CHRISTIAN HAUBELT · JÜRGEN TEICH

Digitale Hardware/ Software-Systeme

Spezifikation und Verifikation

eXamen.press

eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Christian Haubelt · Jürgen Teich

Digitale Hardware/ Software-Systeme

Spezifikation und Verifikation

Christian Haubelt
Universität Erlangen-Nürnberg
Lehrstuhl Hardware-Software-Co-Design
Am Weichselgarten 3
91058 Erlangen
Deutschland
haubelt@informatik.uni-erlangen.de

Jürgen Teich
Universität Erlangen-Nürnberg
Lehrstuhl Hardware-Software-Co-Design
Am Weichselgarten 3
91058 Erlangen
Deutschland
teich@informatik.uni-erlangen.de

ISSN 1614-5216
ISBN 978-3-642-05355-9 e-ISBN 978-3-642-05356-6
DOI 10.1007/978-3-642-05356-6
Springer Heidelberg Dordrecht London New York

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© Springer-Verlag Berlin Heidelberg 2010
Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.
Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Einbandentwurf: KuenkelLopka GmbH

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Vorwort

Getrieben durch neue Technologien und Anwendungen wird der Entwurf eingebetteter Systeme zunehmend komplexer. Dabei ist eine Umsetzung als Hardware/Software-System heutzutage der Stand der Technik. Die Minimierung von Fehlern im Entwurf dieser Systeme ist aufgrund deren Komplexität eine der zentralen Herausforderungen unserer heutigen Zeit. Bereits heute wird mehr Aufwand in die Verifikation, also in die Überprüfung der Korrektheit, eines eingebetteten Systems gesteckt als in den eigentlichen Entwurf. Um so erstaunlicher ist es, dass das Thema Verifikation eingebetteter Systeme in der Ausbildung und Lehre nach wie vor keinen entsprechenden Stellenwert besitzt. Ein überwiegender Anteil der Lehrveranstaltungen zu diesem Thema konzentriert sich dabei ausschließlich auf die Verifikation von Hardware oder Software. Aber gerade das Zusammenspiel beider Bestandteile garantiert die Realisierbarkeit komplexer Systeme und bedarf selbstverständlich, wie andere Aspekte, auch der Verifikation.

Dieses Lehrbuch widmet sich der Verifikation digitaler Hardware/Software-Systeme. Es ist als zweiter Band zu dem Buch „Digitale Hardware/Software-Systeme – Synthese und Optimierung“ [426] konzipiert mit dem Ziel, grundlegende Methoden und Verfahren zur Verifikation digitaler Hardware, Software, aber auch von ganzen Systemen zu vermitteln. Der Leser wird damit in die Lage versetzt, die Komplexität und Grenzen der Verifikation zu verstehen sowie Verifikation durchzuführen und dabei ihre Effektivität einzuschätzen. Dies ist die Voraussetzung, um im praktischen Einsatz geeignete Verifikationsziele zu setzen, passende Verifikationsmethoden auszuwählen sowie die damit verbundenen Risiken abzuschätzen.

Im Gegensatz zu vielen anderen Lehrbüchern, welche oftmals den Schwerpunkt auf den Aspekt Hardware oder Software legen, zielt der vorliegende Band darauf ab, dem Leser genau die notwendigen Methoden an die Hand zu geben, um moderne eingebettete Systeme bestehend aus Hardware- und Software-Komponenten auf ihren korrekten Entwurf hin zu überprüfen. Hierzu gehört sowohl die funktionale Verifikation als auch die Verifikation des Zeitverhaltens. Der Stoff ist Lehrinhalt von Vorlesungen, die teilweise seit mehreren Jahren vom Lehrstuhl für Hardware-Software-Co-Design angeboten und mit großer Resonanz von den Studierenden der Friedrich-Alexander-Universität Erlangen-Nürnberg angenommen werden.

Beim Lesen dieses Buches kann man feststellen, dass Verifikationsmethoden für Hardware und Software auf identischen Prinzipien aufsetzen und ähnliche Lösungen hervorbringen. Dies ist eine analoge Erkenntnis zu der im ersten Band aufgezeigten Dualität für den Entwurf von Hardware und Software. Als gemeinsames Modell für beide Bände dient deshalb das Doppeldachmodell, welches den idealisierten Entwurfsfluss für digitale Hardware/Software-Systeme darstellt. In diesem Band wird ein Doppeldachmodell für den Verifikationsprozess vorgestellt, das zur Definition grundlegender Verifikationsaufgaben in der Entwicklung von digitalen Hardware/Software-Systemen dient. Die verwendeten Modellierungsansätze für digitale Systeme, namentlich Petri-Netze, endliche Automaten, Datenflussmodelle sowie ausgewählte heterogene Modelle, bilden eine weitere Gemeinsamkeit beider Bände. Eine dritte Gemeinsamkeit, die beide Bände verbindet, ist eine Entwurfsumgebung mit dem Namen SystemCoDesigner, welche als aktuelles Forschungsprojekt am Lehrstuhl für Hardware-Software-Co-Design an der Universität Erlangen-Nürnberg entwickelt wird. SystemCoDesigner setzt die im Band Synthese und Optimierung beschriebenen Synthese- und Optimierungsverfahren um und unterstützt die in diesem Band beschriebenen Spezifikations- und Verifikationsmethodiken.

Die Autoren möchten Ihren Dank den anonymen Gutachtern des Springer-Verlags aussprechen, die maßgeblich geholfen haben, die Idee zu diesem Buch zu fokussieren. Darüber hinaus möchten wir uns bei den wissenschaftlichen Mitarbeitern bedanken, die durch ihre Ideen und ihr Mitwirken geholfen haben, die Entwurfsumgebung SystemCoDesigner zu verwirklichen. Insbesondere möchten wir uns bei Dipl.-Inf. Michael Glaß, Dipl.-Inf. Martin Streubühr und Dipl.-Inf. Christian Zebelein bedanken, die durch ihre zahlreichen Vorschläge geholfen haben, das vorliegende Buch zu verbessern. Unserer besonderer Dank gilt Prof. Dr. rer. nat. Rolf Wanka, der uns in langen Diskussionen geholfen hat, Ergebnisse aus dem Bereich der Theoretischen Informatik zu interpretieren. Schließlich möchten wir uns bei Dipl.-Inf. Jens Gladigau bedanken, der durch seine Kommentare und Anregungen maßgeblich zum Gelingen dieses Buches beigetragen hat.

Erlangen, im Frühjahr 2010 C. Haubelt und J. Teich

Inhaltsverzeichnis

1 Einleitung ... 1
1.1 Motivation ... 1
1.2 Der Verifikationsprozess ... 11
1.2.1 Das V-Modell ... 13
1.2.2 Das Doppeldachmodell des Entwurfsprozesses ... 14
1.2.3 Das Doppeldachmodell des Verifikationsprozesses ... 18
1.3 Eine kurze Geschichte der Verifikation ... 22
1.4 Beispiele ... 29
1.5 Ausblick ... 34
1.6 Literaturhinweise ... 34

2 Spezifikation digitaler Systeme ... 37
2.1 Wie spezifiziert man ein System? ... 37
2.2 Formale Verhaltensmodelle ... 41
2.2.1 Petri-Netze ... 41
2.2.2 Endliche Automaten ... 47
2.2.3 Datenflussgraphen ... 51
2.2.4 Heterogene Modelle ... 56
2.3 Ausführbare Verhaltensmodelle ... 59
2.3.1 SystemC ... 60
2.3.2 SysteMoC ... 68
2.4 Formale Spezifikation funktionaler Anforderungen ... 72
2.4.1 Temporale Strukturen ... 73
2.4.2 Temporale Aussagenlogik ... 75
2.4.3 Die Zusicherungssprache PSL ... 83
2.5 Formale Spezifikation nichtfunktionaler Anforderungen ... 88
2.6 Literaturhinweise ... 91

3 Verifikation ... 95
3.1 Verifikationsaufgabe, -ziel und -methode ... 95
3.1.1 Verifikationsziel ... 97

3.1.2 Verifikationsmethode ... 99
3.2 Beobachtbarkeit und Steuerbarkeit ... 107
3.3 Gesteuerte zufällige Simulation ... 111

4 Äquivalenzprüfung ... 115
4.1 Implizite Äquivalenzprüfung ... 117
4.1.1 Kanonische Funktionsrepräsentationen ... 117
4.1.2 Taylor-Expansions-Diagramme ... 120
4.1.3 Reduktion und Normalisierung von TEDs ... 120
4.1.4 Kanonizität von TEDs ... 124
4.1.5 Implizite Äquivalenzprüfung mit TEDs ... 125
4.2 Explizite Äquivalenzprüfung ... 129
4.2.1 Regressionstest ... 131
4.2.2 Bereichstest ... 133
4.2.3 Pfadbereichstest ... 134
4.2.4 Fehleroffenbarende Unterbereiche ... 139
4.3 Sequentielle Äquivalenzprüfung ... 141
4.3.1 Automaten-Äquivalenz ... 142
4.3.2 Zustandsraumtraversierung ... 144
4.3.3 Symbolische Zustandsraumtraversierung ... 148
4.3.4 Erzeugung von Gegenbeispielen ... 151
4.4 Strukturelle Äquivalenzprüfung ... 152
4.5 Literaturhinweise ... 154

5 Eigenschaftsprüfung ... 155
5.1 Prüfung funktionaler Eigenschaften ... 156
5.1.1 Eigenschaftsprüfung auf Erreichbarkeitsgraphen ... 158
5.1.2 Strukturelle Eigenschaftsprüfung von Petri-Netzen ... 167
5.1.3 Partialordnungsreduktion ... 172
5.2 Explizite Modellprüfung ... 178
5.2.1 CTL-Modellprüfung ... 179
5.2.2 LTL-Modellprüfung ... 185
5.2.3 Zusicherungsbasierte Eigenschaftsprüfung ... 188
5.3 Symbolische Modellprüfung ... 197
5.3.1 BDD-basierte CTL-Modellprüfung ... 197
5.3.2 SAT-basierte Modellprüfung ... 199
5.4 Prüfung nichtfunktionaler Eigenschaften ... 207
5.4.1 Zeitbehaftete Petri-Netze ... 207
5.4.2 Zeitbehaftete Automaten ... 214
5.4.3 Zeitbehaftete SDF-Graphen ... 222
5.5 Literaturhinweise ... 232

