zeigt die Notwendigkeit einer stärkeren Integration der Solver, die einen besseren Austausch von Information über Nichtlösungsgebiete zwischen den beteiligten Solvern gewährleisten. Vorschläge dazu findet man in [Ach05, PICW04].

Beide Ansätze integrieren sowohl SAT- und ILP- als auch CLP-Techniken zu einer einheitlichen Entscheidungsprozedur. Die Konstantenpropagierung wird in beiden Ansätzen durch spezialisierte Algorithmen zur Propagierung von Einschränkungen der Wertebereiche von Variablen verallgemeinert. Allerdings verfügt nur [Ach05] über Algorithmen zur Konfliktanalyse für diese verallgemeinerte Propagierung. In [PICW04] wird deshalb der Schnitt im Implikationsgraphen so gewählt, dass nur binäre Variablen darin vorkommen

und deshalb eine Konfliktklausel generiert werden kann.
In [PICW04] wird eine Fourier-Motzkin-Elimination als letztes Mittel eingesetzt, um inkonsistente Belegungen im Datenpfad zu erkennen, nachdem alle binären Variablen fixiert sind. Im Gegensatz dazu wird in [Ach05] die Auswertung einer LP-Relaxierung des Problems nach jedem Entscheidungsschritt des Algorithmus vorgenommen. Dazu wird der duale Simplex-Algorithmus verwendet, der eine effiziente erneute Lösung der LP-Relaxierung nach weiteren Wertebereichseinschränkungen unterstützt. Die globale Sicht der LP Relaxierung erlaubt es dem Solver, unlösbare Teilprobleme wesentlich früher zu erkennen, als dies mit CLP- und SAT-Techniken alleine möglich wäre. Darüber hinaus werden gelöste LPs zur Steuerung der weiteren Suche verwendet.
Alle in diesem Kapitel vorgestellten Ansätze zur Integration verschiedener Entscheidungsverfahren reduzieren gegenüber SAT die Anzahl der nötigen Entscheidungen zum Beweis oder der Widerlegung der Erfüllbarkeit einer Instanz. Dies wird allerdings mit einem erhöhten Aufwand bei jeder Entscheidung erkauft. Besonders in den Problemteilen der Bitebene zahlt sich dieser Aufwand häufig nicht aus, da nur unwesentlich mehr deduziert werden kann, als dies bei SAT-Solvern der Fall ist. Da Probleme aus der industriellen Eigenschaftsprüfung stets einen nicht unwesentlichen Anteil von Bitebenen-Constraints beinhalten, sind hybride Solver momentan als Alternative zu SAT noch nicht attraktiv.

Kapitel 3

Eigenschaftsprüfung

Ziel der Eigenschaftsprüfung für digitale Schaltungen ist es, das Vertrauen des Designers in seinen Entwurf zu stärken. Ein CAD-Werkzeug zur formalen Eigenschaftsprüfung muss es deshalb ermöglichen, bestimmte Aspekte des Verhaltens einer Schaltung übersichtlich und leicht verständlich zu beschreiben und automatisch gegen einen vorhandenen Schaltungsentwurf zu prüfen.

In diesem Kapitel führen wir deshalb die notwendigen Modelle ein, um das zeitliche Verhalten einer digitalen Schaltung beschreiben zu können. Während ein Hardwaredesign alle seine möglichen Verhaltensweisen implizit beschreibt, können sich Eigenschaften auf bestimmte Aspekte dieses Verhaltens konzentrieren und diese wesentlich kompakter darstellen. Eine ausreichend große Menge solcher Eigenschaften, welche das Verhalten des Designs komplett überdeckt, bildet damit eine orthogonale Sicht auf das Design.

Der zweite Teil dieses Kapitels ab Abschnitt 3.3 führt gängige Methoden zu automatischen Verifikation von Eigenschaften ein. Es wird deutlich, dass die Mächtigkeit der Eigenschaftssprache einen entscheidenden Einfluss auf die Berechnungskomplexität der entstehenden Verifikationsprobleme hat. Beide Faktoren korrelieren miteinander und beeinflussen die praktische Anwendbarkeit der Verfahren. Es gilt also, einen ausgewogenen Kompromiss zwischen maximaler Mächtigkeit der Sprache und niedriger Komplexität der Verifikationsprobleme zu finden. In der Praxis hat sich hier das so genannte Bounded Interval Model Checking durchgesetzt. Bei diesem Verfahren wird das Design ausschließlich in einem beschränkten Zeitintervall untersucht. Dies erlaubt sowohl die präzise Formulierung als auch die effiziente Verifikation vieler wichtiger Eigenschaften heutiger Entwürfe.

3.1 Modellierung sequentieller Systeme

Die mathematischen Modelle aus Kapitel 2 zur Beschreibung digitaler Schaltungen eignen sich nur zur Beschreibung rein kombinatorischer Schaltungen. Solche Schaltungen sind dadurch gekennzeichnet, dass die Ausgabe der Schaltung unmittelbar von den Eingaben abhängt. Bei sequentiellen Schaltungen ist dies in der Regel nicht der Fall. Diese

Schaltungen verfügen über Speicherelemente (z.B. Flipflops) und können darin Zustandsinformation ablegen. Im Allgemeinen hängt hier die Ausgabe sowohl von der aktuellen Eingabe als auch vom gespeicherten Zustand ab. Sequentielle Schaltungen werden deshalb üblicherweise als endliche Automaten modelliert. In dieser Arbeit verwenden wir dazu sogenannte Mealy-Automaten.

Definition 3.1. ***(Endlicher Automat)***
Ein endlicher (Mealy-) Automat M ist ein 6-Tupel $M = (I, S, \delta, S_0, O, \lambda)$ bestehend aus einem Eingabealphabet I, einer Zustandsmenge S, einer Zustandsübergangsfunktion $\delta : S \times I \longrightarrow S$, einer Menge von Anfangszuständen S_0, einem Ausgabealphabet O und einer Ausgabefunktion $\lambda : S \times I \longrightarrow O$.

Ein Automat wird oftmals durch seinen Zustandsübergangsgraphen beschrieben. Dabei entsprechen die Knoten den Zuständen und die Kanten den Zustandsübergängen. Die Kanten werden mit den zugehörigen Ein- und Ausgaben markiert. Abbildung 3.1 zeigt ein Beispiel für einen Zustandsübergangsgraphen eines Automaten.

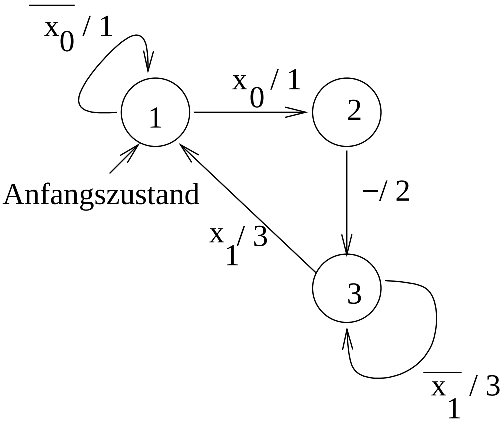


Abbildung 3.1: Beispiel eines Zustandsübergangsgraphen

Eine sequentielle Schaltung mit n Eingängen, m Ausgängen und k Flipflops kann durch einen Automaten $M = (\mathbb{B}^n, \mathbb{B}^k, f_S, S_0 \subseteq \mathbb{B}^k, \mathbb{B}^m, f_O)$ mit $f_S : \mathbb{B}^k \times \mathbb{B}^n \rightarrow \mathbb{B}^k$ und $f_O : \mathbb{B}^k \times \mathbb{B}^n \rightarrow \mathbb{B}^m$ modelliert werden. Im Folgenden verstehen wir unter einem *Hardwaredesign* stets einen solchen Automaten.
Wir gehen dabei davon aus, dass S_0 durch die charakteristische Funktion $\chi_{S_0} : \mathbb{B}^k \rightarrow \mathbb{B}$ spezifiziert wird. Diese Funktion nimmt genau dann den Wert $\chi_{S_0}(x) = 1$ an, wenn $x \in S_0$ gilt. Unter den *Registern* eines Designs verstehen wir im Folgenden die Komponentenfunktionen f_{S_i} der Zustandsübergangsfunktion f_S, d.h., wir identifizieren die Flipflops jeweils mit ihrer Ansteuerfunktion.
Sind die Funktionen f_S und f_O durch Bitvektornetzlisten beschrieben, so sprechen wir von einem Design auf *Register-Transfer (RT) Ebene*. Liegt dagegen jeweils nur eine Gatternetzliste vor, so sprechen wir von der Booleschen Ebene oder auch einem Bitebenen-Design.

Kodiert man die Zustände des Beispielautomaten aus Abbildung 3.1 durch ihre Binärwerte s_1, s_0, erhält man das Hardwaredesign $D = (\mathbb{B}^2, \mathbb{B}^2, f_S, \{(0,1)\}, \mathbb{B}^2, f_0)$ mit $f_0(s_1, s_0, x_1, x_0,) = (s_1, s_0)$ und $f_S(s_1, s_0, x_1, x_0) = (s_1\overline{s_0} + s_1\overline{x_1} + \overline{s_1}x_0, s_1 + \overline{x_0})$. Natürlich sind auch andere Kodierungen für diesen Automaten möglich. Neben der binären Kodierung wird beispielsweise die One-hot-Kodierung oft benutzt. Dabei wird jeder Zustand durch genau ein Flipflop implementiert. Ist das Flipflop auf den Wert 1 gesetzt, so befindet sich der Automat in dem entsprechenden Zustand. Dies impliziert natürlich, dass stets nur eines dieser Flipflops den Wert 1 annehmen kann.

In der formalen Verifikation interessiert uns nun das Ein-/Ausgabeverhalten eines Hardwaredesigns. Dieses wird durch die Menge aller möglichen Ein-/Ausgabesequenzen charakterisiert. Die Zustandsübergangsfunktion eines Automaten bildet jeweils einen Zustand und eine Eingabe auf einen Nachfolgezustand ab. Dieser Modellierung liegt also eine diskrete Zeitvorstellung zu Grunde. Der Automat bildet somit eine Folge von Eingaben auf eine Folge von Ausgaben ab. Eine solche Abbildung wird *Berechnung* genannt. Dies wird durch die folgende Definition 3.2 formalisiert:

Definition 3.2. ***(Berechnungen)***
Sei $M = (I, S, \delta, S_0, O, \lambda)$ ein endlicher Automat. Eine Folge $(i_t, o_t, s_t)_{t\in\mathbb{N}_0}$ mit $i_t \in I, o_t \in O, s_t \in S$ heißt genau dann eine Berechnung von M, wenn gilt:

a) $\forall t \in \mathbb{N}_0 : o_t = \lambda(i_t, s_t)$

b) $\forall t \in \mathbb{N}_0 : s_{t+1} = \delta(i_t, s_t)$

Die Folge $(i_t)_{t\in\mathbb{N}_0}$ wird dabei Eingabefolge, die Folge $(o_t)_{t\in\mathbb{N}_0}$ Ausgabefolge und die Folge $(s_t)_{t\in\mathbb{N}_0}$ Zustandsfolge der Berechnung genannt. Ein konkretes Glied i_t, o_t bzw. s_t dieser Folgen wird Eingabe, Ausgabe bzw. Zustand zum Zeitpunkt t genannt.

Wir unterscheiden zwischen gültigen und ungültigen Berechnungen. Eine Berechnung $(i_t, o_t, s_t)_{t\in\mathbb{N}_0}$ ist gültig, wenn sie in einem Anfangszustand beginnt, d.h. $s_0 \in S_0$. Die Zustandsfolge einer gültigen Berechnung unseres Beispielautomaten M aus Abbildung 3.1 könnte wie folgt beginnen: $s = (1, 2, 3, 1, 1, 2, 3, 3 \dots)$. Hingegen wäre eine Berechnung $(2, 3, 1 \dots)$ nicht gültig, da 2 kein Anfangszustand ist. Die Zustandsfolge $(1, 2, 1 \dots)$ kommt in keiner Berechnung vor, da der Zustand 1 kein Nachfolger des Zustands 2 ist.

Berechnungen erlauben es, Zusammenhänge zwischen Zustandsmengen eines Designs herzustellen. Beispielsweise lässt sich die Frage stellen, ob ein Zustand des Automaten, von einer bestimmten Zustandsmenge ausgehend, erreicht werden kann. Wir definieren deshalb die Menge der erreichbaren Zustände wie folgt:

Definition 3.3. ***(Erreichbarkeit)***
Sei $M = (I, S, \delta, S_0, O, \lambda)$ ein endlicher Automat. Ein Zustand $s \in S$ ist aus einer Zustandsmenge $X \subseteq S$ erreichbar, wenn es eine Berechnung $(i_t, o_t, s_t)_{t\in\mathbb{N}_0}$ und einen Zeitpunkt $t \in \mathbb{N}_0$ gibt, so dass $s_0 \in X$ und $s_t = s$ gilt.
Die Menge $Reach_M(X)$ sei die Menge aller Zustände $s \in S$, die aus X erreichbar sind. $Reach_M := Reach_M(S_0)$ heißt Menge der erreichbaren Zustände des Automaten.

Bemerkungen: $Reach_M$ enthält also alle Zustände, die in gültigen Berechnungen vorkommen. Es gilt offensichtlich $Reach_M(Reach_M(X)) = Reach_M(X)$. Eine Menge von Zuständen Y mit $Reach_M(Y) = Y$ nennt man bezüglich Erreichbarkeit abgeschlossen.
Für den Automaten M aus Abbildung 3.1 gilt beispielsweise: $Reach_M = \{1, 2, 3\}$.
Das Erreichbarkeitsproblem für Graphen und damit auch das für Automaten liegt in der Komplexitätsklasse NLOGSPACE $\subseteq$ P. Dabei wird die Größe des Problems in der Anzahl der Knoten des Graphen bzw. Zustände des Automaten gemessen. Jedoch wächst die Anzahl der möglichen Zustände eines Designs exponentiell mit der Anzahl der Flipflops. Beim Erreichbarkeitsproblem für Schaltungen wird nun gerade diese Anzahl als Maß für die Problemgröße herangezogen. Man kann zeigen, dass das Problem, die Menge der erreichbaren Zustände eines Hardwaredesigns zu berechnen, PSPACE-vollständig [ASB93, DLMM02] ist. Es gibt trotz dieser hohen Berechnungskomplexität eine Flut von Arbeiten [JPHS91, CCQ96, CCL$^+$96, QCC$^+$96, CCLQ97, CHM$^+$94, RS95, SWWK04], die verschiedene exakte und approximative Verfahren vorstellen, um die Erreichbarkeitsanalyse für den industriellen Einsatz zu ertüchtigen. Trotzdem ist eine Erreichbarkeitsanalyse bis heute nur für relative kleine Schaltungen praktikabel.
Nicht nur für die Erreichbarkeitsanalyse von Interesse sind die Begriffe der sequentiellen Tiefe und des Diameters eines Automaten. Für jeden erreichbaren Zustand s gibt es einen kürzesten Pfad $s_0, \ldots, s_{n-1} = s$ im Zustandsübergangsgraphen, der in einem Anfangszustand $s_0 \in S_0$ beginnt. Die Länge n dieses Pfades wird sequentielle Tiefe $depth(s)$ des Zustands s genannt. Die sequentielle Tiefe des Automaten ist die maximale sequentielle Tiefe aller erreichbaren Zustände $m = max\{depth(s) | s \in Reach_M\}$. Der Diameter eines Automaten ist die maximale Länge n eines zyklusfreien Pfades $s_0, \ldots, s_{n-1}$ im Zustandsübergangsgraphen, wobei s_0 nicht notwendig ein Anfangszustand ist.
Die Frage nach der Erreichbarkeit von Zuständen des Systems ist allerdings nicht die einzige für die Eigenschaftsprüfung relevante Problemstellung. Häufig ist man vielmehr an temporalen Beziehungen zwischen Zustandsmengen eines sequentiellen Systems interessiert. Als Grundlage zur Spezifikation solcher Zusammenhänge eignet sich das sogenannte Kripke-Modell, welches eng mit dem vorgestellten Automatenmodell verwandt ist. Das Kripke-Modell erlaubt die Markierung von Zuständen mit atomaren Formeln zur Auszeichnung von Zustandsmengen mit bestimmten Eigenschaften.

Definition 3.4. ***(Kripke-Modell)***
Ein Kripke-Modell ist ein Quintupel $K = (S, S_0, R, A, l)$. Dabei ist:

- S *eine endliche Menge von Zuständen,*
- $S_0 \subseteq S$ *eine Menge von Anfangszuständen,*
- $R \subseteq S \times S$ *Übergangsrelation des Modells,*
- A *eine Menge atomarer Formeln,*
- l *eine Abbildung* $l : A \rightarrow P(S)$.

Eine atomare Formel $p \in A$ *gilt in einem Zustand* $s \in S$ *genau dann, wenn* $s \in l(a)$ *gilt.*

Ein Hardwaredesign $M = (\mathbb{B}^n, \mathbb{B}^k, f_S, S'_0 \subseteq \mathbb{B}^k, \mathbb{B}^m, f_O)$ kann in ein Kripke-Modell umgewandelt werden. Die Zustandsmenge dieses Kripke-Modells besteht aus Eingaben, Ausgaben und Zuständen des Designs. Außerdem ist die Übergangsrelation verträglich mit der Zustandsübergangsfunktion und der Ausgabefunktion des Design.
Formal definiert man $K_M = (S, S_0, R, A, l)$ mit:

- $S := \mathbb{B}^{n+k+m}$
- $S_0 := \{(i, s, o) | i \in \mathbb{B}^n, s \in S'_0, o = f_O(i, s)\}$
- $R := \{(s, s') \in S \times S | f_S(s_0, \ldots, s_{n+k-1}) = (s'_n, \ldots, s'_{n+k-1})$
  $\cdot f_O(s'_0, \ldots, s'_{n+k-1}) = (s'_{n+k}, \ldots, s'_{n+k+m-1})\}$
- $A := \{S \to \mathbb{B}\}$
- $l(f) := f^{-1}(1) := \{s \in S : f(s) = 1\}$.

Da aus dem Kontext stets klar ist, ob ein Hardwaredesign M oder sein zugehöriges Kripke-Modell K_M gemeint ist, benutzen wir beide Begriffe synonym.

3.2 Eigenschaftssprachen

Ein Hardwaredesign spezifiziert implizit alle seine möglichen Berechnungen. Um zu überprüfen, ob das Design genau das gewünschte Verhalten zeigt, müsste man für jede dieser Berechnungen einzeln überprüfen, ob diese auch gewünscht ist. Dies ist die Idee der erschöpfenden Simulation.
Natürlich ist eine solche erschöpfende Simulation praktisch undurchführbar. In der Eigenschaftsprüfung werden deshalb die gewünschten Berechnungen eines Designs durch eine Menge temporaler Formeln beschrieben. Jede dieser Formeln wird dann eine Eigenschaft genannt. Die Eigenschaften können mit einem sogenannten Eigenschaftsprüfer auf Gültigkeit für ein Design getestet werden. Der Eigenschaftsprüfer erzeugt als Ausgabe entweder eine Widerlegungssequenz oder einen Beweis für die Gültigkeit der Eigenschaft. Unter einer Widerlegungssequenz verstehen wir dabei eine gültige Berechnung, die in einem Zustand endet, in dem die Eigenschaft nicht gilt. Eine ausreichende Menge von Eigenschaften legt das Verhalten eines Automaten explizit fest, wohingegen der Automat nur eine implizite Beschreibung seines eigenen Verhaltens ist. Bei den zu spezifizierenden Eigenschaften unterscheidet man im Wesentlichen zwischen Sicherheits- und Lebendigkeitseigenschaften. Während Eigenschaften der ersten Kategorie fordern, dass ein bestimmtes (Fehler-)Verhalten niemals auftritt, fordern Lebendigkeitseigenschaften, dass ein bestimmtes (Wunsch-)Verhalten immer wieder, unendlich oft, eintritt.
Biere et al. haben in [BAS02] gezeigt, dass sich Lebendigkeitseigenschaften auf Sicherheitseigenschaften abbilden lassen. Damit ist gerechtfertigt, dass sich diese Arbeit auf Sicherheitseigenschaften konzentriert.

3.2.1 Temporale Logik

Im einfachsten Fall lässt sich eine Eigenschaft des Designs als Boolescher Ausdruck über Signalen der Zustandsübergangsfunktion beschreiben.

Beispiel 3.1. *Ein Automat M mit drei Zuständen soll durch ein Design mit drei Flipflops implementiert werden. Die Zustände sollen one-hot kodiert sein, d.h. es hat stets genau eines dieser Flipflops den Wert 1. Wir haben also eine Zustandsübergangsfunktion $f_S : \mathbb{B}^3 \times I \to \mathbb{B}^3$. Die one-hot Kodierung kann durch die Boolesche Formel*

$$(s'_1 \oplus s'_2 \oplus s'_3) \cdot \overline{s'_1 \cdot s'_2 \cdot s'_3}$$

für die Nachfolgezustände des Automaten spezifiziert werden. Überprüft werden muss, ob die Anfangszustände und die Bildzustände der Zustandsübergangsfunktion f_S dieser Formel genügen. Wir erhalten die Formel:

$$\forall (s'_1, s'_2, s'_3) \in \{f_S(x) | x \in \mathbb{B}^3 \times I\} \cup S_0 : (s'_1 \oplus s'_2 \oplus s'_3) \cdot \overline{s'_1 \cdot s'_2 \cdot s'_3}.$$

Bei genauerer Betrachtung dieses Beispiels stellt sich allerdings heraus, dass die Spezifikation der Eigenschaft etwas zu eng ist. Es genügt bereits zu fordern, dass die folgende Formel erfüllt ist:

$$\forall (s'_1, s'_2, s'_3) \in \{f_S(x) | x \in \{(1,0,0),(0,1,0),(0,0,1)\} \times I\} \cup S_0 : (s'_1 \oplus s'_2 \oplus s'_3) \cdot \overline{s'_1 \cdot s'_2 \cdot s'_3}.$$

Wir haben also die uns bekannte Erreichbarkeitsinformation für den Automaten genutzt, um die Eigenschaft zu präzisieren. Viele wichtige Eigenschaften eines Designs lassen sich leider nicht auf die gerade geschilderte Weise modellieren. Auch hierzu sei ein Beispiel angeführt:

Beispiel 3.2. *Für eine Implementierung eines Automaten M mit 2^n möglichen Ausgaben soll überprüft werden, dass eine bestimmte Ausgabe $(o_1, \ldots, o_n)$ niemals erfolgt.*

Um diese Eigenschaft nachzuweisen, muss überprüft werden, ob $(o_1, \ldots, o_n)$ in einer gültigen Berechnung vorkommt. Um dies formal notieren zu können, wird die Aussagenlogik um temporale Operatoren erweitert. Die Semantik dieser Operatoren wird gewöhnlich über ein Kripke-Modells definiert.

Bei den Operatoren einer temporalen Logik unterscheidet man zwischen Zustandsoperatoren, die Zustandsformeln erzeugen, und Pfadoperatoren, welche Pfadformeln erzeugen. Für eine Zustandsformel kann man in jedem Zustand entscheiden, ob sie gilt oder nicht. Die Gültigkeit einer Pfadformel kann nur durch Betrachtung von Berechnungen untersucht werden. Formeln der Eigenschaftssprache CTL* werden rekursiv aus den atomaren Formeln eines Kripke-Modell aufgebaut. Dabei können die Operationen der Booleschen Algebra, die Pfadoperatoren F, G, X, U und die Zustandsoperatoren A und E verwendet werden. Zunächst wird nun die syntaktische Struktur von CTL*-Formeln formal definiert.

Definition 3.5. ***(Syntax von CTL*-Formeln)***
Jede atomare Formel ist eine Zustandsformel. Sind p und q Formeln, so sind auch $p + q$, $\overline{p}$ und $p \cdot q$ Formeln. Zustandsformeln sind auch Pfadformeln. Für Pfadformeln p und q sind Fp, Gp, Xp und pUq Pfadformeln. Für Pfadformeln p und q sind Ap und Ep Zustandsformeln.

Als zweiter Schritt folgt nun die Festlegung der Bedeutung diese Formeln.

Definition 3.6. ***(Semantik von CTL*-Formeln)***
Sei $K = (S, S_0, R, A, l)$ ein Kripke-Modell, $s \in S$ ein Zustand und $\pi = s_0, s_1, ...$ ein (unendlicher) Pfad in K. Ferner sei p eine atomare Formel, g_1, g_2 Pfadformeln und f_1, f_2 Zustandsformeln. Für Zustandsformeln bedeutet die Notation $K, s \models p$, dass die Formel p im Zustand s des Kripke-Modells K gilt. Die Notation $K, \pi \models p$, bedeutet, dass die Pfadformel p entlang dem Pfad π im Kripke-Modell K gültig ist. Zuletzt sei π^i der Suffix $(s_i, s_{i+1}, ...)$ ab Position i von π.

- $K, s \models p :\Leftrightarrow p \in l(s)$.
- $K, s \models \overline{f_1} :\Leftrightarrow \overline{K, s \models f_1}$
- $K, s \models f_1 \cdot f_2 :\Leftrightarrow (K, s \models f_1) \cdot (K, s \models f_2)$
- $K, s \models f_1 + f_2 :\Leftrightarrow (K, s \models f_1) + (K, s \models f_2)$
- $K, s \models Eg_1 :\Leftrightarrow$ *es gibt einen Pfad π mit Anfangszustand s und $K, \pi \models g_1$.*
- $K, s \models Ag_1 :\Leftrightarrow$ *für jeden Pfad π mit Anfangszustand s gilt: $K, \pi \models g_1$.*
- $K, \pi \models f_1 :\Leftrightarrow K, s \models f_1$ *wobei s der erste Zustand von π ist.*
- $K, \pi \models \overline{g_1} :\Leftrightarrow \overline{K, \pi \models g_1}$
- $K, \pi \models g_1 \cdot g_2 :\Leftrightarrow (K, \pi \models g_1) \cdot (K, \pi \models g_2)$
- $K, \pi \models g_1 + g_2 :\Leftrightarrow (K, \pi \models g_1) + (K, \pi \models g_2)$
- $K, \pi \models Xg_1 :\Leftrightarrow K, \pi^1 \models g_1$
- $K, \pi \models Fg_1 :\Leftrightarrow$ *es existiert ein $k \geq 0$ mit $K, \pi^k \models g_1$*
- $K, \pi \models Gg_1 :\Leftrightarrow$ *für alle $k \geq 0$ gilt: $K, \pi^k \models g_1$*
- $K, \pi \models g_1 U g_2 :\Leftrightarrow$ *es existiert ein $k \geq 0$ mit $K, \pi^k \models g_2$ und $K, \pi^j \models g_1$ für $0 \leq j < k$.*

Eine Formel f gilt für K genau dann, wenn für alle Anfangszustände $s_0 \in S_0$ gilt:

$$K, s_0 \models f.$$

Mit anderen Worten: Fp gilt, wenn die Eigenschaft p irgendwann in der Zukunft gilt. Gp gilt, wenn die Eigenschaft p ab jetzt bis in alle Zukunft gilt. Xp gilt, wenn die Eigenschaft p im nächsten Schritt gilt und $p\,U q$ gilt, falls p solange gilt, bis q gilt. Die Pfadquantifizierungen A und E besagen, dass eine Formel für alle Pfade bzw. für mindestens einen Pfad in die Zukunft gilt.
Die Eigenschaft aus Beispiel 3.2 lässt sich nun in CTL* durch die Formel

$$AG(o \neq (o_1, \ldots, o_n))$$

notieren.
Interessante Teilmengen von CTL* erhält man durch folgende Einschränkungen:

a) CTL (Computation Tree Logic) erlaubt Pfadoperatoren nur in direkter Verbindung mit Pfadquantoren A oder E.

b) LTL (Linear Temporal Logic) erlaubt nur Formeln der Form Af, wobei in der Pfadformel f nur atomare Formeln als Zustandsformeln erlaubt sind.

c) ACTL ist CTL in positiver Normalform, d.h. der Operator E ist nicht erlaubt.

d) Man kann temporale Logiken angeben, bei denen die Länge der betrachteten Pfade beschränkt ist. Man spricht dann von einer *Bounded Temporal Logic (BTL)*.

e) Anstatt die Gültigkeit einer BTL-Formel ausgehend vom Anfangszustand zu fordern, fordert man die Gültigkeit ausgehend von einem beliebigen Zustand. Solche Eigenschaften werden wir im Abschnitt 3.3.3 *Intervall-Eigenschaft* nennen. Sprachen dieser Klasse nennen wir im Folgenden *Interval Temporal Logic (ITL)*.

Für eine detaillierte Einführung in temporale Logik sei der interessierte Leser auf [CGP99] verwiesen.

3.3 Verifikationsmethoden

Um zu überprüfen, ob eine Formel in temporaler Logik für ein gegebenes Kripke-Modell gültig ist, existieren vollautomatische Werkzeuge [CCGR99, Hol97, McM]. Dabei hat die Wahl der temporalen Logik entscheidenden Einfluss auf die Berechnungskomplexität der zu verwendenden Prüfalgorithmen. In [ASB93, DLMM02] wurde gezeigt, dass die Überprüfung von CTL- bzw. LTL-Formeln zur Klasse der PSPACE-vollständigen Probleme gehört. Dagegen können BTL- und ITL-Formeln auf Erfüllbarkeitsprobleme (SAT) abgebildet werden, und SAT ist ein klassisches NP-vollständiges Problem. Es wird vermutet, dass sich diese beiden Klassen unterscheiden. In der Praxis spiegelt sich der Komplexitätsunterschied unmittelbar in der Größe der handhabbaren Systeme wieder. Dieser Abschnitt befasst sich zunächst mit der Reduktion des CTL-Model-Checking-Problems auf das Erreichbarkeitsproblem. Dabei wird insbesondere auf die mögliche Effizienzsteigerung durch symbolische BDD-basierte Repräsentation von Zustandsmengen hingewiesen. Anschließend wird die Konstruktion von Bounded-(Interval)-Model-Checking-Problemen erläutert und deren Reduktion auf Erfüllbarkeitsprobleme aufgezeigt.

3.3.1 Bildberechnung

Beim klassischen CTL- bzw. LTL-Model Checking muss nach Definition 3.6 festgestellt werden, ob die Menge der Anfangszustände eines Kripke-Modells in der Menge der Zustände enthalten ist, für die eine CTL*-Formel f gilt. In diesem Abschnitt wird deshalb dargestellt, wie diese zweite Menge effizient berechnet werden kann. Eine CTL-Formel wird dazu von innen nach außen ausgewertet. Man geht von den Zustandsmengen aus, die im Kripke-Modell den atomaren Formeln aus A zugeordnet sind. Es genügt, sich auf Formeln der Form $\overline{f_1}$, $f_1 + f_2$, EXf_1, $E[f_1Uf_2]$ und EGf_1 zu beschränken. Alle anderen Formeln lassen sich auf diese Formeln zurückführen.
Die Formeln $\overline{f_1}$ und $f_1 + f_2$ lassen sich leicht durch die Komplementbildung bzw. Vereinigung von Zustandsmengen berechnen. Entscheidend für die Auswertung der Formeln $E[f_1Uf_2]$ bzw. EGf_1 ist, dass sich die Zustandsmengen, in denen die Formel gilt, in beiden Fällen durch eine Fixpunktiteration berechnen lassen. In beiden Iterationen wird dabei die Auswertung einer Formel der Form EXf benutzt. Dazu muss das Urbild der Menge aller Zustände berechnet werden, in denen f gilt. Die Bild- bzw. Urbildberechnung erweist sich als Kernoperation zur Auswertung von CTL-Formeln. Unter Bild und Urbild versteht man die Menge aller Zustände, die von einer Zustandsmenge Z durch einmalige Anwendung der Übergangsrelation R erreicht werden, d.h.:

$$Bild(Z, R) := \{s' : \exists s \in Z : (s, s') \in R\}$$

$$Urbild(Z, R) := Bild^{-1}(Z, R) := \{s : \exists s' \in Z : (s, s') \in R\}.$$

Es stellt sich die Frage, wie diese Mengen effizient berechnet und dargestellt werden können. Eine bahnbrechende Idee in diesem Zusammenhang hatte McMillan [BCMD90, BCL$^+$94], als er vorschlug, Mengen von Zuständen durch ihre charakteristische Funktion, und diese wiederum als BDD darzustellen. Die Bildberechnung kann dann durch BDD-Operationen (UND-Verknüpfung und existenzielle Quantifizierung) ausgeführt werden. Gleichzeitig sind BDDs nützlich für die Fixpunkterkennung, da die Vergleichsoperation für BDDs extrem effizient ist. Da BDDs symbolische Darstellungen für Zustandsmengen sind, wird dieses Verfahren allgemein als symbolische Eigenschaftsprüfung (engl. Symbolic Model Checking (SMC)) bezeichnet.
Die symbolische Eigenschaftsprüfung leidet allerdings an der sogenannten *Zustandsexplosion*. Die Anzahl der Zustände wächst exponentiell mit der Anzahl der Variablen eines Systems. Dies führt in der Praxis häufig zu exponentiell großen BDDs. Selbst wenn der BDD am Ende der Fixpunktiteration sehr klein ist, zum Beispiel wenn alle 2^n Zustände des Systems erreichbar sind, kann jede mögliche Boolesche Funktion als Zwischenergebnis auftreten. Es muss also fast immer mit dem worst-case eines exponentiell großen BDDs gerechnet werden. Damit ist das Verfahren nur für sehr kleine Designs anwendbar. Eine ausführliche Darstellung der symbolischen Eigenschaftsprüfung findet der Leser in [McM93].

3.3.2 Bounded Model Checking (BMC)

Um unvorhersehbare Laufzeiten des SMC zu vermeiden und um die Nutzung anderer Funktionsdarstellungen als BDDs zu ermöglichen, liegt es nahe, die Anzahl der Iterationen l in der Fixpunktiteration zu beschränken. Dies ist die Kernidee des von Biere et al. [BCC$^+$99] vorgeschlagenen Bounded Model Checking Verfahrens (BMC). Mit diesem Verfahren können BTL-Formeln auf Gültigkeit für ein Modell überprüft werden. Wird innerhalb von l Iterationen ein Gegenbeispiel bzw. ein Zeuge (für Lebendigkeit) gefunden, oder der Fixpunkt erreicht, so ist damit auch die entsprechende unbeschränkte Formel widerlegt bzw. bewiesen. Lediglich in dem Fall, dass weder ein Gegenbeispiel noch ein Fixpunkt erreicht wird, macht das Verfahren keine Aussage. Die Erhöhung der Schranke ist dann allerdings noch eine Option. Ist die Schranke größer als die sequentielle Tiefe des Designs, ist BMC vollständig.

Ein wichtige Eigenschaft des BMC-Ansatzes, die dazu führte, dass erstmals Verifikationsprobleme von realistischer industrieller Größe behandelt werden konnten, besteht darin, dass das zu lösende Verifikationsproblem auf ein Erfüllbarkeitsproblem (SAT) für Boolesche Funktionen abgebildet werden kann.

Wir rekapitulieren hier exemplarisch die Konstruktion dieses SAT-Problems für Sicherheitseigenschaften der Form AGp. Lebendigkeitseigenschaften können ähnlich übersetzt werden. Zunächst führen wir Boolesche Variablen $s_0, s_1, \cdots, s_l$ ein, die den Zuständen zum Zeitpunkt $t = 0, \ldots, l$ entsprechen. Die charakteristische Funktion χ_R der Übergangsrelation R wird nun wie folgt abgerollt:

$$f_{R,l}(s_0, \ldots s_l) := \chi_{S_0}(s_0) \cdot \chi_R(s_0, s_1) \cdot \ \ldots \ \cdot \chi_R(s_{l-1}, s_l).$$

Hier bezeichnet χ_{S_0} die charakteristische Funktion der Menge der Anfangszustände des Designs. Die Funktion $f_{R,l}$ nimmt genau dann den Wert 1 an, wenn die Zustände $s_0, \ldots, s_l$ in einer gültigen Berechnung auftreten.

Analog zur Schaltung kann auch die Eigenschaft abgerollt werden. Wir erhalten:

$$p_l = p(s_0) \cdot \ \ldots \ \cdot p(s_l).$$

Eine Widerlegungssequenz der Länge l für die Eigenschaft $AG(p)$ muss die Funktion $bmc(AGp) := f_{R,l} \cdot \overline{p_l}$ erfüllen. Diese Übersetzung in ein SAT-Problem lässt neben den bereits erwähnten BDDs natürlich auch SAT-Techniken als Beweistechnik für BMC zu.

Abschließend sei noch erwähnt, dass ein Hardwaredesign $M = (\mathbb{B}^n, \mathbb{B}^k, f_S, S_0' \subseteq \mathbb{B}^k, \mathbb{B}^m, f_O)$ für BMC nicht unbedingt vorher in sein Kripke-Modell überführt werden muss. Es genügt, die Zustandsübergangsfunktion f_S sowie die Ausgabefunktion f_O abzurollen und um die Eigenschaft zu ergänzen, um ein äquivalentes SAT-Problem zu erzeugen. Die abgerollte Zustandsübergangsfunktion wird iteratives Schaltungsmodell genannt. Dies ist leicht einzusehen, wenn man sich verdeutlicht, dass die Konstruktion des Kripke-Modells für ein Design keinerlei Einschränkungen für die Eingaben des Design enthält. In Abbildung 3.2 ist ein solches iteratives Schaltungsmodell für vier Takte dargestellt, wobei auch die Ausgabefunktion f_O in jedem Takt dargestellt ist.

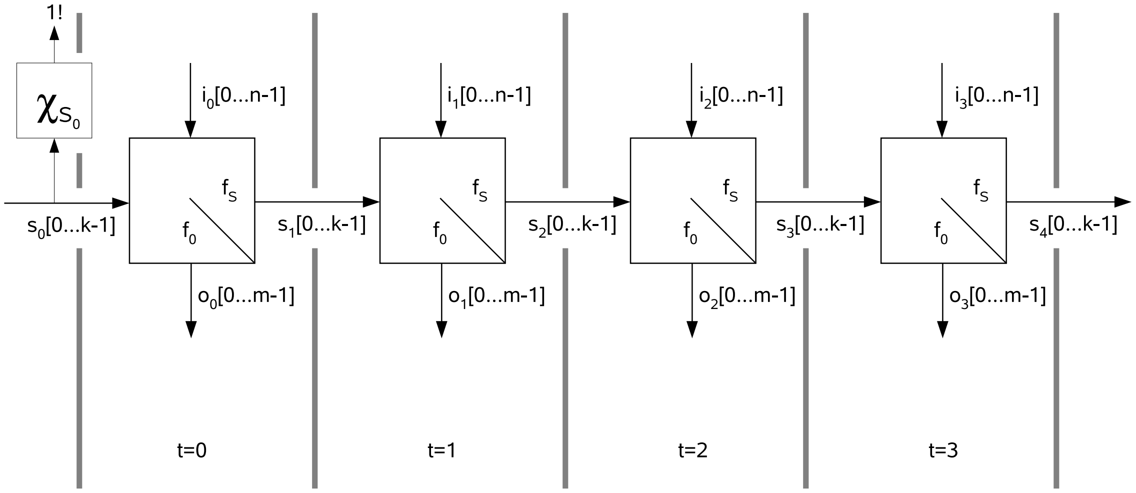


Abbildung 3.2: Iteratives Schaltungsmodell über vier Takte

3.3.3 Bounded Interval Model Checking (BIMC)

Bei allen bisherigen Verfahren zur Eigenschaftsprüfung wurde die Gültigkeit einer Eigenschaft stets für gültige Berechnungen eines Designs überprüft. Beim Bounded Model Checking führt dies dazu, dass das Verfahren prinzipiell nur Aussagen über die ersten l Takte der Schaltung macht. Bei Schaltungen mit großer sequentieller Tiefe müssten daher sehr lange Zeitfenster untersucht werden, um zumindest einen Großteil des möglichen Verhaltens der Schaltung innerhalb des beschränkten Zeitfensters beobachten zu können. Fordert man nun zusätzlich, dass die Eigenschaft auch in allen ungültigen Berechnungen gilt, gelangt man zum sogenannten Bounded Intervall Model Checking (BIMC). Die abgerollte Übergangsrelation lautet in diesem Fall:

$$f'_{R,l}(s_0,\dots s_l) := \chi_R(s_0,s_1)\cdot \;\dots\; \cdot \chi_R(s_{l-1},s_l),$$

und das zu lösende Erfüllbarkeitsproblem lautet:

$$bimc(AGp) := f'_{R,l}\cdot \overline{p_l}.$$

Ein iteratives Schaltungsmodell, wie es bei BIMC verwendet wird, ist in Abbildung 3.3 dargestellt. Im Folgenden bezeichnen wir mit dem iterativen Schaltungsmodell stets die BIMC-Variante.

Die Erfüllbarkeit von $bimc(AGp)$ ist nun ein hinreichendes aber nicht mehr unbedingt notwendiges Kriterium für die Gültigkeit der CTL-Formel AGp. Bei einem Gegenbeispiel muss überprüft werden, ob der Zustand s_0, mit dem dieses Gegenbeispiel beginnt, tatsächlich erreichbar ist. Ist dies nicht der Fall, so kann dieses Gegenbeispiel im laufenden System nicht auftreten. Man bezeichnet solche ungültigen Gegenbeispiele auch als *False-Negative*. Insofern ist BIMC ein orthogonaler Ansatz zum BMC. Es sei daran erinnert, dass ein BMC-Gegenbeispiel die CTL-Formel widerlegt, ein BMC-Beweis aber noch nicht hinreichend für die Allgemeingültigkeit der Formel ist.

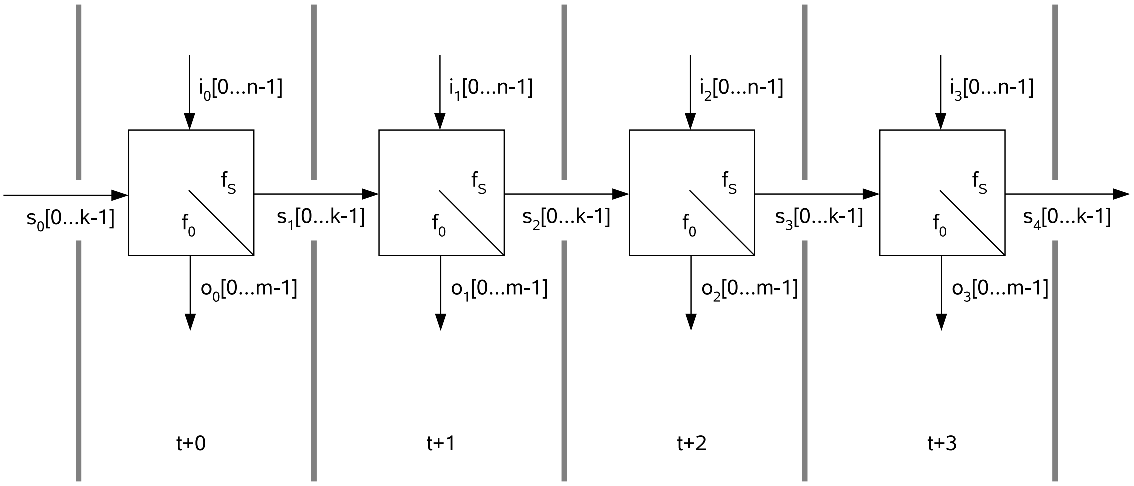


Abbildung 3.3: Iteratives Schaltungsmodell für BIMC über vier Takte

Es hat sich als praktikabel erwiesen, die Funktion p_l aus der BIMC-Instanz nicht durch Abrollen einer CTL-Formel zu ermitteln, sondern vom Anwender direkt als sogenannte beschränkte Intervall-Eigenschaft (engl. Bounded-Interval-Property) spezifizieren zu lassen. In Definition 3.7 wird dies festgehalten.

Definition 3.7. ***(Intervall-Eigenschaft)***
Sei $D = (I, S, f_S, F_0, O, f_O)$ ein Hardwaredesign. Eine Intervall-Eigenschaft der Länge $l \in \mathbb{N}$ ist eine Boolesche Funktion

$$p : I^{l+1} \times S^{l+2} \times O^{l+1} \rightarrow \mathbb{B}.$$

Konvention: Oftmals wird eine Intervall-Eigenschaft p durch einen Term für den Funktionswert $p(i_0, \ldots, i_l, s_0, \ldots, s_{l+1}, o_0, \ldots, o_l)$ in den Variablen $i_0, \ldots, i_l, s_0, \ldots, s_{l+1}, o_0, \ldots, o_l$ spezifiziert. Um in der Notation anzudeuten, dass eine gültige Intervall-Eigenschaft zu jedem Zeitpunkt t jeder Berechnung des Designs gilt, sagen wir für $j = 1, \ldots, l$:
i_j, s_j und o_j sind Eingang, Zustand und Ausgang zum Zeitpunkt $t + j$ und schreiben $i@(t + j)$, $s@(t + j)$ und $o@(t + j)$.
Eine Intervall-Eigenschaft p der Länge l ist genau dann gültig, wenn p, durch das iterative Schaltungsmodell $f_{R,l}$ beschränkt, stets den Wert 1 annimmt, d.h. $p(f_{R,l}) = 1$ gilt. Das vollständige Entscheidungsproblem für eine Intervall-Eigenschaft ist in Abbildung 3.4 dargestellt. Eine Belegung x der Eingänge $s_0, i_0, i_1, \ldots, i_{l-1}$ des iterativen Schaltungsmodells, für die $p(f_{R,l}(x)) = 0$ gilt, genügt deshalb zur Widerlegung der Eigenschaft.

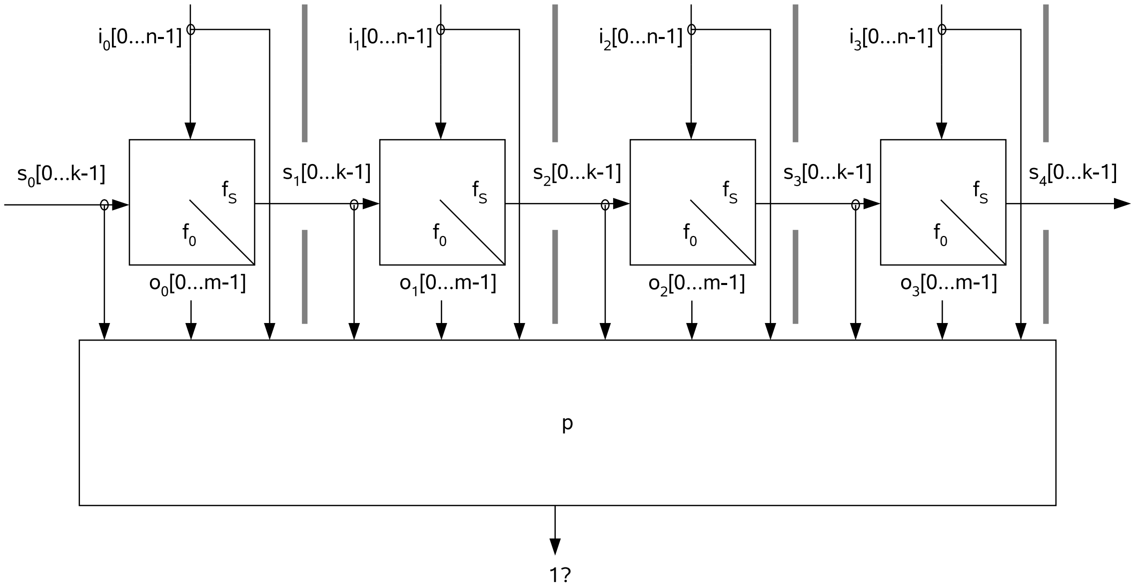


Abbildung 3.4: BIMC Problem über vier Takte

Beispiel 3.3. *Für unser Beispiel M aus Abbildung 3.1 wollen wir spezifizieren, dass man zum Zeitpunkt $t+3$ im Zustand $(1,1)$ ist, falls zu keinem Zeitpunkt $t+k$ mit $k=0\ldots 2$ eine Eingabe $x_1=1$ anliegt und zusätzlich im Intervall $[t+0,t+1]$ einmal die Eingabe $x_0=1$ erfolgt ist.*
Eine entsprechende Intervall-Eigenschaft lautet:

$$p = \overline{x_1@(t+0)+x_1@(t+1)+x_1@(t+2)} \cdot (x_0@(t+0)+x_0@(t+1))$$

$$\Rightarrow (s_1@(t+3)\cdot s_0@(t+3))$$

Dieses einfache Beispiel hat bereits eine typische Struktur für Eigenschaften, die bei der Verifikation eines Designs überprüft werden. Es sollen die Verpflichtungen (engl. commitments) rechts vom Implikationspfeil unter der Voraussetzung überprüft werden, dass die Umgebung des Designs bestimmte Annahmen (engl. assumptions) links vom Implikationspfeil erfüllt. Häufig werden dabei auch Einschränkungen bezüglich der Zustände des Designs gemacht, in denen die Eigenschaft gelten soll. Durch diese Maßnahme können False-Negatives vermieden werden. Benutzt man als Voraussetzung $s_0\in S_0$, so kann man mit BIMC das entsprechende BMC-Problem lösen. Auf diese Weise haben wir damit das BMC-Problem als spezielles BIMC-Problem dargestellt.

3.3.4 Temporale Induktion

Im letzten Abschnitt haben wir gesehen, dass BMC und BIMC zwei orthogonale Verfahren zur Eigenschaftsprüfung in beschränkten Zeitintervallen sind. Indem man beide Verfahren kombiniert, gelangt man zu den induktionsbasierten Verfahren. Solche Verfahren sind BIMC überlegen, wenn es darum geht, Eigenschaften nachzuweisen, die nur für erreichbare Zustände gelten. Die Induktion übernimmt hier in gewisser Weise die Analyse der möglichen False-Negatives des BIMC-Verfahrens.

Dieser Abschnitt gibt deshalb einen Überblick auf induktionsbasierte Verfahren zur Eigenschaftsprüfung, wie sie in [SSS00, ABE00] vorgeschlagen werden. Wir betrachten dabei, ohne Beschränkung der Allgemeinheit, nur Eigenschaften, die sich über Zustandsmengen charakterisieren lassen. Eine beliebige Intervall-Eigenschaft p der Länge n kann mit der Menge $P \subseteq S$ aller Zustände identifiziert werden, für die es keine Widerlegungssequenz $((i_l, s_l, o_l) : l = 0 \ldots n-1)$ mit $s_0 \in P$ gibt.

Für ein Design $D = (I, S, \delta, S_0, O, \lambda)$ möchte man nun überprüfen, ob eine Eigenschaft P in allen erreichbaren Zuständen gilt, d.h. ob $Reach_D \subseteq P$ gilt. In diesem Fall wird das Design als P-sicher bezeichnet. Die Eigenschaft P wird im Folgenden mit der Menge aller Zustände $s \in S$ identifiziert, in denen P gilt. Die Zustände aus P werden ebenfalls P-sicher genannt. Ein Pfad ist P-sicher, wenn jeder Zustand des Pfades in P liegt. Ist aus dem Kontext klar, um welche Eigenschaft es sich handelt, werden Automaten und Zustände sicher genannt. Das folgende Lemma nennt eine hinreichende Bedingung, unter der ein Automat P-sicher ist.

Lemma 3.1. *Ein endlicher Automat ist P-sicher, wenn die folgenden Bedingungen gelten:*

- *Alle Anfangszustände sind P-sicher, also $S_0 \subseteq P$.*
- *Die Nachfolger eines P-sicheren Zustand $s \in S$ sind auch P-sicher, d.h. $s \in P \Rightarrow \forall i \in I : \delta(s, i) \in P$.*

Eine Eigenschaft P wird Invariante genannt, wenn die zweite Bedingung des Lemmas zutrifft. Eine Invariante, die alle Anfangszustände enthält, ist damit eine gültige Eigenschaft. Umgekehrt ist aber nicht jede gültige Eigenschaft eine Invariante.

Das Zustandsdiagramm aus Abbildung 3.5 veranschaulicht dieses Problem und zeigt damit die Unzulänglichkeit dieses Lemmas für die Eigenschaftsprüfung. Die gefärbten Zustände stellen dabei die sicheren Zustände dar. Da der Nachfolger des sicheren Zustands 011 nicht sicher ist, ist die zweite Bedingung von Lemma 3.1 verletzt. Trotzdem ist der Automat offensichtlich sicher.

Dieses Problem kann behoben werden, indem ein längerer Induktionsanfang gewählt wird. Dies wird im nächsten Lemma formuliert.

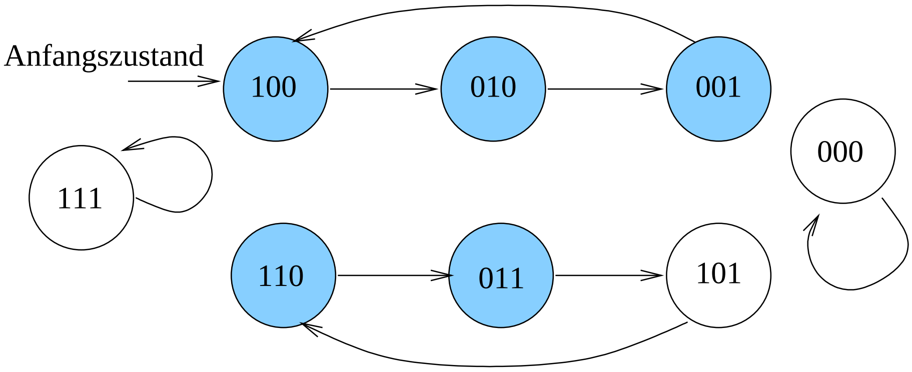


Abbildung 3.5: Zustandsdiagramm eines 3 Bit Ringzählers

Lemma 3.2. *Ein endlicher Automat M ist P-sicher, wenn eine Induktionstiefe $k \geq 0$ existiert, so dass die folgenden Bedingungen gelten:*

1.) *Für alle Pfade $s_0, s_1, ..., s_k$ von einem Anfangszustand $s_0 \in S_0$ im Zustandsübergangsgraphen von M gilt P in allen Zuständen s_i.*

2.) *Gilt P für alle Zustände eines beliebigen Pfades $s_0, ..., s_k$ im Zustandsübergangsgraphen, dann sind auch alle Nachfolger von s_k P-sicher, d.h.*
$\forall i \in I : \delta(s_k, i) \in P$.

Beweis der Lemmata 3.1 und 3.2:
Lemma 3.1 folgt aus Lemma 3.2. Wir nehmen deshalb an, dass die Bedingungen aus Lemma 3.2 gelten und der Automat unsicher ist. Da es sich um einen endlichen Automaten handelt, gibt es deshalb eine endliche Widerlegungssequenz w für P mit den Zuständen $s_0, \ldots, s_n$. Nach Bedingung 1 muss $n > k$ gelten, wobei die Zustände $s_0, \ldots, s_k$ alle sicher sind. $n - k$-malige Anwendung von Bedingung 2 liefert die Sicherheit der Zustände $s_{k+1}, \ldots, s_n$. Also ist w keine Widerlegungssequenz und unsere ursprüngliche Annahme ad absurdum geführt. Damit gilt das Lemma 3.2. □
Mit der Induktionstiefe $k = 2$ lässt sich die Eigenschaft aus Abbildung 3.5 nun zeigen. Allerdings ist dieses Kriterium immer noch nicht ausreichend, wie das Beispiel aus Abbildung 3.6 zeigt.
Wieder ist der Automat eindeutig sicher. Durch die Selbstkante in Zustand 110 scheitert der Induktionsschluss allerdings für jede Induktionstiefe $k > 0$. Um dieses Problem zu lösen, muss das Kriterium weiter verschärft werden. Da das Problem offensichtlich durch Zyklen ausgelöst wird, liegt es nahe, diese im Induktionsschluss von Lemma 3.2 zu verbieten.

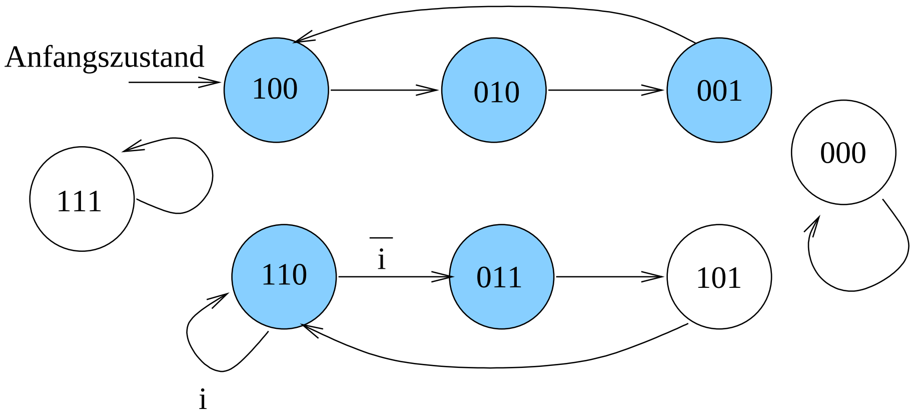


Abbildung 3.6: Zustandsdiagramm mit nicht erreichbarem P-sicherem Zyklus

Lemma 3.3. *Ein endlicher Automat M ist P-sicher, wenn eine Induktionstiefe $k \geq 0$ existiert, so dass die folgenden Bedingungen gelten:*

1.) Für alle Pfade $s_0, s_1, ..., s_k$ von einem Anfangszustand $s_0 \in S_0$ im Zustandsübergangsgraphen von M gilt P in allen Zuständen s_i.

2.) Gilt P für alle Zustände eines Pfades $s_0, ..., s_k$ im Zustandsübergangsgraphen mit paarweise verschiedenen Zuständen s_i, dann sind alle Nachfolger von s_k P-sicher, d.h. $\forall i \in I : \delta(s_k, i) \in P$.

Beweis für Lemma 3.3:
Da man aus einer Widerlegungssequenz stets alle Zyklen entfernen kann, um eine kürzere und zyklenfreie Widerlegungssequenz zu erhalten, können wir im Beweis zu Lemma 3.2 auch eine azyklische Widerlegungssequenz fordern. Der Rest der Argumentation erfolgt dann analog Lemma 3.2. □
Mit diesem Kriterium kann man jetzt die Sicherheit eines Automaten in jedem Fall zeigen. Dazu muss k nur groß genug gewählt werden. Dies ist die Aussage von Lemma 3.4.

Lemma 3.4. *Für jeden P-sicheren Automaten M gibt es eine Induktionstiefe $k \geq 0$, so dass die Bedingungen aus Lemma 3.3 gelten.*

Beweis Lemma 3.4:
Wähle $k = |S|$. Bedingung 1 ist erfüllt, da M nach Voraussetzung sicher ist. Bedingung 2 ist trivialerweise erfüllt, weil ein Pfad der Länge $|S|+1$ mindestens einen Zustand doppelt enthält. □
Natürlich ist eine Induktionstiefe $k = |S|$ viel zu groß, um den Beweis einer Sicherheitseigenschaft effizient durchführen zu können.

Zur Verifikation einer Sicherheitseigenschaft wird deshalb in der Regel eine Suche nach einer Induktionstiefe k durchgeführt, für die entweder beide Bedingungen aus Lemma 3.3 gelten oder die erste Bedingung verletzt ist. Im ersten Fall ist die Eigenschaft bewiesen, im zweiten widerlegt. Ein Algorithmus dazu ist in Tabelle 3.1 angegeben. Die Funktionen *induction_base* und *induction_step* testen dabei die erste bzw. die zweite Bedingung des Lemmas. Der Algorithmus lässt sich durch die Wahl der Startwertes *start_k* für k und die Strategie zur Erhöhung von k in der Funktion *increase* parametrisieren.
Der Algorithmus terminiert bei gültigen Eigenschaften auf Grund von Lemma 3.4 und bei ungültigen Eigenschaften, sobald k die Länge des kürzesten Gegegenbeispiels erreicht. Liefert der Algorithmus den Wert true, so sind beide Bedingungen von Lemma 3.3 erfüllt und die Eigenschaft gilt. Andernfalls ist die erste Bedingung dieses Lemmas nicht erfüllt und damit die Eigenschaft widerlegt. Also ist der Algorithmus korrekt.

```
induction(P)
{
   k := start_k;
   while (induction_base(P, k))
   {
      if (induction_step(P, k))
         return true;
      else
         increase(k);
   }
   return false;
}
```

Tabelle 3.1: Algorithmus zur induktionsbasierten Eigenschaftsprüfung

In der Praxis ist solch ein Algorithmus allerdings nur für kleine Induktionstiefen effizient. Dies liegt in erster Linie am Wachstum der zu lösenden SAT-Instanzen. Für die Induktionstiefe k enthält die SAT-Instanz $k + 2$ Kopien der Zustandsübergangsfunktion und eine quadratische Anzahl von Vergleichern. Dieses Wachstum ist kritisch, da heutige SAT-Solver eine Laufzeit haben, die exponentiell mit der Größe der Instanzen wächst. Darüber hinaus erzeugen die Vergleiche zwischen den Zustandsvariablen global rekonvergente Signale, die sich ebenfalls negativ auf die Performanz des SAT-Solvers auswirken.

Kapitel 4

Induktionsbasierte Eigenschaftsprüfung

Induktionsbasierte Verfahren zur Eigenschaftsprüfung [ABE00, SSS00] sind nur dann effizient, wenn die benötigte Induktionstiefe klein bleibt. Hier ist es Stand der Technik, dass der Anwender seine Eigenschaften mit zusätzlichen Bedingungen ausstattet, deren Gültigkeit er für das Design annimmt. Im günstigsten Fall enthalten die Zusatzbedingungen genug Erreichbarkeitsinformation, dass die neue Eigenschaft ausschließlich in erreichbaren Zuständen gilt, und mit Induktionstiefe $k = 0$ bewiesen werden kann. Zumindest erhofft man sich, dass einzelne nicht erreichbare Zustände aus langen P-sicheren Pfaden die zusätzlichen Bedingungen nicht erfüllen. Dadurch wird dann die Induktionstiefe wenigstens reduziert.

Die manuelle Bestimmung dieser Erreichbarkeitsinformation ist im Allgemeinen ein sehr mühsamer Prozess. Bisher gibt es keine Verfahren, die diesen Schritt automatisieren. Lediglich für den Fall des induktionsbasierten Äquivalenzvergleichs wurden in [BC00] Invarianten generiert, die die Induktionstiefe klein halten. Jedoch lässt sich diese Methode nicht direkt auf die Eigenschaftsprüfung übertragen.

Im Rahmen dieses Kapitels wird untersucht, welche Maßnahmen bei der Modellgenerierung im Front-End des Eigenschaftsprüfers getroffen werden können, um die automatische Generierung von Invarianten zu unterstützen und damit die manuelle Interaktion des Benutzers mit dem Werkzeug zu vermeiden.

Die Zustandskodierung von Design-Komponenten wird dabei als Freiheitsgrad identifiziert um geeignete Modelle für den Induktionsansatz zu erzeugen. Es ist festzustellen, dass diese Kodierung unabhängig von der späteren Implementierung gewählt werden kann, da Eigenschaften des Entwurfs verifiziert werden.

Die korrekte Implementierung durch ein Synthese-Werkzeug wird später mittels Äquivalenzvergleich sichergestellt. Ziel muss es daher sein, die Zustandskodierung für die Eigenschaftsprüfung so zu wählen, dass die Generierung von Invarianten unterstützt wird. Hierbei kann unter anderem auf RT-Informationen zurückgegriffen werden. Des Weiteren kann die Information über die Zustandskodierung unmittelbar als Invariante verwendet werden.

Die hier gewählte Zustandskodierung erlaubt es, mit Hilfe der *strukturellen Automatentraversierung* [SK97, van98, WSK02, SWWK04], eine Überapproximation, d.h. eine Obermenge, der Menge der erreichbaren Zustände zu berechnen und als Invariante für den Induktionbeweis zu benutzen.

4.1 Zustandskodierung für Automatentraversierung

Die Zustandkodierung eines Designs kann die Performanz eines Verfahrens zur Erreichbarkeitsanalyse beeinflussen. So wurde im Kontext der *symbolischen Automatentraversierung* in [KB01] gezeigt, dass Retiming einen großen Einfluss auf die Größe der entstehenden BDDs zur Repräsentation der Menge der erreichten Zustände hat. Durch Verschieben interner Register des Designs kann deren Anzahl und damit die erwartete Größe der BDDs reduziert werden. Es ist klar, dass diese Form des Retiming die Zustandskodierung massiv verändert.

In dieser Arbeit wird nun die Zustandskodierung so gewählt, dass die Induktionstiefe k des Induktionsbeweises reduziert wird. Dazu verwenden wir Informationen aus der RT-Beschreibung und zusätzliche Erreichbarkeitsinformation, die wir mit einer *strukturellen Automatentraversierung* [SK97, van98, WSK02, SWWK04] berechnen. Letztere liefert nur dann befriedigende Ergebnisse, wenn die Zustandskodierung geeignet gewählt wird.

Als erstes soll der Zusammenhang zwischen Erreichbarkeit und Induktionstiefe analysiert werden. Interessanterweise hängt die minimale Induktionstiefe zum Beweis einer Eigenschaft P bei einem P-sicheren Automaten M nicht von der sequentiellen Tiefe des Designs ab. Es sind vielmehr die nicht erreichbaren Zustände, die die Induktionstiefe beeinflussen. So lässt sich z.B. der Extremfall $P = Reach_M$ stets mit Induktionstiefe $k = 0$ beweisen. Eine genaue Charakterisierung dieser Induktionstiefe liefert das folgende Theorem.

Satz 4.1. *Für jeden P-sicheren Automaten M entspricht die minimale Induktionstiefe k der Länge des längsten Pfades $p_1, ..., p_k$ mit folgenden Eigenschaften:*

- $p_i \neq p_j$, *für* $i \neq j$
- $p_i \in P$, *für* $i \in \{1...k\}$
- p_k *hat einen Nachfolger* $p_{k+1} = \delta(i, p_k)$ *mit* $p_{k+1} \notin P$.

Beweis zu Theorem 4.1:

Für die angegebene Induktionstiefe sind beide Kriterien des Lemma 3.3 erfüllt. Das erste Kriterium folgt aus der Sicherheit des Automaten. Die zweite Bedingung ist erfüllt, da sonst der o.g. Pfad nicht die maximale Länge hätte. Also ist k tatsächlich eine Induktionstiefe, für welche die Bedingungen aus Lemma 3.3 gelten. Für $k' < k$ ist nun $p_{k-k'}, ..., p_k$ ein Gegenbeispiel für die zweite Bedingung aus Lemma 3.3. Also ist k minimal. □

Nach Theorem 4.1 hängt die Induktionstiefe k also nur von P-sicheren Zuständen ab, die nicht erreichbar sind. Gibt es wenige unerreichbare Zustände, die der Eigenschaft genügen, kann man eine kleine Induktionstiefe erwarten. Daher macht es Sinn, eine Eigenschaft durch Erreichbarkeitsinformation zu verstärken. Dies wird bisher dem Benutzer überlassen. Wir werden nun für die beiden häufigsten Zustandskodierungen angeben, wie diese Information automatisch generiert werden kann.

4.1.1 Binäre Kodierung

Bei der binären Kodierung eines Automaten mit n Zuständen wird ein Register $R[0, ..., m]$ mit $m = \lceil log_2(n) \rceil$ angelegt und die Zustände durch die Binärdarstellung von $0, ..., n-1$ repräsentiert. Die Anzahl der Zustände ist auf RT-Ebene in der Regel bekannt. Die damit verstärkte Eigenschaft lautet deshalb:

$$P' = P \wedge (n > \sum_{i=0}^{m} 2^i R[i]). \tag{4.1}$$

Besteht das Design nur aus einem einzigen Automaten, so wird P' nur in erreichbaren Zuständen gelten, es sei denn, der Designer hat explizit nicht erreichbare Zustände deklariert. Auf jeden Fall gilt die Eigenschaft P' nicht mehr in Zuständen, die entstehen, weil die Anzahl der Zustände keine Zweierpotenz ist und daher das Register nicht voll genutzt wird. In vielen praktisch relevanten Fällen wird diese Verstärkung also die Induktionstiefe auf $k = 0$ reduzieren. Hat der Designer doch einzelne unerreichbare Zustände deklariert, so kann man trotzdem eine Reduktion der Induktionstiefe erwarten.
Wir wenden uns nun dem wichtigeren Fall zu, dass das Design mehrere Automaten enthält, die von einander abhängen. Die einfachste Form der Automatenkopplung erhält man durch Verwendung gemeinsamer Eingänge. Abbildung 4.1 zeigt zwei solche Automaten. Diese Automaten haben unabhängige Eingänge x_1 und x_2 und eine Menge gemeinsamer Eingänge x. Formal definieren wir gekoppelte Automaten wie folgt:

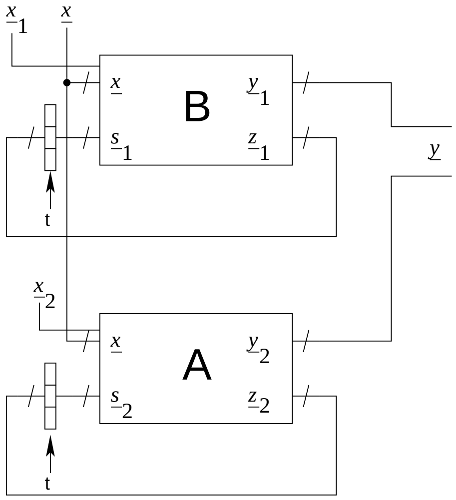


Abbildung 4.1: Gekoppelte Automaten mit unabhängigen Eingängen

Definition 4.1. *Seien* A, B *endliche Automaten mit:*

- $A = (X_2 \times X, S_A, \delta_A, S_{A_0}, O_A, \lambda_A)$ *und*
- $B = (X_1 \times X, S_B, \delta_B, S_{B_0}, O_B, \lambda_B)$.

Ferner sei $M = (I, S, \delta, S_0, O, \lambda)$ *ein Automat mit:*

- $I = X_1 \times X_2 \times X$*,*
- $S = S_A \times S_B$*,*
- $\delta((s_A, s_B), (x_1, x_2, x))$
  $= (\delta_A(s_A, (x_2, x)), \delta_B(s_B, (x_1, x)))$
- $S_0 = S_{A_0} \times S_{B_0}$
- $O = O_A \times O_B$
- $\lambda((s_A, s_B), (x_1, x_2, x))$
  $= (\lambda_A(s_A, (x_2, x)), \lambda_B(s_B, (x_1, x)))$.

Dann heißt M partiell gekoppelter Automat aus A *und* B.

Diese Definition erlaubt zwei Extremfälle. Ist $X = \emptyset$, so sind die Automaten unabhängig und alle Zustände des Produktraums erreichbar. Genauer gilt: $Reach_M = Reach_A \times Reach_B$. In diesem Fall zahlt sich die Verstärkung einer Eigenschaft mit den Schranken aus Gleichung (4.1) aus.

Im anderen Extremfall gibt es keine unabhängigen Eingänge, d.h., $X_1 = X_2 = \emptyset$. Dann ist M eine Produktmaschine wie beim Äquivalenzvergleich. Allerdings können wir bei der Eigenschaftsprüfung nicht davon ausgehen, dass die beteiligten Automaten irgendwelche strukturellen Ähnlichkeiten aufweisen, wie dies beim Äquivalenzvergleich ausgenutzt wird.

An dieser Stelle können wir also lediglich festhalten, dass $Reach_M \subseteq Reach_A \times Reach_B$ gilt. Gleichung (4.1) reicht also im Allgemeinen nicht zur vollständigen Charakterisierung von $Reach_M$ aus. Damit können die Induktionstiefen beliebig groß werden. Weitere Möglichkeiten der Automatenkopplung entstehen, wenn man Ausgänge eines Automaten als Eingänge eines anderen Automaten verwendet. Hierbei muss beachtet werden, dass durch diese Verschaltung keine rein kombinatorischen Rückkopplungen entstehen dürfen. Eine detaillierte Einführung in kommunizierende Automaten findet sich in [Kur94].

4.1.2 One-hot-Kodierung

Um die Komplexität moderner Anwendungen, z.B. in der Telekommunikation, bewältigen zu können, enthalten Designs häufig eine große Anzahl miteinander kommunizierender Automaten. Dabei sind die einzelnen Automaten relativ klein, damit der Designer ihre Funktion gut überblicken kann. In diesem Fall kann man sich eine One-hot-Kodierung für die Komponenten des Designs leisten. Der Vorteil dieser Kodierung besteht darhin, dass jeder Zustand durch ein explizites Register dargestellt wird.

Definition 4.2. *(One-hot-Kodierung)*
Ein Design $M = (\mathbb{B}^n, \mathbb{B}^k, f_S, S_0 \subseteq \mathbb{B}^k, \mathbb{B}^m, f_O)$ *ist one-hot kodiert genau dann, wenn* $Reach_M \subseteq \{(\delta_{i,j} | i = 1 \ldots k) | j = 1 \ldots k\}$ *mit* $\delta_{i,j} = \begin{cases} 1, i = j \\ 0, i \neq j \end{cases}$ *gilt.*

Bemerkungen:
Das Symbol $\delta_{i,j}$ aus dieser Definition wird Kronecker-Symbol genannt. Ist ein Design one-hot kodiert, sagen wir, der Zustand $s_i = (\delta_{i,j} | j = 1 \ldots k) \in Reach_M$ wird durch das Register f_{S_i} kodiert.
Die Attraktivität dieser Kodierung im Hinblick auf unser Ziel, Erreichbarkeitsinformation zu berechnen, soll nun kurz motiviert werden. Dazu betrachten wir die Zustandsdiagramme aus Abbildung 4.2.

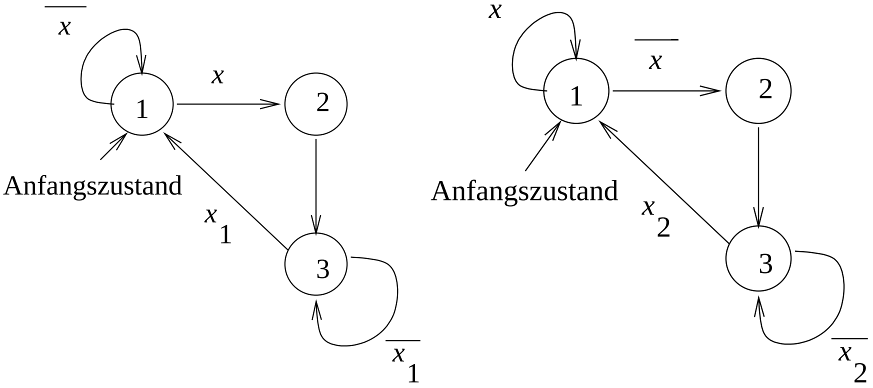


Abbildung 4.2: Beispiel gekoppelter Automaten

Wir nehmen an, dass x gemeinsamer Eingang der beiden Automaten ist und x_1 und x_2 unabhängig sind. Man sieht leicht, dass der Zustand $(2, 2)$ des gekoppelten Automaten niemals erreicht wird. Bei einer binären Kodierung wird diese Information über alle Register des Designs verteilt. In diesem Fall würde man die Automaten durch Register $r_i[1 : 0]$ realisieren, und die gerade genannte Abhängigkeit ließe sich durch folgende Implikationen beschreiben:

$$r_1[1] \wedge \overline{r_1[0]} \Rightarrow r_2[0] \vee \overline{r_2[1]} \text{ und } r_2[1] \wedge \overline{r_2[0]} \Rightarrow r_1[0] \vee \overline{r_1[1]}.$$

In diesen Implikationen besteht die Voraussetzung aus einer Konjunktion mehrerer Literale. Die Berechnung solcher Multiliteralimplikationen ist sehr aufwendig. Kodiert man die Automaten one-hot, so erhält man zwei Register $r_1[2], r_2[2]$, die jeweils den Zustand 2 repräsentieren. Die o.g. Abhängigkeiten lassen sich damit durch die einfachen Implikationen $r_1[2] \Rightarrow \overline{r_2[2]}$ und $r_2[2] \Rightarrow \overline{r_1[2]}$ darstellen. Implikationen dieses Typs lassen sich mit Hilfe der strukturellen Automatentraversierung effizient berechnen.
Bei paarweiser Abhängigkeit der beteiligten Automaten kann die Menge der erreichbaren Zustände sogar exakt durch solche Implikationen beschrieben werden. Damit ist eine Reduktion der Induktionstiefe auf $k = 0$ möglich.