6 Hardware-Verifikation 235
6.1 Äquivalenzprüfung kombinatorischer und sequentieller Schaltungen 236
6.1.1 Implizite Äquivalenzprüfung auf der Logikebene 236
6.1.2 Explizite Äquivalenzprüfung auf der Logikebene 246
6.1.3 Formale explizite Äquivalenzprüfung von Schaltwerken 258
6.1.4 Strukturelle Äquivalenzprüfung auf der Logikebene 263
6.2 Äquivalenzprüfung arithmetischer Schaltungen 273
6.2.1 Implizite Äquivalenzprüfung auf der Architekturebene 273
6.2.2 Äquivalenzprüfung zwischen Architektur- und Logikebene . 280
6.2.3 Äquivalenzprüfung auf der Architekturebene 283
6.3 Formale Verifikation von Prozessoren 291
6.3.1 Äquivalenzprüfung für Prozessoren mit Fließbandverarbeitung 293
6.3.2 Berücksichtigung von Multizyklen-Funktionseinheiten, Ausnahmebehandlung und Sprungvorhersage 308
6.3.3 Äquivalenzprüfung für Prozessoren mit dynamischer Instruktionsablaufplanung 313
6.4 Funktionale Eigenschaftsprüfung 323
6.4.1 Zusicherungsbasierte Eigenschaftsprüfung 323
6.4.2 SAT-basierte Modellprüfung 331
6.5 Zeitanalyse 345
6.5.1 Zeitanalyse synchroner Schaltungen 345
6.5.2 Zeitanalyse latenzinsensitiver Systeme 351
6.6 Literaturhinweise 356

7 Software-Verifikation 361
7.1 Formale Äquivalenzprüfung eingebetteter Software 362
7.1.1 Äquivalenzprüfung von Assemblerprogrammen 362
7.1.2 Strukturelle Äquivalenzprüfung von Assemblerprogrammen 368
7.1.3 Äquivalenzprüfung von C-Programmen 373
7.2 Testfallgenerierung zur simulativen Eigenschaftsprüfung 391
7.2.1 Funktionsorientierte Testfälle 391
7.2.2 Kontrollflussorientierte Testfälle 400
7.2.3 Datenflussorientierte Testfälle 410
7.3 Formale funktionale Eigenschaftsprüfung von Programmen 416
7.3.1 Statische Programmanalyse 416
7.3.2 SAT-basierte Modellprüfung von C-Programmen 422
7.3.3 Modellprüfung durch Abstraktionsverfeinerung 425
7.4 Zeitanalyse 431
7.4.1 BCET- und WCET-Analyse 432
7.4.2 Echtzeitanalyse für Einprozessorsysteme 438
7.5 Literaturhinweise 448

8 Systemverifikation . . . 451
8.1 Funktionale Eigenschaftsprüfung von SystemC-Modellen . . . 452
8.1.1 Symbolische CTL-Modellprüfung von SysteMoC-Modellen 452
8.1.2 Modellprüfung von SystemC-Modellen . . . 466
8.1.3 Formale Modellprüfung von Transaktionsebenenmodellen . . 476
8.1.4 Zusicherungsbasierte Eigenschaftsprüfung für Transaktionsebenenmodelle . . . 484
8.2 Zeitanalyse auf Systemebene . . . 490
8.2.1 Simulative Zeitbewertung . . . 492
8.2.2 Kompositionale Zeitanalyse über Ereignisströme . . . 499
8.2.3 Modulare Zeitanalyse mit RTC . . . 508
8.3 Literaturhinweise . . . 520

Anhang . . . 523

Notation . . . 523
A.1 Mengen . . . 523
A.2 Relationen und Funktionen . . . 524
A.3 Aussagenlogik . . . 527
A.4 Prädikatenlogik erster Ordnung . . . 528
A.5 Graphen . . . 529

Binäre Entscheidungsdiagramme . . . 533
B.1 Entscheidungsdiagramme . . . 533
B.2 Binäre Entscheidungsdiagramme . . . 534
B.3 Verallgemeinerte binäre Entscheidungsdiagramme . . . 537

Algorithmen . . . 541
C.1 Klassifikation von Algorithmen . . . 541
C.2 SAT-Solver . . . 542
C.3 SMT-Solver . . . 551
C.4 CTL-Fixpunktberechnung . . . 556

Literatur . . . 561

Sachverzeichnis . . . 587

1 Einleitung

Dieses einleitende Kapitel vermittelt einen Einblick, warum die Verifikation ein unverzichtbarer Bestandteil des Entwurfs von digitalen Hardware/Software-Systemen darstellt. Dabei werden die wesentlichen Aspekte der Verifikation vorgestellt.

1.1 Motivation

Digitale Hardware/Software-Systeme sind aus der heutigen Gesellschaft nicht mehr wegzudenken. Oftmals treten sie in Form sog. *eingebetteter Systeme* auf. Eingebettete Systeme sind Computer, die für einen speziellen Einsatz konzipiert und optimiert sind. Heutzutage sorgen eingebettete Systeme dafür, dass wir mobil auf Daten zugreifen können, Autos fahren, Flugzeuge fliegen, Patienten überwacht werden etc.

Im Unterschied zu eingebetteten Systemen werden *Vielzweckrechner*, wie beispielsweise Server und PCs, vorrangig auf deren Rechengeschwindigkeit optimiert. Während Verlustleistung, Platzbedarf und Kosten bei Vielzweckrechnern eher untergeordnete Ziele sind, werden eingebettete Systeme primär bezüglich dieser nichtfunktionalen Ziele optimiert. Eine ausreichende Rechengeschwindigkeit gilt lediglich als Randbedingung, die es zu erfüllen gilt. Beispielsweise ist es nicht tolerierbar, dass ein Mobiltelefon bereits nach wenigen Minuten oder gar Sekunden die gesamte, in einem vollständig geladenen Akku gespeicherte Energie verbraucht. Auch gibt es die Kundenanforderungen, dass Mobiltelefone klein und leicht zu sein haben. Diesen Zielen entgegen stehen allerdings die Forderungen nach immer komplexeren Anwendungen, die auf dem eingebetteten System ausgeführt werden. Ein modernes Mobiltelefon sollte heutzutage neben den Sprach- und Datendiensten zumindest mit einer integrierte Digitalkamera, einem MP3-Player und einem Internet-Zugang ausgestattet sein. Auch das Abspielen von Videos auf dem Mobiltelefon gehört bereits zum Standard.

Auch in anderen Bereichen kann man eine zunehmende Komplexität der Anwendungen sehen: Ein prominentes Beispiel sind *Fahrerassistenzsysteme* in heutigen Fahrzeugen mit gehobener Ausstattung. Neben ABS (Antiblockiersystem) und ESP

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_1, © Springer-Verlag Berlin Heidelberg 2010

(elektronisches Stabilitätsprogramm) gehören heutzutage Spur- und Personenerkennung sowie Einparksysteme zum Funktionsumfang. Die steigende Komplexität ist hierbei auch der Grund, warum immer häufiger eingebettete Systeme aus spezialisierten, interagierenden Hardware- und Software-Komponenten bestehen.

Durch den steigenden Funktionsumfang wird auch die Implementierung eingebetteter Systeme immer komplexer. Zukünftige Systeme werden nicht mehr lediglich aus einem zentralen Prozessor und etwas Zusatzhardware, sondern aus vielen zusammenwirkenden Prozessorkernen und Hardware-Beschleunigern, die auf einem einzelnen Chip integriert werden (engl. *Multi-Processor System-on-Chip, MPSoC*), bestehen. Ein Beispiel hierfür ist in Abb. 1.1 zu sehen: Man sieht im oberen Teil mehrere Prozessoren zu einem Prozessorsubsystem zusammengefasst. Jeder einzelne Prozessor verfügt über einen eigenen Cache für Daten und Instruktionen. Hardware-Beschleuniger sind in einer eigenen Einheit zusammengefasst. Die Kommunikation zwischen Prozessoren und Hardware-Beschleunigern erfolgt über ein blockierungsfreies Verbindungsnetzwerk. Der Datentransport zwischen Prozessoren und Hauptspeicher wird durch ein spezielles Speichersubsystem mit eigenem Cache gesteuert, welches ebenfalls über das Verbindungsnetzwerk an die anderen Komponenten des MPSoC angebunden ist. Schließlich ist eine weitere Kommunikationseinheit (I/O) direkt auf dem Chip integriert, um Daten schnell mit Komponenten außerhalb des Chips austauschen zu können.

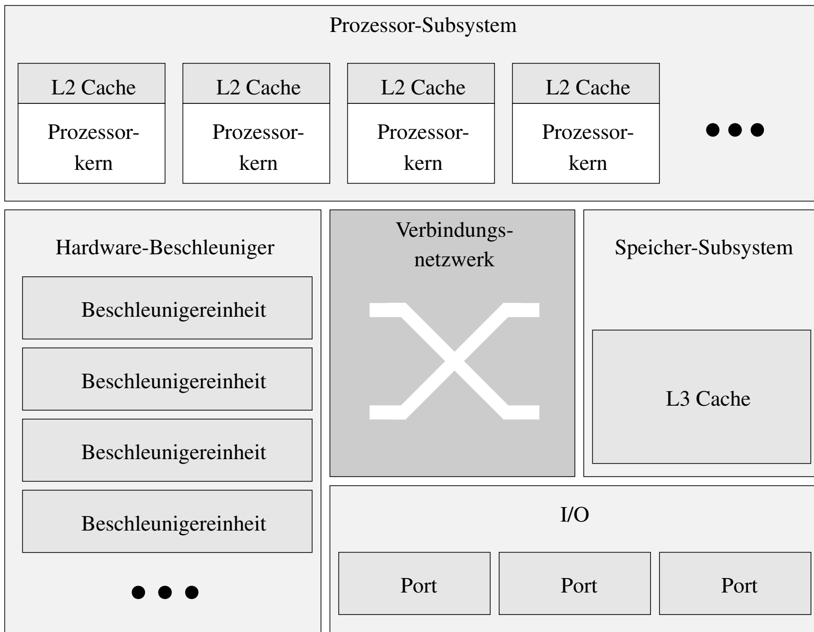


Abb. 1.1. Beispiel eines Mehrprozessorsystems (MPSoC) nach [240]

Während in Abb. 1.1 noch eine recht überschaubare Anzahl an Prozessoren und Hardware-Beschleunigern zu sehen ist, wird dies in zukünftigen Systemen nicht mehr so sein. Die International Technology Roadmap for Semiconductors (ITRS) [240] geht in ihrem Bericht von 2007 davon aus, dass sich das Mooresche Gesetz auch auf zukünftige MPSoCs übertragen lässt, d. h. man wird einen exponentiellen Anstieg der Prozessoren auf einem einzelnen Chip feststellen können (siehe Abb. 1.2). Das Mooresche Gesetz besagt, dass sich die Komplexität integrierter Schaltungen etwa alle zwei Jahre verdoppelt. Das Gesetz wurde 1965 von George Moore aufgestellt [332] und gilt bis heute, wobei Moore mit Komplexität die Anzahl der Gatter meinte.