Satz 4.2. *Ein Design M ist genau dann one-hot kodiert, wenn für alle erreichbaren Zustände $(s_1, \ldots, s_k) \in Reach_M \subseteq \mathbb{B}^k$ die Implikationen $s_j \Rightarrow \overline{s_i}$ für alle $i \neq j$ gelten und stets ein $s_l = 1$ existiert.*

Beweis von Theorem 4.2:
Wir nehmen zunächst an, M sei one-hot kodiert. Für $(s_1, \ldots, s_k) \in Reach_M \subseteq \mathbb{B}^k$ gilt dann aufgrund der One-hot-Kodierung: $s_i = \delta_{i,l}$ und $s_j = \delta_{j,l}$ für ein $l \in 1 \ldots k$. Damit gilt $\overline{s_i} + \overline{s_j}$ für $i \neq j$, also die Implikation. Außerdem existiert für alle erreichbaren Zustände ein l mit $s_l = \delta_{l,l} = 1$. Setzen wir für $(s_1, \ldots, s_k) \in Reach_M \subseteq \mathbb{B}^k$ umgekehrt die Implikation $s_j \Rightarrow \overline{s_i}$ für alle $i \neq j$ sowie $s_l = 1$ voraus, so gilt stets $s_j = \delta_{j,l}$. Also ist das Design one-hot kodiert. □
Theorem 4.2 besagt, dass sich die one-hot Eigenschaft eines Designs mit k Registern durch Implikationen und eine zusätzliche Klausel $s_1 + \cdots + s_k$ repräsentieren lässt.

Satz 4.3. *Seien M_1, M_2 one-hot kodierte Designs mit den Registern $r_i[1 \ldots k_i](i = 1, 2)$ und M der gekoppelte Automat aus M_1 und M_2. Die Menge der erreichbaren Zustände von M lässt sich durch eine Menge von Implikationen I zwischen den Registern r_1 und r_2 und eine Menge konstanter Register C_0 (mit Wert 0) exakt darstellen.*

Beweis von Theorem 4.3:
Für alle Zustände $(s_1, s_2) \in (Reach_{M_1} \times Reach_{M_2}) \setminus Reach_M$, enthalte I die Implikation $r_1 \Rightarrow \overline{r_2}$ und $r_2 \Rightarrow \overline{r_1}$ der entsprechenden Register. Ferner sei π_k die Projektion von $Reach_M$ auf $Reach_{M_k}$. Für alle Zustände aus $s_k \in Reach_{M_k} \setminus \pi_k(Reach_M)$ enthalte C_0 das entsprechende Register r_k.
Wir zeigen nun, dass die Mengen I und C_0 zusammen mit der one-hot Eigenschaft der Komponenten M_i die Zustandsmenge $Reach_M$ exakt beschreiben, d.h. es gilt für alle Zustände $(s_1, s_2) \in Reach_{M_1} \times Reach_{M_2}$:

$$(s_1, s_2) \in Reach_M \;\Leftrightarrow r_1, r_2 \notin C_0 \cdot (r_1 \Rightarrow \overline{r_2}) \notin I \cdot (r_2 \Rightarrow \overline{r_1}) \notin I$$

wobei s_1 durch r_1 und s_2 durch r_2 kodiert sei.
$\Rightarrow$: Sei $(s_1, s_2) \in Reach_M$. Also ist $s_1 \in \pi_1(Reach_M)$ und $s_2 \in \pi_2(Reach_M)$ und somit $r_1, r_2 \notin C_0$. $(r_1 \Rightarrow \overline{r_2}) \notin I$ und $(r_2 \Rightarrow \overline{r_1}) \notin I$ folgt unmittelbar aus der Definition von I.
$\Leftarrow$: Wir nehmen an, dass $r_1, r_2 \notin C_0$, $(r_1 \Rightarrow \overline{r_2}) \notin I$ und $(r_2 \Rightarrow \overline{r_1}) \notin I$ gilt. Ist $(s_1, s_2) \notin Reach_M$ folgt sofort $(r_1 \Rightarrow \overline{r_2}) \in I$ und $(r_2 \Rightarrow \overline{r_1}) \in I$. Dies ist ein Widerspruch, womit $(s_1, s_2) \in Reach_M$ folgt. □
Theorem 4.3 besagt in Verbindung mit Theorem 4.2, dass Implikationen, Konstanten und zwei Klauseln eine exakte Darstellung des Zustandsraums zweier gekoppelter Automaten zulassen. Damit eignet sich die hier vorgeschlagene Vorgehensweise insbesondere für die Überprüfung paarweiser Abhängigkeiten, wie Bus-Konflikten, Acknowledge-Requests etc..
Bei komplexeren Abhängigkeiten zwischen mehr als zwei Automaten ist die exakte Darstellung des Zustandsraumes durch einfache Implikationen im Allgemeinen nicht mehr möglich, wie das folgende Beispiel zeigt.

Beispiel 4.1. *Wir betrachten einen Automaten M mit Komponenten A,B und C. Ferner existiere ein Zustand $(s_A, s_B, s_C) \in Reach_A \times Reach_B \times Reach_C$, so dass $Reach_M = Reach_A \times Reach_B \times Reach_C \setminus \{(s_A, s_B, s_C)\}$ gilt.*

Mit anderen Worten, in der Umgebung M erreicht C den Zustand s_C nur dann, wenn A und B nicht gleichzeitig in den Zuständen s_A bzw. s_B sind. Es gelten also die Multiliteralimplikationen $r_A \cdot r_B \Rightarrow \overline{r_C}$, $r_A \cdot r_C \Rightarrow \overline{r_B}$ und $r_B \cdot r_C \Rightarrow \overline{r_A}$. Durch einfache Implikationen kann dieser Sachverhalt nicht dargestellt werden.
Es zeigt sich jedoch in den Experimenten, dass Implikationen auch im Fall komplexerer Abhängigkeiten zumindest eine Überapproximation des Zustandsraumes erlauben, die ausreicht, um die Induktionstiefe drastisch zu reduzieren.

4.2 Strukturelle Automatentraversierung

Ein Verfahren zur effizienten Berechnung einer solchen Überapproximation ist die *strukturelle Automatentraversierung*. Eine detaillierte Beschreibung dieses Verfahrens findet der Leser in [SWWK04]. Im Rahmen dieser Arbeit wurde eine Variante dieses Verfahrens implementiert, mit der sich insbesondere Implikationsbeziehungen zwischen Registern eines Designs berechnen lassen.

4.2.1 Automatentraversierung mit dem iterativen Schaltungsmodell

Zur Erreichbarkeitsanalyse für einen endlichen Automaten, wird in der Regel der Zustandsübergangsgraph durchlaufen und die dabei auftretenden Zustände in einer geeigneten Datenstruktur festgehalten. Dies nennt man dann eine Traversierung des Automaten. Verfahren zur Traversierung eines endlichen Automaten benötigen somit eine Datenstruktur zur Darstellung von Zustandsmengen des Automaten. Liegt der Automat als Schaltung vor, stellt sich die Frage, wie die Struktur dieser Schaltung ausgenutzt werden kann, um solche Zustandsmengen darzustellen. Dass dies in der Tat möglich ist, kann man sich an Hand des iterativen Schaltungsmodells aus Abschnitt 3.3.2 leicht überlegen.
Die Schaltfunktion des iterativen Schaltungsmodells eines bounded model checkers bildet danach Eingabefolgen $i_0, \ldots, i_t$ auf Zustände s_t und Ausgabefolgen $o_0, \ldots, o_t$ ab. In Abbildung 4.3 ist noch einmal ein solches Modell für vier Takte dargestellt. Beschränkt man sich auf die Abbildung der $i_0, \ldots, i_t$ auf die Zustände s_t und betrachtet das Bild dieser Abbildung, dann sind darin genau die Zustände der Schaltung enthalten, die nach genau t Takten in der Schaltung erreichbar sind. Das iterative Schaltungsmodell der Länge t ist damit eine Darstellung der Menge $R(t)$ aller Zustände, die nach exakt t Takten erreicht werden.
Wir halten dies in der folgenden Definition fest:

Definition 4.3. *(Menge erreichbarer Zustände)*
Die Menge $R_M(t)$ bezeichnet die Menge aller Zustände, die nach genau t Zustandsübergängen von einem Anfangszustand eines endlichen Automaten M erreichbar sind.

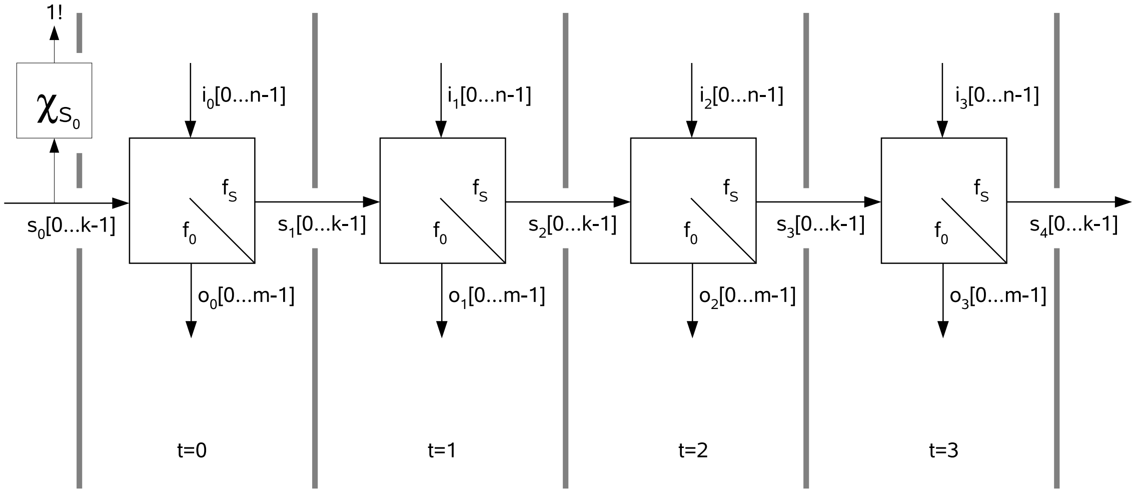


Abbildung 4.3: Iteratives Schaltungsmodell über vier Takte

Bemerkung:
Ist aus dem Kontext klar, auf welchen Automaten M sich die betrachteten Zustandsmengen beziehen, wird die Schreibweise $R(t) := R_M(t)$ verwendet.
Da wir von einem endlichen Automaten M ausgehen, gibt es für jeden Zustand $s \in Reach_M$ einen Zeitpunkt t_s mit $s \in R(t_s)$. Die Folge $R(t)$ durchläuft den Zustandsgraphen von M und besucht dabei jeden erreichbaren Zustand. Damit lässt sich in natürlicher Weise der Algorithmus aus Tabelle 4.1 zur Traversierung des Automaten formulieren.

```
FSM_traversal($\delta$, $S_0$)
{
  $t := 0$;
  $R(t) := S_0$;    /* initial state set */
  repeat
    $t := t + 1$;
    $R(t) :=$ new_states($\delta$, $R(t-1)$);
  until finished($\{R(0), R(1), \ldots, R(t)\}$, $t$);
}
```

Tabelle 4.1: (Vorwärts-)traverierung eines Automaten

Dieser Algorithmus unterscheidet sich von den üblichen (BDD-basierten) Traversierungsalgorithmen für endliche Automaten in der Art und Weise, wie die Funktionen *new_states()* und *finished()* implementiert sind. In jeder Iteration des Algorithmus wird in *new_states()* eine neue Kopie der Zustandsübergangsfunktion an das iterative Schaltungsmodell angefügt. Die Funktion berechnet also das Bild der Menge $R_M(t-1)$ unter dieser Funktion:

$$\textit{new\_states}(\delta, S) := \textit{img}(\delta, S).$$

Hier unterscheidet sich die Vorgehensweise von der klassischen Implementierung, die eine Vereinigung der Menge S mit ihrem Bild *img*(δ, S) durchführt.

Natürlich ist dieses einfache Abrollen des Automaten nur sehr begrenzt praktikabel. Zum einen wächst das iterative Schaltungsmodell linear mit der Anzahl der Iterationen, was zu einem konstanten Wachstum der Repräsentation der Zustandsmenge führt. Dies ist für große Designs nicht sehr lange handhabbar. Zum anderen ist ad hoc nicht klar, wann alle Zustände besucht wurden und die Iteration abgebrochen werden kann.

4.2.2 Existenzielle Quantifizierung

Die Frage nach dem Abbruchkriterium für die Iteration steht in enger Beziehung zum Konvergenzverhalten der Zustandsmengen $R(t)$. In [SK00, SWWK04] findet man eine Analyse dieses Konvergenzverhaltens. Dort wird gezeigt, dass für jeden Automaten ein Zeitpunkt t_{fix} und eine Oszilationsperiode T mit

$$R(t) = R(t-T) \quad \text{für } t \geq t_{fix}$$

existiert. Zu diesem Zeitpunkt sind alle erreichbaren Zustände besucht worden. Das Abbruchkriterium für die Iteration lautet damit:

$$\textit{finished}() := (\text{exists } T \text{ such that } R(t) = R(t-T)).$$

Interessanterweise kann man für die meisten praktischen Schaltungen eine kleine Oszilationsperiode T erwarten. Für viele Schaltungen wird diese sogar den Wert 1 annehmen. Hierzu genügt es zum Beispiel, dass die Schaltung einen Reset-Eingang hat oder im Zustandsübergangsgraphen ein anderer Zyklus der Länge 1 vorhanden ist.

Als nächstes soll kurz der Frage nachgegangen werden, wie die oben genannte Abbruchbedingung ausgewertet werden kann. Dies ist anhand der nicht kanonischen Darstellung durch das iterative Schaltungsmodell natürlich schwieriger als für eine kanonische Darstellung wie beispielsweise BDDs. Zur Fixpunkterkennung muss untersucht werden, ob die iterativen Schaltungsmodelle der Länge t und $t + T$ das gleiche Bild besitzen.

Um dies zu erkennen, führt man eine strukturelle Art der existenziellen Quantifizierung ein. Dabei wird eine funktionale Dekomposition des iterativen Schaltungsmodells mit einem Schnitt durch die Schaltung kombiniert, wie dies in Abbildung 4.4 dargestellt ist.

Der obere Teil der Abbildung 4.4 stellt ein iteratives Schaltungsmodell dar. Dieses wird nun zerlegt in eine bildäquivalente Schaltung (engl. stub circuit) und eine Restschaltung R. Durch Entfernen der Restschaltung verliert man dann die Abhängigkeiten der Zustandsvariablen von den Originaleingängen des Schaltungsmodells. Das Bild der Schaltung, auf dem momentan unser Fokus liegt, bleibt allerdings erhalten. Der Schnitt durch die dekomponierte Schaltung führt also eine Art existentielle Quantifizierung durch und vereinfacht die Darstellung der Zustandsmenge. Die *stub*-Schaltung kann im weiteren Verfahren anstatt des iterativen Schaltungsmodells verwendet werden.

Neben dieser Vereinfachung erleichtert diese Dekomposition auch die Erkennung des Fixpunktes. Gelingt es, für die Darstellungen von $R(t)$ und $R(t + T)$ die gleichen Dekompositionsschritte durchzuführen, so ist der Fixpunkt erreicht.

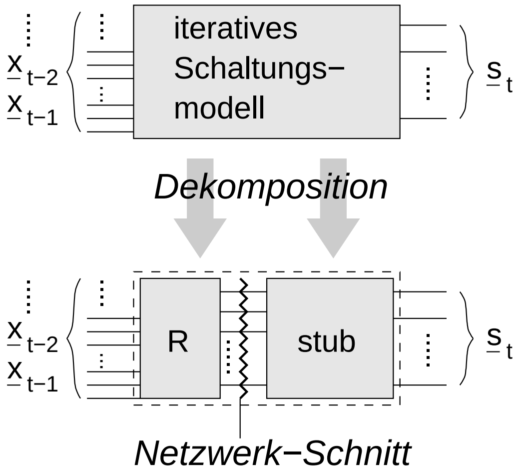


Abbildung 4.4: Strukturelle Form der existenziellen Quantifizierung

Bleibt die Frage offen, wie die bildäquivalente Schaltung berechnet werden kann. Dies ist ein sehr schwieriges Problem, welchem in [Sto99] nachgegangen wurde. Das dort vorgestellte exakte Verfahren ist allerdings für eine praktische Anwendung nicht einsetzbar. Jedoch sind inzwischen eine Reihe anwendungsspezifischer Approximationen entwickelt worden.

Eine Möglichkeit zur Approximation besteht darin, Äquivalenzen, Konstanten und Implikationen zwischen internen Signalen des iterativen Schaltungsmodells zu bestimmen und als zusätzliche Constraints in der Schaltungsdarstellung abzuspeichern. Jeder Schnitt durch die Schaltung führt dann zu einer Überapproximation der Zustandsmenge, da die Abhängigkeiten zwischen den Signalen entlang des Schnitts verloren geht und deshalb mehr Werte an die *stub*-Schaltung angelegt werden können. Die gespeicherten Äquivalenz- bzw. Implikationsbeziehungen zwischen einzelnen Signalen der *stub*-Schaltung bleiben allerdings als zusätzliche Randbedingungen erhalten. Es sind also nur solche Eingangswerte erlaubt, die diese Abhängigkeiten respektieren. Gelingt es nun im nächsten Schritt der Iteration, die gleichen Beziehungen für die Signale der nächsten Übergangsfunktion erneut zu beweisen und den Schnitt an der selben Stelle durchzuführen, so ist ein Fixpunkt erreicht und der *stub* bildet zusammen mit den darin gespeicherten Abhängigkeiten eine Überapproximation für die Menge der erreichbaren Zustände. In der im Rahmen dieser Arbeit entwickelten Variante der strukturellen Automatentraversierung für die Eigenschaftsprüfung wird nun der Schnitt immer entlang der ursprünglichen Registerpositionen vorgenommen, so dass sich die Fixpunkterkennung darauf reduziert, die gleichen Implikationen erneut zu beweisen.

4.3 Implementierung des Eigenschaftsprüfers

An dieser Stelle soll auf die Implementierung des induktionsbasierten Eigenschaftsprüfers eingegangen werden, mit dem die vorgeschlagene Vorgehensweise experimentell evaluiert wurde [WSK03, WSK04b]. Der Flussplan aus Abbildung 4.5 visualisiert den Ablauf dieses Werkzeugs.

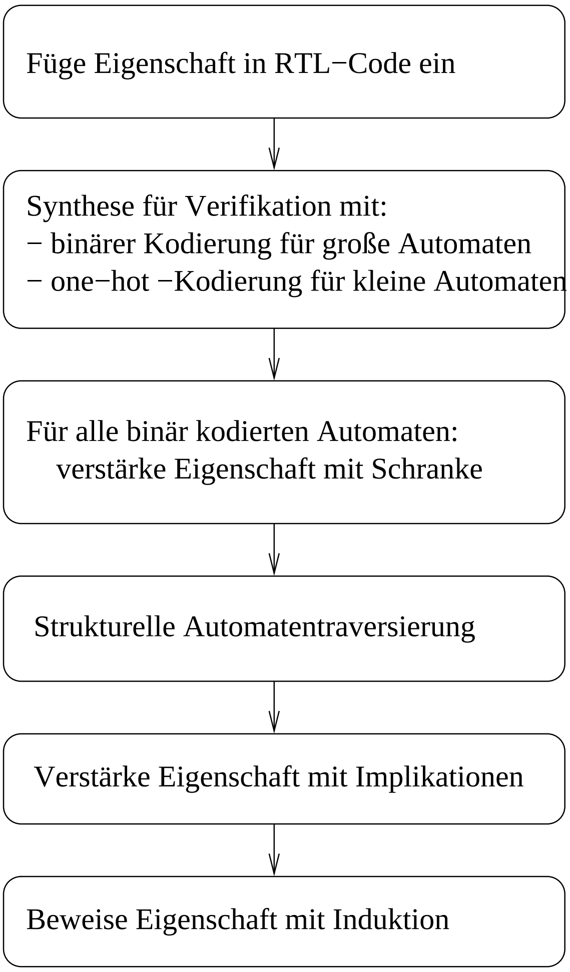


Abbildung 4.5: Flussdiagramm für Eigenschaftsprüfung

Zunächst wird eine Monitorschaltung zur Auswertung der Eigenschaft zur RT-Beschreibung des Designs hinzugefügt. Das Front-End eines industriellen Eigenschaftsprüfers wird anschließend genutzt, um das Design in eine Gatternetzliste zu übersetzen. Dabei wird für Komponenten des Designs die one-hot Zustandskodierung verwendet, sofern die Anzahl der Zustände dies zulässt. Andernfalls wird die binäre Kodierung verwendet. An dieser Stelle sei nochmals darauf hingewiesen, dass die Zustandskodierung während der Eigenschaftsprüfung unabhängig von der späteren Implementierung gewählt werden darf.

Für binär kodierte Designkomponenten wird die obere Schranke des genutzten Wertebereichs der Zustandsvariablen aus Gleichung (4.1) zur Verstärkung der Eigenschaft herangezogen. Zuletzt wird eine strukturelle Automatentraversierung durchgeführt, und die dabei ermittelten Konstanten und Implikationen werden zur Eigenschaft hinzugefügt. Natürlich wird auch die one-hot-Eigenschaft der entsprechenden Register zur Verstärkung der Eigenschaft herangezogen.

Zuletzt wird ein Induktionsbeweis auf dem so entstandenen sequentiellen Modell von Design und Eigenschaft durchgeführt. Dabei starten wir mit Induktionstiefe 0 und erhöhen diese sukzessiv, bis ein Beweis oder ein Gegenbeispiel gefunden wurde. Gelingt dies bis zu einer benutzerdefinierten maximalen Induktionstiefe nicht, so wird der Beweis ohne Ergebnis abgebrochen.

Die Erfüllbarkeitsbeweise in jeder Iteration des Induktionsbeweises werden mit Hilfe des SAT-Solvers ZCHAFF [MMZ$^+$01] durchgeführt. Dieser Solver ist nach wie vor einer der performantesten verfügbaren und quelloffenen Solver. Zur Berechnung der Implikationen während der strukturellen Automatentraversierung wird *Recursive Learning* [KP94] verwendet.

4.4 Experimentelle Ergebnisse

Das vorgeschlagene Verfahren wurde mit einer Reihe von Benchmark-Schaltungen und dazugehörigen Eigenschaften evaluiert. Bei diesen Schaltungen stoßen sowohl die einfache induktionsbasierte Eigenschaftsprüfung, als auch die symbolische Modellprüfung an ihre Grenzen. Dies ist zum Beispiel bei großen Modulo-($2^k + 2^{k-1}$)-Zählern der Fall. Da die Anzahl der Zustände exponentiell mit k wächst, scheidet die One-hot-Kodierung schon für kleine k aus. Für die binäre Kodierung werden $k + 1$ Register benötigt. Es entstehen also 2^{k+1} Zustände. Will man nun beweisen, dass der Zustand $2^{k+1} - 1$ nicht erreichbar ist, so benötigt die reine Induktionsmethode eine Induktionstiefe von $2^k - 1$. Die Verstärkung der Eigenschaft durch die obere Schranke reduziert diese Induktionstiefe auf $k = 0$. Die folgende Tabelle 4.6 zeigt Laufzeiten und Induktionstiefen für einige Werte von k. Dabei bezeichnet P die ursprüngliche Eigenschaft und P' die verstärkte Eigenschaft.

Das zweite untersuchte Design enthält nun zwei Zähler, wobei der erste vorwärts von 0 bis $2^k - 1$ zählt und der zweite rückwärts von $2^k - 1$ bis 0 zählt. Dadurch ist für $k > 1$ gesichert, dass der Zustand $(0, 1)$ des kombinierten Automaten niemals erreicht wird. Tabelle 4.7 zeigt die Induktionstiefen und Laufzeiten für dieses Problem, wobei P und P' wieder die ursprüngliche bzw. verstärkte Eigenschaft referenzieren. Die Tabelle zeigt recht anschaulich die Grenzen induktionsbasierter Eigenschaftsprüfung, wenn nur die Beschränkung des Wertebereichs für einzelne Zustandsvariablen zur Eigenschaftsverstärkung herangezogen wird.

Das nun anschließende Beispiel ist durch ein industrielles Design aus einer Telekommunikationsanwendung motiviert. In diesem Design gibt es sehr viele Instanzen eines Automaten mit relativ wenigen Zuständen. Um dies nachzuahmen, wurde ein Design ent-

k	**Induktionstiefe**		**CPU-Zeit (hh:mm:ss)**	
	P	**P'**	**P**	**P'**
5	15	0	00:01	<00:01
6	31	0	00:07	<00:01
7	63	0	01:53	<00:01
8	127	0	11:58:33	<00:01
15	>1000	0	abgebrochen	<00:01
23	>1000	0	abgebrochen	00:01
30	>1000	0	abgebrochen	00:03

Abbildung 4.6: Ergebnisse für Modulo-$\mathbf{2^k} + \mathbf{2^{k-1}}$-Zähler

k	**Induktionstiefe**		**CPU-Zeit (hh:mm:ss)**	
	P	**P'**	**P**	**P'**
5	63	33	0:04:06	0:00:33
6	127	65	1:32:31	0:10:36
7	255	129	abgebrochen	3:33:24
15	>1000	>1000	abgebrochen	abgebrochen

Abbildung 4.7: Ergebnisse bei gekoppelten Modulo-$\mathbf{2^k} + \mathbf{1}$-Zählern

worfen, in dem der Automat aus Abbildung 4.8 mehrfach instantiiert wurde.

Jede Instanz des Automaten hat einen unabhängigen Eingang. Wenn dieser Eingang gesetzt ist, sendet der Automat ein Request, um eine Ressource zu belegen. Diese wird von einem Arbiter entgegengenommen, der bei freier Ressource ein grant-Signal setzt. Außerdem setzt der Arbiter ein finish-Signal, wenn ein zweiter Request für die Ressource ansteht. Nach einem Bearbeitungs- und Ausgabetakt überprüft jede Automateninstanz, ob dies der Fall ist und gibt gegebenenfalls die Ressource an den Arbiter zurück.

Im Folgenden bezeichnet k stets die Anzahl der Instanzen des Automaten. Wir überprüfen den gegenseitigen Ausschluss der Instanzen im Zustand run. Damit überprüfen wir nicht nur, dass der Arbiter korrekt arbeitet, sondern auch, dass die Instanzen des Automaten die Arbiter-Signale richtig verarbeiten.

Zum Abschluss der Evaluierung wurde der Arbiter durch einen 2-Ressourcen-Arbiter ersetzt. Dieser verwaltet zwei identische Ressourcen. Damit lautet die zu verifizierende Eigenschaft, dass niemals mehr als zwei Automaten im Zustand run sind. Diese Eigenschaft ist nicht binär, weshalb die Menge der erreichbaren Zustände im Design durch Implikationen zwischen Registern nicht exakt dargestellt werden kann. Trotzdem zeigen die Ergebnisse, dass die *strukturelle Automatentraversierung* eine gute Überapproximation dieses Zustandsraums liefert und die Induktionstiefe signifikant reduziert.

In den Tabellen 4.9 und 4.10 ist zunächst die Anzahl der Register der Modelle für dieses Beispiel festgehalten. Die Modelle, die vom Front-End generiert wurden, enthalten bis zu

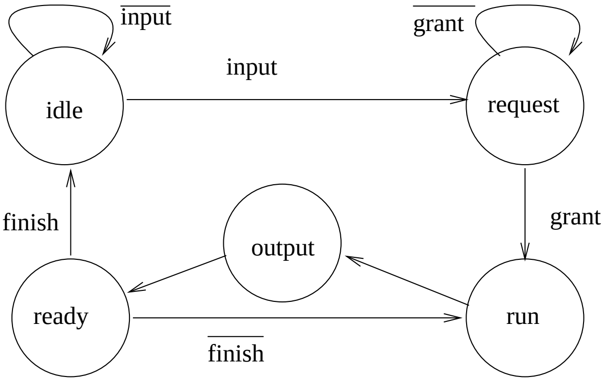


Abbildung 4.8: Zustandsübergangsdiagramm einer Automateninstanz

170 Register. Damit sind die größeren Beispiele auch mit symbolischen Methoden nicht zu behandeln.

k	**Kod.**	**# Reg.**	**Kod.**	**# Reg.**
2	B	19	O	23
4	B	41	O	49
8	B	78	O	94
16	B	138	O	170

Abbildung 4.9: Größe des gekoppelten Automaten mit binären Abhängigkeiten

k	**Kod.**	**# Reg.**	**Kod.**	**# Reg.**
4	B	47	O	63
5	B	52	O	74
6	B	63	O	92
7	B	72	O	109

Abbildung 4.10: Größe des gekoppelten Automaten mit nicht-binären Abhängigkeiten

Die Ergebnistabellen 4.11 und 4.13 sind wie folgt aufgebaut:

- Die erste Spalte gibt die Anzahl der Automaten an.
- Die Spalten zwei und fünf geben die Kodierung der einzelnen Automaten an ((B)inär/(O)ne-hot).
- Die Spalten drei und sechs enthalten die Induktionstiefe für den Beweis der Originaleigenschaft.
- Die Spalten vier und sieben enthalten die gleiche Information für die Eigenschaft nach Verstärkung mit den Implikationen aus der strukturellen Automatentraversierung.

Die Tabellen 4.12 und 4.14 sind ähnlich strukturiert. Allerdings wird hier die CPU-Zeit angegeben.

Anhand der Ergebnisse dieses Kapitels wird deutlich, dass sich die strukturelle Automatentraversierung zur Generierung kraftvoller Invarianten eignet. Optimale Invarianten werden generiert, wenn die Komponenten des Designs eine Kodierung aufweisen, die eine exakte Darstellung oder zumindest eine Approximation des Zustandsraums durch Implikationen erlaubt. Dies ist bei der One-hot-Kodierung der Fall.

k	**Induktionstiefe**					
	enc	**P**	**P$\wedge$Impl**	**enc**	**P**	**P$\wedge$Impl**
2	B	14	14	O	14	**0**
4	B	28	20	O	28	**0**
8	B	>40	>40	O	>40	**0**
16	B	>40	>40	O	>40	**0**

Abbildung 4.11: Induktionstiefe bei binären Abhängigkeiten

k	**CPU-Zeit (mm:ss)**					
	enc	**P**	**P$\wedge$Impl**	**enc**	**P**	**P$\wedge$Impl**
2	B	00:09	0:00:11	O	00:13	**00:01**
4	B	40:05	0:04:53	O	41:57	**00:08**
8	B	abgebrochen	abgebrochen	O	abgebrochen	**01:25**
16	B	abgebrochen	abgebrochen	O	abgebrochen	**06:56**

Abbildung 4.12: CPU-Zeiten bei binären Abhängigkeiten

k	**Induktionstiefe**					
	enc	**P**	**P$\wedge$Impl**	**enc**	**P**	**P$\wedge$Impl**
4	B	>32	24	O	>32	**3**
5	B	>32	26	O	>32	**3**
6	B	>32	21	O	>32	**3**
7	B	>32	>27	O	>32	**3**

Abbildung 4.13: Induktionstiefe bei nicht binären Eigenschaften

k	**CPU-Zeit (hh:mm:ss)**					
	enc	**P**	**P$\wedge$Impl**	**enc**	**P**	**P$\wedge$Impl**
4	B	abgebrochen	0:56:02	O	abgebrochen	**0:00:57**
5	B	abgebrochen	2:49:22	O	abgebrochen	**0:01:16**
6	B	abgebrochen	10:09:31	O	abgebrochen	**0:02:15**
7	B	abgebrochen	abgebrochen	O	abgebrochen	**0:05:59**

Abbildung 4.14: CPU-Zeiten bei nicht binären Abhängigkeiten

Kapitel 5

Arithmetische Schaltungen

Obwohl sich Eigenschaften arithmetischer Schaltungen sehr gut in beschränkten Zeitintervallen formulieren lassen, stößt der BIMC-Ansatz auch hier auf Komplexitätsprobleme. Grund dafür sind die bekannten Schwierigkeiten, die SAT-Solver beim Umgang mit Arithmetik haben. In der Regel scheitern diese Solver deshalb beim Beweis von Eigenschaften für arithmetische Schaltungen. Um dieser Problematik zu begegnen können bereits bei der Modellgenerierung im Frondend des Eigenschaftsprüfers Maßnahmen getroffen werden. Es wird in diesem Kapitel eine Normalisierungstechnik vorgestellt. Dieses Verfahren arbeitet auf einer *arithmetischen Bitebenen-Beschreibung (engl. arithmetic bit level description, ABL)* für die entsprechenden Teile von Eigenschaft und Schaltung. Eine solche Beschreibung erlaubt eine einheitliche Modellierung arithmetischer Einheiten sowohl der Bitebene als auch der Wortebene. Damit wird der Notwendigkeit Rechnung getragen, dass Designs für arithmetische Schaltungen aus Performanzgründen häufig auf der Bitebene entworfen werden. Es wird ausgenutzt, dass die Arithmetik in solchen Entwürfen in der Regel aus elementaren arithmetischen Funktionen der Bitebene zusammengesetzt wird. Ist dies der Fall, kann die ABL leicht im Front-End des Eigenschaftsprüfers generiert werden. Die vorgeschlagene Normalisierung [WSK05] vereinfacht die zu lösenden SAT-Instanzen dramatisch. Dadurch ist es erstmals möglich, die Integer-Pipeline eines komplexen industriellen Mikroprozessors mit zusätzlicher Funktionalität für die digitale Signalverarbeitung (DSP) vollständig formal zu verifizieren.

5.1 Eigenschaften arithmetischer Schaltungen

An dieser Stelle werden zunächst Eigenschaften analysiert, wie sie bei der Verifikation arithmetischer Schaltungen typischerweise auftreten. Zur Spezifikation solcher Intervall-Eigenschaften verwenden wir die Eigenschaftssprache ITL des Eigenschaftsprüfers Gateprop der Firma OneSpin Solutions GmbH. Diese Eigenschaftssprache wurde beispielsweise in [WSFT04] verwendet. Diese Sprache erlaubt es dem Designer oder Verifikationsingenieur, die gewünschten Eigenschaften der Schaltung in einer an Verilog bzw. VHDL angelehnten Syntax zu spezifizieren. Da die Semantik dieser Sprache sehr intuitiv ist, soll hier auf eine vollständige formale Einführung verzichtet werden.

Anhand einer Beispieleigenschaft sollen die wesentlichen Sprachmerkmale veranschaulicht werden. Diese Eigenschaft in Tabelle 5.1 stellt einen Prototyp für eine ganze Reihe von Eigenschaften dar, die in der praktischen Anwendung spezifiziert werden. Hier wird zum Beispiel festgelegt, dass das Ergebnis eines Multiplikationsbefehls nach vier Takten am Ausgang der Integer Pipeline *ip_res* vorliegt, wenn bestimmte Annahmen über die Umgebung des Designs erfüllt sind.
Eine Eigenschaft wird durch ein Theorem beschrieben und besteht im Wesentlichen aus zwei Teilen:

- Annahmen (engl. assumptions), eingeleitet durch das Schlüsselwort **assume**
- Beweisziele (engl. commitments), eingeleitet durch das Schlüsselwort **prove**.

Annahmen und Beweisziele bestehen jeweils aus Booleschen Ausdrücken über den Signalen der Schaltung und einer relativen Zeitangabe. Um Signale aus verschiedenen Taktzyklen in Beziehung setzen zu können, gibt es darüber hinaus die Möglichkeit, diese mit Aliassen zu benennen, die in jedem Zeitfenster des iterativen Schaltungsmodells sichtbar sind. Dazu wird das Schlüsselwort **freeze** verwendet. Um die Lesbarkeit von Eigenschaften zu erhöhen und die Wiederverwendbarkeit von Annahmen und Beweiszielen zu ermöglichen, können Makros definiert werden. Beispielsweise könnte das Makro *no_reset* durch die Formel *reset==0* definiert werden. Ändert sich die Phase des reset-Signals, muss dies später nur einmal für alle Eigenschaften geändert werden.

```
theorem mulXXX;
  freeze: op1_at_t=op1@t, op2_at_t=op2@t;
  assume:
    during[t,t+4]:no_reset;
    during[t,t+4]:environment_assumptions;
    at t:command(mulXXX,op1,op2);
  prove:
    at t+4:ip_res=command_res(mulXXX,op1_at_t,op2_at_t);
end theorem;
```

Tabelle 5.1: Skizze einer arithmetischen Eigenschaft

Eigenschaften der in Tabelle 5.1 skizzierten Art treten häufig auf, wenn die korrekte Implementierung des Befehlssatzes eines Prozessors nachgewiesen werden soll. Handelt es sich bei dem zu verifizierenden Befehl um eine arithmetische Operation des Prozessors, so kann das gewünschte Operationsergebnis durch einen arithmetischen Ausdruck spezifiziert werden. Damit ergibt sich für die zu lösende SAT-Instanz die in Abbildung 5.1 dargestellte Grobstruktur. Dort wird das iterative Schaltungsmodells um eine Schaltung ergänzt, deren Ausgang genau dann den Wert 1 annimmt, wenn die Annahmen der Eigenschaft zutreffen. Zusätzlich wird eine Schaltung für den arithmetischen Ausdruck aus dem Beweisziel der Eigenschaft aufgebaut und mit dem Ergebnissignal aus dem iterativen Schaltungsmodell verglichen. Vereinfacht wurden alle internen Signale als 1-Bit-Signale dargestellt, obwohl in der Realität natürlich ganze Bitvektoren aus dem Schaltungsmodell abgegriffen werden. Gesucht ist eine Belegung für die Eingänge des iterativen Schaltungsmodells, die einerseits die Annahmen erfüllt und andererseits das Beweisziel widerlegt. Herkömmliche SAT-Techniken sind zwar in der Lage, ungültige Eigenschaften dieser Art sehr schnell zu widerlegen, scheitern aber am Beweis einer gültigen Eigenschaft. In diesem Kapitel wird deshalb ein neues Verfahren entwickelt, welches gültige Eigenschaften effizient beweisen kann.

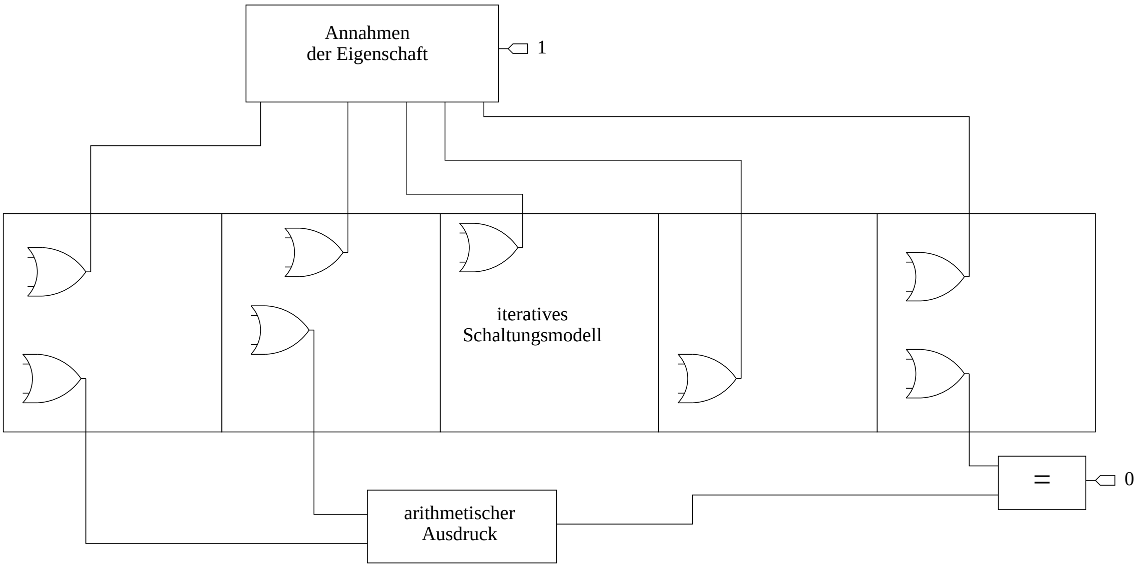


Abbildung 5.1: Grobstruktur der unterliegenden SAT-Instanz

Die Schwierigkeit der nicht erfüllbaren SAT-Instanzen hat im Wesentlichen zwei Gründe. Zunächst kann im Allgemeinen natürlich keinerlei strukturelle Ähnlichkeit zwischen abgerollter Schaltung und dem synthetisierten Ausdruck erwartet werden. Damit kann das Problem nicht ohne weiteres in kleinere Teilprobleme zerlegt werden, wie dies beim Äquivalenzvergleich üblich ist. Darüber hinaus wird die abgerollte Schaltung immer noch einen signifikanten Anteil an Kontrolllogik enthalten. Deshalb muss eine generelle Methodik zur Lösung dieser Probleme zwei Aspekte berücksichtigen.

a) Es ist zu vermeiden, dass der SAT-Solver über das gesamte rekonvergente Netzwerk aus Schaltung und Ausdruck schließen muss. Dieser Aspekt wird in den nun folgenden Abschnitten 5.2 und 5.3 dieses Kapitels behandelt. Für den arithmetischen Teil des Problems wird hier eine Beschreibung auf der Bitebene (ABL) eingeführt. Diese kann mit dem vorgeschlagenen Normalisierungsansatz stark vereinfacht werden.

b) Die eingebettete Kontrolllogik muss verarbeitet werden können. Auf diese Problematik geht Abschnitt 5.4 ein. Dabei stellt sich heraus, dass man SAT-Techniken nutzen kann, um die Kontrolllogik zu behandeln.

5.2 Arithmetische Bitebenen-Beschreibung (ABL)

In diesem Abschnitt wird eine Beschreibung einer arithmetischen Schaltung auf der Bitebene (ABL) eingeführt. Diese Beschreibung setzt sich aus drei Klassen von Objekten zusammen: *Additionsnetzwerken, partiellen Produkten und Vergleichern*. Im Folgenden werden nun zunächst diese Basisobjekte definiert und anschließend die ABL als Graph definiert, der diese drei Typen als Knoten verwendet.

Mit Additionsnetzwerken wollen wir Hardwarestrukturen beschreiben, die ausschließlich aus 1-Bit-Additionseinheiten, wie Halbaddierern, Volladdierern und XOR-Gattern, zusammengesetzt werden. Daher berechnet ein Additionsnetzwerk eine gewichtete Summe einer Menge Boolescher Variablen, wobei die Gewichte ganze Zahlen sind. Formal wird das Additionsnetzwerk wie folgt definiert:

Definition 5.1. *Ein Additionsnetzwerk N ist ein 4-Tupel (A, r, w, c), wobei die Komponenten wie folgt definiert sind:*

- *A ist eine Menge Boolescher Variablen mit $A = A_0 \cup \cdots \cup A_n$,*
- *$r = (r_0, \ldots, r_n)$ ist ein Vektor Boolescher Variablen,*
- *$w = (w_0, \ldots, w_n)$ ist eine Vektorfunktion mit:*
  $w_i : A_i \to \mathbb{Z}$,
- *$c = (c_0, \ldots, c_n)$ ist ein Vektor mit $c_i \in \mathbb{Z}$.*

Wir nennen A die Menge der Summanden $a \in A$ und $n+1$ die Anzahl der Spalten des Additionsnetzwerks N. Ein Summand $a_i \in A_i$ gehört zur Spalte i. Für jeden Summanden a nennen wir $w_i(a)$ das Gewicht von a in Spalte i. r_i heißt Ergebnis und c_i konstanter Offset der Spalte i.

Der Ergebnisvektor r von N wird durch folgende Gleichung definiert:

$$\mathbb{Z}(r) = \Big(\sum_{i=0 \ldots n} 2^i \big(c_i + \sum_{a \in A_i} w_i(a) a\big)\Big) \; mod \; 2^{n+1}. \tag{5.1}$$

Bemerkung:
Ein Summand kann in mehreren Spalten eines Additionsnetzwerks aufsummiert werden, d.h. $A_i \cap A_k = \emptyset$ gilt im Allgemeinen nicht. Darüber hinaus kann jedes Additionsnetzwerk mit einer Booleschen Funktion identifiziert werden, die eine Komposition von Halbaddierern ist. Für Additionsnetzwerke ohne negative Gewichte ist dies unmittelbar klar. Glücklicherweise kann man negative Gewichte wie folgt eliminieren:
Angenommen, es gilt $w_i(a) < 0$ für einen Summanden a mit $i < n$. Wir gehen davon aus, dass a auch in der nächsten Spalte als Summand vorkommt, d.h. es gilt $a \in A_{i+1}$. Andernfalls setzen wir $A_{i+1} = A_{i+1} \cup \{a\}$ mit Gewicht $w_{i+1}(a) = 0$ und erhalten ein äquivalentes Netzwerk. Wir verschieben das negative Gewicht in die nächste Spalte, indem wir die Gewichtsfunktion w' mit $w'_i(a) = -w_i(a)$ und $w'_{i+1}(a) = w_{i+1}(a) + w_i(a)$ definieren. Auf diese Weise können alle negativen Gewichte in die oberste Spalte des Additionsnetzwerks verschoben werden.
In der obersten Spalte eines Additionsnetzwerks können wir das Gewicht einfach invertieren, d.h. $w'_n(a) = -w_n(a)$, um ein äquivalentes Netzwerk zu erhalten. Um die gerade gemachten Aussagen zu verifizieren, genügt es, Gleichung 5.1 heranzuziehen. Aus

$$x + 2^n w_n(a) - (x - 2^n w_n(a)) = 2^{n+1} w_n(a)$$

folgt

$$(x - 2^n w_n(a)) \bmod 2^{n+1} = (x + 2^n w_n(a)) \bmod 2^{n+1},$$

d.h., bei Änderung des Vorzeichens des Summanden $2^n w_n(a) a$ bleibt die rechte Seite der Gleichung 5.1 unverändert. Damit sind beide Netzwerke äquivalent.
Obwohl negative Gewichte prinzipiell nicht notwendig sind, wurden sie in die Definition des Additionsnetzwerks aufgenommen, um die Modellierung vorzeichenbehafteter arithmetischer Operationen zu erleichtern. Da wir gerade gezeigt haben, wie diese negativen Gewichte eliminiert werden können, wird im Folgenden davon ausgegangen, dass alle Gewichte positiv sind.
Manchmal ist es sinnvoll, alle Gewichte eines Additionsnetzwerks auf 1 zu normieren. Dass dies möglich ist, formuliert das folgende Lemma:

Lemma 5.1. *Zu jedem Additionsnetzwerk N existiert ein äquivalentes Additionsnetzwerk N', so dass in N' für alle Gewichte $w_i(a) = 1$ gilt.*

Beweis: Wir können, wie oben gezeigt, $w_i(a) > 0$ voraussetzen. Gilt $w_i(a) > 1$ für einen Summanden a in Spalte i, so setzen wir $w'_i(a) = w_i(a) \bmod 2$ und $w'_{i+1}(a) = w_{i+1}(a) + w_i(a)/2$, falls i nicht bereits die oberste Spalte ist. Anhand von Gleichung 5.1 überzeugt man sich leicht, dass dies eine Äquivalenztransformation ist. Durch iteriertes Anwenden dieser Transformation normiert man ausgehend von Spalte 0 in jeder Spalte alle Gewichte auf 1. □
Bemerkung:
Völlig analog kann man auch die konstanten Offsets auf 0 bzw. 1 normieren.

Beispiel 5.1. *In Abbildung 5.2 ist eine Schaltung zur Addition der partiellen Produkte eines 2-Bit Multiplizierer dargestellt. Diese Schaltung kann durch ein Additionsnetzwerk modelliert werden, mit:*

- $A_0 = \{x_0\}$, $A_1 = \{x_1, x_2\}$, $A_2 = \{x_3\}$, $A_3 = \emptyset$,
- $r = (y_0, y_1, y_2, y_3)$,
- $w_i(x) = 1$ *für alle* $x \in A_i$ *und alle* i,
- $c = (0, 0, 0, 0)$.

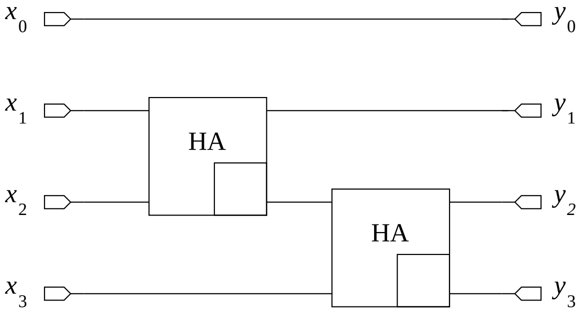


Abbildung 5.2: Additionsnetzwerk eines 2-Bit Multiplizierer

Gleichung 5.1 besagt, dass alle Überträge, die in einer Spalte eines Additionsnetzwerks entstehen können, als Summanden in die nächste Spalte eingehen. So entstehen im Additionsnetzwerk aus Abbildung 5.2 in den Spalten 1 und 2 Überträge, welche in die Spalten 2 bzw. 3 eingehen. Formal definieren wir die Überträge eines Additionsnetzwerks wie folgt:

Definition 5.2. *Ein Additionsnetzwerk erzeugt in Spalte* k *einen Übertrag, wenn eine Belegung der Summanden existiert, so dass die folgende Bedingung erfüllt ist:*

$$s_k := (\sum_{i=0..k} 2^i (c_i + \sum_{a \in A_i} w_i(a)a)) \geq 2^{k+1} \textit{ oder } s_k \leq -2^{k+1} + 1.$$

Mit Additionsnetzwerken sind wir in der Lage, elementare 1-Bit Additionen und Subtraktionen sowie n-Bit Additionen bzw. Subtraktionen in einem durchgängigen Modell zu modellieren. Dies soll nun durch die Repräsentation von 1-Bit Multiplikationsergebnissen vervollständigt werden. Zwar ist mit Additionsnetzwerken prinzipiell auch die Darstellung von partiellen Produkten als Übertrag einer 1-Bit Addition möglich, doch würde durch diese Darstellung die Erkennung dieser Produkte in einer ABL sehr schwierig. Die Unterscheidung von partiellen Produkten und Additionsnetzwerken ist jedoch eine wichtige Voraussetzung für das im nächsten Abschnitt beschriebene Verfahren zur Normalisierung der ABL. Als nächstes sollen deshalb partielle Produkte modelliert werden. Dazu definieren wir einen Partialproduktgenerator wie folgt:

Definition 5.3. *Ein Partialproduktgenerator P ist ein Tripel (o_1, o_2, p), wobei $o_1 = (o_{1,0}, \dots, o_{1,n})$, $o_2 = (o_{2,0}, \dots, o_{2,m})$ und $p = (p_0, \dots, p_{n*m})$ Vektoren Boolescher Variablen sind. Dabei heißen die $o_{i,k}$ Bitebenen-Operanden und die p_i partielle Produkte. Ferner nennen wir o_1 und o_2 Operanden der Wortebene. Die partiellen Produkte hängen von den Operanden wie folgt ab:*

$$p_{i+nk} = o_{1,i} \cdot o_{2,k}.$$

Abbildung 5.3 zeigt exemplarisch die Modellierung des Partialproduktgenerators eines 2x2-Multipliziers. Alternativ können wir die partiellen Produkte auch als Menge angeben.

- $o_1 = (a[0], a[1])$,
- $o_2 = (b[0], b[1])$,
- $p = (e, f, g, h)$.

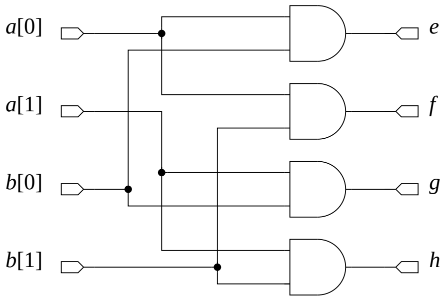


Abbildung 5.3: Beispiel: Partialproduktgenerator eines 2x2-Multiplizierers

Dies ist Gegenstand des folgenden Lemmas.

Lemma 5.2. *Zu jeder endlichen Menge von partiellen Produkten P' existiert ein Partialproduktgenerator P, der alle Produkte aus P' erzeugt.*

Beweis: Sei $P' = \{p_i = o_{1,i}o_{2,i} | i = 0 \dots n\}$. Der Partialproduktgenerator P mit den Operanden $o_1 = (o_{1,i} | i = 0 \dots n)$ und $o_2 = (o_{2,i} | i = 0 \dots n)$ erzeugt diese Produkte. □.

Definition 5.4. *Seien P' und P wie in Lemma 5.2 definiert. Dann heißt $\langle P' \rangle := P$ der von P' erzeugte Partialproduktgenerator.*

Durch Kombination von Additionsnetzwerken und Partialproduktgeneratoren können nun die arithmetischen Bitvektorfunktionen $+$ und $*$ realisiert werden. Nutzt man die negativen Gewichte aus, so können auch Subtraktion und vorzeichenbehaftete Multiplikation modelliert werden. Die Subtraktion der Bitvektoren $(a_k, \dots, a_0)$ und $(b_k, \dots, b_0)$ wird durch ein Additionsnetzwerk mit $A_i = \{a_i, b_i\}$, $w_i(a_i) = 1$, $w_i(b_i) = -1$ und $c_i = 0$ modelliert. Die vorzeichenbehaftete Multiplikation der Bitvektoren $(a_k, \dots, a_0)$ und $(b_l, \dots, b_0)$ wird realisiert, indem die partiellen Produkte $a_k b_i (i < l)$ und $a_j b_l (j < k)$ aus den entsprechenden Spalten des Additionsnetzwerks subtrahiert werden, d.h., $w_{k+i}(a_k b_i) = -1 (i < l)$ und $w_{j+l}(a_j b_l) = -1 (i < l)$.

Als letzter Baustein für ABLs wird nun der Vergleicher eingeführt. Dadurch sollen Bitvektorfunktionen wie $=$ und $\neq$ beschrieben werden. Wichtig ist dabei die Eigenschaft des Vergleichers, dass sein Ergebnis invariant gegenüber der Addition eines Wertes x zu beiden Operanden des Vergleichers ist. Diese Invariante wird durch den Normalisierungprozess ausgenutzt.
An dieser Stelle sei bemerkt, dass diese Invariante für die Vergleiche $<, >, \leq$ und $\geq$ nicht gilt. Es zeigt sich jedoch, dass bestimmte Restriktionen für die Additionsnetzwerke im Fanin eines solchen Vergleichers ausreichen, um auch hier eine Normalisierung durchführen zu können, wie sie im Folgenden für $=$ und $\neq$ beschrieben wird.

Definition 5.5. *Ein Vergleicher C ist ein 4-Tupel (c_1, c_2, o, f) bestehend aus zwei Vektoren Boolescher Variablen $c_1 = (c_{1,0}, \ldots, c_{1,n})$ und $c_2 = (c_{2,0}, \ldots, c_{2,n})$, einer Booleschen Variable o und einer Abbildung $f : B^n \times B^n \to B$ mit $f(x, y) = f(x + z, y + z)$ für alle $z \in B^n$. Hierbei bezeichnet $+$ die übliche Bitvektor-Addition.*
Die c_i werden Operanden, o Ergebnis und f Vergleichsfunktion genannt. Das Ergebnis o ist durch die Boolesche Funktion

$$o = f(c_1, c_2)$$

definiert.

Als Beispiel zeigt Abbildung 5.4 die Modellierung eines 4-Bit ($\neq$)-Vergleichers.

- $c_1 = (e, f, g, h)$,
- $c_2 = (i, j, k, l)$.
- $o = 1?$
- $f(x, y) = \begin{cases} 1, x \neq y \\ 0, x = y \end{cases}$

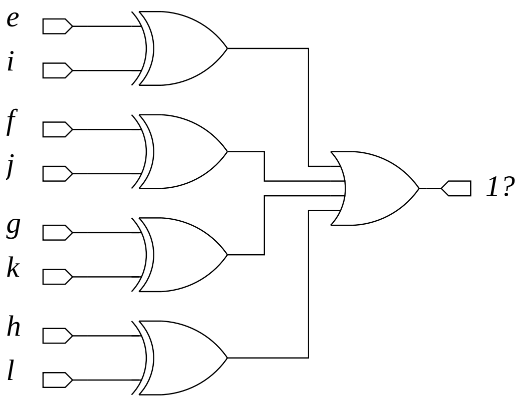


Abbildung 5.4: Beispiel: 4-Bit Vergleicher

Aus den nun vorhandenen Basiskomponenten wird die arithmetische Bitebenen-Beschreibung in Definition 5.7 zusammengesetzt. Wir wollen darin Komponenten, die gleiche Variablen benutzen, durch Kanten zu einem Graphen verbinden. Dazu führen wir nun zunächst für jeden Komponententyp den Begriff einer Fanin- bzw. Fanout-Variable ein.

Definition 5.6. *Sei v eine Boolesche Variable.*

a) *v heißt* Fanin-*Variable eines Additionsnetzwerks N genau dann, wenn v ein Summand von N ist.*

b) *v heißt* Fanin-*Variable eines Partialproduktgenerators P genau dann, wenn v ein Operand von P ist.*

c) *v heißt* Fanin-*Variable eines Vergleichers C genau dann, wenn v ein Operand von C ist.*

d) *v ist heißt* Fanout-*Variable eines Additionsnetzwerks N, genau dann, wenn v ein Ergebnis von N ist.*

e) *v heißt* Fanout-*Variable eines Partialproduktgenerators P genau dann, wenn v ein partielles Produkt von P ist.*

Bemerkung: Man beachte, dass Vergleicher keine Fanout-Variablen haben.
Mit Hilfe des Fanin- und des Fanout-Begriffs können wir nun die Kantenmenge einer ABL festlegen. Dies geschieht in der folgenden Definition.

Definition 5.7. *Sei $\mathcal{N}$ eine Menge von Additionsnetzwerken, $\mathcal{P}$ eine Menge von Partialproduktgeneratoren und $\mathcal{C}$ eine Menge von Vergleichern. Eine arithmetische Bitebenen-Beschreibung (ABL) ist ein gerichteter azyklischer Graph $G = (V, E)$ mit $V = \mathcal{N} \cup \mathcal{P} \cup \mathcal{C}$. Für die Kanten $(x, y) \in E$ existiere dabei stets eine Boolesche Variable l, so dass l Fanin-Variable von y und Fanout-Variable von x ist.*

Bemerkung: Da Vergleicher keine Fanout-Variablen haben, sind diese stets Senken in der ABL. Nach ihrer Definition repräsentieren die Knoten jeder ABL Boolesche Vektorfunktionen. Damit entspricht die ABL der Komposition dieser Funktionen. Analog zur Bitvektornetzliste können wir eine Globalfunktion für jede Fanout-Variable der ABL angeben. Somit lässt sich der Boolesche Äquivalenzbegriff auf ABLs wie folgt übertragen.

Definition 5.8. *Zwei ABLs A, B heißen genau dann äquivalent, wenn sie für gemeinsame Fanout-Variablen die gleiche Funktion repräsentieren.*

Für unsere Zwecke ist dieser Äquivalenzbegriff jedoch etwas zu restriktiv, wie das folgende Beispiel zeigt.

Beispiel 5.2. *Wir betrachten ABLs A und B für die folgenden Vergleiche:*

a) $r = ((a + c) = (b + c))$

b) $r = ((a + d) = (b + d))$.

Dabei seien die Ausdrücke $a + c$ und $b + c$ bzw. $a + d$ und $b + d$ jeweils durch Additionsnetzwerke mit den Ergebnissen r_a und r_b modelliert.

Da die Globalfunktionen von r_a in A und B nicht äquivalent sind, sind die ABLs ebenfalls nicht äquivalent, obwohl die Globalfunktionen von r in beiden ABLs identisch sind. Dies motiviert die Einführung eines etwas schwächeren Äquivalenzbegriffs, der lediglich die Äquivalenz der Globalfunktionen für eine bestimmte Menge gemeinsamer Variablen fordert.

Definition 5.9. *Zwei ABLs A, B heißen genau dann äquivalent bezüglich einer Teilmenge M der gemeinsamen Fanout-Variablen, wenn sie für jede Variable aus M die gleiche Funktion repräsentieren.*

Die ABLs aus Beispiel 5.2 sind äquivalent bezüglich $\{r\}$. Häufig ist man, wie in diesem Beispiel, besonders am Ergebnis der Vergleicher einer ABL interessiert. Sind zwei ABLs bezüglich der Fanout-Variablen der Vergleicher äquivalent, so nennen wir die ABLs auch äquivalent bezüglich der Vergleicher.
Damit sind wir nun in der Lage den arithmetischen Kern von Problemen der Eigenschaftsprüfung zu modellieren. Im nächsten Abschnitt widmen wir uns nun dem Problem, aus einer gegebenen ABL eine möglichst einfache bezüglich der Vergleicher äquivalente ABL zu berechnen.

5.3 Normalisierung

Dieser Abschnitt widmet sich der Definition einer Normalform für die arithmetische Bitebenen-Beschreibung (ABL) und beschreibt ein Verfahren, mit dem eine ABL effizient normalisiert werden kann. Zunächst soll die zugrunde liegende Idee an einem kleinen Beispiel motiviert werden. Angenommen, es soll eine Schaltung verifiziert werden, die das Polynom $p = ax^2 + bx + c$ auswertet. Diese Schaltung verwende eine kombinatorische Multiply-/Accumulate-Einheit mit der Ausgabefunktion $o = xy + z$, um das Ergebnis in zwei Taktzyklen zu ermitteln. Rollt man das Design zwei Takte ab, so ergibt sich der Ausdruck $p = (ax + b)x + c$. Dieser Ausdruck führt zu einer strukturell völlig anders aufgebauten Schaltung als der Ausdruck $p = ax^2 + bx + c$ aus der Spezifikation. Auf der Wortebene lässt sich das Verifikationsproblem in diesem Fall leicht durch Rewriting [Bri03] lösen. Dabei wird das Distributivgesetz verwendet, um beide Ausdrücke auf die Form $p = ax^2 + bx + c$ zu bringen. Diese Verfahren kann man allerdings nur dann effizient einsetzen, wenn das Problem ausschließlich auf Wortebene vorliegt. Bei heutigen industriellen Hochleistungsprozessoren ist dies meist nicht der Fall. Aus Performanzgründen sind hier Designer in der Regel gezwungen, spezielle Arithmetikblöcke zu entwerfen. Diese werden in der Regel auf der Bitebene hart kodiert, d.h., der Designer instantiiert Halbaddierer, Volladdierer und Zusatzlogik, um arithmetische Funktionen zu berechnen. So können beim Tricore 2 von Infineon über 600 verschiedene Varianten von Multiply-/Accumulate-Instruktionen auf der gleichen Hardware ausgeführt werden. Für die formale Verifikation solcher Designs gab es bisher keine befriedigende Technik.

Eine ähnliche Normalisierung wie auf der Wortebene kann allerdings auch auf der Bitebene erreicht werden. Nehmen wir dazu an, dass alle beteiligten Variablen a, b, c und x, aus dem obigen Beispiel, 2-Bit Zahlen sind. Dann erhält man für die Spezifikation $p = ax^2 + bx + c$ die folgende pseudo-Boolesche Formel:

$$p = (2a_1 + a_0)(2x_1 + x_0)^2 + (2b_1 + b_0)(2x_1 + x_0) + (2c_1 + c_0).$$

Diese wird mit Hilfe des Distributivgesetzes und des Kommutativgesetzes wie folgt normalisiert:

$$p = 8a_1x_1+8a_1x_0x_1+4a_0x_1+4a_0x_0x_1+4b_1x_1+2a_1x_0+2b_0x_1+b_0x_0+2b_1x_0+2c_1+a_0x_0+c_0.$$

Diese Termrepräsentation erweist sich jedoch in der Praxis als untauglich, da selbst für relativ kleine arithmetische Schaltungen sehr große Terme entstehen. Allerdings lässt sich der gerade intuitiv beschriebene Ansatz auf die im vorigen Abschnitt eingeführte ABL übertragen.

Definition 5.10. *Eine ABL* $(\mathcal{N} \cup \mathcal{P} \cup \mathcal{C}, E)$ *ist in* **Normalform** *genau dann, wenn* E *keine Kante* (N, P) *mit* $N \in \mathcal{N}$ *und* $P \in \mathcal{P}$ *enthält.*

Eine ABL in Normalform benutzt also keine Ergebnisse von Additionsnetzwerken, um neue partielle Produkte zu berechnen.

Definition 5.11. *Eine ABL* $(\mathcal{N} \cup \mathcal{P} \cup \mathcal{C}, E)$ *in Normalform ist* **reduziert** *genau dann, wenn folgende Bedingungen gelten:*

a) Alle Additionsnetzwerke $N_1, N_2 \in \mathcal{N}$*, für die ein Vergleicher* $C \in \mathcal{C}$ *mit* $(N_1, C) \in E$ *und* $(N_2, C) \in E$ *existiert, haben keine gemeinsamen Summanden.*

b) Es gibt keine Kante $(N_1, N_2) \in E$ *zwischen Additionsnetzwerken* $N_1, N_2 \in \mathcal{N}$*.*

ABLs in reduzierter Normalform kaskadieren also keine Additionsnetzwerke und vereinfachen Vergleiche durch Subtraktion gemeinsamer Summanden von beiden Additionsnetzwerken. Hier sei bemerkt, dass eine reduzierte Normalform keine kanonische Darstellung der Schaltung ist. So kann z.B. das konstante Offset für die oberste Spalte eines Additionsnetzwerks um 2 erhöht werden, um ein äquivalentes Netzwerk zu erhalten. Des Weiteren ist zu klären, ob eine Normalform für jede ABL existiert. Dies ist Inhalt der folgenden Sätze 5.1 und 5.2 .

Satz 5.1. *Für jede Boolesche Funktion* $f : \mathbb{B}^n \to \mathbb{B}$ *existiert eine äquivalente ABL in reduzierter Normalform.*

Beweis:
Durch Entwicklung nach der positiven Davio-Dekomposition erhalten wir eine Darstellung von f der Form $f(x_1, \ldots, x_n) = p_1 \oplus \cdots \oplus p_k$, wobei die p_i Produktterme sind, in denen nur positive Variablen vorkommen. Jeder dieser Produktterme kann durch kaskadierte Partialproduktgeneratoren repräsentiert werden. Unsere ABL besteht dann aus diesen Partialproduktgeneratoren und einem Additionsnetzwerk mit einer Spalte und den p_i als Summanden. Diese ABL liegt in reduzierter Normalform vor. $\square$

Satz 5.2. *Für jede ABL existiert eine äquivalente ABL in reduzierter Normalform.*

Beweis: Da eine ABL eine Boolesche Funktion darstellt, folgt Satz 5.2 unmittelbar aus Satz 5.1. □
Aus dem Beweis dieses Satzes lässt sich prinzipiell direkt ein Algorithmus zur Normalisierung einer ABL ableiten. Allerdings ist der Umweg über die positive Davio-Normalform im Allgemeinen viel zu komplex. Während die ABLs, die sich aus der Davio-Normalform ergeben, nur Additionsnetzwerke mit einer Spalte enthalten, lässt sich die Schaltung oftmals wesentlich kompakter mit mehrspaltigen Additionsnetzwerken darstellen. Das folgende Beispiel veranschaulicht dies.

Beispiel 5.3. *Wir betrachten einen vorzeichenlosen 2-Bit Addierer* $(r_2, r_1, r_0) = (a_1, a_0) + (b_1, b_0)$. *Dieser lässt sich durch ein Additionsnetzwerk mit drei Spalten darstellen, d.h.* $A_0 = \{a_0, b_0\}$, $A_1 = \{a_1, b_1\}$ *und* $A_2 = \emptyset$. *Alle Gewichte sind* 1 *und die konstanten Offsets sind* 0. *Zuletzt sind die* r_i *die Spaltenergebnisse. Eine ABL, die nur aus diesem Additionsnetzwerk besteht, ist trivialerweise in reduzierter Normalform.*
Als nächstes betrachten wir die positiven Davio-Normalformen der Spaltensummen. Diese lauten:

$$r_0 = a_0 \oplus b_0,$$

$$r_1 = a_0 b_0 \oplus a_1 \oplus b_1$$

$$r_2 = b_1 a_1 \oplus a_1 a_0 b_0 \oplus a_0 b_1 b_0.$$

Damit erhalten wir eine ABL in reduzierter Normalform mit den Partialproduktgeneratoren $p_1 = a_0 b_0$, $p_2 = b_1 a_1$, $p_3 = p_1 a_1$, $p_4 = p_1 b_1$ *und einspaltigen Additionsnetzwerken* N^i *für die Ergebnisse* r_i *mit den Summanden* $A_0^0 = \{a_0, b_0\}$, $A_0^1 = \{a_1, b_1, p_1\}$ $A_0^2 = \{p_2, p_3, p_4\}$. *Beide ABLs sind in Abbildung 5.5 visualisiert.*

ABLs, die durch die Anwendung der Davio-Dekomposition erzeugt werden, enthalten nur einspaltige Additionsnetzwerke. Diese Beschränkung führt dazu, dass Überträge in eine höhere Spalte als partielle Produkte berechnet werden. Die Struktur der ursprünglichen ABL wird dabei vollständig zerstört. Darüber hinaus ergibt sich bei großen Additionsnetzwerken eine unüberschaubar große Anzahl partieller Produkte.

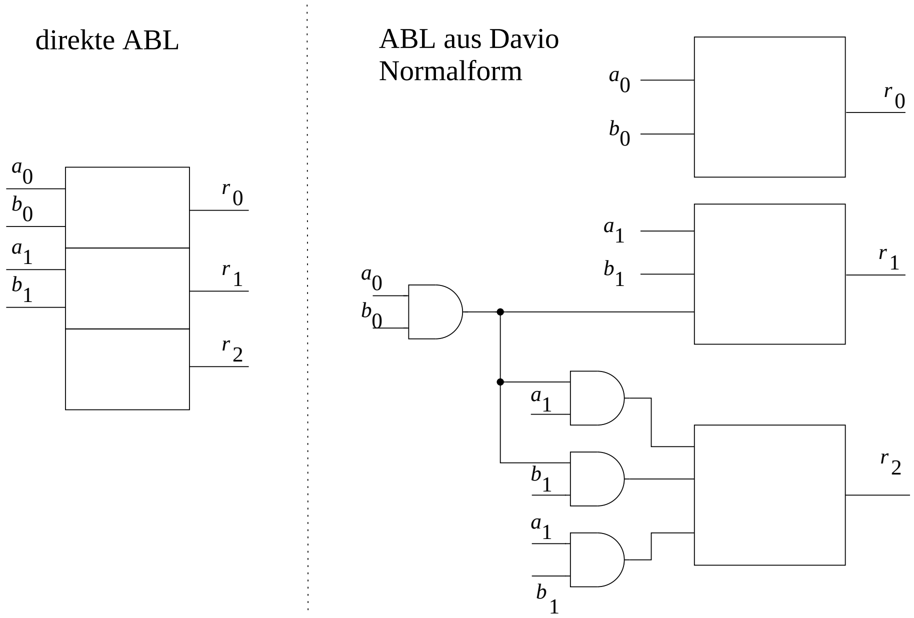


Abbildung 5.5: Zwei ABLs in reduzierter Normalform für einen 2-Bit-Addierer

Ziel muss es deshalb sein, die Struktur einer gegebenen ABL weitestgehend zu erhalten, um daraus die ABL in reduzierter Normalform zu gewinnen. Wir wenden uns deshalb nun einer heuristischen Methode zu, die aus einer gegebenen ABL durch lokale Transformationen schrittweise eine normalisierte ABL erzeugt. Wir führen dazu zunächst einige lokale Operationen ein.

Definition 5.12. *Zwei Additionsnetzwerke N_1 und N_2 können zusammengefasst werden genau dann, wenn ein Additionsnetzwerk N_3 existiert, so dass die ABLs $(V, E) = (\{N_3\}, \emptyset)$ und $(V', E') = (\{N_1, N_2\}, \{(N_1, N_2)\})$ äquivalent sind.*

N_1 und N_2 können also durch N_3 ersetzt werden, ohne dass sich die Funktion ändert, die durch die ABL dargestellt wird.

Lemma 5.3. *Gegeben seien zwei Additionsnetzwerke N_1, N_2 einer ABL mit $(N_1, N_2) \in E$. Ferner erzeugt N_1 keine Überträge in der obersten Spalte. N_1,N_2 können zusammengefasst werden, wenn eine der beiden folgenden Bedingungen erfüllt ist:*

1. *Es existiert eine Spalte $k \geq 0$ des Netzwerks N_1, so dass das Ergebnis r_j jeder Spalte $j \geq k$ als Summand in Spalte $j - k$ des Netzwerks N_2 eingeht.*
2. *Es existiert eine Spalte $k \geq 0$ des Netzwerks N_2, so dass das Ergebnis r_i jeder Spalte i des Netzwerks N_1 ein Summand der Spalte $i + k$ des Netzwerks N_2 ist.*

Die Spalten i des Netzwerks N_1 und j des Netzwerks N_2 können zusammengefasst werden, falls das Ergebnis von i ein Summand von j ist.

Beweis:
Wir behandeln den Fall 1, der zweite Fall kann analog bewiesen werden. Für $i = 1, 2, 3$ seien $A_i = \bigcap_{k=0,\ldots,n_i} A_{i,k}$ die Summanden, $w_{i,k}(k = 0, \ldots, n_i)$ die Gewichtsfunktionen, $c_{i,k}(k = 0, \ldots, n_i)$ die konstanten Offsets, $r_i = (r_{i,k}|k = 0, \ldots, n_i)$ die Ergebnisvektoren der Netzwerke N_1, N_2 und N_3.
Wir definieren N_3 wie folgt:

- $A_{3,i} = \begin{cases} A_{1,i} & , i < k \\ A_{1,i} \cup A_{2,i-k} & , k \leq i \leq n_1 \\ A_{2,i-k} & , n_1 < i \leq n_2 + k \end{cases}$
- $c_{3,i} = \begin{cases} c_{1,i} & , i < k \\ c_{1,i} + c_{2,i-k} & , k \leq i \leq n_1 \\ c_{2,i-k} & , n_1 < i \leq n_2 + k \end{cases}$
- $w_{3,i} = \begin{cases} w_{1,i} & , i < k \\ w_{1,i} + w_{2,i-k} & , k \leq i \leq n_1 \\ w_{2,i-k} & , n_1 < i \leq n_2 + k, \end{cases}$
  wobei im zweiten Fall die Gewichtsfunktionen um den Wert 0 für nicht gemeinsame Summanden ausgedehnt werden.
- $r_{3,i} = \begin{cases} r_{1,i} & , i < k \\ r_{2,i-k} & k \leq i \leq n_2 + k \end{cases}$

Durch diese Konstruktion ist unmittelbar sichergestellt, dass die $r_{1,i}$ und die $r_{3,i}$ für $i < k$ äquivalent sind. Bleibt zu zeigen, dass auch die $r_{3,i}$ und die $r_{2,i-k}$ für $k \leq i \leq n_2 + k$ äquivalent sind. Für $i \leq n_1$ ist auch dies unmittelbar klar. Für größere i nutzt man aus, dass N_1 keine Überträge in diese Spalten produziert. □
Zwei Additionsnetzwerke können also in linearer Zeit bezüglich der Anzahl der Summanden zusammengefasst werden. Dazu werden alle Summanden von N_1 in die entsprechenden Summandenmengen von N_2 eingefügt und die Gewichtsfunktion und konstanten Offsets entsprechend angepasst. Abbildung 5.6 visualisiert das Ergebnis dieses Prozesses.