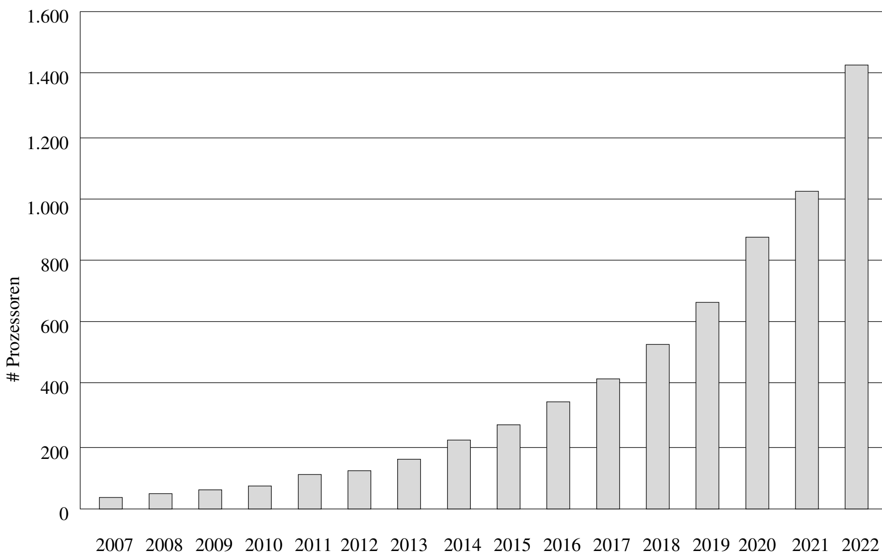


Abb. 1.2. Die ITRS geht davon aus, dass im Jahr 2020 nahezu 1.000 Prozessoren auf einem einzelnen Chip integrierbar sind [240]

Mit steigendem Funktionsumfang erobern eingebettete Systeme immer neue Anwendungsgebiete in unserem täglichen Leben, so dass sie häufig gar nicht mehr als solche wahrgenommen werden. Diese oftmals unauffälligen Helfer treten häufig erst in unser Bewusstsein, wenn sie ihre Aufgabe nicht mehr zu unserer Zufriedenheit erfüllen oder sich nicht mehr entsprechend ihrer Vorgaben verhalten. Betrachtet man die enorme Anzahl solcher Systeme, mit denen wir tagtäglich interagieren, so verwundert es nicht, dass es hin und wieder zu Störungen kommt. Aber gerade wegen dieser „wenigen Störungen" ist es ein beeindruckendes Phänomen unserer Gesellschaft, dass wir solchen Systemen immer öfter unser Geld oder sogar unser Leben anvertrauen.

Korrektheit und Fehler

Als Benutzer eines eingebetteten Systems geht man davon aus, dass dieses korrekt funktioniert. Diese Erwartung ist aber leider so schwierig zu erfüllen, wie sie trivial scheint. Dies beginnt bereits damit, wie sich Korrektheit eigentlich definiert. Der IEEE-Standard 610.12 [232] definiert die Korrektheit eines Systems auf drei unterschiedliche Arten:

1. Korrektheit ist der Grad der Fehlerfreiheit eines Systems in seiner Spezifikation, in seinem Entwurf und seiner Implementierung.
2. Korrektheit ist der Grad, mit dem ein System die spezifizierten Anforderungen erfüllt.
3. Korrektheit ist der Grad, mit dem ein System die Kundenanforderungen und Kundenerwartungen erfüllt. Unabhängig davon, ob diese spezifiziert sind oder nicht.

Hier kann man zumindest bereits erahnen, wie umfangreich der Spielraum bei der Interpretation von *Korrektheit eines Systems* ausfallen kann.

Genau dies wird deshalb im Folgenden genauer betrachtet. Hierzu wird zunächst die *Entwicklung* eines eingebetteten Systems näher untersucht. Die Entwicklung eines digitalen Hardware/Software-Systems (siehe Abb. 1.3) beginnt mit der Produktidee. Nach der *Anforderungsanalyse* entsteht eine Spezifikation, welche die Grundlage für den Entwurfsprozess bildet. Die Umsetzung der Spezifikation in eine Implementierung wird als *Entwurf* bezeichnet. Erfolgt die Umsetzung einer Spezifikation in eine Implementierung automatisch, so spricht man von *Synthese* [426]. Die Implementierung wiederum ist die Basis für die Herstellung eines Produktes, welches anschließend in *Betrieb* geht.

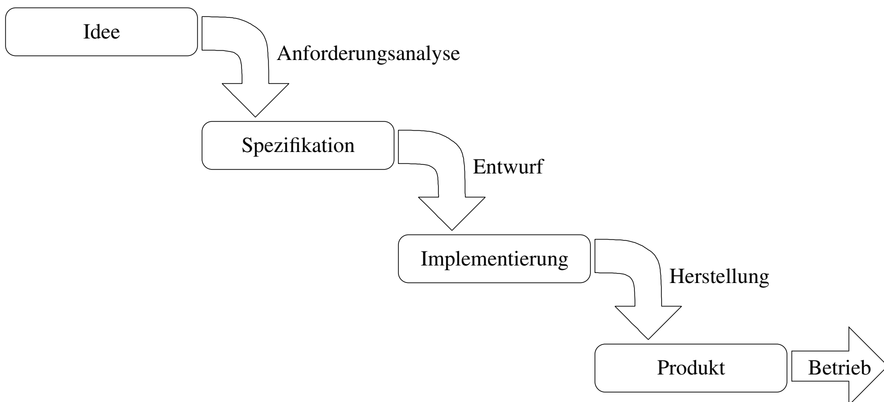


Abb. 1.3. Entwicklung eines eingebetteten Systems

Im Betrieb eines eingebetteten Systems kann es aus verschiedensten Gründen zu einem *Fehlverhalten* bzw. einem *Ausfall* (engl. *failure*) kommen. Fehlverhalten oder Ausfälle wird man erst bei der Benutzung eines Systems feststellen können. Sie sind Auswirkungen von *Fehlern* (engl. *faults*) im System. Dementsprechend sind Fehler die Ursachen für ein Fehlverhalten bzw. einen Ausfall.

Es gibt vielfältige Möglichkeiten, wie Fehler entstehen können. Ursachen für Fehler sind *Irrtümer* (engl. *errors*) oder Tippfehler. Irrtümer werden oftmals durch eine falsche Interpretation ungenauer Informationen hervorgerufen. Irrtümer, Fehler und Fehlverhalten stehen oft in einer Beziehung zueinander [305]: Einerseits kann es passieren, dass das zu entwickelnde System fehlerhaft spezifiziert wurde. Hierbei kann es sich schon um einen *Spezifikationsfehler* handeln, sobald die Spezifikation nicht eindeutig oder nicht vollständig ist, d. h. ein mögliches Szenario im späteren Betrieb vergessen wurde. Ob dies schwerwiegende Folgen nach sich zieht, hängt zum einen von der endgültigen Umsetzung des Systems und zum anderen davon ab, ob das Szenario tatsächlich eintritt. Spezifikationsfehler können nur durch eine ausgiebige *Validierung* minimiert werden. Hierbei wird immer wieder kritisch hinterfragt, ob die Spezifikation das gewünschte Verhalten beschreibt und ob die Spezifikation vollständig, eindeutig und konsistent ist.

Auf der anderen Seite können Fehler beim Entwurf des Systems entstehen, also bei der Umsetzung der Spezifikation in eine *Implementierung* (siehe Abb. 1.3). Hierbei handelt es sich um sog. *Entwurfsfehler*. Entwurfsfehler entstehen beispielsweise, wenn ein fehlerhaftes Synthesewerkzeug eingesetzt wird oder wenn ein Entwickler die Spezifikation falsch interpretiert. Unter *Verifikation* versteht man den Prozess, der den korrekten Entwurf eines digitalen Systems bezüglich einer Spezifikation nachweist. Dies umfasst zum einen die *funktionale Korrektheit* des Systems. Andererseits kann es auch durch Fehler, z. B. im zeitlichen Verhalten, zu schwerwiegenden Folgen kommen, weshalb auch die Korrektheit *nichtfunktionaler Eigenschaften* zu zeigen ist.

Man sieht, dass die Verifikation versucht, die Frage zu beantworten, ob eine Implementierung korrekt im Sinne der Spezifikation ist. Erfüllt eine Implementierung alle Anforderungen der Spezifikation und sind ferner alle von der Spezifikation geforderten Funktionen umgesetzt, gilt die Implementierung als korrekt. Falls gewisse Fälle, z. B. fehlerhafte oder widersprüchliche Eingabedaten, in der Spezifikation nicht berücksichtigt wurden, so kann die Implementierung für diese Fälle durchaus fehlerhaft sein – die Implementierung bleibt aber weiterhin korrekt. Eine *robuste* Implementierung ist eine Implementierung, die auf viele mögliche Betriebsszenarien korrekt reagiert. Somit ist Robustheit aber auch keine Eigenschaft der Implementierung, sondern genau genommen der Spezifikation, da wie oben dargestellt eine Implementierung korrekt bezüglich ihrer Spezifikation ist.