Definition 5.13. *Seien P_1, P_2 und P_3 Partialproduktgeneratoren. P_1 und P_2 können genau dann zu P_3 zusammengefasst werden, wenn für alle partiellen Produkte von P_1 und P_2 ein äquivalentes partielles Produkt von P_3 existiert.*

Lemma 5.4. *Zwei Partialproduktgeneratoren P_1 und P_2 können zusammengefasst werden, wenn sie einen gemeinsamen Operanden haben.*

Beweis: Seien o, o_1 die Operanden von P_1 und o, o_2 die Operanden von P_2. Der Partialproduktgenerator P_3 mit den Operanden o und $o_3 = (o_1, o_2)$ erzeugt alle partiellen Produkte von P_1 und P_2. □

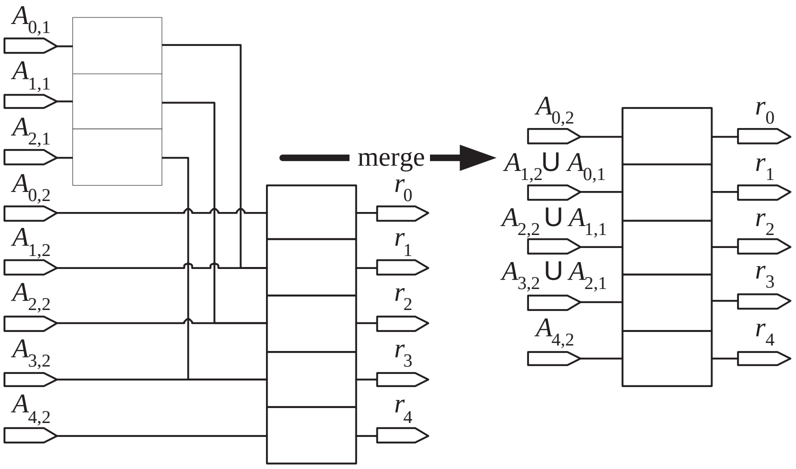


Abbildung 5.6: Zusammenfassen von Additionsnetzwerken

Beim Zusammenfassen von Partialproduktgeneratoren werden die nicht gemeinsamen Operanden konkateniert. Diese Operation kann in linearer Zeit bezüglich der Anzahl der Operanden durchgeführt werden. Der Vergleich der Operanden kann ebenfalls in linearer Zeit durchgeführt werden, wenn die Operanden sortiert sind. Letzteres kann bezüglich der Komplexität mit $O(n\ log\ n)$ abgeschätzt werden. Damit kann die gesamte Zusammenfassung in $O(n\ log\ n)$ Zeit durchgeführt werden.

Definition 5.14. *Sei P ein Partialproduktgenerator und N ein Additionsnetzwerk mit $(N, P) \in E$. P kann genau dann durch ein Additionsnetzwerk N verschoben werden, wenn ein Partialproduktgenerator P' und Additionsnetzwerke $N'_1 \dots N'_k$ existieren, so dass die Substitution von N und P durch $N'_1 \dots N'_k$ und P' eine äquivalente ABL erzeugt.*

Lemma 5.5. *Sei P ein Partialproduktgenerator und N ein Additionsnetzwerk mit $(N, P) \in E$ und $(N, X) \in E \Rightarrow X = P$. P kann durch N verschoben werden, wenn eine der Operandenmengen von P die Menge der Ergebnisse des Additionsnetzwerks enthält.*

Beweis:
Seien o_1, o_2 die Operanden der Wortebene von P. Ferner sei A die Menge der Summanden von N. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass o_2 die Ergebnisse des Additionsnetzwerks enthält. Sei $o_1 = (o_{1,0}, \dots, o_{1,n})$ und $o_2 = (o_{2,0}, \dots, o_{2,m})$. Die Menge O enthalte alle $o_{2,k}$, die kein Ergebnis von N sind. P' sei nun der Partialproduktgenerator $P' =< \{o_{1,k}a | k = 0 \dots n, a \in A\} \cup \{o_{1,k}o' | k = 0 \dots n, o' \in O\} >$.
Für $i = 0 \dots n$ sei N_i das Additionsnetzwerk mit Summanden $o_{1,i}a$. Die Gewichte der Summanden $o_{1,i}a$ in N_i entsprechen den Gewichten von a in der entsprechenden Spalte von N. Die konstanten Offsets von N_i und N seien ebenfalls identisch. Die Ergebnisse der N_i sind dann offensichtlich die partiellen Produkte von P, die von Ergebnissen von N abhängen. Alle anderen partiellen Produkte werden bereits in P' generiert. □

Bemerkung: Die Bedingung aus Lemma 5.5 kann durch Duplikation von Additionsnetzwerks (im Fanout- Fall) und Erweiterung der Partialproduktgenerators um fehlende Operanden immer erfüllt werden.
Abbildung 5.7 zeigt das Verschieben eines 2×1-Partialproduktgenerators durch einen Halbaddierer. Wenn n die Größe des unabhängigen Operanden ist, entstehen beim Verschieben eines Partialproduktgenerators durch ein Additionsnetzwerk n Duplikate des Additionsnetzwerks. Es zeigt sich, dass die entstehenden Additionsnetzwerke gewöhnlich sofort mit dem Additionsnetzwerk im Fanout des Partialproduktgenerators zusammengefasst werden können. Somit führt dieses temporäre Wachstum in der Regel nicht zu einem übermäßigem Speicherplatzbedarf.

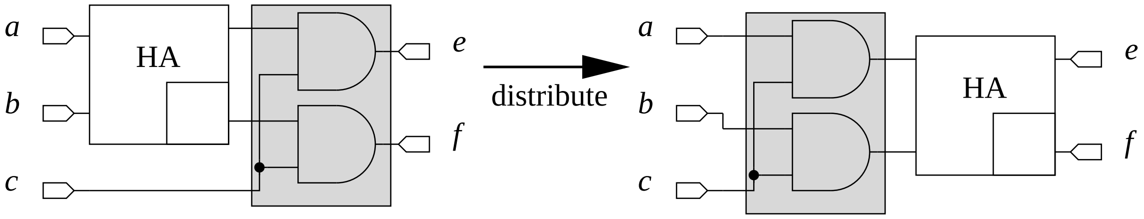


Abbildung 5.7: Verschieben von Partialproduktgeneratoren durch Additionsnetzwerke

Durch diese vier Operationen sind wir nun in der Lage, alle partiellen Produktgeneratoren vor die Additionsnetzwerke in ihrem Fanin zu verschieben und die dabei entstehenden neuen Additionsnetzwerke zusammenzufassen. Durch Entfernen äquivalenter partieller Produkte lässt sich daraus leicht eine reduzierte Normalform erzeugen.
Damit wird die reduzierte Normalform durch folgenden Prozess erzeugt:

1. Zusammenfassen von Additionsnetzwerken
2. Zusammenfassen von Partialproduktgeneratoren
3. Verschieben der Partialproduktgeneratoren
4. Zusammenfassen von Additionnetzwerken
5. Ermittlung äquivalenter partieller Produkte
6. Entfernen äquivalenter partieller Produkte aus verglichenen Additionsnetzwerken.

Dabei dient alleine Schritt 3 der Normalisierung. Die übrigen Schritte werden benötigt, um die ABL zu reduzieren. Aufgrund des Entfernens äquivalenter partieller Produkte aus Additionsnetzwerken ist die entstehende ABL lediglich bezüglich der Vergleicher äquivalent zur Ausgangs-ABL.

Das Verfahren wird nun anhand eines kleinen Beispiels illustriert. Dazu betrachten wir den Ausdruck $(a \cdot b) \cdot c$ und nehmen an, dass dieser auf der Bitebene spezifiziert ist. Alle Variablen seien zwei Bit breit. Abbildung 5.8 zeigt eine Implementierung auf der Bitebene, wobei dunkelgrau unterlegt bereits einige partielle Produkte zu Partialproduktgeneratoren und hellgrau unterlegt einige Addierer zu Additionsnetzwerken zusammengefasst sind.

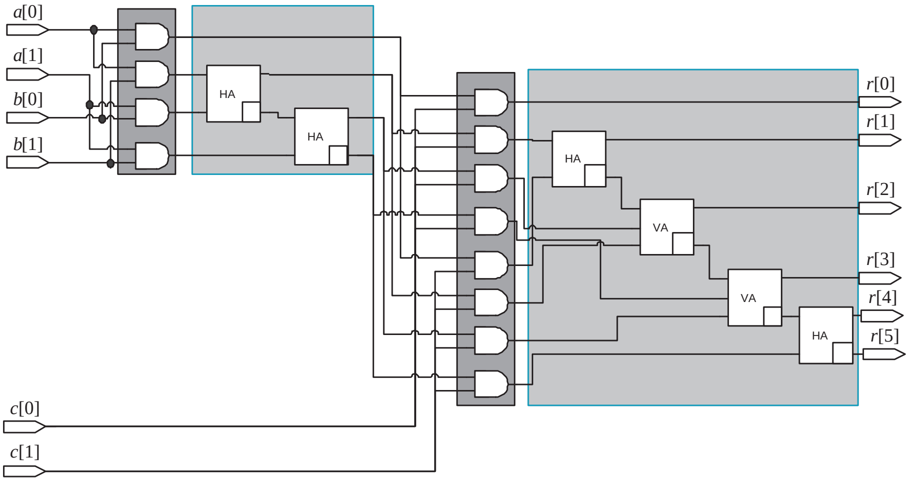


Abbildung 5.8: ABL nach Schritt 2

Das Ergebnis nach Verschieben des rechten Partialproduktgenerators ist in Abbildung 5.9 visualisiert.

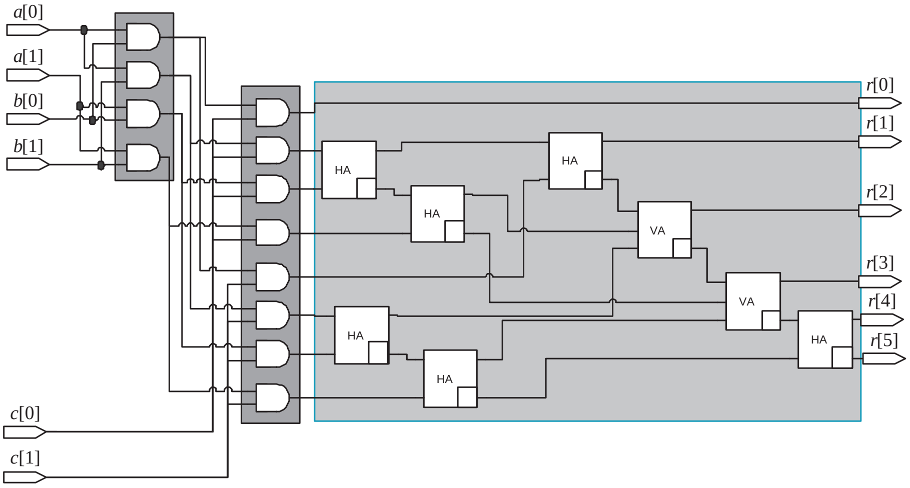


Abbildung 5.9: ABL nach Schritt 4

Man sieht leicht ein, dass eine andere Implementierung dieses Ausdrucks ebenfalls auf diese ABL führt.

5.3.1 Booth-Encoding: ein Anwendungsbeispiel für ABL Normalisierung

Eine in der Praxis häufig genutzte Architektur für Multiplizierer verwendet für die partiellen Produkte eine spezielle Kodierung, die so genannte Radix-4-Booth-Kodierung. Dabei werden partielle Produkte gebildet, die von mehreren Bits der beiden Operanden abhängen, wodurch die Anzahl dieser Produkte reduziert wird. In diesem Abschnitt werden wir zeigen, dass auch in diesem Fall durch eine geeignete arithmetische Modellierung der partiellen Produkte nach Booth-Encoding die Normalisierung angewendet werden kann. Gleichzeitig illustrieren wir damit das Verfahren anhand eines etwas komplexeren Beispiels. Dazu sei ein kombinatorisches Design zur Berechnung des Ausdrucks $a * b + c$ angenommen. Der Multiplizierer in diesem Design verwendet Radix-4-Booth-Encoding zur Berechnung der partiellen Produkte.
Für dieses Design soll die Eigenschaft *th_mac* aus Tabelle 5.2 nachgewiesen werden.

```
theorem th_mac;
  prove:
    at t: o==a*b+c;
end theorem;
```

Tabelle 5.2: Eigenschaft für eine Multiply-/Accumulate-Einheit

Um das Beispiel überschaubar zu halten, nehmen wir nun eine Operandenbreite von vier Bit an. Die ABL zu diesem Problem wird nun in Abbildung 5.10 dargestellt.

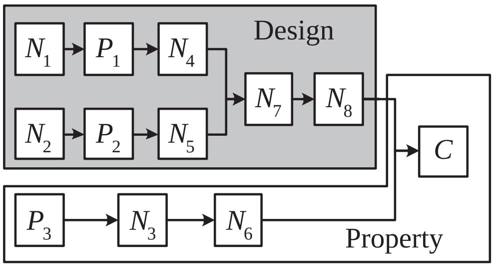


Abbildung 5.10: ABL einer 4-Bit-Instanz von der MAC-Eigenschaft

Hierbei wird der Multiplizierer $a*b$ aus dem Beweisziel der Eigenschaft durch den Partialproduktgenerator P_3 und das Additionsnetzwerk N_3 modelliert. Das Additionsnetzwerk N_6 implementiert die Addition von c zu diesem Multiplikationsergebnis.
Schwieriger gestaltet sich die Modellierung der partiellen Produkte aus der Implementierung. Es ist wohlbekannt [Kor98, Par00], dass die partiellen Produkte p_i, die nach dem Booth-Verfahren kodiert sind, für gerade Bitbreiten der folgenden Gleichung genügen:

$$p_i = (-2a[i] + a[i-1] + a[i-2]) * b \text{ für } i = 2k+1 \leq n.$$

In der ABL aus Abbildung 5.10 wird der Ausdruck

$$p_1 = (-2a[1] + a[0]) * b$$

durch die Additionsnetzwerke N_1 und N_4 sowie den Partialproduktgenerator P_1 implementiert. Analog implementieren N_2, P_2 und N_5 den Ausdruck

$$p_3 = (-2a[3] + a[2] + a[1]) * b.$$

Man beachte, dass Boolesche Methoden auch für größere Bitbreiten sehr leicht die Äquivalenz dieser Ausdrücke mit einer Booleschen Implementierung nachweisen können, da der linke Operand stets die Bitbreite 2 hat.
Die Addition der beiden Produktvektoren p_1 und p_3 wird schließlich mit N_7 modelliert. Um das Beispiel überschaubar zu halten, gehen wir hier davon aus, dass diese Addition auf der Wortebene kodiert wurde und deshalb auf diese Weise modelliert werden kann. Bei einer Implementierung auf der Bitebene würde N_7 lediglich in eine Reihe von Halbaddierern und Volladdierern zerlegt. Diese Zerlegung würde dann im ersten Schritt des Normalisierungsverfahren rückgängig gemacht. Damit erhielten wir für den zweiten Schritt des Verfahrens die gleiche ABL.

Generator	o_1	o_2	Ergebnisse
P_1	b	r_1	$pp_1[0..11]$
P_2	b	r_2	$pp_2[0..11]$
P_3	a	b	$pp_3[0..15]$

Tabelle 5.3: Paritialproduktgeneratoren aus Abbildung 5.10

Abschließend müssen noch die Ergebnisse der Additionsnetzwerke N_6 und N_8 verglichen werden. Dazu enthält die ABL den Vergleicher C,
In den Tabellen 5.3 und 5.4 findet man die exakte Spezifikation aller Additionsnetzwerke bzw. Partialproduktgeneratoren der ABL.

Netzwerk	Summanden	Ergebnisse
N_1	$A_0 = \{a[0]\}, A_1 = \{-a[1]\}, A_2 = \emptyset$	$r_1[0..2]$
N_2	$A_0 = \{a[1], a[2]\}, A_1 = \{-a[3]\}, A_2 = \emptyset$	$r_2[0..2]$
N_3	$A_0 = \{a[0]b[0]\}, A_1 = \{a[0]b[1], a[1]b[0]\}$ $A_2 = \{a[0]b[2], a[1]b[1], a[2]b[0]\}$ $A_3 = \{-a[0]b[3], a[1]b[2], a[2]b[1], -a[3]b[0]\}$ $A_4 = \{-a[1]b[3], a[2]b[2], -a[3]b[1]\}$ $A_5 = \{-a[2]b[3], -a[3]b[2]\},$ $A_6 = \{a[3]b[3]\}, A_7 = \emptyset$	$r_3[0..7]$
N_4	$A_0 = \{pp_1\}, A_1 = \{pp_1[1], pp_1[4]\},$ $A_2 = \{pp_1[2], pp_1[5], -pp_1[8]\}$ $A_3 = \{-pp_1[3], pp_1[6], -pp_1[9]\}$ $A_4 = \{-pp_1[7], -pp_1[10]\}$ $A_5 = \{pp_1[11]\}, A_6 = \emptyset$	$r_4[0..6]$
N_5	$A_0 = \{pp_2[0]\}, A_1 = \{pp_2[1], pp_2[4]\}$ $A_2 = \{pp_2[2], pp_2[5], -pp_2[8]\}$ $A_3 = \{-pp_2[3], pp_2[6], -pp_2[9]\}$ $A_4 = \{-pp_2[7], -pp_2[10]\}$ $A_5 = \{pp_2[11]\}, A_6 = \emptyset$	$r_5[0..6]$
N_6	$A_0 = \{r_3[0], c[0]\}, A_1 = \{r_3[1], c[1]\},$ $A_2 = \{r_3[2], c[2]\}, A_3 = \{r_3[3], c[3]\},$ $A_4 = \{r_3[4], c[4]\}, A_5 = \{r_3[5], c[5]\},$ $A_6 = \{r_3[6], c[6]\}, A_7 = \{r_3[7], c[7]\},$	$r_6[0..7]$
N_7	$A_0 = \{r_4[0]\}, A_1 = \{r_4[1]\},$ $A_2 = \{r_4[2], r_5[0]\}, A_3 = \{r_4[3], r_5[1]\},$ $A_4 = \{r_4[4], r_5[2]\}, A_5 = \{r_4[5], r_5[3]\},$ $A_6 = \{r_4[6], r_5[4]\}, A_7 = \{r_5[5]\},$	$r_7[0..7]$
N_8	$A_0 = \{r_7[0], c[0]\}, A_1 = \{r_7[1], c[1]\},$ $A_2 = \{r_7[2], c[2]\}, A_3 = \{r_7[3], c[3]\},$ $A_4 = \{r_7[4], c[4]\}, A_5 = \{r_7[5], c[5]\},$ $A_6 = \{r_7[6], c[6]\}, A_7 = \{r_7[7], c[7]\},$	$r_8[0..7]$

Tabelle 5.4: Additionsnetzwerke aus Abbildung 5.10

Im ersten Schritt der Normalisierung werden nun die Additionsnetzwerke N_4, N_5, N_7 mit N_8 sowie das Netzwerk N_3 mit N_6 zusammengefasst. Als Ergebnis erhalten wir die ABL aus Abbildung 5.11.

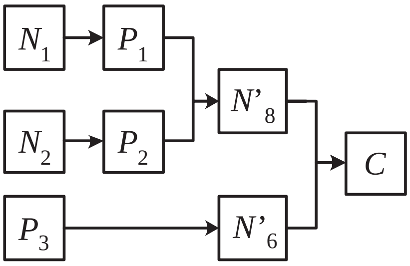


Abbildung 5.11: ABL nach Zusammenführung von Additionsnetzwerken

Die Summanden von N_6' und N_8' sind in Tabelle 5.5 angegeben.

Netzwerk	Summanden
N_6'	$A_0 = \{c[0], pp_3[0]\}$, $A_1 = \{c[1], pp_3[1], pp_3[4]\}$ $A_2 = \{c[2], pp_3[2], pp_3[5], pp_3[8]\}$ $A_3 = \{c[3], -pp_3[3], pp_3[6], pp_3[9], -pp_3[12]\}$ $A_4 = \{c[4], -pp_3[7], pp_3[10], -pp_3[13]\}$ $A_5 = \{c[5], -pp_3[11], -pp_3[14]\}$, $A_6 = \{c[6], pp_3[15]\}, A_7 = \{c[7]\}$,
N_8'	$A_0 = \{c[0], pp_1[0]\}$, $A_1 = \{c[1], pp_1[1], pp_1[4]\}$, $A_2 = \{c[2], pp_1[2], pp_1[5], -pp_1[8], pp_2[0]\}$, $A_3 = \{c[3], -pp_1[3], pp_1[6], -pp_1[9], pp_2[1], pp_2[4]\}$, $A_4 = \{c[4], -pp_1[7], -pp_1[10], pp_2[2], pp_2[5], -pp_2[8]\}$, $A_5 = \{c[5], pp_1[11], -pp_2[3], pp_2[6], -pp_2[9]\}$, $A_6 = \{c[6], -pp_2[7], -pp_2[10]\}$, $A_7 = \{c[7], pp_2[11]\}$,

Tabelle 5.5: Additionsnetzwerke aus Abbildung 5.11

Da in der hier vorliegenden ABL keine Partialproduktgeneratoren zusammengefasst werden können, wird das Verfahren nun mit Schritt drei fortgesetzt. In diesem Schritt des Verfahrens werden nun die Partialproduktgeneratoren P_1 und P_2 durch die Netzwerke N_1 bzw. N_2 verschoben. Damit erhält man die ABL aus Abbildung 5.12.

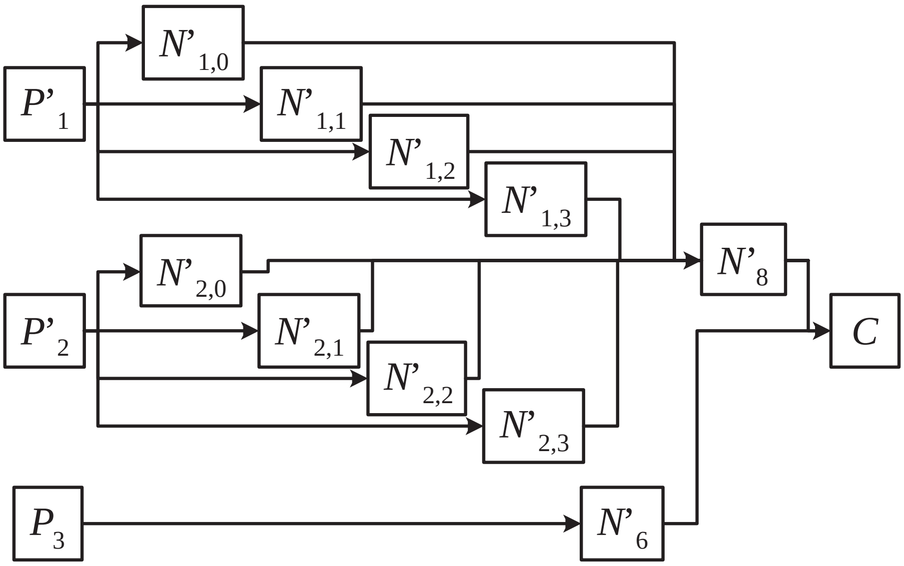


Abbildung 5.12: ABL nach dem Verschieben von Partialproduktgeneratoren

Die Struktur der neuen Partialproduktgeneratoren P'_1, P'_2 geht aus Tabelle 5.6 hervor. Die neu entstandenen Additionsnetzwerke werden in Tabelle 5.7 angegeben.

Generator	o_1	o_2	Ergebnisse
P'_1	b	$a[0:1]$	$pp'_1[0..7]$
P'_2	b	$a[1:3]$	$pp'_2[0..11]$

Tabelle 5.6: Partialproduktgeneratoren aus Abbildung 5.12

Netzwerk	Summanden	Ergebnisse
$N'_{1,0}$	$A_0 = \{pp'_1[0]\}, A_1 = \{-pp'_1[4]\}, A_2 = \emptyset$	$pp_1[0,4,8]$
$N'_{1,1}$	$A_0 = \{pp'_1[1]\}, A_1 = \{-pp'_1[5]\}, A_2 = \emptyset$	$pp_1[1,5,9]$
$N'_{1,2}$	$A_0 = \{pp'_1[2]\}, A_1 = \{-pp'_1[6]\}, A_2 = \emptyset$	$pp_1[2,6,10$
$N'_{1,3}$	$A_0 = \{pp'_1[3]\}, A_1 = \{-pp'_1[7]\}, A_2 = \emptyset$	$pp_1[3,7,11$
$N'_{2,0}$	$A_0 = \{pp'_2[0], pp'_2[4]\}, A_1 = \{-pp'_2[8]\}, A_2 = \emptyset$	$pp_2[0,4,8]$
$N'_{2,1}$	$A_0 = \{pp'_2[1], pp'_2[5]\}, A_1 = \{-pp'_2[9]\}, A_2 = \emptyset$	$pp_2[1,5,9]$
$N'_{2,2}$	$A_0 = \{pp'_2[2], pp'_2[6]\}, A_1 = \{-pp'_2[10]\}, A_2 = \emptyset$	$pp_2[2,6,10]$
$N'_{2,3}$	$A_0 = \{pp'_2[3], pp'_2[7]\}, A_1 = \{-pp'_2[11]\}, A_2 = \emptyset$	$pp_2[3,7,11]$

Tabelle 5.7: Additionsnetzwerke aus Abbildung 5.12

Die neuen Additionsnetzwerke $N'_{i,k}$ können nun im vierten Schritt mit dem Additionsnetzwerk N'_8 zusammengefasst werden. Danach erhält man die in Abbildung 5.13 dargestellte ABL. Als Ergebnis dieser Zusammenfassung entsteht das Netzwerk N''_8, dessen Summanden aus Tabelle 5.8 hervorgehen.

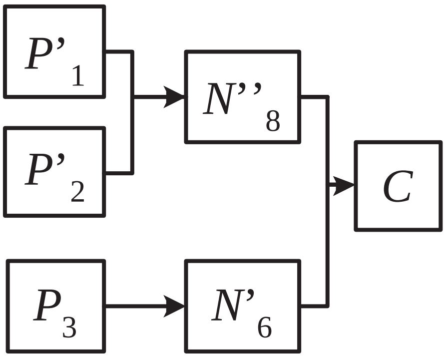


Abbildung 5.13: ABL nach dem Zusammenfassen von Additionsnetzwerken

Netzwerk	Summanden
N''_8	$A_0 = \{c[0], pp'_1[0]\}$, $A_1 = \{c[1], pp'_1[1], -pp'_1[4]\}$, $A_2 = \{c[2], pp'_1[2], -pp'_1[5], pp'_2[0], pp'_2[4]\}$, $A_3 = \{c[3], -pp'_1[3], -pp'_1[6], pp'_2[1], pp'_2[5], -pp'_2[8]\}$, $A_4 = \{c[4], pp'_1[7], pp'_2[2], pp'_2[6], -pp'_2[9], \}$, $A_5 = \{c[5], -pp'_2[3], -pp'_2[7], -pp'_2[10]\}$, $A_6 = \{c[6], pp'_2[11]\}$, $A_7 = \{c[7]\}$,

Tabelle 5.8: Additionsnetzwerke zu Abbildung 5.13

Der folgende Schritt fünf ermittelt nun äquivalente partielle Produkte. Als Ergebnis erhält man die folgenden Äquivalenzen:

$$pp'_1[0..7] \equiv pp_3[0, 4, 8, 12, 1, 5, 9, 13]$$

und

$$pp'_2[0..11] \equiv pp_3[1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15].$$

Damit reduziert sich die ABL wie in Abbildung 5.14 dargestellt.
Die Summanden der modifizierten Netzwerke findet man in Tabelle 5.9.
Anschließend wird im letzten Schritt des Verfahrens festgestellt, dass beide Netzwerke paarweise identische Summanden haben. Damit ist die Eigenschaft bewiesen.

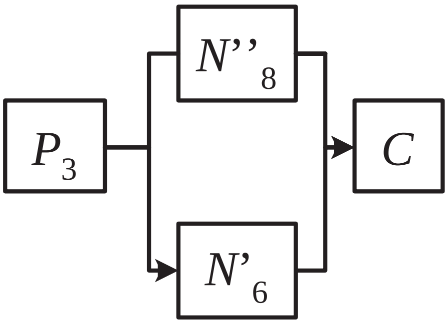


Abbildung 5.14: ABL nach Identifikation äquivalenter partieller Produkte

Netzwerk	Summanden
N'_6	$A_0 = \{c[0], pp_3[0]\}$, $A_1 = \{c[1], pp_3[4], -pp_3[1]\}$, $A_2 = \{c[2], pp_3[8], -pp_3[5], pp_3[1], pp_3[2]\}$, $A_3 = \{c[3], -pp_3[12], -pp_3[9], pp_3[5], pp_3[6], -pp_3[3]\}$, $A_4 = \{c[4], pp_3[13], pp_3[9], pp_3[10], -pp_3[7]\}$, $A_5 = \{c[5], -pp_3[13], -pp_3[14], -pp_3[11]\}$, $A_6 = \{c[6], pp_3[15]\}$, $A_7 = \{c[7]\}$,
N''_8	$A_0 = \{c[0], pp_3[0]\}$, $A_1 = \{c[1], pp_3[4], pp_3[1]\}$, $A_2 = \{c[2], pp_3[8], pp_3[5], pp_3[2]\}$, $A_3 = \{c[3], -pp_3[12], pp_3[9], pp_3[6], -pp_3[3]\}$, $A_4 = \{c[4], -pp_3[13], pp_3[10], -pp_3[7]\}$, $A_5 = \{c[5], -pp_3[14], -pp_3[11]\}$, $A_6 = \{c[6], pp_3[15]\}$, $A_7 = \{c[7]\}$,

Tabelle 5.9: Additionsnetzwerke aus Abbildung 5.14

5.3.2 Anwendungsbereich der ABL-Normalisierung

Wie bei jedem heuristischen Verfahren stellt sich natürlich auch bei der hier vorgestellten ABL-Normalisierung die Frage, in welchem Anwendungsbereich das Verfahren effizient eingesetzt werden kann.

Zunächst stellen wir fest, dass ABL Beschreibungen zu Booleschen Netzwerken degenerieren können, indem UND-Gatter durch Partialproduktgeneratoren oder Übertragsausgänge von Halbaddierern implementiert werden. Der Summenausgang des Halbaddierers liefert das XOR-Gatter, aus dem durch Einsetzen der Konstante 1 ein Inverter gebildet werden kann. Also bilden die Konstrukte der ABL eine Basis der Booleschen Algebra, auf die jede Gatternetzliste abgebildet werden kann.

Es ist sofort klar, dass man von der Normalisierung keine Effizienz für eine ABL erwarten kann, die auf diese Weise aus einer Gatternetzliste gewonnen wurde. Die Stärke des Verfahrens liegt genau darin, dass die arithmetischen Informationen, die in der Regel auf der RT-Ebene noch vorhanden sind, ausgenutzt werden. In der folgenden Analyse wird deshalb untersucht, in wie weit sich die ABL-Normalisierung auf industriell relevante Probleme anwenden lässt.

Als ersten Schritt in diese Richtung wollen wir uns zunächst dem Spezialfall von Satz 5.1 zuwenden, dass eine ABL aus einem arithmetischen Ausdruck erzeugt wurde, der die Operationen der Wortebene $\{+,-,*\}$ benutzt. Für diesen Spezialfall geben wir einen konstruktiven Beweis an. Dabei wird eine ABL erzeugt, die ein Additionsnetzwerk für das gesamte Ergebnis des arithmetischen Ausdrucks enthält. Es sei daran erinnert, dass im Beweis von Satz 5.1 für jedes Bit des Ergebnisses ein einspaltiges Additionsnetzwerk erzeugt wurde.

Satz 5.3. *Für jeden arithmetischen Ausdruck der Wortebene, der aus den Operanden* $\{+,-,*\}$ *gebildet werden kann, existiert eine ABL in reduzierter Normalform mit genau einem Additionsnetzwerk.*

Beweis: Jeder arithmetische Ausdruck e kann als Summe von Produkten $e = \sum_i \lambda_i \prod_k v_{i,k}$ mit $\lambda_i \in \{-1,1\}$ dargestellt werden. Die einzelnen Variablen $v_{i,k}$ können wiederum als gewichtete Summe ihrer Bits $v_{i,k}[l]$ dargestellt werden:

$$v_{i,k} = -2^{b_{i,k}} v_{i,k}[b_{i,k}] + \sum_{l=0\ldots b_{i,k}-1} 2^l v_{i,k}[l]$$

wobei $b_{i,k}$ der Bitbreite der Variable $v_{i,k}$ entspricht. Damit erhalten wir

$$e = \sum_i \lambda_i \prod_k (-2^{b_{i,k}} v_{i,k}[b_{i,k}] + \sum_{l=0\ldots b_{i,k}-1} 2^l v_{i,k}[l]).$$

Durch Anwendung des Distributivgesetzes erhalten wir eine gewichtete Summe von Produkten der Bits der Variablen mit ganzzahligen Gewichten δ_j:

$$e = \sum_j \delta_j \prod v_{i_j,k_j}[l_j].$$

Jedes dieser Produkte $\prod v_{i_j,k_j}[l_j]$ kann durch Kaskadierung von Partialproduktgeneratoren erzeugt werden. Die gewichtete Summe kann durch ein Additionsnetzwerk A berechnet werden, wobei zunächst alle Partialproduktgeneratoren Summanden der untersten Spalte sind und mit δ_j gewichtet werden. □

Bemerkung:

Die Gewichte in der untersten Spalte des Additionsnetzwerk A aus diesem Beweis können natürlich wieder auf $-1, 1$ und 0 normiert werden. Summanden mit betragsmäßig größeren Gewichten gehen dann zusätzlich in höhere Spalten ein.

Die Struktur der im Beweis zu Satz 5.3 erzeugten ABL unterscheidet sich erheblich von der Struktur der in Satz 5.1 erzeugten ABL. Statt ein Additionsnetzwerk pro Spalte des Ergebnisses zu erzeugen, wird hier lediglich ein Additionsnetzwerk für das gesamte Ergebnis erzeugt. Die dabei verwendeten Operationen entsprechen dabei dem Verschieben von Partialproduktgeneratoren bzw. dem Zusammenfassen von Additionsnetzwerken. Damit garantiert dieser Satz, dass die ABL Normalisierung für solche Instanzen erfolgreich durchgeführt werden kann.

Sind die Instanzen jedoch auf der arithmetischen Bitebene gegeben, ist diese Aussage nicht mehr allgemein gültig. Es ist im Gegenteil möglich, eine ABL für einen Addierer zu konstruieren, die sich durch paarweises Zusammenfassen der Additionnetzwerke nicht in ein einziges äquivalentes Additionsnetzwerk überführen lässt.

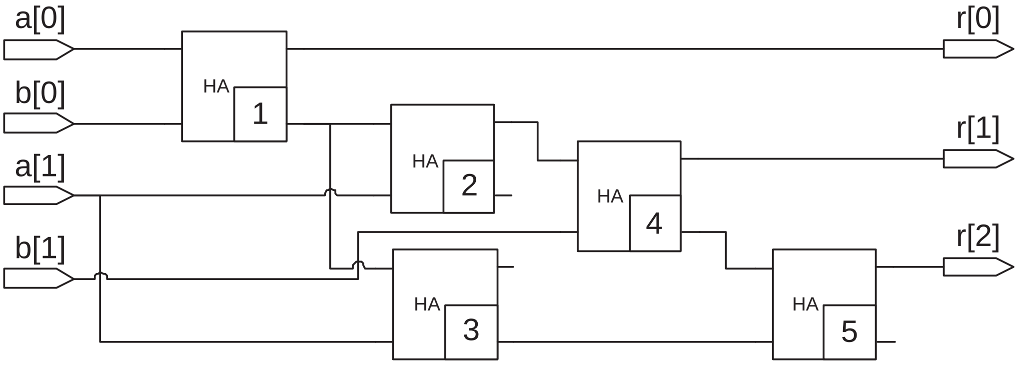


Abbildung 5.15: Halbaddierer-Netzwerk eines 2-Bit Addierers

Abbildung 5.15 zeigt eine Netzliste aus Halbaddierern, die einen 2-Bit Addierer implementiert. Jeder Halbaddierer ist also ein Additionsnetzwerk mit zwei Spalten und zwei Summanden in der unteren Spalte. Wie man sieht, haben die Halbaddierer 2 und 3 identische Eingänge. Allerdings wird jeweils nur ein Ausgang für weitere Berechnungen genutzt.

In dieser Situation können die Halbaddierer 4 und 5 zu einem neuen Additionsnetzwerk N zusammengefasst werden. Anschließend kann aber weder der Halbaddierer 2 noch der Halbaddierer 3 mit N zusammengefasst werden. In einem Fall fehlt die Summe, im anderen den Übertrag als Summand in N.

Tatsächlich wird durch die Duplikation des Halbaddierers seine arithmetische Natur zerstört. Der Halbaddierer 2 degeneriert zu einem XOR-Gatter, Halbaddierer 3 entspricht einem UND-Gatter. Dies läuft auf die gleiche Problematik hinaus wie das weiter oben beschriebene Problem, eine ABL-Beschreibung aus einer Gatternetzliste zu gewinnen. Ein Verfahren zur Extraktion der ABL aus einer Gatternetzliste wurde in [SK01] vorgeschlagen.

Ist das zu verifizierende Design durch eine Gatternetzliste beschrieben, kann dieses Verfahren eingesetzt werden, um zunächst eine ABL-Beschreibung zu erzeugen, wie sie für das hier vorgeschlagene Verfahren zur Eigenschaftsprüfung erforderlich ist.

In der Eigenschaftsprüfung konzentrieren wir uns deshalb auf RT-Designs, in denen die arithmetischen Blöcke mindestens auf der arithmetischen Bitebene beschrieben sind. Da die Eigenschaftsprüfung in der Regel zu einem sehr frühen Zeitpunkt im Designflow eingesetzt wird, ist davon auszugehen, dass Optimierungen, die den Einsatz einer Beschreibung durch Gatter erfordern, erst später vorgenommen und dann mit einem Equivalence Checker überprüft werden.

Die explizite Implementierung der arithmetischer Funktionen auf der arithmetischen Bitebene ermöglicht es dem Designer, die volle Kontrolle über die Topologie der Additionsnetzwerke zu behalten. Um optimale Performanz zu erreichen, wird der Designer weiterhin stets bemüht sein, keine Redundanz in diese Netzwerke einzubauen.

Die folgende Definition charakterisiert nun eine Klasse von Implementierungen von Additionsnetzwerken, die insofern ohne Redundanz sind, dass alle Zwischenergebnisse auch weiterverwendet werden.

Definition 5.15. *Ein Menge von Additionsnetzwerken* $\{N_1, \ldots, N_k\}$ *implementiert ein Additionsnetzwerk* N *genau dann, wenn die ABL bestehend aus* N *zu der ABL bestehend aus* $\{N_1, \ldots, N_k\}$ *äquivalent ist.*

Eine Implementierung heißt redundanzfrei, wenn alle Ergebnisse r_i *eines der Netzwerke* N_j*, die kein Ergebnis von* N *und nicht konstant* 0 *sind, in mindestens einem anderen Netzwerk als Summand benutzt werden.*

Bemerkung: Für jede redundanzfreie Implementierung $\{N_1, \ldots, N_k\}$ eines Additionsnetzwerks N können wir annehmen, dass genau ein Additionsnetzwerk N_j die Ergebnisse von N als Ergebnisse hat. Andernfalls fügen wir einen Puffer N_{k+1} für diese Ergebnisse hinzu. Wir gehen im Folgenden stets von einer solchen Implementierung aus.

Problematisch für die Theorie stellen sich die Zwischensummen in einer solchen Implementierung heraus, die konstant den Wert 0 annehmen. Die Addition einer solchen Konstante mit einem beliebigen anderen Signal ändert offensichtlich nichts an der Funktion der ABL. Dadurch können Implementierungen erzeugt werden, die sich durch paarweises Zusammenführen der Additionsnetzwerke nicht mehr in ein Additionsnetzwerk zurückführen lassen.

Beispiel 5.4. *Wir betrachten das Additionsnetzwerk* N *mit drei Spalten, den Ergebnissen* r_0, r_1, r_2 *und einem Summanden* a *in Spalte* 1. N *kann durch die ABL aus Abbildung 5.16 implementiert werden. Da das Summenbit des Halbaddierers stets den Wert 0 annimmt,*

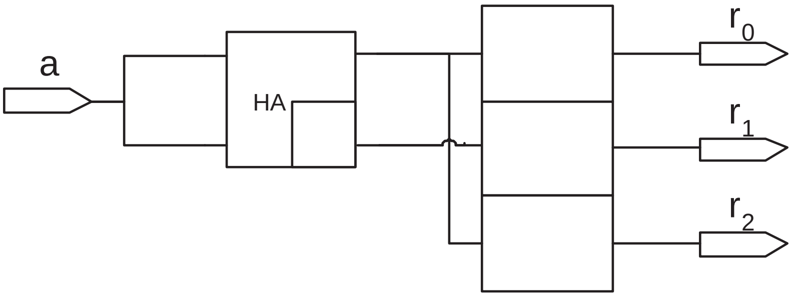


Abbildung 5.16: Ausnutzung von Konstanten in einer ABL

wird dies ebenfalls benutzt, um das Ergebnis r_2 *zu berechnen. Dies führt dazu, dass man die beiden Additionsnetzwerke mit unserer Heuristik nicht mehr zusammenfassen kann.*

Ziel der folgenden Analyse ist es nun, ein möglichst einfaches Kriterium für die Verwendung von konstanten Signalen (mit Wert 0) festzulegen, das die Anwendbarkeit der ABL-Normalisierung sicherstellt. Für die Praxis sind dabei zwei Fälle von Bedeutung.

a) Der Designer überblickt, dass eine Zwischensumme stets den Wert Null hat. Deshalb wird er dieses Signal bei der weiteren Addition nicht mehr berücksichtigen.

b) Ist dies nicht der Fall, wird der Designer die Zwischensumme in die entsprechende Spalte des Ergebnisses aufaddieren.

Bei diesen Entscheidungen des Designers können natürlich Fehler auftreten. Diese führen dann in der Regel zu einem Fehlverhalten der Schaltung und können, wie bereits in Abschnitt 5.1 erläutert, leicht mit SAT-Techniken gefunden werden.

Für die Normalisierung können sich zwei verschiedene Konsequenzen ergeben. Im einen Fall, z.B. wenn ein primärer Summand in eine falsche Spalte addiert wird, werden die Additionsnetzwerke zu einem Additionsnetzwerk zusammengefasst. Allerdings werden hier Summanden den falschen Spalten zugeordnet sein und damit bei der abschließenden Vereinfachung nicht vollständig eliminiert. Dadurch wird natürlich die Generierung des Gegenbeispiels vereinfacht.

Schwerwiegender ist der Fall, dass durch den Fehler die vollständige Zusammenführung der Additionsnetzwerke verhindert wird. Dadurch wird der abschließende Vereinfachungsschritt des Normalisierungsverfahrens in Mitleidenschaft gezogen. Man ist wieder gezwungen, auf SAT-Techniken zurückzugreifen, um das Gegenbeispiel zu generieren.

Im Folgenden steht nun der Fall der gültigen Eigenschaft, in dem der Designer die oben genannten Schritte korrekt durchgeführt hat, im Fokus.

In einer redundanzfreien Implementierung $\{N_1, \ldots, N_k\}$ eines Additionsnetzwerks N kann man jedem internen Signal, d.h., den Ausgängen r_j der Additionsnetzwerke N_i die kleinste Ergebnisspalte $l(r_j)$ von N_k im transitiven Fanout zuordnen, also

- Für N_k gilt $r_j \mapsto l(r_j) := j$ für alle j.
- Für $N_i, i < k$ seien $r'_1, \ldots, r'_m$ die Spalten der Additionsnetzwerke, in die r_j als Summand eingeht. Dann gelte:

$$r_j \mapsto l(r_j) := min\{l(r_1), \ldots, l(r_m), +\infty\}$$

Der Wert $+\infty$ zeigt dabei an, dass ein solches Signal nicht mehr verwendet wird. Nach Definition 5.15 muss dieses Signal dann konstant den Wert 0 annehmen.
Für den Spezialfall einer Implementierung von N durch Halbaddierer können wir nun beweisen, dass durch iteriertes paarweises Zusammenführen von Additionsnetzwerken das Originalnetzwerk N erzeugt wird. Einen Halbaddierer modellieren wir dabei mit einem Additionsnetzwerk mit zwei Spalten und maximal zwei Summanden, die beide in die untere Spalte 0 eingehen. Einer oder beide Summanden können dabei auch durch konstante Offsets (0 oder 1) ersetzt werden.

Satz 5.4. *Sei N ein Additionsnetzwerk. Ferner sei $\{N_1, \ldots, N_{k+1}\}$ eine redundanzfreie Implementierung von N durch eine Halbaddierer-Netzliste $\{N_1, \ldots, N_k\}$ und einen Puffer N_{k+1}. Für die Ergebnisse $r_{i,0}, r_{i,1}$ der $N_i, i = 1 \ldots k$ gelte stets*

$$l(r_{i,0}) = l(r_{i,1}) - 1 \textit{ oder } l(r_{i,0}) = \infty \textit{ oder } l(r_{i,1}) = \infty. \tag{5.2}$$

Dann gilt: Durch iteriertes paarweises Zusammenführen der N_i oder äquivalenter N'_i erhält man ein zu N äquivalentes Netzwerk.

Beweis: Es genügt zu zeigen, dass man durch iteriertes paarweises Zusammenführen zu einem einzigen Additionsnetzwerk N' gelangt. Da das Zusammenführen eine Äquivalenztransformation ist, ist N' zu N äquivalent. Aufgrund von Lemma 5.1 können wir ohne Beschränkung der Allgemeinheit davon ausgehen, dass alle Gewichte von N auf 1 normiert sind.
Darüber hinaus gehen wir davon aus, dass die Halbaddierer der Implementierung durch Additionsnetzwerke $N_1, \ldots, N_k$ in topologischer Ordnung modelliert sind. Der Puffer für die Ergebnisse von N sei durch ein Additionsnetzwerk N_{k+1} modelliert, welches die r_i sowohl als Summand als auch als Ergebnis der Spalte i hat.
Damit ist sofort klar, dass $N_1, \ldots, N_k$ ebenfalls eine Implementierung von N ist. Wir beweisen den Satz nun durch Induktion über k. Im Fall $k = 1$ haben wir $N = N_1$. Deshalb kann N_1 mit dem Puffer N_2 zusammengefasst werden, und wir erhalten erneut N. Für den Induktionsschritt nehmen wir die Gültigkeit des Satzes für $k > 0$ an.
Sei nun N durch Halbaddierer $N_1, \ldots, N_{k+1}$ und einen Puffer N_{k+2} implementiert. Wir können topologische Sortierung der N_j annehmen. Deshalb müssen die Summanden a_1, a_2 von N_1 primäre Summanden von N sein.

Im Fall 1: $a := a_1 = a_2$ nimmt der Summeneingang r_0 von N_1 konstant den Wert Null an. Wir streichen dieses Signal aus allen Additionsnetzwerken $N_2, \ldots, N_{k+1}$, in denen es als Summand verwendet wird. Ferner ersetzen wir r_1 überall durch a. Damit ist klar, dass die so konstruierten Netzwerke $N'_2, \ldots, N'_{k+1}$ ebenfalls eine Implementierung von N darstellen. Es ist ferner klar, dass auch $N'_2, \ldots, N'_k$ Halbaddierer sind. Damit können diese Netzwerke nach Induktionshypothese zu einem Additionsnetzwerk N', das zu N äquivalent ist, zusammengefasst werden.

Im Fall 2: $a := a_1 = \overline{a_2}$ gilt $r_0 = 1$ und $r_1 = 0$ für die beiden Ausgänge von N_1. Analog zum ersten Fall kann N_1 eliminiert werden.

Ähnlich verläuft die Argumentation, wenn einer oder beide Summanden konstant sind.

Bleibt also nur noch der Fall, dass beide Summanden unabhängig und nicht konstant sind. Damit sind auch die Ergebnisse r_0, r_1 von N_1 nicht konstant. Nach Definition 5.15 werden die Ergebnisse r_0, r_1 von N_1 als Summanden in anderen Netzwerken $N_j, N_{j'}$ $(j, j' \in \{2 \ldots k+2\})$ verwendet. Nun sei i die unterste Spalte von N_{k+2}, die im transitiven Fanout von r_0 liegt. Also implementieren $N_2, \ldots, N_{k+2}$ das Additionsnetzwerk N', das sich durch Subtraktion von a_1, a_2 aus Spalte i von N und Hinzufügen von r_0 zu Spalte i und r_1 zu Spalte $i+1$ von N ergibt. Gegebenenfalls müssen bei N' die Gewichte auf 1 normalisiert werden.

Nach Induktionshypothese $N_2, \ldots, N_{k+1}$ erhalten wir durch iteriertes paarweises Zusammenführen ein zu N' äquivalentes Netzwerk N''. Dieses können wir aufgrund der Konstruktion von N' mit N_1 zusammenführen und erhalten ein zu N äquivalentes Netzwerk.
$\square$

Bemerkungen:

- Die Einführung des Puffers in Satz 5.4 wird eigentlich nur benötigt, um den Sonderfall zu behandeln, dass eine Spalte von N keinen Übertrag generiert. In diesem Fall können durch das paarweise Zusammenführen voneinander unabhängige Netzwerke entstehen, die nicht mehr zusammengeführt werden können. Die experimentelle Erfahrung zeigt, dass es in der Praxis reicht, einen Puffer für die Eingänge der Vergleicher zur ABL hinzuzufügen.

- Die Bedingung 5.2 fordert, dass konstante Signale mit Wert 0 entweder gar nicht oder in die ”richtige” Spalte aufsummiert werden. Außerdem wird gefordert, dass Summe und Übertrag eines Halbaddierers stets in aufeinanderfolgende Spalten des Netzwerks eingehen.

Damit ist nun gezeigt, dass das Normalisierungsverfahren unter den weiter oben genannten Randbedingungen korrekte Implementierungen eines Additionsnetzwerks zusammenfassen kann.

Abschließend bleibt anzumerken, dass die Sätze 5.3 und 5.4 sowohl die Intuition als auch die experimentellen Ergebnisse stützen, dass die Normalisierung für praktische Designs effektiv anwendbar ist, die sinnvolle arithmetische Funktionen berechnen.

5.4 Behandlung der Kontrolllogik

In Abschnitt 5.1 wurde für die Eigenschaftsprüfung von arithmetischen Schaltungen die Behandlung von Kontrolllogik als zweiter wesentlicher Aspekt identifiziert. Dieser Abschnitt beschreibt deshalb, wie die Kontrolllogik, welche die arithmetischen Bestandteile eines Problems der Eigenschaftsprüfung separiert, eliminiert werden kann. Ziel ist es dabei, das Problem auf ein arithmetisches Kernproblem zu reduzieren.

5.4.1 Konstantenpropagierung

Wie bereits in Kapitel 3 Abschnitt 3.3.3 beschrieben wurde, haben BIMC-Probleme in der Regel die Form:

$$p = \overline{(a \Rightarrow c)} \cdot f'_{R,k}.$$

Dabei ist a eine Boolesche Funktion, die die Voraussetzungen (engl. assumptions) beschreibt, unter denen ein bestimmtes Beweisziel c (engl. commitment) Gültigkeit besitzt. $f'_{R,k}$ ist bekanntlich das iterative Schaltungsmodell der Länge k. Sowohl Voraussetzung als auch Beweisziel sind in der Regel Konjunktionen aus Teilvoraussetzungen bzw. Teilzielen von der Form

$$a = a_0 \cdot \ \ldots \ \cdot a_n \text{ bzw. } c = c_0 \cdot \ \ldots \ \cdot c_m.$$

Damit p erfüllt ist, muss also $a_i = 1$ für alle i und $c_i = 0$ für ein i gelten. Dies ist der erste Schritt der Konstantenpropagierung. Die Konstanten $a_i = 1$ propagieren nun in der Regel weiter, z.T. sogar sehr tief in das iterierte Schaltungsmodell hinein.

Die Eigenschaft in Tabelle 5.1 auf Seite 66 macht beispielsweise die Annahme, dass zu Beginn des betrachteten Zeitfensters ein bestimmter Multiplikationsbefehl zur Bearbeitung durch die Integer-Pipeline eines Prozessors ansteht. Nach der Konstantenpropagierung sind dann eine ganze Reihe interner Kontrollsignale spezifiziert.

5.4.2 Logiksubstitution

Bekanntlich ist die Konstantenpropagierung unvollständig. Auch nach der Propagierung bleibt deshalb bei industriellen Problemen stets ein signifikanter Logikanteil zwischen den einzelnen arithmetischen Blöcken des Problems erhalten. Allerdings kann man mit einem SAT-Solver in der Regel leicht zeigen, dass diese Logik unter den Voraussetzungen der Eigenschaft transparent wird, d.h., die Eingänge eines arithmetischen Blocks sind äquivalent zu den Ausgängen eines anderen. Darüber hinaus kann ebenfalls leicht gezeigt werden, dass bestimmte Kontrollsignale unter denselben Voraussetzungen konstante Werte annehmen.

Diese Äquivalenzen und Konstanten können dann genutzt werden, um Substitutionen durchzuführen, die die verbliebene Kontrolllogik eliminieren. Es gilt der folgende Satz:

Satz 5.5. *Seien* $f_1, f_2, a, b : \mathbb{B}^n \to \mathbb{B}$ *Boolesche Funktionen, wobei* a, b *interne Signale von* f_1 *und* f_2 *seien, d.h., es existieren Boolesche Funktionen* $f_1', f_2' : \mathbb{B}^{n+2} \to \mathbb{B}$ *mit* $f_i(x_1 \dots x_n) = f_i'(x_1 \dots x_n, a(x_1 \dots x_n) b(x_1 \dots x_n))$. *Zuletzt sei* $f_2^{b \leftarrow a} : \mathbb{B}^n \to \mathbb{B}$ *mit* $f_2^{b \leftarrow a}(x_1 \dots x_n) = f_2'(x_1 \dots x_n, a(x_1 \dots x_n) a(x_1 \dots x_n))$ *die Funktion, die durch Substitution von* b *durch* a *entsteht. Dann gilt:*

$$(f_1 \Rightarrow (a = b)) \Rightarrow (f_1 \Rightarrow (f_2 = f_2^{b \leftarrow a}))$$

Beweis:

$$\begin{aligned} & (f_1 \Rightarrow (a = b)) \Rightarrow (f_1 \Rightarrow (f_2 = f_2^{b \leftarrow a})) \\ \Leftrightarrow\ & \overline{(\overline{f_1} + (a = b))} + (\overline{f_1} + (f_2 = f_2^{b \leftarrow a})) \\ \Leftrightarrow\ & f_1 \overline{(a = b)} + \overline{f_1} + (f_2 = f_2^{b \leftarrow a}) \\ \Leftrightarrow\ & \overline{f_1} + \overline{(a = b)} + (f_2 = f_2^{b \leftarrow a}) \\ \Leftrightarrow\ & \overline{f_1} + ((a = b) \Rightarrow (f_2 = f_2^{b \leftarrow a})) \\ \Leftrightarrow\ & 1 \end{aligned}$$

□

Dieser Satz besagt, dass man interne Signale in Beweiszielen substituieren darf, wenn deren Äquivalenz aus den Voraussetzungen folgt. In der Voraussetzung ist diese Substitution im Allgemeinen nicht erlaubt.

Beispiel 5.5. *Wir betrachten die arithmetischen Ausdrücke:* $c = ab, d = a + b, e = c + d$ *Dann gilt* $(d = a) \Rightarrow (e = a)$. *Nach Satz 5.5 genügt es,* $(d = a) \Rightarrow (e = d)$ *zu zeigen. Dagegen gilt weder* $(a = a) \Rightarrow (e = a)$ *noch* $(d = d) \Rightarrow (e = d)$.

5.4.3 Abstraktion bei gemischten Problemen

Das Beweisziel einer Eigenschaft lässt sich in der Praxis jedoch nicht immer durch einen reinen arithmetischen Ausdruck charakterisieren. Also enthalten diese Eigenschaften einen kleinen Kontrollanteil. Beispielhaft sei hier die Multiplikation mit Saturierung genannt. Wenn das Ergebnis einer Multiplikation den darstellbaren Zahlenbereich verlässt, möchte man bei einigen DSP-Anwendungen, dass das Ergebnis einen bestimmten Wert, z.B. den größten bzw. den kleinsten darstellbaren Wert, annimmt. In solchen Fällen erweist sich die Normalisierung als taugliches Werkzeug, um Wortebenenabstraktionen für die arithmetischen Bestandteile des Problems zu erhalten. Eine solche Situation wird in den Abbildungen 5.17 und 5.18 illustriert.

Wir betrachten ein Design bestehend aus einem hart kodierten arithmetischen Block und einer Überlauferkennung für die Saturierung. Aus Performanzgründen schätzt man mögliche Überläufe gerne bereits aufgrund der Operanden (Eingänge des arithmetischen Blocks) ab. Damit sind nicht mehr alle Ausgänge des Arithmetikblocks notwendig, um die korrekte Saturierungsbedingung zu berechnen. Dadurch verkürzt sich der kritische Pfad für die Timing-Analyse der Schaltung.

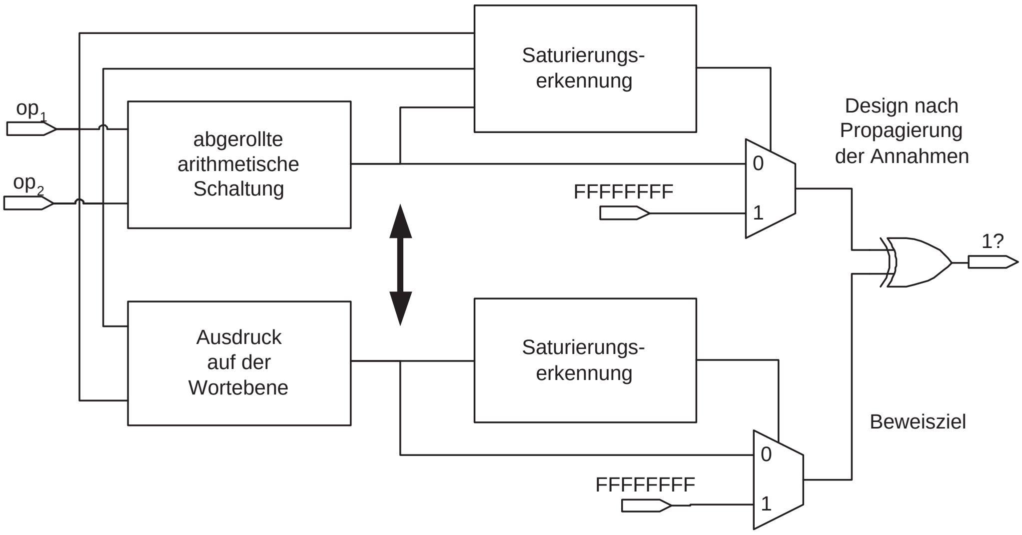


Abbildung 5.17: SAT-Instanz vor Abstraktion

Natürlich wird man in der Eigenschaft die Saturierungsbedingung allein aufgrund des unsaturierten exakten Ergebnisses berechnen. Also muss der SAT-Solver in dieser Situation implizit die Äquivalenz des Arithmetikblocks und des Ausdrucks aus der Eigenschaft nachweisen. Dies führt natürlich zu untragbaren Rechenzeiten.

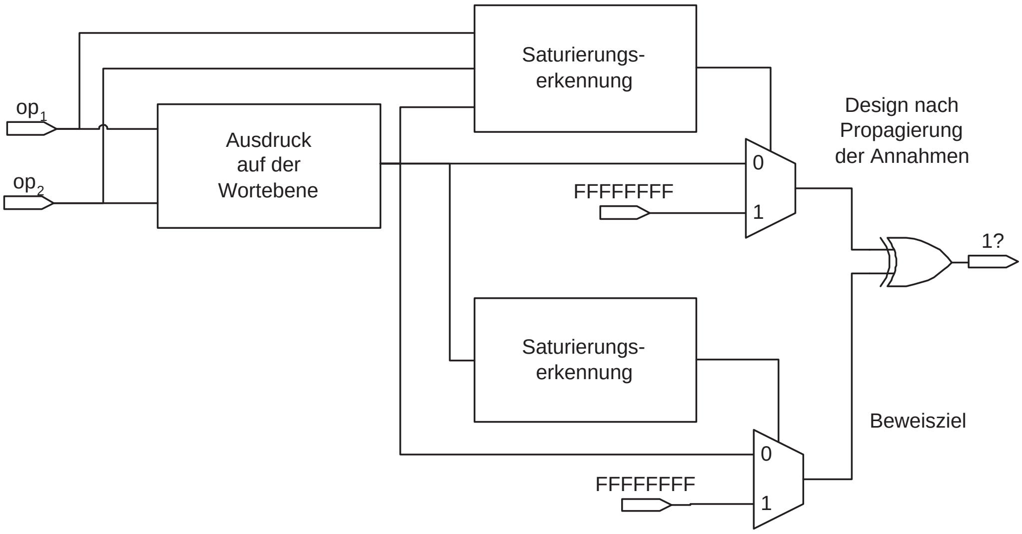


Abbildung 5.18: SAT-Instanz nach Abstraktion

Durch Normalisierung verifiziert man nun leicht, dass der arithmetische Block den arithmetischen Ausdruck aus der Eigenschaft implementiert. Diese Information kann dann genutzt werden, um das Problem zu vereinfachen, indem der Ausdruck auf der Wortebe-

ne, wie er in der Eigenschaft vorliegt, als Treiber für beide Signale verwendet wird. Der Arithmetikblock wird also durch den Ausdruck *abstrahiert*. Diese abstrahierte Instanz kann dann einem SAT-Solver zur Lösung übergeben werden.
Experimentell hat sich gezeigt, dass für vorzeichenlose Multiplikation nach der Abstraktion sogar 32-Bit-Instanzen dieser Problemklasse noch effizient mit einem SAT-Solver gelöst werden können. Natürlich können an dieser Stelle auch hybride Solver [ABC$^+$02, CK03] eingesetzt werden. Bei vorzeichenbehafteter Arithmetik kann man ähnlich vorgehen. Allerdings kann das Ergebnis des Ausdrucks hier an beiden Enden aus dem darstellbaren Zahlenbereich herausfallen. Experimentell hat sich gezeigt, dass der SAT-Solver dabei in Komplexitätsprobleme gerät, falls negative Operanden auftreten. Lösen lässt sich dieses Problem durch eine Fallunterscheidung anhand der Vorzeichen der Operanden und geeignete Abstraktionen für jeden Fall. Dieses Vorgehen soll exemplarisch für vorzeichenbehaftete Multiplikation mit Saturierung diskutiert werden.
Wir betrachten die Operanden a und b dieser Multiplikation. Ferner sei $i = a \cdot b$ das unsaturierte Zwischenergebnis. Wir verwenden die Abstraktionen

$$i = \begin{cases} a \cdot b, \text{ falls } a \geq 0, b \geq 0 \\ (-a) \cdot (-b), \text{ falls } a < 0, b < 0 \\ -(a \cdot (-b)), \text{ falls } a \geq 0, b < 0 \\ -((-a) \cdot b), \text{ falls } a < 0, b \geq 0 \end{cases} .$$

Man beachte, dass hier zur Darstellung von $-a$ bzw. $-b$ jeweils eine Vorzeichenerweiterung durchgeführt werden muss. In jedem der vier Fälle sind die Operanden des Multiplizierers stets positiv und damit auch das Ergebnis der Multiplikation. In den ersten beiden Fällen kann es aufgrund eines Überlaufs und in den anderen Fällen aufgrund eines Unterlaufs zur Saturierung kommen.
Im Folgenden wird für i eine Bitbreite von $2n + 1$ und für a und b eine Bitbreite von n angenommen. Ferner wird angenommen, dass das Ergebnis auf n Bit saturiert werden soll. In den ersten beiden Fällen lässt sich der Überlauf durch $i[2n, \ldots, n-1] \neq 0$ charakterisieren. Der Unterlauf in den anderen Fällen wird analog durch $\overline{i[2n, \ldots, n-1]} \neq 0$ charakterisiert.
Liegt das Ergebnis innerhalb des darstellbaren Zahlenbereichs, führt diese Charakterisierung in allen Fällen dazu, dass durch Propagierung die Bits $2n, \ldots, n$ des Multiplizierers den Wert 0 erhalten. Damit müssen alle partiellen Produkte dieser Spalten verschwinden. Nach dieser Propagierung sind sehr viele Bits von a und b bereits stark eingeschränkt, so dass dem SAT-Solver die Lösung gelingt.
Bei der Verwendung der einfachen Abstraktion $i = a \cdot b$ ist dies nicht immer der Fall. Für $a < 0, b \geq 0$ z.B. können diese Bits entweder alle verschwinden oder alle den Wert 1 annehmen. Damit ist durch Propagierung keines der partiellen Produkte und damit auch kein zusätzliches Bit der Eingänge festgelegt.
Zusammenfassend lässt sich damit festhalten, dass durch die geeignete Wahl der Abstraktionen auch im vorzeichenbehafteten Fall die korrekte Saturierung mit einem SAT-Solver nachgewiesen werden kann.

5.5 Experimente

Das vorgeschlagen Normalisierungsverfahren wurde in einem Prototypen (ABLProp) implementiert und über eine Dateischnittstelle mit dem industriellen Eigenschaftsprüfer Gateprop der Firma OneSpin Solutions GmbH gekoppelt.
Bei der experimentellen Evaluierung des Verfahrens wurden zwei Schritte durchgeführt.

a) Die Skalierbarkeit des Verfahrens bezüglich der Operandengrößen wurde untersucht. Dazu wurde die ABL einiger Ausdrücke generiert und anschließend normalisiert.

b) Ein skalierbarer Multiplizierer mit Booth-Encoding wurde als weiteres Beispiel implementiert. Dieser ist vollständig auf der Bitebene implementiert. Die partiellen Produkte werden dabei mit den arithmetischen Operationen von Verilog berechnet, um deren arithmetische Spezifikation in die Normalisierung einbeziehen zu können.

c) Das Verfahren wurde genutzt, um die Integer-Pipeline des Prozessors Tricore 2 der Firma Infineon zu verifizieren. Erstmals ist es gelungen, die Abarbeitung von Befehlen mit Multiplikation vollständig zu verifizieren.

5.5.1 Skalierbarkeit

Die Skalierbarkeit des Verfahrens wird untersucht, indem die folgenden Gleichungen in eine ABL überführt und anschließend normalisiert werden:

$$ab + ac = a(b + c) \tag{5.3}$$

$$a(bc) = (ab)c \tag{5.4}$$

$$\begin{aligned} a * b = {} & a[0 : n-1] * b[0 : n-1] \\ & +2^n a[n : 2n-1] * b[0 : n-1] \\ & +2^n a[0 : n-1] * b[n : 2n-1] \\ & +2^{2n} a[n : 2n-1] * b[n : 2n-1] \end{aligned} \tag{5.5}$$

Dabei wird angenommen, dass alle Operationen auf der Wortebene spezifiziert werden. Es sei angemerkt, dass Gleichung (5.5) entsteht, wenn ein n-Bit Multiplizierer verwendet wird, um in vier Takten das Ergebnis einer $2n$-Bit Multiplikation zu berechnen. Die CPU-Zeiten für den Beweis dieser Gleichungen werden in Tabelle 5.10 für verschiedene Bitbreiten angegeben. Dabei wird in der ersten Spalte stets die Bitbreite eines einzelnen Operanden angegeben. Für Gleichung (5.5) entspricht die angegebene Bitbreite dem Wert für n.
Eine genauere Analyse dieser Zahlen bestätigt die Skalierbarkeit der Methode bezüglich der Operandenbitbreite. Besonders wichtig ist allerdings, dass der Ansatz geeignet ist, sehr große arithmetische Blöcke zu verifizieren. So wurden z.B. alle 192 Ausgabe-Bits der größten Instanz von Gleichung 5.4 in ca. 40 Minuten CPU-Zeit verifiziert. Dies zeigt, dass auch sehr große Operandenbreiten noch zu akzeptablen Rechenzeiten führen.

Bitbreite	CPU-Zeit [s] für Gleichung		
	(5.3)	(5.4)	(5.5)
4	0.01	$<$0.01	$<$0.01
8	0.03	0.02	0.02
16	0.43	0.57	0.09
24	3.21	5.59	0.23
32	15.92	30.06	0.46
48	193.46	386.41	1.37
64	1151.86	2303.47	3.16

Tabelle 5.10: Experimentelle Ergebnisse zur Skalierbarkeit

5.5.2 Booth Encoding

In Abschnitt 5.3.1 wurde das Normalisierungsverfahren exemplarisch für eine MAC-Unit mit einem Multiplizierer, dessen partiellen Produkte durch Booth-Encoding berechnet werden, durchgeführt. Dort wurde das Additionsnetzwerk des Multiplizierers allerdings bereits als vollständig zusammengefasst modelliert, was im ersten Schritt des Verfahrens ohnehin erreicht wird. Das dies tatsächlich auch für große realistische Beispiele funktioniert, soll in diesem Abschnitt gezeigt werden.
Dazu wurden auf der Bitebene eine Reihe von Verilog-Designs für Multiplizierer mit Radix-4-Booth-Encoding für verschiedene Bitbreiten generiert und anschließend mit dem Normalisierungsverfahren verifiziert. In Abbildung 5.19 ist exemplarisch der Code für den 4x4 Multiplizierer wiedergegeben.

Die Tabelle 5.11 zeigt die Laufzeiten der ABL-Normalisierung für einige Instanzen bis zu 56 Bit Operandenbreite. Außer der 4-Bit-Instanz ist keine dieser Instanzen mit SAT-Techniken zu lösen. Um die Instanzen nicht nur über die Bitbreite der Operanden zu charakterisieren, sind in den Spalten 2 und 3 der Tabelle außerdem die Anzahl der Additionsnetzwerke und partiellen Produktgeneratoren in der ABL angegeben, mit der das Normalisierungsverfahren startet.

```
// generated by: ./gen_mult_adder
-a 4 -b 4 -e 2 -o mult_ub_4x4.v -v 1

// unsigned 4x4 Booth-encoded multiplier with primary inputs: a, b
// primary output: p


module mult_ub_4x4(p, a, b);

// The input-vectors of the circuit #0:
 input [3:0] a;
 input [3:0] b;

// The only output-vector:
 output [7:0] p;

// The partial products of the circuit #0:
 wire [7:0] pp_0_0;
 wire [5:0] pp_0_1;
 wire [3:0] pp_0_2;

// The adders of the circuit #0:
 wire [1:0] ha_0_0;
 wire [1:0] ha_0_1;
 wire [1:0] fa_0_0;
 wire [1:0] fa_0_1;
 wire [1:0] fa_0_2;
 wire [1:0] fa_0_3;
 wire [1:0] fa_0_4;
 wire [1:0] fa_0_5;
 wire [1:0] fa_0_6;
 wire [1:0] fa_0_7;

// *****  Partial products *****
 assign pp_0_0 = (-({b[1],1’b0}) + b[0])*a[3:0];
 assign pp_0_1 = (-({b[3],1’b0}) + b[2] + b[1])*a[3:0];
 assign pp_0_2 = b[3]*a[3:0];
// *****  Partial products *****

// *****  Adders *****
 assign ha_0_0 = ({1’b0,pp_0_1[0]} + {1’b0,pp_0_0[2]});
 assign fa_0_0 = ({1’b0,ha_0_0[1]} + {1’b0,pp_0_0[3]} + {1’b0,pp_0_1[1]});
 assign fa_0_1 = ({1’b0,fa_0_0[1]} + {1’b0,pp_0_0[4]} + {1’b0,pp_0_1[2]});
 assign ha_0_1 = ({1’b0,pp_0_2[0]} + {1’b0,fa_0_1[0]});
 assign fa_0_2 = ({1’b0,fa_0_1[1]} + {1’b0,ha_0_1[1]} + {1’b0,pp_0_0[5]});
 assign fa_0_3 = ({1’b0,fa_0_2[0]} + {1’b0,pp_0_1[3]} + {1’b0,pp_0_2[1]});
 assign fa_0_4 = ({1’b0,fa_0_2[1]} + {1’b0,fa_0_3[1]} + {1’b0,pp_0_0[6]});
 assign fa_0_5 = ({1’b0,fa_0_4[0]} + {1’b0,pp_0_1[4]} + {1’b0,pp_0_2[2]});
 assign fa_0_6 = ({1’b0,fa_0_4[1]} + {1’b0,fa_0_5[1]} + {1’b0,pp_0_0[7]});
 assign fa_0_7 = ({1’b0,fa_0_6[0]} + {1’b0,pp_0_1[5]} + {1’b0,pp_0_2[3]});
// *****  Adders *****

// ***   outputs   ***
assign p[0] = pp_0_0[0];
assign p[1] = pp_0_0[1];
assign p[2] = ha_0_0[0];
assign p[3] = fa_0_0[0];
assign p[4] = ha_0_1[0];
assign p[5] = fa_0_3[0];
assign p[6] = fa_0_5[0];
assign p[7] = fa_0_7[0];

endmodule
```

Abbildung 5.19: Sourcecode eines 4x4 Multiplizierers auf der arithmetischen Bitebene

Instanz	# ANW	#PPG	CPU-Zeit(s)
mult_ub_4x4	29	4	0,008
mult_ub_8x8	103	6	0,064
mult_ub_12x12	225	8	0,396
mult_ub_16x16	395	10	1,564
mult_ub_20x20	613	12	5,164
mult_ub_24x24	879	14	13,116
mult_ub_28x28	1193	16	29,941
mult_ub_32x32	1555	18	61,787
mult_ub_40x40	2423	22	214,897
mult_ub_48x48	3483	26	587,612
mult_ub_56x56	4735	30	1485,790

Tabelle 5.11: Ergebnisse für Multiplizierer mit Booth-Encoding

5.5.3 Industrielle Anwendung

Die industrielle Verwendbarkeit des Verfahrens wurde anhand der Integer-Pipeline des Infineon Tricore 2 Prozessors [Tri] evaluiert. Dabei handelt es sich um einen superskalaren 32-Bit-Prozessor der nächsten Generation mit zusätzlichen DSP-Funktionalitäten. Da dieser Prozessor in Zukunft auch für sehr kritische Anwendungen, z.B. Steuergeräte im Auto, eingesetzt werden soll, wurde bereits im Entwurf auf die bestmögliche Qualität geachtet. Aus diesem Grund wurde der gesamte Entwurf formal mit dem industriellen Eigenschaftsprüfer Gateprop verifiziert.

Unter anderem waren dabei die Befehle zur Integer-Arithmetik auf korrekte Abarbeitung zu überprüfen. Dabei hat sich gezeigt, dass ein Eigenschaftsprüfer, der dem Stand der Technik entspricht, lediglich bei den Befehlen mit Multiplikation in Komplexitätsprobleme gerät. Nun kann die o.g. Pipeline allerdings über 600 verschiedene Varianten von Multiply-/Accumulate-Befehlen ausführen. Bis auf einige Spezialfälle, bei denen ein Operand der Multiplikation einen festen Wert hat, konnte keine dieser Varianten mit dem SAT-basierten Ansatz verifiziert werden. Auch Wortebenen-Solver sind an dieser Stelle nicht einsetzbar, da fast die gesamte Arithmetik auf der Bit-Ebene hart kodiert ist.

Die Vielfalt an Befehlsvarianten entsteht durch die Kombination verschiedener Bitbreiten mit verschiedenen Adressierungsarten und verschiedenen Verfahren zur Nachbearbeitung des Ergebnisses. So können gleichermaßen vorzeichenbehaftete und vorzeichenlose Multiplikationen auf 32- sowie 16-Bit-Operanden durchgeführt werden. Das Ergebnis kann wahlweise abgeschnitten, saturiert oder gerundet werden. Zwei 16-Bit-Operationen können zudem parallel abgearbeitet werden.

Alle diese Varianten werden für alle Befehle auf der gleichen Hardware ausgeführt. Aus diesem Grund ist die Arithmetik von einer komplexen Kontrolllogik durchzogen. Zuletzt sind die Zwischenergebnisse an den Pipeline-Grenzen nicht auf der Wortebene zu modellieren, wodurch die Instanzen für ILP- und CLP-Ansätze nicht zugänglich sind.