Erfolgt die Herstellung einer Implementierung, so entsteht das Produkt (siehe Abb. 1.3). Hierbei können *Herstellungsfehler* auftreten, z. B. aufgrund von Defektstellen auf dem Wafer, oder Codierungsfehlern in der Software. Herstellungsfehlern begegnet man heutzutage mit speziellen *Testverfahren* [76, 283, 305, 306]. Hardware wird beispielsweise gezielt auf Kurzschlüsse zwischen zwei Signalleitungen (engl. *bridging faults*), konstante Ausgänge von Gattern (Haftfehler, engl. *stuck-at faults*)

oder Leerlauf der Eingänge (engl. *open fault*) getestet. Bei Software wird getestet, ob es beispielsweise nichterreichbaren Quelltext (engl. *dead code*) oder falsche Bereichsgrenzen bei Schleifen oder Arrays gibt. Die Herausforderung dabei ist es, die Implementierung so zu gestalten, dass diese Herstellungsfehler, sofern sie auftreten, steuerbar und beobachtbar sind. Beim Hardware-Entwurf helfen hierbei spezielle Testschaltungen, die auf dem Chip integriert und für die eigentliche Funktionalität nicht benötigt werden. In Software können sog. *Zusicherungen* (engl. *assertions*) Verwendung finden. Da das Auffinden von Herstellungsfehlern entscheidend für die Produktqualität ist, ist die Testbarkeit und der Entwurf von testbaren Systemen ein zentrales Forschungsthema geworden (engl. *design for testability*) [282, 1].

Nach der Herstellung geht das System in den Betrieb (siehe Abb. 1.3). Auch während des Betriebs kann es zu Fehlern kommen, z. B. durch den Ausfall von Teilkomponenten bedingt durch Verschleiß oder durch unvorhergesehene Betriebsmodi. *Betriebsfehlern* wirkt man durch *Fehlerkorrekturmaßnahmen* entgegen. Das Ziel ist es, die *Zuverlässigkeit* des Systems zu erhöhen [246, 51, 458, 267].

Fehler können zu unterschiedlichen Zeitpunkten in der Entwicklung eines eingebetteten Systems entstehen. Bis zu dem Zeitpunkt, da diese Fehler erkannt werden, kann allerdings unterschiedlich viel Zeit vergehen. Je später ein Fehler erkannt wird, desto teurer ist im Allgemeinen dessen Korrektur. Wird ein Fehler vor dem Entwurf erkannt, erzeugt dies in der Regel moderate Kosten, da lediglich die Spezifikation anzupassen ist. Wird ein Fehler nach dem Entwurf, aber vor der Herstellung entdeckt, entstehen überschaubare Kosten. In diesem Fall müssen Entwurfsentscheidungen, die sich im schlimmsten Fall auf die Hardware, die Software aber auch die Schnittstellen beziehen, neu überdacht und getroffen werden. Eventuell ist eine Änderung der Spezifikation ebenfalls erforderlich.

Fehler, die erst nach der Herstellung, aber dennoch vor der Inbetriebnahme erkannt werden, können bereits enorme Kosten nach sich ziehen. Dies gilt insbesondere für Hardware, die als ASIC (engl. *Application Specific Integrated Circuit*), also als anwendungsspezifischer Chip, implementiert wurde. Ein Fehler kann in diesem Fall eine komplette Neuentwicklung nach sich ziehen. Dies bedeutet, dass es zu einer Änderung der Spezifikation, einer Überprüfung der Entwurfsentscheidungen und einer erneuten Herstellung kommen kann. Insbesondere letzteres kann bei Maskenkosten von mehreren 100.000€ schnell hohe Kosten verursachen. Unüberschaubar können schließlich die Kosten werden, wenn ein Fehler erst im Betrieb erkannt wird, wie die folgenden Beispiele illustrieren:

Am 22. Juli 1962 startete die NASA (National Aeronautics and Space Administration) eine Trägerrakete mit der Mariner 1 Venus-Sonde. 293 Sekunden nach dem Start kam die Rakete vom Kurs ab und musste gesprengt werden. Die Steuerung rechnete mit Rohdaten anstatt mit geglätteten Messwerten. Die Ursache lag in der fehlerhaften Umsetzung der Spezifikation, indem an einer Stelle im Programm statt einem Komma ein Punkt verwendet wurde. Die Kosten für diese Verwechslung werden auf damals 18,5 Millionen US$ geschätzt.

Im November 1994 wurde ein Fehler im Pentium-Prozessor der Firma Intel bekannt. Zu diesem Zeitpunkt war der Prozessor bereits eineinhalb Jahre auf dem Markt. Bei Gleitkomma-Divisionen mit bestimmten Werten lieferte der Prozessor

ein falsches Ergebnis, so ergab z. B. die Rechnung

$$4.195.825,0 - (4.195.825,0/3.145.727,0) \cdot 3.145.727,0 = 0$$

auf einem fehlerhaften Pentium-Prozessor das Ergebnis 256. Der Grund hierfür lag in dem verwendeten Divisionsalgorithmus, der auf eine Tabelle mit 1.066 Werten zugreift. Hiervon wurden allerdings aufgrund einer fehlerhaft programmierten Schleife lediglich 1.061 Werte geladen. Laut Intel würde bei einem Normalanwender statistisch gesehen dieser Fehler nur alle 27.000 Jahre einmal auftreten. Andere Stimmen hingegen behaupteten, dass dieser Fehler sogar alle sechs Stunden auftreten könne. Am Ende spielte es keine Rolle, wer Recht hatte. Intel musste ca. eine Million fehlerhafte Prozessoren austauschen. Der Verlust für Intel wird mit mehr als 400 Millionen US$ beziffert.

Ähnlich teuer kam der ESA (European Space Agency) ein Software-Fehler beim Erstflug der Ariane 5 Rakete zu stehen. Am 4. Juni 1996 sprengte sich eine Ariane 5 Rakete 36,7 Sekunden nach dem Start zusammen mit vier Satelliten an Bord selbst in die Luft. Grund für die Selbstzerstörung war der Absturz des Bordcomputers, als er versuchte, eine 64 Bit Gleitkommazahl in eine 16 Bit Ganzzahl zu wandeln. Der Wert der Zahl war größer als 2^{15} und erzeugte einen Überlauf. Als Folge stellte das Lenksystem einen Fehler fest und die Flugkontrolle wurde an eine zweite Einheit übergeben, welche die Zerstörung der Rakete aus Sicherheitsgründen einleitete. Die verwendete Software stammte noch aus der Ariane 4 und diente nur Startvorbereitungen. Für den eigentlichen Flug wurde sie nicht benötigt. Der Schaden wird auf ca. 500 Millionen US$ für die Rakete plus 7 Milliarden US$ Entwicklungskosten für die Satelliten beziffert.

An diesen Beispielen sieht man, dass Fehler, die erst im Betrieb erkannt werden, unüberschaubare Kosten nach sich ziehen können. Neben monetären Verlusten, die ein Hersteller hierbei direkt zu spüren bekommt, kommt häufig die Schädigung des öffentlichen Ansehens eines Unternehmens. Dieser Reputationsverlust führt dann wahrscheinlich zu weiteren Verlusten.

Der Zusammenhang zwischen Fehlerentstehung, -korrektur und -kosten ist in Abb. 1.4 für ein Software-Projekt dargestellt [330]. Neben den vier oben diskutierten Phasen (Anforderungsanalyse, Entwurf, Herstellung und Betrieb) sind in der Abbildung zwei Verifikationsphasen zusätzlich dargestellt. Während der Spezifikation entstehen verhältnismäßig wenig Fehler. Da diese aber nicht sofort erkannt werden, können auch diese Fehler signifikante Kosten erzeugen. Der Großteil der Fehler wird allerdings im Entwurf und der Herstellung gemacht. Dem gegenüber steht eine relativ kleine Anzahl an bis dahin entdeckten Fehlern. Dies ist besonders bedauerlich, da die Kosten zur Fehlerkorrektur vor der Herstellung noch recht moderat sind (in [330] wird von 500 DM je Fehlerkorrektur ausgegangen). Ein Software-Produkt zu diesem Zeitpunkt ungeprüft in Betrieb zu nehmen, wäre nicht sinnvoll.

Aus diesem Grund folgt nach der Herstellung die Verifikation. Die Verifikation ist in Abb. 1.4 mit zwei Phasen angegeben. In der *Modulverifikation* werden einzelne Software-Einheiten, z. B. Klassen, auf ihre Korrektheit überprüft. Auch wenn man hierbei eine erhebliche Anzahl an Fehlern erkennen und korrigieren kann, sieht man, dass der größte Teil von Fehlern erst bei der Interaktion aller Module auftritt.

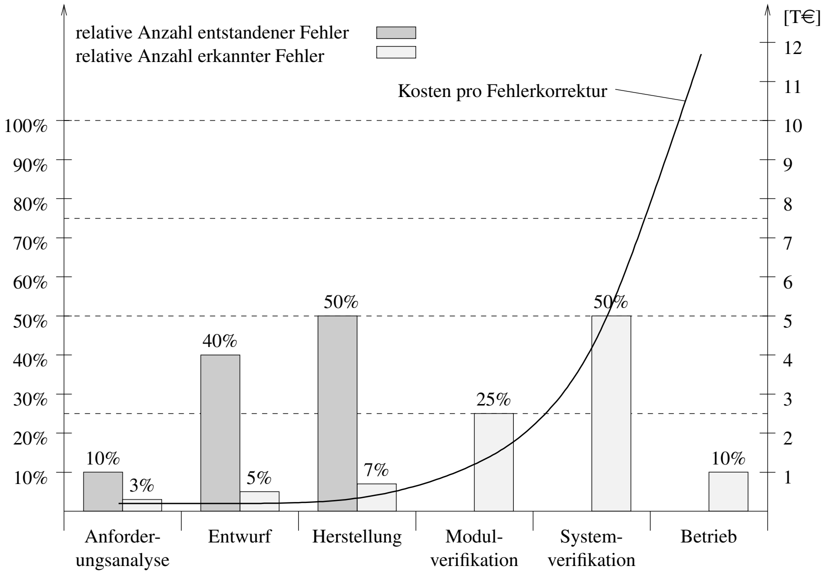


Abb. 1.4. Anzahl entstandener und erkannter Fehler sowie Kosten pro Fehlerkorrektur [330]

Diese Fehler werden in der Systemverifikation erkannt. Man sieht in Abb. 1.4, dass sich der zusätzliche Aufwand der Verifikation in den Kosten zur Fehlerkorrektur niederschlägt. So vervierfachen sich bereits die Kosten aus der Entwurfsphase in der Modulverifikation. Eine Verzwölffachung erkennt man in der Systemverifikation.