Das Normalisierungswerkzeug ABLProp, welches im Rahmen dieser Forschungsarbeit

entwickelt wurde, ist mit dem state-of-the-art Eigenschaftsprüfer Gateprop der Firma OneSpin Solutions GmbH integriert worden. Letzterer wurde als Front-End verwendet, um Design und Eigenschaft in eine Netzliste von Bitvektorfunktionen zu übersetzen. Diese Netzliste enthält dedizierte Signale p und a, mit denen Assumptions und Commitments referenziert werden. Es ist also die Implikation $a \Rightarrow p$ nachzuweisen.
Dazu weisen wir zunächst dem Signal a den Wert 1 zu und propagieren diesen in die Netzliste. Die arithmetischen Bitvektorfunktionen der so modifizierten Netzliste werden nun in eine ABL übersetzt und diese dann normalisiert. Zum Abschluss der Modellgenerierung wird die normalisierte ABL zusammen mit den nicht arithmetischen Bitvektorfunktionen in eine KNF übersetzt und dem SAT-Solver Minisat [ES03] übergeben. Dieser Ablauf wird in Abbildung 5.20 visualisiert.

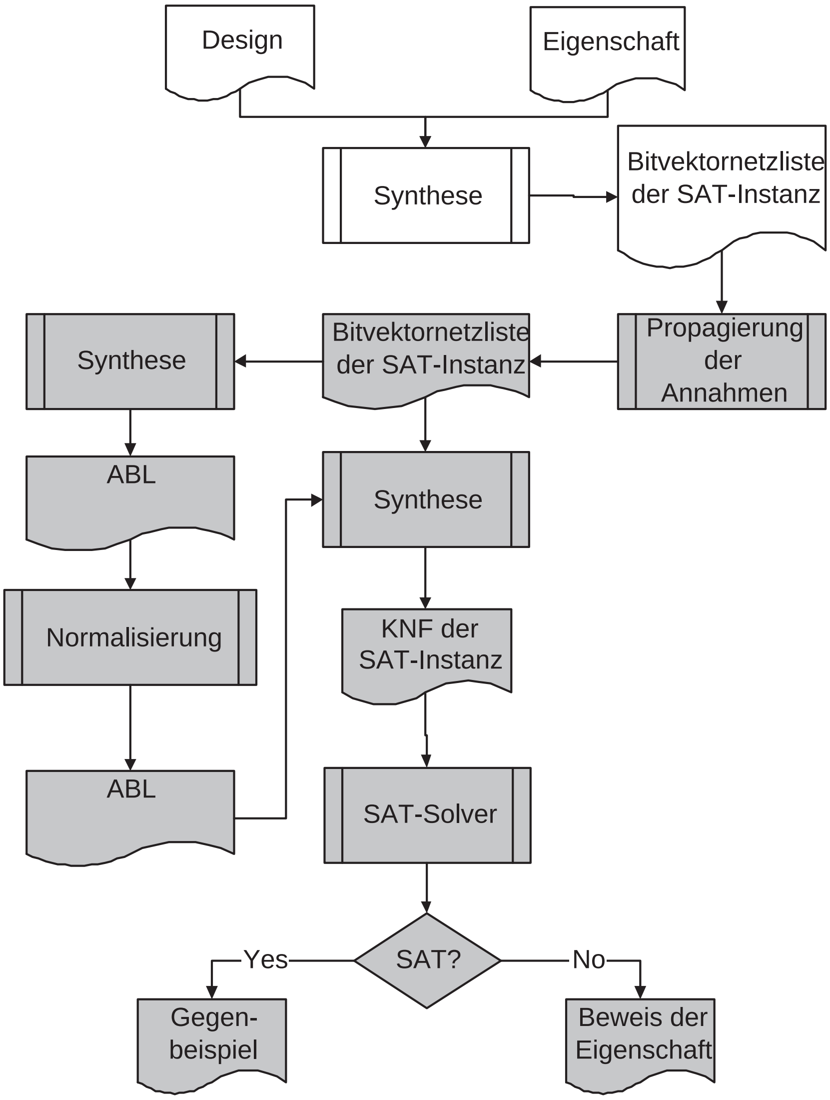


Abbildung 5.20: Ablauf der Eigenschaftsprüfung mit Normalisierung

Mit Hilfe von ABLProp ist es gelungen, für alle Arithmetikbefehle mit Multiplikation des Tricore 2 Prozessors das arithmetische Ergebnis vollständig zu verifizieren. Im Anhang A findet der interessierte Leser die Laufzeittabellen für die Beweise sämtlicher Instruktionsvarianten. Dabei werden die CPU-Zeiten für die grau hinterlegten Schritte des Algorithmus präsentiert.

Tabelle 5.12 gibt einen repräsentativen Ausschnitt dieser Tabellen wieder. Als Erklärung wird für einige Instruktionsvarianten das Beweisziel angegeben. Die erste Spalte der Tabelle gibt den Namen des Theorems an. Das Beweisziel wird in der zweiten Spalte gezeigt. Die Variable *res* bezeichnet dabei das Ergebnisregister der Integer-Pipeline. Die Operandenregister im Dekoder des Prozessors werden mit op1, op2 und op3 bezeichnet. Zuletzt wird durch $<<$ die Linksverschiebung und mit $+$ und $*$ die Addition bzw. Multiplikation bezeichnet, wobei jeweils angegeben ist, ob die Bitvektoren als vorzeichenbehaftet zu interpretieren sind.

Instanz	**Beweisziel**	**CPU-Zeit(s)**
madd_1	res[31:0]=op3[31:0]+(op1[31:0]*op2[31:0]);signed	26,28
madd_h2n1	res[31:0]=op3[31:0]+(op1[15:0]*op2[31:16]<<1);signed res[63:32]=op3[63:32]+(op1[31:16]*op2[15:0]<<1);signed	10,78
madd_q4n1	res[31:0]=op3[31:0]+(op1[31:16]*op2[31:16]<<1);signed	11,13
maddr_h2n1x8000lo	res[15:0]=rnd16(op3[31:0]+(7FFFFFFF));signed res[31:16]=rnd16(op3[63:32]+(op1[31:16]*op2[15:0]<<1)); signed	5,87
madds_5	Beweise Abstraktion für: res[31:0]=sat(op3[31:0]+(op1[31:0]*op2[31:0]));signed	28,21
maddsum_h4	res[63:0]=op3[63:0]+(op1[31:16]*op2[15:0]<<16) -(op1[31:16]*op2[31:16]<<16)	14,76
msub_3	res[63:0]=op3[31:0]-(op1[31:0]*op2[31:0]);signed	30,25
msubm_h3	res[63:0]=op3[63:0]-(op1[31:16]*op2[15:0]<<16) -(op1[15:0]*op2[15:0]<<16)	14,90
mul_u3	res[63:0]=op1[31:0]*op2[31:0];unsigned	30,62
mulr_q1	res[31:16]=rnd16(op1[31:16]*op2[31:16]);signed	5,34

Tabelle 5.12: Ausschnitt der experimentellen Ergebnisse auf industriellen Beispielen

Zur Verifikation von Instruktionen mit Saturierung wurde die Normalisierung genutzt, um eine Abstraktion auf die Wortebene zu verifizieren, wie sie in Abschnitt 5.4 beschrieben wurde. Die angegebene CPU-Zeit bezieht sich dabei auf genau diesen Beweis.

Auf dem abstrahierten Modell konnte die Korrektheit des Ergebnisses *ip_res* allein mit Gateprop bewiesen werden. Tabelle 5.13 zeigt die CPU-Zeiten von Gateprop zum Beweis der korrekten Saturierung für einen Multiplikationsbefehl für vorzeichenlose Operanden. Hierbei wurden zwei Eigenschaften bewiesen. Entsteht ein Überlauf, muss saturiert werden, andernfalls wird das korrekte Multiplikationsergebnis geliefert. Der Beweis dieser Eigenschaften wurde für die nicht abstrahierten Instanzen nach 2500 Sekunden erfolglos abgebrochen.

Eigenschaft	CPU Zeit [s]	
	abstrahiertes Modell	ursprüngliches Modell
no overflow	152.32	$>$2500
overflow	56.61	$>$2500

Tabelle 5.13: CPU-Zeiten für Saturierungseigenschaften

Für Instruktionen mit vorzeichenbehafteten Operanden können ähnliche Ergebnisse erzielt werden, wenn man eine Fallunterscheidung nach Vorzeichen der Operanden durchführt.

Zusammenfassend lässt sich feststellen, dass das Normalisierungsverfahren, welches in diesem Kapitel beschrieben wurde, ein attraktive Methode zur Verifikation hochoptimierter arithmetischer Schaltungen darstellt.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Mit zunehmender Integration von immer mehr Funktionalität in zukünftigen SoC-Designs erhöht sich die Bedeutung der funktionalen Verifikation auf der Blockebene. Nur Block-Entwürfe mit extrem niedriger Fehlerrate erlauben eine schnelle Integration in einen SoC-Entwurf. Diese hohen Qualitätsansprüche können durch simulationsbasierte Verifikation nicht erreicht werden. Aus diesem Grund rücken Methoden zur formalen Entwurfsverifikation in den Fokus. Auf der Blockebene hat sich die Eigenschaftsprüfung basierend auf dem iterativen Schaltungsmodell (BIMC) als erfolgreiche Technologie herausgestellt. Trotzdem gibt es immer noch einige Design-Klassen, die für BIMC schwer zu handhaben sind. Hierzu gehören Schaltungen mit hoher sequentieller Tiefe sowie arithmetische Blöcke.

Die fortlaufende Verbesserung der verwendeten Beweismethoden, z.B. der verwendeten SAT-Solver, wird der zunehmenden Komplexität immer größer werdender Blöcke alleine nicht gewachsen sein. Aus diesem Grund zeigt diese Arbeit auf, wie bereits in der Problemaufbereitung des Front-Ends eines Werkzeugs zur formalen Verifikation Maßnahmen zur Vereinfachung der entstehenden Beweisprobleme ergriffen werden können.

In den beiden gerade angesprochenen Problemfeldern wurden dazu exemplarisch geeignete Freiheitsgrade bei der Modellgenerierung im Front-End identifiziert und zur Vereinfachung der Beweisaufgaben für das Back-End ausgenutzt.

Eigenschaften auf unbeschränkten Zeitintervallen

Zunächst wurde in Kapitel 4 dargestellt, wie eine geeignete Zustandskodierung für Komponenten eines Designs gewählt werden kann, um Invarianten zu generieren, die die Tiefe eines SAT-basierten Induktionsbeweises reduzieren. Es hat sich gezeigt, dass die sogenannte One-hot-Kodierung für kleine Komponenten eines Designs eine attraktive Alternative zur üblichen binären Kodierung darstellt. Diese Kodierung unterstützt in besonderem Maße die Generierung von Invarianten, da sich paarweise Abhängigkeiten zwischen one-hot kodierten Komponenten durch Implikationen zwischen einzelnen Zustandsvariablen ausdrücken lassen. Darüber hinaus steht mit der *strukturellen Automatentraversierung* [SWWK04] ein leistungsfähiger Algorithmus zur Verfügung, um diese Abhängigkeiten zu berechnen. Auf diese Weise erlangt man kraftvolle Invarianten zur Unterstützung der Induktionsbeweise für die zu verifizierenden Eigenschaften.

Bei dem hier vorgestellten Verfahren handelt es sich lediglich um einen ersten Schritt zur konsequenten Ausnutzung von Strukturinformation bei der Problemaufbereitung für die Eigenschaftsprüfung. Die Erkenntnis, dass sich Wissen über Struktur und Komponenten eines Designs zur Invariantengenerierung für die Eigenschaftprüfung ausnutzen lässt, hat sich dabei zu einem eigenständigen Thema entwickelt, welches momentan in weiteren Arbeiten erforscht wird. Erste Publikationen [NSK05, NSWK05] nutzen zum Beispiel die Existenz eines überschaubaren zentralen Steuerwerkes aus, wie man es in vielen Protokollimplementierungen findet.

Der Zustandsübergangsgraph dieses Automaten wird genutzt, um eine Dekomposition sowohl der Zustandsübergangsfunktion des kompletten Designs, als auch der Menge der erreichbaren Zustände durchzuführen. Dadurch gelingt es, wertvolle Invarianten für die Eigenschaftprüfung solcher Designs zu ermitteln. Insbesondere ist der Aufwand für Codeinspektion zur manuellen Elimination von False-Negatives drastisch reduziert worden.

Eigenschaftsprüfung für arithmetische Schaltungen

In der RT-Beschreibung eines Datenpfades findet sich Arithmetik auf unterschiedlichen Abstraktionsebenen wieder. Um hoch optimierte Implementierungen für spezielle arithmetische Funktionen zu erhalten, werden diese häufig direkt auf der Bitebene implementiert. Die Designer instantiieren dabei manuell elementare arithmetische Komponenten wie Halbaddierer, Volladdierer oder partielle Produkte. Diese Vorgehensweise ist notwendig, um einen hohen Grad an Ressourcenteilung zu erzielen, ohne die Performanz des Gesamtsystems zu gefährden. Die so entstandenen hart kodierten arithmetischen Blöcke sind für konventionelle Verifikationsmethoden nicht handhabbar. Diese Problematik wird im wesentlichen durch zwei Faktoren ausgelöst.

- Boolesche Methoden der Bitebene versagen bei der Behandlung arithmetischer Schaltungen.
- Eine Wortebenenabstraktion für das gesamte Problem ist in der Regel nicht möglich.

In Kapitel 5 wurden deshalb Probleme aus der Eigenschaftsprüfung für arithmetische Schaltungen auf der arithmetischen Bitebene (ABL) beschrieben. Eine ABL ist aus Additionsnetzwerken, Generatoren für partielle Produkte und Vergleichern zusammengesetzt. Es erweist sich als einfach, diese ABL im Front-End des Verifikationswerkzeugs zu generieren.

Mit der ABL steht dann eine Beschreibung zur Verfügung, die es erlaubt, Arithmetik auf den verschiedenen Abstraktionsebenen der Wort- und der Bitebene in einem einheitlichen Modell zu erfassen und mit einem einheitlichen Kalkül zu behandeln.

Ebenfalls in Kapitel 5 wurde deshalb ein Verfahren zur Normalisierung einer ABL vorgestellt. Kern dieses Verfahrens sind Operationen zur lokalen Transformation der ABL, die zur schrittweisen Reduzierung des Problems führen. Diese Transformationen nutzen dabei im wesentlichen das Kommutativgesetz sowie das Distributivgesetz aus. Die ausgeführten Transformationen steigern schrittweise die Abstraktion innerhalb der ABL und bringen so die Beschreibung der Arithmetik aus Design und Eigenschaft zur Deckung. Schlussendlich führt dies zur drastischen Vereinfachung der unterliegenden SAT-Instanz und damit zur Reduktion des Beweisaufwands.

Nach dem Wissen des Autors ist dies die erste Verifikationsmethode, die in der Lage ist, hart kodierte arithmetische Datenpfade vollständig zu verifizieren. Der Ansatz kann nahtlos in bestehende SAT-basierte Eigenschaftsprüfer integriert werden, wie die Experimente mit einem industriellen Werkzeug gezeigt haben. Darüber hinaus können natürlich auch hybride Solver von den Abstraktionen, die die Normalisierung liefert, profitieren.

6.2 Ausblick

Die Methoden dieser Arbeit stellen einen ersten Schritt zur Verbesserung der Performanz der Eigenschaftsprüfung dar. Weitere Forschung ist notwendig, um mit der immer weiter ansteigenden Größe der Designs Schritt zu halten. Darüberhinaus wird nach Möglichkeiten gesucht, die Anwendbarkeit der Eigenschaftsprüfung auf neue Anwendungsfelder auszudehnen.

Im Folgenden werden einige Felder für mögliche zukünftige Entwicklungen skizziert.

6.2.1 Dynamische Normalisierung

In Abschnitt 5.4 aus Kapitel 5 wurde dargelegt, wie man das Verifikationsproblem durch Ausnutzung interner Äquivalenzen auf einen arithmetischen Kern reduzieren kann, der dann mit Hilfe der ABL-Normalisierung gelöst werden kann.

Dabei wurde zunächst gezeigt, dass die Kontrolllogik zwischen arithmetischen Blöcken unter den Annahmen der aktuell untersuchten Eigenschaft transparent wird, d.h. einen bestimmten Datenfluss zwischen den arithmetischen Einheiten einstellt. Die daraus resultierenden Äquivalenzen lassen sich mit SAT-Techniken leicht ermitteln und dann zur Logikersetzung ausnutzen.

Obwohl in unserem Fall eine industrielle MAC-Einheit vollständig verifiziert werden konnte, sind auch Fälle denkbar, in denen dies nicht der Fall ist. Es ist möglich, dass ein Zwischenergebnis auf der arithmetischen Bitebene wahlweise auf verschiedenen Einheiten berechnet wird. In vielen Fällen wird dann der Ansatz der Abstraktion durch lokale Normalisierung aus Abschnitt 5.4.3 aus Kapitel 5 greifen.

Schwierig wird dies allerdings, wenn die Ressourcenänderung auch mit der Anwendung unterschiedlicher Algorithmen einhergeht, und die verwendeten Komponenten jeweils unterschiedliche Zwischenergebnisse liefern. Im Zuge der Einführung rekonfigurierbarer Datenpfade, wäre ein Szenario ähnlich dem folgenden Beispiel vorstellbar.

Beispiel 6.1. *Stehen in einer Pipeline mehrere Addierer und mehrere Multiplizierer zur Verfügung, kann es, je nach Situation der Pipeline, sinnvoll sein, statt des Ausdrucks* $(a+b)c$ *den äquivalenten Ausdruck* $ab+bc$ *zu berechnen. Die dazu notwendige Kontrolllogik hängt allerdings nicht von der auszuführenden Operation, sondern von dem aktuellen Zustand der Pipeline ab. Dieser wird in einer Eigenschaft typischerweise nicht festgelegt, um die Eigenschaft so allgemein wie möglich zu halten. Damit ist es nicht möglich, die Kontrolllogik unter den Annahmen der Eigenschaft zu eliminieren. Abbildung 6.1 visualisiert den Ausschnitt aus dem iterativen Schaltungsmodell dieser Pipeline, der für die hier betrachtete Operation relevant ist.*

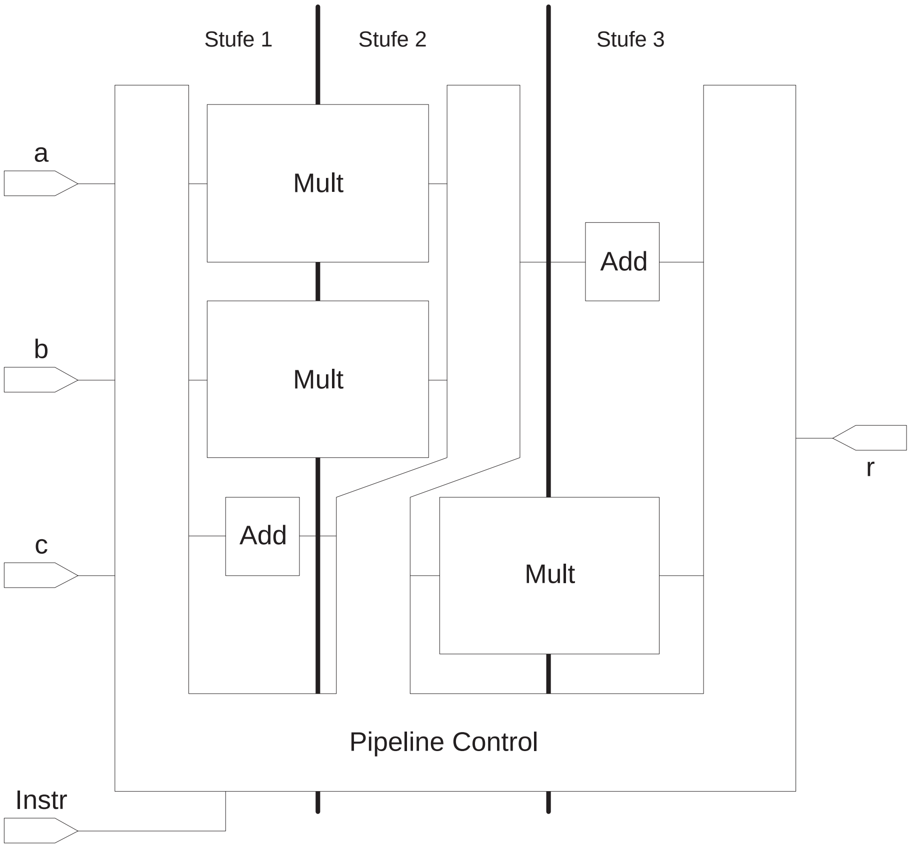


Abbildung 6.1: Struktur einer Pipeline

In einer Situation wie dem gerade beschriebenen Beispiel kann die Normalisierung erst dann sinnvoll eingesetzt werden, wenn der Datenfluss durch vorangehende Entscheidungen und Wertezuweisungen für Kontrollvariablen fest eingestellt ist.

Eine erste naheliegende Lösung könnte also sein, durch Enumerierung aller möglichen Datenflüsse das Problem in Teile zu zerlegen, die durch Normalisierung gelöst werden können.

Die reine Enumerierung der Kontrollvariablen wird im Allgemeinen allerdings eine Vielzahl ähnlicher Probleme für die Normalisierung erzeugen. Es bedarf also der Entwicklung neuer Konzepte, die es erlauben, eine solche Enumerierung effizient zu gestalten und Wissen aus bereits erfolgreichen Normalisierungen wieder zu verwenden.

6.2.2 Gröbner-Basen

Die Methoden dieser Arbeit nutzen bestimmte Charakteristika moderner Entwurfsprozesse aus, um mit sehr spezialisierten Verfahren eine bestimmte Klasse praktisch relevanter Probleme zu lösen. Dem gegenüber stehen die generischen Methoden, wie z.B. die Erfüllbarkeitsprüfung, die eine sehr große Klasse von Problemen lösen, wobei auf Grund ihrer Allgemeingültigkeit nicht immer die optimale Performanz erzielt wird.

Bei der Suche nach alternativen generischen Methoden für die Eigenschaftsprüfung, die Potential zur Behandlung von Hardware mit Ganzzahlarithmetik aufweisen, erweist sich die Computeralgebra als Fundgrube. Erste Ansätze zur Verwendung algebraischer Methoden in der formalen Verifikation findet man in [SKEG05]. Dabei wird der Datenpfad eines Designs durch ein multivariates Polynom über dem Ring $\mathbb{Z}/2^n$ modelliert. In der oben genannten Arbeit wird die Äquivalenz einer RT-Implementierung mit einer MATHLAB-Simulink-Spezifikation in Festpunktarithmetik sichergestellt, indem die jeweiligen Polynome mit algebraischen Methoden bearbeitet werden. Dazu wird ein Algorithmus zur Reduktion eines multivariaten Polynoms über $\mathbb{Z}/2^n$ in eine kanonische Darstellung angegeben. Der Äquivalenzvergleich ist dann durch einfachen Koeffizientenvergleich leicht möglich.

Für die Eigenschaftsprüfung von RT-Designs greift dieser Ansatz allerding noch zu kurz, da weder eingebettete Kontrolllogik noch Variablen verschiedener Bitbreiten behandelt werden können. Darüber hinaus gelten die Äquivalenzen, die im Beweisziel einer Eigenschaft gefordert werden, in der Regel nicht global, sondern lediglich unter bestimmten Annahmen, die ebenfalls in der Eigenschaft spezifiziert werden.

Es stellt sich also die Frage, wie sich das Problem der Eigenschaftsprüfung als algebraische Fragestellung formulieren lässt. Ein Ansatz zur Modellierung von Bitvektorfunktionen mit Variablen verschiedener Bitbreiten könnte die Verwendung des Rings $\mathbb{Z}/2^{max}$ sein, wobei max die maximale Bitbreite aller vorkommenden Variablen ist. Damit ist es möglich, das iterative Schaltungsmodell und die Eigenschaft auf eine Menge von Polynomen über diesem Ring abzubilden.

Modellierung der Eigenschaftsprüfung in $\mathbb{Z}/2^{max}$

In diesem Abschnitt soll in aller Kürze eine Modellierung der Probleme der Eigenschaftsprüfung als Polynomsystem über $\mathbb{Z}/2^{max}$ skizziert werden.

Zunächst muss sichergestellt werden, dass Variablen der Bitbreite $n < max$ auch nur Werte aus Bereich $0, \ldots, 2^n - 1$ annehmen können. Für den Spezialfall $n = 1$ kann dies durch das Polynom $x * (x - 1) = x^2 - x$ erreicht werden. Diese Vorgehensweise kann allerdings nicht problemlos für $n > 1$ verallgemeinert werden. So hat das Polynom $x(x-1)(x-2)(x-3)$ in $\mathbb{Z}/8$ die Nullstelle 7. Durch das Polynom $x - \sum_{i=0}^{n-1} 2^i x_i$ und die Nebenbedingungen $x_i^2 - x_i$ gelingt zwar, die Beschreibung dieser Teilmenge, wird allerdings durch die Anwesenheit vieler Nullteiler in den Koeffizienten erkauft.

Nun wenden wir uns der Modellierung von Bitvektorfunktionen durch Polynome zu. Beispielhaft sei hier die Modellierung der Multiplikation $r = a*b$ von 5-Bit Zahlen in $\mathbb{Z}/2^{20}$ genannt, diese kann durch das Polynom $ab - r + 2^5 s$ erfolgen. Aufgrund der Bitbreitenbeschränkung für r benötigen wir die zusätzliche sogenannte Schlupfvariable s, um die oberen Bits des Multiplikationsergebnisses aufzunehmen. Analog zur Multiplikation lassen sich die anderen Operationen $+$ und $-$ umsetzen.

Schwieriger gestaltet sich die Modellierung nicht arithmetischer Bitvektorfunktionen. Hier bleibt in einigen Fällen nur der Umweg über die oben bereits erwähnte Zerlegung einer Variablen in die einzelnen Bits und die Bitweise Implementierung von $\overline{x} = 1 - x$ und $x \wedge y = xy$. Die anderen Funktionen können dann aus dieser Basis der Booleschen Algebra abgeleitet werden.

Einige Operationen, wie z.B. $ite(s, a, b) = sa + (1 - s)b$, lassen sich auch direkt auf der Wortebene modellieren. Hierzu gehört auch die Konkatenation von Vektoren a und b der Bitbreite n bzw. m: $concat(a, b) = 2^m a + b + 2^{m+n} s$, wobei hier im Fall $m + n < max$ erneut eine Schlupfvariable s eingeführt werden muss. Hat man nun eine Menge von Polynomen, die alle Bitvektorfunktionen des iterativen Schaltungsmodells und der Eigenschaft repräsentiert, so stellt sich die Frage, wie man anhand dieser Menge ermitteln kann, ob die Eigenschaft gültig ist.

Sei dazu S diese Polynommenge und p die 1-Bit Variable, die die Eigenschaft kodiert, also $p = 1$ genau dann, wenn die Eigenschaft gilt. Dies ist genau dann der Fall, wenn die Varietät der Menge $S \cup \{1 - p\}$, d. h. die Menge $V(S \cup \{1 - p\})$ der gemeinsamen Nullstellen aller dieser Polynome leer ist. Um eine handhabbare Darstellung für eine Varietät $V(S)$ zu erlangen, betrachtet man nun das Ideal $< S >$ der von S erzeugten Polynome im Polynomring. Enthält dieses Ideal ein von Null verschiedenes konstantes Polynom, so ist die Varietät $V(S)$ leer und umgekehrt.

Damit ist das Problem der Eigenschaftsprüfung in ein algebraisches Problem überführt, welches den Methoden der Computeralgebra zugänglich ist. Ein Ansatz, der hier zielführend erscheint, liegt darin, spezielle Gröbner-Basen-Algorithmen für Ringe zu implementieren. Auf Grund einer Gröbner-Basis kann dann leichter entschieden werden, ob im Ideal $< S >$ ein von Null verschiedenes konstantes Polynom enthalten ist.

Normalisierung zur Modellgenerierung

Der gerade beschriebene Ansatz der Verwendung von Gröbner-Basen in der Eigenschaftsprüfung hat insbesondere dann Potential, wenn die Arithmetik im vorliegenden Problem überwiegend auf der Wortebene spezifiziert ist. Die vorliegende Arbeit hat gezeigt, dass die ABL-Normalisierung eingesetzt werden kann, um für Problemteile, deren Arithmetik auf der Bitebene spezifiziert ist, geeignete Abstraktionen zu beweisen. Diese Abstraktionen bilden dann die Grundlage für die im letzten Abschnitt beschriebene Modellierung des übergreifenden Problems als Polynomsystem.

6.2.3 Neue Anwendungsdomänen

Fortschritte bei der Eigenschaftsprüfung für digitale arithmetische Schaltungen können in Zukunft genutzt werden, um neue Anwendungsdomänen für die Eigenschaftsprüfung zu erschließen. Exemplarisch soll an dieser Stelle auf die formale Verifikation analoger Schaltungen eingegangen werden. Die Möglichkeit, digitale und analoge Schaltungen mit ähnlichen Techniken zu verifizieren, ist eine Grundvoraussetzung, um die formale Verifikation von mixed-signal-Schaltungen in Angriff zu nehmen, die heute schon in fast jedem SoC-Entwurf enthalten sind.

Ein Ansatz zur formalen Verifikation von analogen oder mixed-signal Schaltungen sieht die Digitalisierung des analogen Schaltungsteils vor. Dazu müssen natürlich die Parameter der analogen Schaltung zunächst diskretisiert werden.

In [HHB02b, HHB02a] wird dazu ein Ansatz vorgestellt, der den kontinuierlichen Zustandsraum einer analogen Schaltung in eine endliche Anzahl von Gebieten aufteilt und eine abstrakte Übergangsrelation zwischen diesen Gebieten berechnet. Das so entstandene Modell der analogen Schaltung wird mit klassischen CTL-Model-Checking-Methoden verifiziert.

Andere Methoden benutzen abstrahierte Modelle der analogen Schaltung, die dann als digitale Schaltung realisiert werden. Bartsch und Schubert [Sch97, BS97] haben beispielsweise eine Methode zur Simulation kleiner analoger Schaltungsteile in einer VHDL'93 Umgebung entwickelt.

In [FSS05] schlagen Freibothe et. al eine Methode zur Verifikation des quasistatischen Verhaltens von mixed-signal Schaltungen vor. Hier werden die analogen Teile der Schaltung durch algebraische Gleichungen beschrieben. Aufgrund der Verwendung von SVC als Beweiser kann dieser Ansatz allerdings nur lineare analoge Schaltungsteile bearbeiten.

Eine alternative Möglichkeit, das Verhalten der analogen Schaltung durch eine digitale Hardware zu modellieren, besteht darin, ein iteratives Näherungsverfahren, z.B das Backward-Euler-Verfahren, für die Differentialgleichungen der Analogschaltung zu implementieren. Das so entstehende digitale Modell enthält dann fast ausschließlich Arithmetik, womit das Potential der neuen Methoden zur Arithmetikverifikation genutzt werden kann.

Die wissenschaftliche Herausforderung besteht nun darin, die Generierung des Modells für die analoge Schaltung weitestgehend zu automatisieren.

Anhang A

Ergebnisstabellen für Tricore 2 Eigenschaften

Instanz	**CPU-Zeit(s)**	**Instanz**	**CPU-Zeit(s)**	**Instanz**	**CPU-Zeit(s)**
madd_1	26,28	maddm_h3n1x8000lo	5,98	maddrs_h7n1	10,32
madd_2	24,86	maddm_h4	8,09	maddrs_h7n1x8000hi	4,72
madd_3	29,51	maddm_h4n1	14,85	maddrs_h7n1x8000lo	5,96
madd_4	29,74	maddm_h4n1x8000hi	5,86	maddrs_h8	9,55
madd_h1	9,84	maddm_h4n1x8000lo	5,99	maddrs_h8n1	9,86
madd_h1n1	10,15	maddms_h5	8,33	maddrs_h8n1x8000hi	4,57
madd_h1n1x8000hi	4,74	maddms_h5n1	14,95	maddrs_h8n1x8000lo	5,81
madd_h1n1x8000lo	6,02	maddms_h5n1x8000hi	9,98	maddrs_h9	9,66
madd_h2	9,04	maddms_h5n1x8000lo	6,31	maddrs_h9n1	10,23
madd_h2n1	10,78	maddms_h6	7,86	maddrs_h9n1x8000hi	4,55
madd_h2n1x8000hi	4,64	maddms_h6n1	14,30	maddrs_h9n1x8000lo	5,80
madd_h2n1x8000lo	5,88	maddms_h6n1x8000hi	9,96	maddrs_q3	6,03
madd_h3	9,29	maddms_h6n1x8000lo	6,18	maddrs_q3n1	10,33
madd_h3n1	10,16	maddms_h7	8,18	maddrs_q3n1x8000lo	6,54
madd_h3n1x8000hi	4,57	maddms_h7n1	15,53	maddrs_q4	4,59
madd_h3n1x8000lo	5,91	maddms_h7n1x8000hi	5,83	maddrs_q4n1	8,47
madd_h4	9,82	maddms_h7n1x8000lo	6,10	madds_5	28,21
madd_h4n1	10,64	maddms_h8	8,10	madds_6	23,13
madd_h4n1x8000hi	4,56	maddms_h8n1	14,34	madds_7	28,51
madd_h4n1x8000lo	5,66	maddms_h8n1x8000hi	5,80	madds_8	27,87
madd_q1	31,32	maddms_h8n1x8000lo	6,04	madds_h5	9,06
madd_q10	5,02	maddr_h1	9,26	madds_h5n1	9,90
madd_q10n1	10,74	maddr_h1n1	9,95	madds_h5n1x8000hi	4,56
madd_q1n1	39,83	maddr_h1n1x8000hi	4,56	madds_h5n1x8000lo	5,78
madd_q2	8,01	maddr_h1n1x8000lo	5,76	madds_h6	9,05
madd_q2n1	12,34	maddr_h2	9,35	madds_h6n1	9,81
madd_q3	8,12	maddr_h2n1	10,18	madds_h6n1x8000hi	4,61
madd_q3n1	11,55	maddr_h2n1x8000hi	4,62	madds_h6n1x8000lo	5,81
madd_q4	5,79	maddr_h2n1x8000lo	5,87	madds_h7	9,13
madd_q4n1	11,13	maddr_h3	9,67	madds_h7n1	9,96
madd_q4n1x8000lo	7,53	maddr_h3n1	9,92	madds_h7n1x8000hi	4,44
madd_q5	4,61	maddr_h3n1x8000hi	4,52	madds_h7n1x8000lo	5,69
madd_q5n1	8,32	maddr_h3n1x8000lo	5,86	madds_h8	9,02
madd_q6	27,96	maddr_h4	9,72	madds_h8n1	11,09
madd_q6n1	37,47	maddr_h4n1	9,71	madds_h8n1x8000hi	4,49
madd_q7	19,90	maddr_h4n1x8000hi	4,69	madds_h8n1x8000lo	5,79
madd_q7n1	8,48	maddr_h4n1x8000lo	5,72	madds_q11	30,69
madd_q8	20,00	maddr_h5	5,23	madds_q11n1	36,27
madd_q8n1	8,77	maddr_h5n1	10,62	madds_q12	8,72
madd_q9	6,45	maddr_h5n1x8000hi	4,70	madds_q12n1	12,20
madd_q9n1	11,37	maddr_h5n1x8000lo	6,00	madds_q13	8,06
madd_q9n1x8000lo	6,52	maddr_q1	7,14	madds_q13n1	11,59
madd_u3	26,45	maddr_q1n1	11,06	madds_q14	5,76
madd_u4	25,85	maddr_q1n1x8000lo	7,95	madds_q14n1	11,39
maddm_h1	8,19	maddr_q2	5,82	madds_q14n1x8000lo	6,33
maddm_h1n1	14,14	maddr_q2n1	11,02	madds_q15	4,57
maddm_h1n1x8000hi	10,46	maddrs_h10	9,95	madds_q15n1	9,01
maddm_h1n1x8000lo	6,10	maddrs_h10n1	11,06	madds_q16	29,47
maddm_h2	8,14	maddrs_h10n1x8000hi	4,79	madds_q16n1	39,55
maddm_h2n1	14,78	maddrs_h10n1x8000lo	6,26	madds_q17	7,79
maddm_h2n1x8000hi	10,98	maddrs_h6	9,46	madds_q17n1	8,36
maddm_h2n1x8000lo	6,03	maddrs_h6n1	10,27	madds_q18	20,06
maddm_h3	7,99	maddrs_h6n1x8000hi	4,55	madds_q18n1	8,30
maddm_h3n1	15,10	maddrs_h6n1x8000lo	5,73	madds_q19	6,31
maddm_h3n1x8000hi	5,79	maddrs_h7	9,62	madds_q19n1	11,18

Tabelle A.1: Experimentelle Ergebnisse auf industriellen Beispielen Teil 1

Instanz	CPU-Zeit(s)	Instanz	CPU-Zeit(s)	Instanz	CPU-Zeit(s)
madds_q19n1x8000lo	6,30	maddsur_h1	9,52	msub_h1n1x8000lo	5,92
madds_q20	4,96	maddsur_h1n1	14,31	msub_h2	9,52
madds_q20n1	11,71	maddsur_h1n1x8000hi	4,52	msub_h2n1	16,06
madds_u5	25,23	maddsur_h1n1x8000lo	5,90	msub_h2n1x8000hi	4,81
madds_u6	25,13	maddsur_h2	9,53	msub_h2n1x8000lo	5,99
madds_u7	24,91	maddsur_h2n1	13,60	msub_h3	9,45
madds_u8	24,50	maddsur_h2n1x8000hi	4,63	msub_h3n1	17,02
maddsu_h1	9,19	maddsur_h2n1x8000lo	5,90	msub_h3n1x8000hi	4,64
maddsu_h1n1	17,41	maddsur_h3	9,31	msub_h3n1x8000lo	5,85
maddsu_h1n1x8000hi	4,64	maddsur_h3n1	14,33	msub_h4	9,52
maddsu_h1n1x8000lo	5,92	maddsur_h3n1x8000hi	4,54	msub_h4n1	17,24
maddsu_h2	9,22	maddsur_h3n1x8000lo	5,97	msub_h4n1x8000hi	4,66
maddsu_h2n1	16,49	maddsur_h4	9,00	msub_h4n1x8000lo	5,83
maddsu_h2n1x8000hi	4,62	maddsur_h4n1	13,95	msub_q1	34,44
maddsu_h2n1x8000lo	5,96	maddsur_h4n1x8000hi	4,43	msub_q10	6,60
maddsu_h3	9,24	maddsur_h4n1x8000lo	5,78	msub_q10n1	11,64
maddsu_h3n1	17,19	maddsurs_h5	9,44	msub_q1n1	54,03
maddsu_h3n1x8000hi	4,65	maddsurs_h5n1	13,90	msub_q2	17,14
maddsu_h3n1x8000lo	5,86	maddsurs_h5n1x8000hi	4,58	msub_q2n1	33,10
maddsu_h4	9,72	maddsurs_h5n1x8000lo	6,09	msub_q2n1tru32	12,73
maddsu_h4n1	17,61	maddsurs_h6	9,59	msub_q2tru32	17,32
maddsu_h4n1x8000hi	5,26	maddsurs_h6n1	14,49	msub_q3	15,80
maddsu_h4n1x8000lo	5,81	maddsurs_h6n1x8000hi	4,57	msub_q3n1	19,31
maddsum_h1	14,65	maddsurs_h6n1x8000lo	5,94	msub_q3n1tru32	14,07
maddsum_h1n1	21,67	maddsurs_h7	9,33	msub_q3tru32	19,12
maddsum_h1n1x8000hi	12,79	maddsurs_h7n1	15,46	msub_q4	9,96
maddsum_h1n1x8000lo	7,62	maddsurs_h7n1x8000hi	4,52	msub_q4n1	17,49
maddsum_h2	14,94	maddsurs_h7n1x8000lo	5,85	msub_q4n1x8000lo	10,00
maddsum_h2n1	20,67	maddsurs_h8	9,52	msub_q5	6,88
maddsum_h2n1x8000hi	12,12	maddsurs_h8n1	15,08	msub_q5n1	12,50
maddsum_h2n1x8000lo	7,68	maddsurs_h8n1x8000hi	4,58	msub_q6	39,14
maddsum_h3	14,64	maddsurs_h8n1x8000lo	5,96	msub_q6n1	53,72
maddsum_h3n1	22,17	maddsus_h5	9,21	msub_q7	9,72
maddsum_h3n1x8000hi	7,00	maddsus_h5n1	16,56	msub_q7n1	24,36
maddsum_h3n1x8000lo	6,78	maddsus_h5n1x8000hi	4,66	msub_q8	10,53
maddsum_h4	14,76	maddsus_h5n1x8000lo	5,90	msub_q8n1	20,85
maddsum_h4n1	21,45	maddsus_h6	9,28	msub_q9	11,11
maddsum_h4n1x8000hi	6,90	maddsus_h6n1	16,52	msub_q9n1	20,99
maddsum_h4n1x8000lo	6,84	maddsus_h6n1x8000hi	4,75	msub_q9n1x8000lo	9,68
maddsums_h5	14,93	maddsus_h6n1x8000lo	5,95	msub_u3	26,90
maddsums_h5n1	20,44	maddsus_h7	9,39	msub_u4	26,85
maddsums_h5n1x8000hi	12,48	maddsus_h7n1	16,91	msubad_h1	10,06
maddsums_h5n1x8000lo	7,50	maddsus_h7n1x8000hi	4,70	msubad_h1n1	10,50
maddsums_h6	14,49	maddsus_h7n1x8000lo	6,02	msubad_h1n1x8000hi	4,79
maddsums_h6n1	21,78	maddsus_h8	9,42	msubad_h1n1x8000lo	5,96
maddsums_h6n1x8000hi	12,45	maddsus_h8n1	19,38	msubad_h2	10,07
maddsums_h6n1x8000lo	7,56	maddsus_h8n1x8000hi	4,73	msubad_h2n1	10,46
maddsums_h7	14,63	maddsus_h8n1x8000lo	5,94	msubad_h2n1x8000hi	4,79
maddsums_h7n1	24,33	msub_1	25,87	msubad_h2n1x8000lo	5,99
maddsums_h7n1x8000hi	6,98	msub_2	24,65	msubad_h3	10,03
maddsums_h7n1x8000lo	6,95	msub_3	30,25	msubad_h3n1	10,53
maddsums_h8	14,68	msub_4	29,45	msubad_h3n1x8000hi	4,65
maddsums_h8n1	22,60	msub_h1	9,36	msubad_h3n1x8000lo	5,89
maddsums_h8n1x8000hi	7,14	msub_h1n1	15,99	msubad_h4	9,96
maddsums_h8n1x8000lo	7,01	msub_h1n1x8000hi	4,86	msubad_h4n1	10,49

Tabelle A.2: Experimentelle Ergebnisse auf industriellen Beispielen Teil 2

Instanz	**CPU-Zeit(s)**	**Instanz**	**CPU-Zeit(s)**	**Instanz**	**CPU-Zeit(s)**
msubad_h4n1x8000hi	4,71	msubadrs_h6n1	10,44	msubms_h8	15,75
msubad_h4n1x8000lo	5,96	msubadrs_h6n1x8000hi	4,76	msubms_h8n1	27,40
msubadm_h1	8,24	msubadrs_h6n1x8000lo	5,98	msubms_h8n1x8000hi	7,68
msubadm_h1n1	13,83	msubadrs_h7	9,56	msubms_h8n1x8000lo	7,42
msubadm_h1n1x8000hi	10,70	msubadrs_h7n1	10,59	msubr_h1	12,30
msubadm_h1n1x8000lo	6,46	msubadrs_h7n1x8000hi	4,68	msubr_h1n1	18,42
msubadm_h2	8,28	msubadrs_h7n1x8000lo	5,92	msubr_h1n1x8000hi	6,11
msubadm_h2n1	14,06	msubadrs_h8	9,97	msubr_h1n1x8000lo	7,53
msubadm_h2n1x8000hi	10,53	msubadrs_h8n1	10,36	msubr_h2	12,41
msubadm_h2n1x8000lo	6,57	msubadrs_h8n1x8000hi	4,67	msubr_h2n1	18,53
msubadm_h3	8,33	msubadrs_h8n1x8000lo	5,80	msubr_h2n1x8000hi	5,48
msubadm_h3n1	14,63	msubads_h5	9,38	msubr_h2n1x8000lo	6,76
msubadm_h3n1x8000hi	6,14	msubads_h5n1	9,88	msubr_h3	10,99
msubadm_h3n1x8000lo	6,35	msubads_h5n1x8000hi	4,70	msubr_h3n1	16,82
msubadm_h4	8,23	msubads_h5n1x8000lo	5,82	msubr_h3n1x8000hi	5,19
msubadm_h4n1	15,14	msubads_h6	9,41	msubr_h3n1x8000lo	7,23
msubadm_h4n1x8000hi	6,16	msubads_h6n1	9,70	msubr_h4	11,55
msubadm_h4n1x8000lo	6,24	msubads_h6n1x8000hi	4,70	msubr_h4n1	17,53
msubadms_h5	8,14	msubads_h6n1x8000lo	5,89	msubr_h4n1x8000hi	5,38
msubadms_h5n1	14,08	msubads_h7	9,40	msubr_h4n1x8000lo	7,46
msubadms_h5n1x8000hi	10,71	msubads_h7n1	9,88	msubr_h5	11,92
msubadms_h5n1x8000lo	6,34	msubads_h7n1x8000hi	4,68	msubr_h5n1	21,00
msubadms_h6	8,07	msubads_h7n1x8000lo	5,87	msubr_h5n1x8000hi	14,66
msubadms_h6n1	14,20	msubads_h8	9,93	msubr_h5n1x8000lo	13,70
msubadms_h6n1x8000hi	10,40	msubads_h8n1	10,41	msubr_q1	9,11
msubadms_h6n1x8000lo	6,42	msubads_h8n1x8000hi	4,69	msubr_q1n1	16,04
msubadms_h7	8,30	msubads_h8n1x8000lo	5,76	msubr_q1n1x8000lo	11,72
msubadms_h7n1	15,06	msubm_h1	14,81	msubr_q2	6,88
msubadms_h7n1x8000hi	6,09	msubm_h1n1	21,34	msubr_q2n1	13,47
msubadms_h7n1x8000lo	6,38	msubm_h1n1x8000hi	12,18	msubrs_h10	14,18
msubadms_h8	8,11	msubm_h1n1x8000lo	7,58	msubrs_h10n1	24,97
msubadms_h8n1	14,12	msubm_h2	15,05	msubrs_h10n1x8000hi	14,02
msubadms_h8n1x8000hi	6,00	msubm_h2n1	21,30	msubrs_h10n1x8000lo	14,26
msubadms_h8n1x8000lo	6,25	msubm_h2n1x8000hi	12,11	msubrs_h6	11,83
msubadr_h1	9,70	msubm_h2n1x8000lo	7,52	msubrs_h6n1	18,56
msubadr_h1n1	10,22	msubm_h3	14,90	msubrs_h6n1x8000hi	6,04
msubadr_h1n1x8000hi	4,68	msubm_h3n1	22,29	msubrs_h6n1x8000lo	7,89
msubadr_h1n1x8000lo	5,92	msubm_h3n1x8000hi	7,36	msubrs_h7	11,50
msubadr_h2	9,94	msubm_h3n1x8000lo	7,11	msubrs_h7n1	21,35
msubadr_h2n1	10,44	msubm_h4	15,69	msubrs_h7n1x8000hi	9,54
msubadr_h2n1x8000hi	4,78	msubm_h4n1	23,87	msubrs_h7n1x8000lo	7,37
msubadr_h2n1x8000lo	5,98	msubm_h4n1x8000hi	7,57	msubrs_h8	11,79
msubadr_h3	9,64	msubm_h4n1x8000lo	7,48	msubrs_h8n1	21,55
msubadr_h3n1	10,25	msubms_h5	16,08	msubrs_h8n1x8000hi	5,64
msubadr_h3n1x8000hi	4,65	msubms_h5n1	22,10	msubrs_h8n1x8000lo	7,03
msubadr_h3n1x8000lo	5,90	msubms_h5n1x8000hi	12,75	msubrs_h9	11,39
msubadr_h4	9,53	msubms_h5n1x8000lo	8,01	msubrs_h9n1	20,77
msubadr_h4n1	10,00	msubms_h6	15,95	msubrs_h9n1x8000hi	6,30
msubadr_h4n1x8000hi	4,58	msubms_h6n1	24,69	msubrs_h9n1x8000lo	6,88
msubadr_h4n1x8000lo	5,97	msubms_h6n1x8000hi	12,58	msubrs_q3	9,31
msubadrs_h5	9,65	msubms_h6n1x8000lo	7,82	msubrs_q3n1	15,18
msubadrs_h5n1	10,24	msubms_h7	15,53	msubrs_q3n1x8000lo	9,93
msubadrs_h5n1x8000hi	4,68	msubms_h7n1	23,79	msubrs_q4	8,85
msubadrs_h5n1x8000lo	5,90	msubms_h7n1x8000hi	8,53	msubrs_q4n1	19,60
msubadrs_h6	9,90	msubms_h7n1x8000lo	7,54	msubs_5	67,72

Tabelle A.3: Experimentelle Ergebnisse auf industriellen Beispielen Teil 3

Instanz	CPU-Zeit(s)	Instanz	CPU-Zeit(s)	Instanz	CPU-Zeit(s)
msubs_6	53,00	mul_h1n1x8000hi	4,49	mulms_h6n1x8000hi	4,39
msubs_7	40,57	mul_h1n1x8000lo	5,53	mulms_h6n1x8000lo	5,63
msubs_8	49,95	mul_h2	18,85	mulms_h7	11,11
msubs_h5	12,67	mul_h2n1	10,40	mulms_h7n1	11,25
msubs_h5n1	18,34	mul_h2n1x8000hi	4,40	mulms_h7n1x8000hi	3,91
msubs_h5n1x8000hi	4,88	mul_h2n1x8000lo	5,50	mulms_h7n1x8000lo	4,42
msubs_h5n1x8000lo	6,11	mul_h3	18,81	mulms_h8	11,21
msubs_h6	9,40	mul_h3n1	10,50	mulms_h8n1	11,12
msubs_h6n1	16,52	mul_h3n1x8000hi	3,90	mulms_h8n1x8000hi	3,86
msubs_h6n1x8000hi	4,72	mul_h3n1x8000lo	5,06	mulms_h8n1x8000lo	4,98
msubs_h6n1x8000lo	5,94	mul_h4	18,89	mulr_h1	9,52
msubs_h7	9,82	mul_h4n1	9,42	mulr_h1n1	10,64
msubs_h7n1	17,39	mul_h4n1x8000hi	3,92	mulr_h1n1x8000hi	4,61
msubs_h7n1x8000hi	4,83	mul_h4n1x8000lo	5,06	mulr_h1n1x8000lo	5,82
msubs_h7n1x8000lo	5,94	mul_q1	5,01	mulr_h2	9,93
msubs_h8	10,05	mul_q1n1	6,23	mulr_h2n1	11,00
msubs_h8n1	16,96	mul_q1n1x8000lo	6,02	mulr_h2n1x8000hi	4,51
msubs_h8n1x8000hi	4,78	mul_q2	3,27	mulr_h2n1x8000lo	5,76
msubs_h8n1x8000lo	5,98	mul_q2n1	4,14	mulr_h3	9,93
msubs_q11	30,51	mul_q3	6,15	mulr_h3n1	11,10
msubs_q11n1	38,49	mul_q3n1	6,52	mulr_h3n1x8000hi	4,04
msubs_q12	15,10	mul_q4	6,26	mulr_h3n1x8000lo	5,31
msubs_q12n1	16,07	mul_q4n1	6,15	mulr_h4	10,27
msubs_q12n1tru32	16,36	mul_q5	70,16	mulr_h4n1	10,66
msubs_q12tru32	15,28	mul_q5n1	67,32	mulr_h4n1x8000hi	3,97
msubs_q13	15,66	mul_q6	7,56	mulr_h4n1x8000lo	5,23
msubs_q13n1	20,50	mul_q6n1	7,70	mulr_q1	5,34
msubs_q13n1tru32	12,34	mul_q7	6,56	mulr_q1n1	7,41
msubs_q13tru32	12,26	mul_q7n1	6,66	mulr_q1n1x8000lo	7,21
msubs_q14	7,55	mul_q8	70,17	mulr_q2	5,54
msubs_q14n1	12,07	mul_q8n1	67,46	mulr_q2n1	5,76
msubs_q14n1x8000lo	8,29	mul_u3	30,62	muls_5	27,73
msubs_q15	6,20	mul_u4	30,03	muls_6	27,87
msubs_q15n1	10,47	mulm_h1	6,44	muls_u5	25,04
msubs_q16	31,09	mulm_h1n1	6,54	muls_u6	23,73
msubs_q16n1	42,79	mulm_h1n1x8000hi	4,36		
msubs_q17	8,30	mulm_h1n1x8000lo	5,58		
msubs_q17n1	15,10	mulm_h2	6,40		
msubs_q18	8,40	mulm_h2n1	6,53		
msubs_q18n1	17,88	mulm_h2n1x8000hi	3,85		
msubs_q19	7,85	mulm_h2n1x8000lo	4,93		
msubs_q19n1	13,38	mulm_h3	6,46		
msubs_q19n1x8000lo	7,90	mulm_h3n1	6,54		
msubs_q20	6,51	mulm_h3n1x8000hi	4,18		
msubs_q20n1	13,42	mulm_h3n1x8000lo	5,47		
msubs_u5	26,47	mulm_h4	7,23		
msubs_u6	26,99	mulm_h4n1	7,17		
msubs_u7	27,09	mulm_h4n1x8000hi	4,18		
msubs_u8	26,84	mulm_h4n1x8000lo	5,54		
mul_1	33,62	mulms_h5	11,04		
mul_2	23,92	mulms_h5n1	10,89		
mul_3	29,23	mulms_h5n1x8000hi	3,92		
mul_4	29,01	mulms_h5n1x8000lo	5,02		
mul_h1	19,07	mulms_h6	11,08		
mul_h1n1	10,52	mulms_h6n1	11,02		

Tabelle A.4: Experimentelle Ergebnisse auf industriellen Beispielen Teil 4

Abbildungsverzeichnis

Literaturverzeichnis

[ABC$^+$02] AUDEMARD, G. ; BERTOLI, P. ; CIMATTI, A. ; KORNILOWICZ, A. ; SEBASTIANI, R.: A SAT-based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In: *Proc. Conference on Automated Deduction (CADE)*, 2002, S. 195–210

[ABE00] ABDULLA, P. A. ; BJESSE, P. ; E'EN, N.: Symbolic Reachability Analysis based on SAT Solvers. In: *Proc. Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-00)*, 2000, S. 411–425

[Ach05] ACHTERBERG, T.: Conflict Analysis in Mixed Integer Programming / Zuse Institute Berlin. 2005 (05-19). – Forschungsbericht. `http://www.zib.de/Publications/abstracts/ZR-05-19/`

[Ake78] AKERS, S. B.: Binary Decision Diagrams. In: *IEEE Transactions on Computers* C-27 (1978), June, Nr. 6, S. 509–516

[AMTW99] APT, K. R. (Hrsg.) ; MAREK, V. W. (Hrsg.) ; TRUSZCZYNSKI, M. (Hrsg.) ; WARREN, D. S. (Hrsg.): *The Logic Programming Paradigm A 25-Year Perspective*. Springer, 1999

[ARMS02] ALOUL, F. A. ; RAMANI, A. ; MARKOV, I. L. ; SAKALLAH, K. A.: Generic ILP versus specialized 0-1 ILP: An Update. In: *Proc. International Conference on Computer-Aided Design (ICCAD-02)*, 2002, S. 450–457

[ASB93] AZIZ, A. ; SINGHAL, V. ; BRAYTON, R. K.: Verifying Interacting Finite State Machines: Complexity Issues / University of California at Berkeley. 1993 (UCB/ERL M93/52). – Forschungsbericht

[BAS02] BIERE, A. ; ARTHO, C. ; SCHUPPAN, V.: Liveness Checking as Safety Checking. In: *Proc. of 7. International Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, 2002

[BC00] BJESSE, P. ; CLAESSEN, K.: SAT-Based Verification without State Space Traversal. In: *Formal Methods in Computer-Aided Design*, 372-389

[BCC$^+$99] BIERE, A. ; CIMATTI, A. ; CLARKE, E. M. ; FUJITA, M. ; ZHU, Y.: Symbolic Model Checking using SAT Procedures instead of BDDs. In: *Proc. International Design Automation Conference (DAC-99)*, 1999, S. 317–320

[BCL$^+$94] BURCH, J. R. ; CLARKE, E. M. ; LONG, D. E. ; MCMILLAN, K. L. ; DILL, D. L.: Symbolic Model Checking for Sequential Circuit Verification. In: *IEEE Transactions on Computer-Aided Design* 13 (1994), April, Nr. 4, S. 401–424

[BCMD90] BURCH, J. R. ; CLARKE, E. M. ; MCMILLAN, K. L. ; DILL, D. L.: Sequential Circuit Verification using Symbolic Model Checking. In: *Proc. International Design Automation Conference (DAC-90)*, 1990, S. 46–51

[BD02] BRINKMANN, R. ; DRECHSLER, R.: RTL-Datapath Verification using Integer Linear Programming. In: *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-02)*. Bangalore, India, 2002

[BJW04] BRINKMANN, R. ; JOHANNSEN, P. ; WINKELMANN, K.: Application of property checking and underlying techniques. In: DRECHSLER, Rolf (Hrsg.): *Advanced Formal Verification*. Boston, MA, USA : Kluwer Academic Publishers, 2004

[BR85] BEST, M. J. ; RITTER, K.: *Linear Programming*. Prentice Hall, 1985

[Bri03] BRINKMANN, R.: *Preprocessing for Property Checking of Sequential Circuit on the Register Transfer Level*, Technische Universität Kaiserslautern, Dissertation, 2003

[Bry86] BRYANT, R.: Graph-based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 35 (1986), August, Nr. 8, S. 677–691

[BS97] BARTSCH, E. ; SCHUBERT, M.: Mixed Analog-Digital Circuit Modeling Using Event-Driven VHDL. In: *IEEE International workshop on Behavioral Modeling and Simulation - BMAS 97* (1997)

[CCGR99] CIMATTI, A. ; CLARKE, E. M. ; GIUNCHIGLIA, F. ; ROVERI, M.: NUSMV: a new Symbolic Model Verifier. In: N. HALBWACHS (Hrsg.) ; D. PELED (Hrsg.): *Proc.International Conference Computer Aided Verification (CAV-99)*. Trento, Italy : Springer, July 1999 (Lecture Notes in Computer Science 1633), S. 495–499

[CCL$^+$96] CABODI, G. ; CAMURATI, P. ; LAVAGNO, L. ; MACII, E. ; PONCINO, M. ; QUER, S. ; SENTOVICH, E.: Enhancing FSM Traversal by Temporary Re-Encoding. In: *Proc. International Conference on Computer Design (ICCD-96)*, 1996

[CCLQ97] CABODI, G. ; CAMURATI, P. ; LAVAGNO, L. ; QUER, S.: Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits. In: *Proc. International Design Automation Conference (DAC-97)*. Anaheim, CA, 1997, S. 728–733

[CCQ96] CABODI, G. ; CAMURATI, P. ; QUER, S.: Improved Reachability Analysis of Large Finite State Machines. In: *Proc. International Conference on Computer-Aided Design (ICCAD-96)*, 1996, S. 354–360

[CGP99] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking*. London, England : MIT Press, 1999

[CHM$^+$94] CHO, H. ; HACHTEL, G. D. ; MACII, E. ; PONCINO, Massimo ; SOMENZI, F.: A Structural Approach for State Space Decomposition for Approximate Reachability Analysis. In: *Proc. International Conference on Computer Design (ICCD-94)*, 1994, S. 236–239

[CK03] CHAI, D. ; KUEHLMANN, A.: A Fast Pseudo-Boolean Constraint Solver. In: *Proc. International Design Automation Conference (DAC-03)*, 2003, S. 830–835

[CK05] CHAI, D. ; KUEHLMANN, A.: A Fast Pseudo-Boolean Constraint Solver. In: *IEEE Transactions on Computer-Aided Design* 24 (2005), March, Nr. 3, S. 305–317

[CPL] *ILOG CPLEX*. – http://www.cplex.com

[CZKR02] CIESIELSKI, M. ; ZENG, Z. ; KALLA, P. ; ROUZEYRE, B.: Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-02)*, 2002, S. 285–291

[DB98] DRECHSLER, R. ; BECKER, B.: *Graphenbasierte Funktionsdarstellung*. Stuttgart : B.G. Teubner, 1998

[DC00] DIAZ, D. ; CODOGNET, P.: The GNU Prolog System and its Implementation. In: *SAC (2)*, 728-732

[DLL60] DAVIS, M. ; LONGEMANN, G. ; LOVELAND, D.: A machine program for theorem proving. In: *Communications of the ACM* (1960), Nr. 7, S. 201–215

[DLMM02] DONINI, F. M. ; LIBERATORE, P. ; MASSACCI, F. ; M.SCHAERF: Solving QBF with SMV. In: *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR 2002)*, 2002, S. 578–589

[ES03] EEN, N. ; SÖRENSSON, N.: An Extensible SAT-solver. In: *Proc. 6. International Conference on Theory and Applications of Satisfiability Testing(SAT 2003)*, 2003

[FA97] FRÜHWIRTH, T. ; ABDENNADHER, S.: *Constraint-Programmierung Grundlagen und Anwendungen*. Springer, 1997

[FDK01] FALLAH, F. ; DEVADAS, S. ; KEUTZER, K.: Functional Vector Generation for HDL Models Using Linear Programming and Boolean Satisfiability. In: *IEEE Transactions on Computer-Aided Design* CAD-20 (2001), August, Nr. 8, S. 994–1002

[FSS05] FREIBOTHE, M. ; SCHÖNHERR, J. ; STRAUBE, B.: Formal Verification of the Quasi-Static Behavior of Mixed-Signal Circuits by Property Checking. In: *Workshop on Formal Verification of Analog Circuits - FAC05* (2005)

[GKP94] GRAHAM, R. L. ; KNUTH, D. E. ; PATASHNI, O.: *Concrete Mathematics, Second Edition*. Addison-Wesley, 1994

[GN02] GOLDBERG, E. ; NOVIKOV, Y.: BerkMin: A Fast and Robust SAT Solver. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-02)*, 142-149

[GT96] GRASSMANN, W. K. ; TREMBLAY, J.-P.: *Logic and Discrete Mathematics*. Prentice Hall, 1996

[HC01] HUANG, C.-Y. ; CHENG, K. T.: Using Word-Level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking. In: *IEEE Transactions on Computer-Aided Design* 20 (2001), March, Nr. 3, S. 381–390

[HD99] HÖRETH, S. ; DRECHSLER, R.: Formal verification of word-level specifications. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-99)*, 1999, S. 52–58

[HHB02a] HARTONG, W. ; HEDRICH, L. ; BARKE, E.: An Approach to Model Checking for Nonlinear Analog Systems. In: *Design, Automation and Test in Europe - Date 2002* (2002)

[HHB02b] HARTONG, W. ; HEDRICH, L. ; BARKE, E.: On Discret Modeling and Model Checking for Nonlinear Analog Systems. In: *Conference on Computer-Aided Verification - CAV 2002* (2002)

[Hol97] HOLZMANN, G. J.: The Model Checker SPIN. In: *IEEE Transactions on Software Engineering*, 23 (1997), May, Nr. 5

[HS96] HACHTEL, Gary D. ; SOMENZI, Fabio: *Logic Synthesis and Verification Algorithms*. Boston : Kluwer Academic Publishers, 1996

[HS02] HASSOUN, S. (Hrsg.) ; SASAO, T. (Hrsg.): *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2002

[ITR03] *International technology roadmap for semiconductors*. 2003

[JD01] JOHANNSEN, P. ; DRECHSLER, R.: Formal Verification on the RT Level Computing One-To-One Design Abstractions by Signal Width Reduction. In: *Proc. IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2001)*. Montpellier, France, 2001

[JM94] J.JAFFAR ; MAHER, M. J.: Constraint logic programming: A survey. In: *The Journal of Logic Programming* 19 (1994), S. 503–582

[JMF95] JAIN, J. ; MUKHERJEE, R. ; FUJITA, M.: Advanced Verification Techniques Based on Learning. In: *Proc. International Design Automation Conference (DAC-95)*, 1995, S. 420 – 426

[Joh01] JOHANNSEN, P.: BOOSTER: Speeding Up RTL Property Checking of Digital Designs by Word-Level Abstraction. In: *Proc.International Conference Computer Aided Verification (CAV-01)*, 2001, S. 373–377

[JPHS91] JEONG, S.-W. ; PLESSIER, B. ; HACHTEL, G. D. ; SOMENZI, F.: Variable Ordering for FSM Traversal. In: *Proc. International Workshop on Logic Synthesis*. MCNC, Research Triangle Park, NC, May 1991

[Kal86] KALMANSON, K.: *Discrete Mathematics and its Applications*. Addison-Wesley, 1986

[Kat94] KATZ, R.: *Contemporary Logic Design*. Benjamin/Cummings, 1994

[KB01] KÜHLMANN, A. ; BAUMGARTNER, J.: Transformation-Based Verification Using Generalized Retiming. In: *Proc.International Conference Computer Aided Verification (CAV-01)*, 2001, S. 104–117

[KK97] KÜHLMANN, A. ; KROHM, F.: Equivalence Checking Using Cuts and Heaps. In: *Proc. International Design Automation Conference (DAC-97)*, 1997, S. 263–268

[Kor98] KOREN, I.: *Computer Arithmetic Algorithms*. Brooside Court Publishers, 1998

[KP94] KUNZ, W. ; PRADHAN, D.: Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems: Test, Verification and Optimization. In: *IEEE Transactions on Computer-Aided Design* 13 (1994), September, S. 1143–1158

[Kun93] KUNZ, W.: An Efficient Tool for Logic Verification Based on Recursive Learning. In: *Proc. International Conference on Computer-Aided Design (ICCAD-93)*, 1993, S. 538–543

[Kur94] KURSHAN, R. P.: *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton, New Jersey : Princeton University Press, 1994

[Lee59] LEE, C.: Representation of switching circuits by binary-decision programs. In: *Bell Systems Technical Journal* 38 (1959), July, S. 985–999

[Mat96] MATSUNAGA, Y.: An Efficient Equivalence Checker for Combinational Circuits. In: *Proc. International Design Automation Conference (DAC-96)*, 1996, S. 629–634

[McM] MCMILLAN, K. L.: *Cadence SMV*. – http://www-cad.eecs.berkeley.edu/˜kenmcmil/

[McM93] MCMILLAN, K. L.: *Symbolic Model Checking*. Boston : Kluwer Academic Publishers, 1993

[MMZ$^+$01] MOSKEWICZ, M. W. ; MADIGAN, C. F. ; ZHAO, Y. ; ZHANG, L. ; MALIK, S.: Chaff: Engineering an Efficient SAT Solver. In: *Proc. International Design Automation Conference (DAC-01)*, 2001, S. 530–535

[MR] MANQUINHO, V. M. ; ROUSSEL, O.: *The First Evaluation of Pseudo-Boolean Solvers (PB'05)*. – To appear in Journal on Satisfiability, Boolean Modeling and Computation

[MSS99] MARQUES-SILVA, P. ; SAKALLAH, K. A.: GRASP: A Search Algorithm for Propositional Satisfiability. In: *IEEE Transactions on Computers* 48 (1999), May, Nr. 5, S. 506–521

[MTP94] MAYOH, B. (Hrsg.) ; TYUGU, E. (Hrsg.) ; PENJAM, J. (Hrsg.): *Constraint Programming*. Springer, 1994

[MVF$^+$02] MCCUNE, W. ; VEROFF, R. ; FITELSON, B. ; HARRIS, K. ; FEIST, A. ; WOS, L.: Short single axioms for boolean algebra. In: *Journal of Automated Reasoning* 29 (2002), S. 1–16

[NSK05] NGUYEN, M. D. ; STOFFEL, D. ; KUNZ, W.: Enhancing BMC-based Protocol Verification Using Transition-By-Transition FSM Traversal. In: *Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*. Bonn, Germany, September 2005

[NSWK05] NGUYEN, M. D. ; STOFFEL, D. ; WEDLER, M. ; KUNZ, W.: Transition-by-Transition FSM Traversal for Reachability Analysis in Bounded Model Checking. In: *Proc. International Conference on Computer Aided Design (ICCAD05).* San Jose, USA, November 2005

[Par00] PARHAMI, P.: *Computer Arithmetic Algorithms and Hardware Design.* Oxford University Press, 2000

[PICW04] PARTHASARATHY, G. ; IYER, M. K. ; CHENG, K.-T. ; WANG, Li.C.: An Efficient Finite-domain Constraint Solver for RTL Circuits. In: *Proc. International Design Automation Conference (DAC-04),* 2004

[QCC$^+$96] QUER, S. ; CABODI, G. ; CAMURATI, P. ; LAVAGNO, L. ; SENTOVICH, E. M. ; BRAYTON, R.K.: Incremental Re-Encoding for Symbolic Traversal of Product Machines. In: *Proc. European Design Automation Conference (Euro-DAC-96),* 1996

[RS95] RAVI, K. ; SOMENZI, F.: High Density Reachability Analysis. In: *Proc. International Conference on Computer-Aided Design (ICCAD-95),* 1995, S. 154–158

[SBW98] SCHOLL, C. ; BECKER, B. ; WEIS, T.M.: Word-level decision diagrams, WLCDs and division. In: *Proc. International Conference on Computer-Aided Design (ICCAD-98)*, 1998, S. 672–677

[Sch90] SCHRIJVER, A.: *Theory of linear and integer programming.* John Wiley & Sons Ltd., 1990

[Sch97] SCHUBERT, M.: Mixed Analog-Digital Signal Modeling Using Event-Driven VHDL. In: *Brazilian Symposium on Integrated Circuit Design - SBCCI* (1997)

[Sch00] SCHÖNING, U.: *Logik für Informatiker, 5. Auflage.* Spektrum Akademischer Verlag, 2000

[Sha38] SHANNON, C.: A Symbolic Analysis of Relay and Switching Circuits. In: *Transactions AIEE* 57 (1938), S. 713–723

[SK97] STOFFEL, D. ; KUNZ, W.: Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification. In: *Proc. International Conference on Computer-Aided Design (ICCAD-97)*, 1997, S. 394–399

[SK00] STOFFEL, D. ; KUNZ, W.: Convergence Behaviour of Structural FSM Traversal. In: *Proc. of the 3rd Conference on Computer-Aided Technologies in Applied Discrete Mathematics (ICADM-00).* Tomsk, Russia, 2000, S. 176–183

[SK01] STOFFEL, D. ; KUNZ, W.: Verification of Integer Multipliers on the Arithmetic Bit Level. In: *Proc. International Conference on Computer-Aided Design (ICCAD-01)*. San Jose, CA, November 2001, S. 183–189

[SKEG05] SHEKHAR, N. ; KALLA, P. ; ENESCU, F. ; GOPALAKRISHNAN, S.: Equivalence verification of polynomial datapath with fixed-size bit-vectors using finite ring algebra. In: *Proc. International Conference on Computer-Aided Design (ICCAD-05)*, 2005

[SKKK04] STOFFEL, D. ; KARIBAEV, E. ; KUFAREVA, I. ; KUNZ, Wolfgang: Equivalence Checking of Arithmetic Circuits. In: DRECHSLER, Rolf (Hrsg.): *Advanced Formal Verification*. Boston, MA, USA : Kluwer Academic Publishers, 2004

[SS05] SHEINI, H. M. ; SAKALLAH, K. A.: Pueblo: A Modern Pseudo-Boolean SAT Solver. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-05)*. Munich, Germany, March 2005

[SSS00] SHEERAN, M. ; SINGH, S. ; STALMARCK, G.: Checking safety properties using induction and a SAT solver. In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)* Bd. 1954, Springer, November 2000

[Sto99] STOFFEL, D.: *Formal Verification of Sequential Circuits Using Reasoning Techniques*, Johann Wolfgang Goethe - Universität Frankfurt am Main, Dissertation, 1999

[SWWK04] STOFFEL, D. ; WEDLER, M. ; WARKENTIN, P. ; KUNZ, W.: Structural FSM-Traversal. In: *IEEE Transactions on Computer-Aided Design* 23 (2004), May, Nr. 5, S. 598–619

[Tri] *Infineon TriCore 2 Architectural Manual*. – http://www.infineon.com/tricore

[van98] VAN EIJK, C. A. J.: Sequential Equivalence Checking without State Space Traversal. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-98)*. Paris, France, March 1998, S. 618–623

[WSFT04] WINKELMANN, K. ; STOFFEL, D. ; FEY, G. ; TRYLUS, H.: Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-04)*. Paris, France, February 2004

[WSK02] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Improving Structural FSM-Traversal by Constraint-Satisfying Simulation. In: *Proc. IEEE Computer Society Annual Symposium on VLSI 2002 (ISVLSI 2002)*, 2002, S. 151–158

[WSK03] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Using RTL statespace information and state encoding for induction based property checking. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-03)*. Munich, Germany, March 2003

[WSK04a] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Arithmetic reasoning in DPLL-based SAT Solving. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-04)*. Paris, France, February 2004

[WSK04b] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Exploiting State Encoding for Invariant Generation in Induction-Based Property Checking. In: *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-04)*. Yokohama, Japan, January 2004

[WSK05] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Normalization at the arithmetic bit level. In: *Proc. International Design Automation Conference (DAC-05)*, 2005

[WSKer] WEDLER, M. ; STOFFEL, D. ; KUNZ, W.: Frontend Model Generation for SAT-Based Property Checking. In: *Proc. 6. International Conference On ASIC (ASICON 2005)*. Shanghai, China, October 2005 (invited paper)

[ZCR01] ZENG, Z. ; CIESIELSKI, M. ; ROUZEYRE, B.: Functional Test Generation using Constraint Logic Programming. In: *Proc. IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2001)*. Montpellier, France, 2001

[ZKC01] ZENG, Z. ; KALLA, P. ; CIESIELSKI, M.: LPSAT: A Unified Approach to RTL Satisfiability. In: *Proc. Conference on Design, Automation and Test in Europe (DATE-01)*. Munich, Germany, March 2001

Lebenslauf

Name: Markus Wedler geb. Losch
Anschrift: Kettelerstr. 34
67663 Kaiserslautern
Geburtsdatum: 05.06.1972
Geburtsort: Wuppertal/Deutschland
Familienstand: verheiratet, 2 Kinder

Ausbildung

1979-1983	Grundschule in Wuppertal
1983-1992	Gymnasium an der Bayreuther Straße in Wuppertal Abschluss: Abitur
9/1992-2/1995	Ausbildung zum Mathematisch-Technischen Assistenten im Computerlernzentrum der Hoechst AG / Frankfurt am Main
10/1993-5/1999	Berufsbegleitendes Studium Mathematik mit Nebenfach Informatik an der Fernuniversität in Hagen Diplom: 2.6.1999

Berufstätigkeit

2/1995-6/1997	DV-Koordinator der Abteilung Planung und Logistik im Bereich Kunststoffe der Hoechst AG/Werk Knapsack
7/1997-3/1999	Entwicklung Client-Server Software HiServ GmbH (IT-Tochter der Hoechst AG)
7/1999-6/2001	Referent für Aus- und Weiterbildung in den IT-Berufen Provadis GmbH (Aus- & Weiterbildungsdienstleister, aus der Hoechst AG hervorgegangen)
seit 7/2001	Wissenschaftlicher Mitarbeiter von Prof. Dr.-Ing. Wolfgang Kunz am Lehrstuhl für Entwurfsmethodik des Fachbereichs Informatik der Universität Frankfurt am Main und am Lehrstuhl Entwurf Informationstechnischer Systeme des Fachbereichs Elektro- & Informationstechnik der Technischen Universität Kaiserslautern