Schließlich zeigt Abb. 1.4, dass in diesem Fall ca. 10% der Fehler unentdeckt geblieben sind. Ob dies Auswirkungen hat, hängt davon ab, ob es durch diese Fehler zu einem Fehlverhalten oder Ausfall kommt. Kommt es allerdings zu einem solchen Fehlverhalten oder Ausfall, entstehen hohe Kosten zur Fehlerkorrektur. Die Anzahl der Fehler, die bei Inbetriebnahme eines eingebetteten Systems noch unentdeckt sind, ist ein umgekehrt proportionales Qualitätsmaß für ein solches System. Ziel der Verifikation ist somit die *Erhöhung* der Qualität eines Systems. Die folgenden Darstellungen finden sich in ähnlicher Form auch in [305].

Qualität

Der Begriff *Qualität* ist bisher sehr vage formuliert worden und beschreibt die Beschaffenheit eines Systems bezüglich seiner Eignung, spezifizierte oder abgeleitete *Qualitätsanforderungen* zu erfüllen. Die Qualitätsanforderungen sind die Gesamtheit aller Einzelanforderungen an die Beschaffenheit eines Systems. Die Beurteilung der Qualität eines Systems basiert auf sog. *Qualitätsmerkmalen*, die Eigenschaften eines Produktes darstellen ohne diese näher zu quantifizieren. Beispiele für Qualitätsmerkmale sind Gefahrlosigkeit, Zuverlässigkeit, Verfügbarkeit, Robustheit, Speicher- und Laufzeiteffizienz, Änderbarkeit, Portierbarkeit, Prüfbarkeit und Be-

nutzbarkeit. Für Hersteller stehen primär die Qualitätsmerkmale Änderbarkeit, Portierbarkeit und Prüfbarkeit im Vordergrund. Ein Kunde hingegen wird sich vor allem für Gefahrlosigkeit, Zuverlässigkeit, Verfügbarkeit, Robustheit, Speicher- und Laufzeiteffizienz und Benutzbarkeit interessieren. Da für einen Hersteller aber auch die Kundenzufriedenheit von großer Bedeutung ist, wird er sich automatisch auch dieser Qualitätsmerkmale annehmen.

Die Quantifizierung von Qualitätsmerkmalen erfolgt in sog. *Qualitätsmaßen*. Dies sind Maße, die Rückschlüsse auf die Ausprägung von Qualitätsmerkmalen zulassen. Als Beispiel sei die sog. *MTTF* (engl. *Mean Time To Failure*) als Qualitätsmaß für die Zuverlässigkeit genannt. Diese bezeichnet die erwartete Zeitspanne bis zum Ausfall eines Systems.

Zwischen Qualitätsmerkmalen können Wechselwirkungen und Zusammenhänge existieren. Eine Verbesserung eines Qualitätsmerkmals führt somit oftmals zur Verschlechterung eines anderen Qualitätsmerkmals. So geht z. B. die Gefahrlosigkeit eines Systems oftmals zu Lasten der Verfügbarkeit, da in einem sicheren System kritische Betriebszustände erkannt werden und das System in einen sicheren Zustand übergeht. In einem solchen sicheren Zustand werden Systeme zunächst einmal nicht die volle Funktionalität zur Verfügung stellen. Somit ist die Verfügbarkeit reduziert. Damit steht auch fest, dass eine Forderung nach allumfassender Qualität unsinnig ist. Wie im Fall des Entwurfs und der Optimierung von digitalen Hardware/Software-Systemen [426], gibt es im Allgemeinen nicht eine optimale Lösung, sondern eine Menge sog. *Pareto-optimaler* Lösungen.

Qualitätsmerkmale, und somit die Eigenschaften eines Produktes, können in *funktionale* und *nichtfunktionale Qualitätsmerkmale* unterteilt werden. Funktionale Qualitätsmerkmale beziehen sich stets auf die zu erbringende Funktion eines Systems. Beispiele für funktionale Qualitätsmerkmale sind Gefahrlosigkeit und Verfügbarkeit. Nichtfunktionale Qualitätsmerkmale beziehen sich auf die Implementierung des Systems. Beispiele für nichtfunktionale Qualitätsmerkmale sind die Zuverlässigkeit und die Speicher- und Laufzeiteffizienz. Die oben beschriebenen Wechselwirkungen können dabei auch zwischen funktionalen und nichtfunktionalen Qualitätsmerkmalen bestehen: Zur Umsetzung eines sicheren Systems bedarf es oftmals zusätzlicher Hardware oder Software, was auf Kosten der Speichereffizienz gehen kann. Auf der anderen Seite kann eine schlechte Laufzeiteffizienz dazu führen, dass ein System zwar die korrekte Funktion erbringt, allerdings dieses viel zu langsam. Dies kann im Fall von echtzeitfähigen Steuerungen, z. B. im Fahrzeug oder Flugzeug, die Gefahrlosigkeit des Systems mindern.

Das für die Verifikation wichtigste Qualitätsmerkmal ist die *Korrektheit*. Korrektheit ist in diesem Kontext ein Synonym für die Fehlerfreiheit als Übereinstimmung zwischen dem beobachteten und dem gewünschten Verhalten. Wie oben dargestellt, muss hierzu eine Spezifikation korrekt in eine Implementierung umgesetzt sein. Dies gilt sowohl für funktionale als auch nichtfunktionale Anforderungen an Qualitätsmerkmale. Somit ist Korrektheit ein aus mehreren Qualitätsmerkmalen zusammengesetztes Qualitätsmerkmal.

Verifikation ist eine komplexe Aufgabe, da zum einen die Anwendungen und zum anderen die Implementierungen heutiger eingebetteter Systeme immer umfang-

reicher und heterogener werden. Die Schwierigkeit besteht in der Verzahnung funktionaler und nichtfunktionaler Eigenschaften, in der sich auch die Durchmischung der Komplexitäten von Anwendung und Implementierung widerspiegelt. So ist es nicht verwunderlich, dass ein großer Teil des Aufwands einer Produktentwicklung in die Verifikation fließt. Eine detaillierte Fallstudie [156] zeigt, dass der Verifikationsaufwand den Entwurfsaufwand sogar übersteigt. Dennoch lässt es die Komplexität heutiger Systeme nicht zu, dass diese nach dem heutigen Stand der Technik nach wirtschaftlichen Aspekten fehlerfrei entwickelt werden können.

Dies wird im *Entwurfsdreieck* in Abb. 1.5 dargestellt. Mit jeder Ecke des Dreiecks ist ein Qualitätsmaß assoziiert, namentlich Qualität, Kosten und Zeit, wobei Qualität für die Produktqualität, Zeit für die Zeit bis zur Markteinführung und Kosten für die Entwicklungskosten stehen sollen. Bei einer Produktplanung kann man die Vorstellungen für dieses Produkt in das Entwurfsdreieck eintragen, wobei ein Qualitätsmaß als um so besser gilt, je näher der Punkt an der entsprechenden Ecke steht. Das Entwurfsdreieck zeigt graphisch die Abhängigkeiten der drei Qualitätsmaße: Es ist nicht möglich, ein Produkt mit höchster Produktqualität zu geringsten Entwicklungskosten in kürzester Zeit auf den Markt zu bringen. Es ist aber z. B. möglich, ein Produkt möglichst kostengünstig zu entwickeln. Dies geht dann aber zu Lasten der Qualität und der Zeit bis zur Markteinführung.

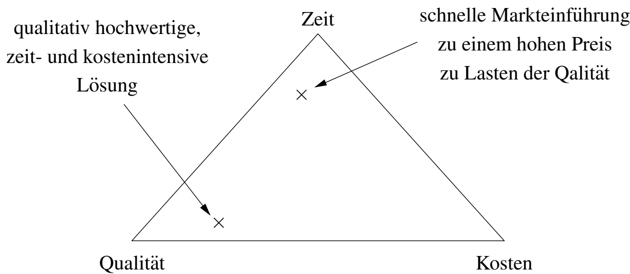


Abb. 1.5. Entwurfsdreieck

Verifikation ist ein qualitätssteigernder Prozess in der Entwicklung eingebetteter Systeme. Mit dem Wissen über das Entwurfsdreieck muss man allerdings immer wieder kritisch hinterfragen, ob die gewünschte Qualität erreicht ist, bzw. wie viel Verifikation noch nötig und wirtschaftlich vertretbar ist. Dabei darf man allerdings nie aus den Augen verlieren, dass ein Fehlverhalten oder Ausfall im Betrieb enorme Kosten nach sich ziehen und sogar einen Reputationsverlust für das Unternehmens bedeuten kann.

1.2 Der Verifikationsprozess

Verifikation ist der Prozess, die Fehlerfreiheit einer Implementierung bezüglich der Spezifikation zu zeigen. Mit anderen Worten: Es soll überprüft werden, ob ein Entwurfs- oder Syntheseschritt das korrekte Ergebnis liefert. Dies wird durch das sog. *Rekonvergenzmodell* (engl. *reconvergence model*) [41] in Abb. 1.6 ausgedrückt. Das Rekonvergenzmodell ist eine konzeptionelle Darstellung des Verifikationsprozesses. Hierbei wird eine Spezifikation im Entwurf in eine Implementierung transformiert. Die Korrektheit des Ergebnisses wird in der Verifikation überprüft.

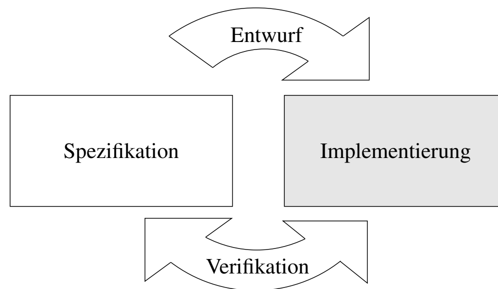


Abb. 1.6. Das Rekonvergenzmodell [41]

Der in Abb. 1.6 gezeigte Ablauf ist allerdings stark vereinfacht dargestellt. Ein wesentlicher Aspekt ist, dass die Spezifikation meistens in einer Form vorliegt, die nicht direkt in eine Implementierung transformierbar ist. Dies bedeutet, dass ein Entwickler zunächst die Spezifikation aufbereiten muss, bevor Entwurfsentscheidungen getroffen werden können. Genau diese Aufbereitung verlangt aber, dass die Spezifikation interpretiert wird. Dies liegt unter anderem daran, dass Spezifikationen oftmals unvollständig, mehrdeutig oder sogar widersprüchlich sind. Die Erweiterung des Rekonvergenzmodells um diesen Aspekt ist in Abb. 1.7 dargestellt.

Einen Schwachpunkt kann man allerdings sofort in Abb. 1.7 erkennen: Ist es bei der Interpretation der Spezifikation notwendig, Unvollständigkeiten, Mehrdeutigkeiten oder Widersprüche aufzulösen, und werden diese Modifikationen nicht in der Spezifikation festgehalten, so wird die Implementierung später nicht gegen die Spezifikation, sondern gegen die Interpretation der Spezifikation verifiziert. Die Interpretation existiert dabei nur als Bild der Spezifikation im Kopf des Entwicklers. Ist die Interpretation selbst fehlerhaft, kann dies nicht durch Verifikation erkannt werden.

Um diesen Sachverhalt aufzulösen, sollte die Implementierung stets gegen die Spezifikation geprüft werden. Hier zeigt sich aber ein zweites Problem: Wie beim Entwurf ist die Spezifikation oftmals nicht für einen direkten Einsatz im Verifikationsprozess einsetzbar. Dies bedeutet, dass ebenfalls eine Interpretation der Spezifikation für die Verifikation notwendig wird. Um dem Fall zu begegnen, dass fehlerhafte

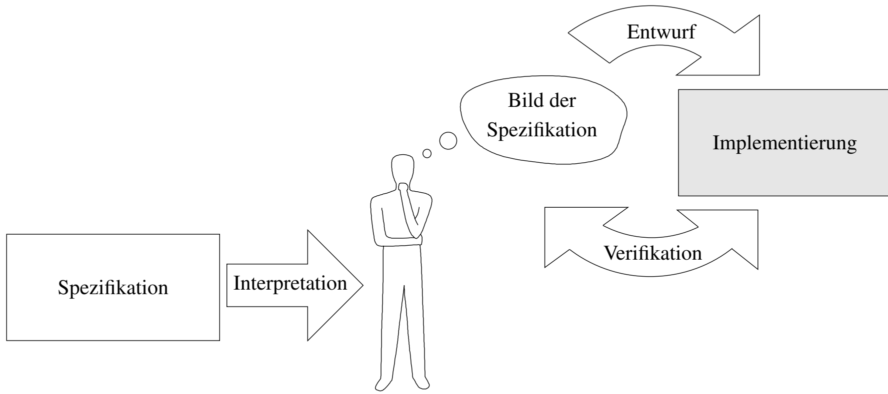


Abb. 1.7. Interpretation und Rekonvergenzmodell

Interpretationen die Qualität eines Entwurfs mindern, sollte deshalb nach Möglichkeit stets darauf geachtet werden, dass ein nicht am Entwurf beteiligter Entwickler die Verifikation übernimmt. Somit ist wahrscheinlich, dass immer zwei Bilder der Spezifikation durch Interpretation entstehen. Damit wird auch die Wahrscheinlichkeit gemindert, dass beide Bilder die selben Fehler enthalten. Dies ist in Abb. 1.8 dargestellt.

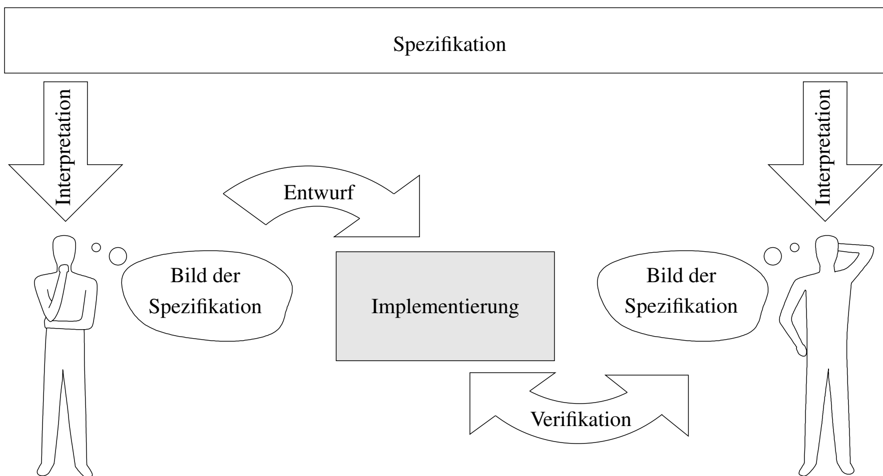


Abb. 1.8. Getrennte Interpretation zum Entwurf und zur Verifikation

1.2.1 Das V-Modell

Das Rekonvergenzmodell (siehe Abb. 1.6) stellt sowohl den Entwurf als auch die Verifikation jeweils als einen einzelnen Schritt dar. Dies ist für heutige Systeme nicht realistisch. Die Komplexität heutiger digitaler Hardware/Software-Systeme verlangt, dass man einen hierarchischen Entwurfsprozess einsetzt. Dies bedeutet, dass man, etwa nach einem Divide&Conquer-Ansatz, die Systemkomplexität zerlegt und den Entwurf somit auf verschiedenen Abstraktionsebenen durchführt (siehe auch [426]). Diesem Aspekt wird durch das *V-Modell* Rechnung getragen (siehe Abb. 1.9).

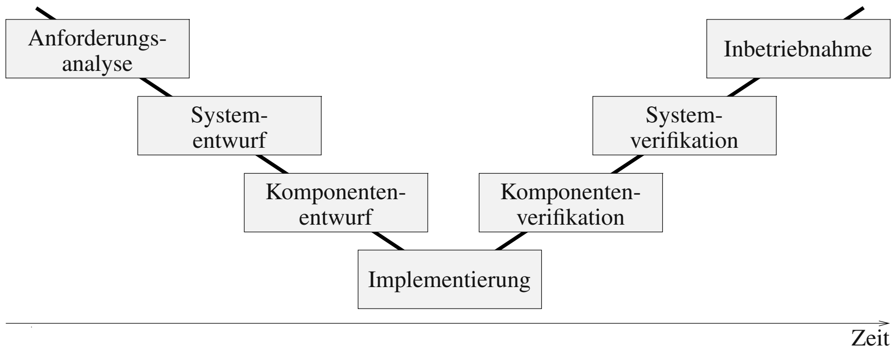


Abb. 1.9. Das V-Modell

Das V-Modell trägt seinen Namen aufgrund seiner Darstellung als „V"-förmiges Diagramm. Die linke Seite des V entspricht dem Entwurfsprozess. Entsprechend beschreibt die rechte Seite den Verifikationsprozess. Der Berührungspunkt der beiden Prozesse ist die *Implementierung*. Die Entwurfsphase startet oben links mit der Analyse der Anforderungen. Diese werden in dem Systementwurf umgesetzt. Dabei erfolgt auch die Aufteilung in Hardware- und Software-Komponenten. Diese werden im Komponentenentwurf entwickelt. Damit endet der Entwurf und es folgt die Implementierung. Die Verifikationsphase beginnt unten auf der anderen Seite des V mit der Verifikation der einzelnen Komponenten. Dieser Schritt ist gefolgt von der Systemverifikation und der anschließenden Inbetriebnahme.

In einem fehlerfreien Entwurf schreitet die Zeit von links nach recht fort und es wird zunächst die eine Seite des V herabgestiegen und anschließend die andere Seite wieder heraufgestiegen. Das Ab- und Aufsteigen entspricht dabei den Wechsel der Abstraktionsebenen. In der Entwurfsphase wird die Implementierung immer detaillierter. In der Verifikation abstrahiert man von bereits verifizierten Komponenten.

In einem realen Systementwurf wird man aber die Abstraktionsebenen hin- und herspringen, da z. B. bereits Teile der Implementierung auf niedrigen Abstraktionsebenen vorliegen. Daneben wird auch zwischen den Seiten des V gesprungen. Wird beispielsweise in der Komponentenverifikation ein Fehler aufgedeckt, wird

man nicht zur Systemverifikation übergehen, sondern zunächst die fehlerhafte Komponente korrigieren.

Das V-Modell gibt es in verschiedensten Versionen mit unterschiedlich vielen Abstraktionsebenen. Dementsprechend gibt es auch spezialisierte Varianten für den Hardware-Entwurf oder für den Software-Entwurf. Dies stellt zugleich einen großen Nachteil des V-Modells heraus: Es gibt keine Varianten, die gleichzeitig zwischen dem Hardware- und dem Software-Entwurf und deren Verifikationsphasen und Interaktionen unterscheiden können. Aus diesem Grund werden im folgenden Abschnitt Modelle für den Entwurfs- und den Verifikationsprozess von digitalen Hardware/Software-Systemen vorgestellt.

1.2.2 Das Doppeldachmodell des Entwurfsprozesses

Die Verifikation eines Systems beschreibt die Überprüfung der korrekten Implementierung einer Spezifikation. Um den Verifikationsprozess zu verstehen, ist es notwendig, zunächst den Entwurfsprozess detaillierter zu betrachten. Da eingebettete Systeme häufig aus interagierenden Hardware- und Software-Komponenten bestehen, muss der Entwurf beider Bestandteile sowie die Interaktion zwischen diesen im Entwurfsprozess berücksichtigt werden. Der idealisierte Entwurf für digitale Hardware/Software-Systeme ist in Abb. 1.10 als *Doppeldachmodell* dargestellt [425, 426].

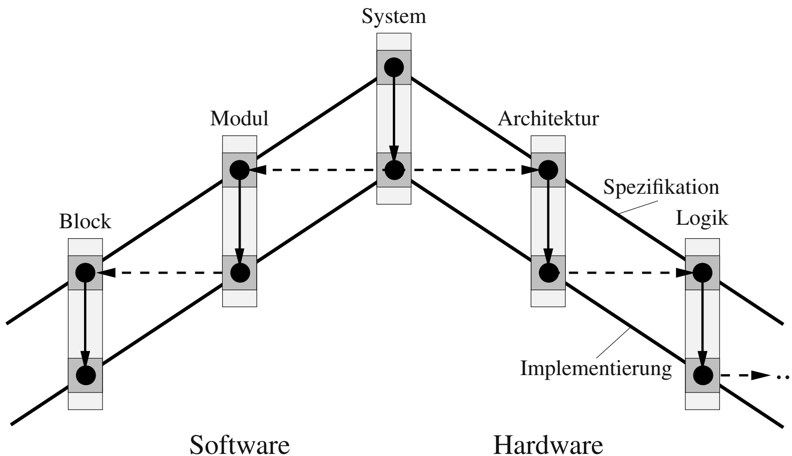


Abb. 1.10. Doppeldachmodell für den Entwurf von digitalen Hardware/Software-Systemen

Die linke Seite des Daches entspricht dem *Software-Entwurfsprozess*, während die rechte Seite dem *Hardware-Entwurfsprozess* zugeordnet ist. Jede Seite ist dabei in unterschiedliche *Abstraktionsebenen* organisiert. Für den Software-Entwurfsprozess sind häufig die *Block*- und *Modulebene* von Interesse. Im Hardware-Entwurfsprozess findet man im Allgemeinen die *Logik*- und *Architekturebene*. In der Mitte des

Doppeldachmodells findet man eine gemeinsame Abstraktionsebene, die sog. *Systemebene* (engl. *Electronic System Level, ESL*). Auf dieser Abstraktionsebene wird nicht zwischen Hardware und Software unterschieden.

Auf jeder Abstraktionsebene wird durch einen *Syntheseschritt* eine *Spezifikation* in eine *Implementierung* transformiert. Der Syntheseschritt wird auf jeder Abstraktionsebene als vertikaler Pfeil im Doppeldachmodell dargestellt. Im Rekonvergenzmodell aus Abb. 1.6 waren alle diese Schritte zu einem Schritt (Entwurf) zusammen gefasst. Horizontale Pfeile beschreiben die Wiederverwendung individueller Elemente einer Implementierung als Spezifikation auf der nächst tieferen Abstraktionsebene.

Der durch das Doppeldachmodell beschriebene Entwurfsprozess beginnt typischerweise auf der Systemebene. Hier besteht die Spezifikation beispielsweise aus einem Prozessgraphen, der das gewünschte Verhalten des Systems als über Kanäle kommunizierende Prozesse beschreibt. Daneben gibt es meistens eine Menge an Abbildungs- und Implementierungsbeschränkungen in der Form von maximalen Flächenanforderungen, minimale zu erreichendem Durchsatz etc. Die Implementierung auf der Systemebene wird auch als Hardware/Software-Architektur bezeichnet und wird in Form von strukturellen Modellen bestehend aus Prozessoren, Bussen, Speichern, Hardware-Beschleunigern und Klassendiagrammen beschrieben. Die Aufgabe der *Systemsynthese* besteht darin, eine geeignete Implementierungsplattform auszuwählen, Prozesse und Kanäle auf diese Plattform abzubilden und eine zeitliche Ordnung der Abarbeitung der Prozesse und der Kommunikation zu bestimmen. Hierdurch entsteht ein verfeinertes Modell, welches alle Entwurfsentscheidungen, die auf dieser Ebene getroffen wurden, beinhaltet. Die einzelnen Komponenten dieses verfeinerten Modells dienen anschließend als Eingabe für den Entwurfsprozess auf der nächst tieferen Abstraktionsebene.

Die Synthese auf tieferen Abstraktionsebenen ähnelt der Systemsynthese dahingehend, dass stets ein Verhaltensmodell in eine strukturelle Implementierung transformiert wird [426]. Dabei unterscheidet sich allerdings die Granularität der betrachteten Objekte. Auf der *Modulebene* werden typischerweise Prozesse, die auf den selben Prozessor abgebildet sind, zeitlich geplant. Dies geschieht häufig unter Zuhilfenahme eines Betriebssystems. Auf der *Blockebene* müssen Prozesse auf den Instruktionssatz des Prozessors (engl. *Instruction Set Architecture, ISA*) abgebildet werden. Hierfür werden in der Regel Compiler und Linker eingesetzt.

Auf *Architekturebene* im Hardware-Entwurf werden Prozesse, welche als Hardware-Beschleuniger implementiert werden, in Beschreibungen auf der *Registertransferebene* (engl. *Register Transfer Level, RTL*) synthetisiert. RTL-Beschreibungen bestehen aus einem Controller zur Steuerung eines Datenpfads, der wiederum aus Funktionseinheiten, Registern und Verbindungen besteht. Dieser Syntheseschritt wird als *Verhaltenssynthese* (engl. *behavioral synthesis* bzw. *high-level synthesis*) oder Architektursynthese bezeichnet. Auf *Logikebene* werden *Boolesche Funktionen* oder *Automatenmodelle* durch Gatter und Flip-Flops implementiert.

Es sei angemerkt, dass der im Doppeldachmodell angedeutete Top-Down-Entwurfsprozess auf Systemebene auf der Verfügbarkeit von Entwurfsprozessen auf Modul- und Architekturebene basiert. Dies heißt, dass tiefere Abstraktionsebenen

einen Zugang zum Entwurfsprozess, entweder durch ein Syntheseverfahren oder durch vordefinierte Module (engl. *IP modules*), bieten muss.

Synthese

Zentral für den Entwurfsprozess ist die Synthese. Die Synthese transformiert eine Spezifikation in eine Implementierung. Dies geschieht auf verschiedenen Abstraktionsebenen, was der heutigen Systemkomplexität Rechnung trägt. Dabei wird auf jeder Abstraktionsebene eine Verhaltensbeschreibung, die Teil der Spezifikation ist, in eine Strukturbeschreibung transformiert (siehe Abb. 1.11).

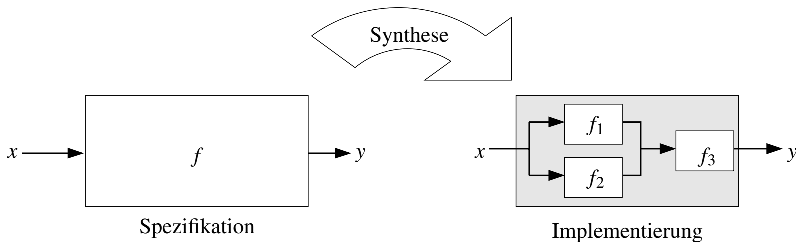


Abb. 1.11. Die Synthese transformiert eine Verhaltensbeschreibung, die Teil der Spezifikation ist, in eine Strukturbeschreibung

In Abb. 1.11 ist das Verhalten der Spezifikation durch die Funktion f beschrieben. Die Funktion $y := f(x)$ transformiert den Eingang x in den Ausgang y. Um hieraus eine Strukturbeschreibung zu erzeugen, muss zunächst die Funktion f in Teilfunktionen, hier f_1, f_2 und f_3, zerlegt werden. Damit die resultierende Implementierung aber das in der Spezifikation vorgegebene Verhalten implementiert, muss zusätzlich angegeben werden, wie die Teilfunktionen zusammenarbeiten, d. h. $y = f_3(f_1(x), f_2(x))$. Somit besteht eine Strukturbeschreibung aus Teilfunktionen und deren Zusammenwirken. Da die Teilfunktionen f_1, f_2, f_3 selbst wieder Verhaltensbeschreibungen sind, können diese gegebenenfalls in weitere Strukturmodelle verfeinert werden.

Neben der Transformation der Verhaltensbeschreibung in eine Strukturbeschreibung spielt bei der Synthese aber auch die Einhaltung der Anforderungen an das System eine zentrale Rolle. Die abstrakte Sicht auf den Syntheseschritt kann mit Hilfe des *X-Diagramms* verfeinert werden (Abb. 1.12). Eine *Spezifikation* besteht aus einem *Verhaltensmodell* und *Anforderungen*. Das Verhaltensmodell beschreibt die geforderte Funktionalität des Systems. Seine Expressivität und Analysierbarkeit wird als *Berechnungsmodell* (engl. *Model of Computation, MoC*) angegeben [298]. Verhaltensmodelle werden häufig in Programmiersprachen, wie C/C++ oder Java, Hardware-Beschreibungssprachen, wie SystemVerilog oder VHDL, oder Systembeschreibungssprachen, wie beispielsweise SystemC oder SpecC, geschrieben.

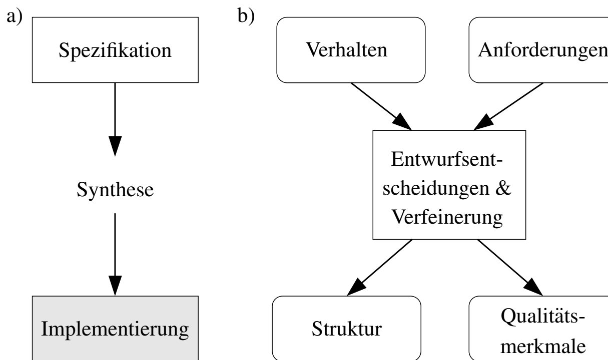


Abb. 1.12. X-Diagramm der Synthese

Die *Anforderungen* (engl. *requirements*) umfassen oftmals implizit oder explizit ein *Plattformmodell*, welches die Vielfalt möglicher struktureller Implementierungen einschränkt, etwa durch Vorgabe verfügbarer Ressourcen, deren Dienste und deren Verbindungen. Beispiele hierfür sind verfügbare Prozessoren mit deren Instruktionssatz, Speicher und Prozessorbusse. Auf Systemebene sind typische Plattformmodelle beispielsweise Einprozessorsysteme, Hardware/Software-Prozessor/Coprozessor-Systeme, homogene, symmetrische oder heterogene, asymmetrische Multiprozessorsysteme [466]. Neben dem Plattformmodell existieren in den Anforderungen darüber hinaus oftmals Abbildungsbeschränkungen für Teile des Verhaltensmodells sowie zusätzliche Anforderungen an nichtfunktionale Eigenschaften, wie maximale Antwortzeit, minimaler Durchsatz, maximale Leistungsaufnahme etc.

Eine *Implementierung* besteht aus einem *Strukturmodell* sowie einer Menge von *Qualitätsmerkmalen*. Das Strukturmodell ist ein verfeinertes Modell des Verhaltensmodells unter Berücksichtigung der Anforderungen aus der Spezifikation. Zusätzlich zu implementierungsunabhängigen Informationen im Verhaltensmodell enthält das Strukturmodell Informationen über die Umsetzung von Entwurfsentscheidungen aus dem vorhergegangenen Syntheseschritt. Ein Beispiel hierfür ist die Abbildung des Verhaltensmodells auf Elemente des Plattformmodells.

Qualitätsmerkmale sind geschätzte Eigenschaften einer Implementierung und werden in einem geeigneten *Qualitätsmaß* quantifiziert, z. B. Antwortzeit, Durchsatz, Latenz, Flächen- oder Leistungsbedarf. Qualitätsmerkmale quantifizieren funktionale oder nichtfunktionale Eigenschaften eines Systems. Zur Abschätzung dienen entweder das Strukturmodell oder geeignete *Bewertungsmodelle*, die, je nach Abstraktionsebene und Umsetzung, unterschiedlich gute Abschätzungen liefern. Beispiele für Bewertungsmodelle für zeitliche Qualitätsmaße sind Instruktionssatz- oder Hardware-Simulationsmodelle.

Die *Synthese* transformiert eine Spezifikation in eine Implementierung. Die zentralen Aufgaben dabei sind das Treffen von *Entwurfsentscheidungen* und die *Verfeinerung* eines Verhaltensmodells in ein Strukturmodell (Abb. 1.12). Auf jeder Ab-

straktionsebene führt die Synthese dabei eine Abbildung der Elemente im Verhaltensmodell in Raum und Zeit durch. Das Treffen von Entwurfsentscheidungen ist somit die *Allokation* von Ressourcen aus dem Plattformmodell, die (räumliche) *Bindung* von Elemente des Verhaltensmodells und deren (zeitliche) *Ablaufplanung*, um Ressourcenkonflikte aufzulösen. *Verfeinerung* ist der Prozess, die Entwurfsentscheidungen in das Verhaltensmodell zu integrieren. Das Ergebnis ist ein Strukturmodell der Implementierung. Daneben können mit den Entwurfsentscheidungen die *Qualitätsmerkmale* abgeschätzt werden.

Da die Menge möglicher Entwurfsentscheidungen typischerweise sehr groß sein kann, wird häufig als Teil der Synthese eine *Entwurfsraumexploration* zur Bestimmung optimaler (oder nahezu optimaler) Implementierungen durchgeführt. Die Entwurfsraumexploration ist ein *Mehrzieloptimierungsproblem*, welches in der Regel nicht nur eine einzelne, sondern eine Menge sog. *Pareto-optimaler* Lösungen bestimmt. Somit kann die Entwurfsraumexploration als Mehrzieloptimierungsproblem der Synthese bezügliche der zu optimierenden Qualitätsmaße aufgefasst werden. Die Aufgaben und Methoden zum Entwurf und zur Optimierung digitaler Hardware/Software-Systeme sind ausführlich in [426] dargestellt.

1.2.3 Das Doppeldachmodell des Verifikationsprozesses

Wie im Entwurfsprozess digitaler Hardware/Software-Systeme, muss auch im Verifikationsprozess die Verifikation von Hardware, von Software und von Systemen unterstützt werden. Außerdem muss auf jeder Abstraktionsebene verifiziert werden, d. h. es muss überprüft werden, ob die jeweilige Spezifikation korrekt in eine Implementierung umgesetzt wurde. Das *Doppeldachmodell für den Verifikationsprozess* in Abb. 1.13 enthält sowohl die Hardware-, die Software- als auch die Systemverifikation. Die vertikalen Pfeile stellen Verifikationsschritte dar, während die horizontalen Pfeile die Komposition einzelner Komponenten beschreiben. Man beachte, dass die Pfeile im Doppeldachmodell für den Verifikationsprozess gegenüber dem Doppeldachmodell für den Entwurfsprozess entgegengesetzt verlaufen.

Auf jeder Abstraktionsebene haben sich eigenständige Ansätze zur automatischen Verifikation etabliert, wobei diese oftmals Gemeinsamkeiten zeigen. Wichtige Ansätze zur automatischen Verifikation von Hardware, Software und Systemen werden in diesem Buch in einer einheitlichen Notation dargestellt. Es wird gezeigt, dass sich viele der vorgestellten Ansätze gleichermaßen auf Hardware, Software oder teilweise auch auf Hardware/Software-Systeme anwenden lassen. Da die beschriebenen Ansätze aber in einem bestimmten Kontext entwickelt wurden, werden sie in dem vorliegenden Buch auch in diesem Zusammenhang vorgestellt.

Auch wenn die Verifikationsansätze viele Gemeinsamkeiten aufweisen, unterscheiden sich diese auf den einzelnen Abstraktionsebenen erheblich. Auf der Logikebene gilt es beispielsweise zu zeigen, dass eine Funktion korrekt in ein Schaltnetz bzw. ein Schaltwerk umgesetzt wurde. Die Spezifikation (die geforderte Funktion) ist dabei ein System Boolescher Funktionen, die Implementierung eine Netzliste aus Gattern. Die Herausforderung besteht darin, zu zeigen, dass die Wahl einer bestimmten Technologie (Gattertypen) und somit des Basissystems und die daraus resultie-

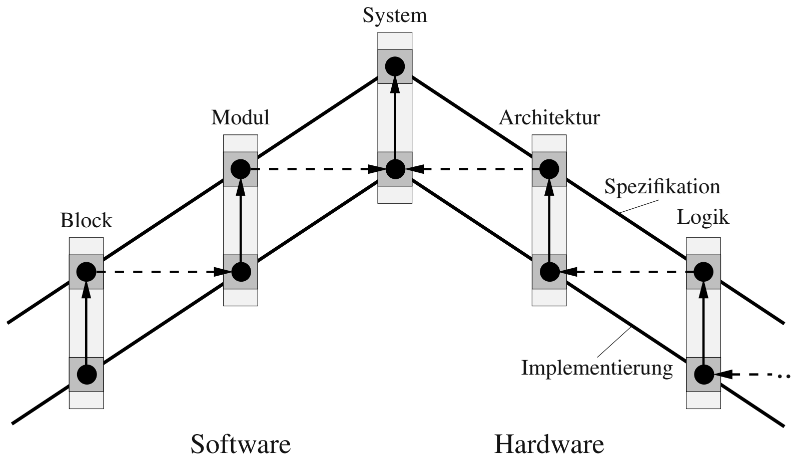


Abb. 1.13. Doppeldachmodell für den Verifikationsprozess digitaler Hardware/Software-Systeme

renden Transformationen, die gewünschte Funktion, beispielsweise eines Multiplexers oder Volladdierers, nicht verfälscht hat. Auf höherer Abstraktionsebene (der Architekturebene) kann diese Frage auch so formuliert werden: Berechnet ein Addierer/Multiplizierer tatsächlich die Summe/das Produkt der beiden Operanden?

Einen Spezialfall auf dieser Ebene stellt die Prozessorverifikation dar. Als Spezifikation dient dabei die Instruktionssatzarchitektur des Prozessors, der als eine sequentielle Implementierung des Prozessors verstanden werden kann, d. h. die Instruktionssatzarchitektur beschreibt, wie die Ausführung einer Instruktion den Zustand der für den Programmierer sichtbaren Register ändert. Die Mikroarchitektur des Prozessors hingegen ist eine Hardware-Implementierung auf Registertransferebene und typischerweise hoch optimiert. So erfolgt die Abarbeitung von Instruktionen häufig verschränkt, d. h. die Mikroarchitektur implementiert eine sog. *Pipeline* mit unterschiedlichen Stufen der Abarbeitung von Instruktionen. Hierdurch können mehrere Instruktionen gleichzeitig in Bearbeitung sein. Andere Optimierungen, die darauf zielen, den Durchsatz des Prozessors zu erhöhen, sind Sprungvorhersage mit spekulativer Ausführung oder parallel arbeitende Funktionseinheiten, die eine dynamische Ablaufplanung der Instruktionen in Abhängigkeit der Verfügbarkeit von Ressourcen ermöglicht. Zu zeigen, dass eine so optimierte Mikroarchitektur den Instruktionssatz korrekt implementiert, ist eine zentrale Aufgabe auf der Architekturebene.

Auf höheren Abstraktionsebenen werden in der Hardware-Verifikation zunehmend nicht nur die Übereinstimmung des spezifizierten Verhaltens mit dem implementierten Verhalten verglichen, sondern auch zusätzlich Eigenschaften der Implementierung überprüft. Hierzu gehören beispielsweise Eigenschaften, die garantieren, dass die Hardware in keinen ungewünschten oder gefährlichen Zustand übergeht bzw. garantiert in einer vorgegebenen Zeitspanne eine gewünschte Reaktion zeigt.

Dabei ist das Zeitverhalten der Hardware stark durch die gewählte Technologie beeinflusst: Die Gatterlaufzeiten bestimmen den kritischen Pfad in einem System und somit die Taktrate bzw. die Tiefe der Pipeline. Diese Effekte sind nicht auf die Logikebene beschränkt, sondern sind auch auf Architekturebene und sogar Systemebene zu berücksichtigen.

Im Bereich der Software-Verifikation trifft man auf ähnliche Aufgaben wie in der Hardware-Verifikation. Auf Blockebene liegt die Implementierung in Form eines Assemblerprogramms für den gewählten Prozessor vor. Die Spezifikation ist in einer Hochsprache verfasst, häufig in eingebetteten Systemen C/C++. Auch auf dieser Ebene kann es wichtig sein, die Äquivalenz von Programmen zu zeigen, da gerade eingebettete Software stark optimiert wird, um den Speicher- und Geschwindigkeitsanforderungen zu genügen. Da eingebettete Systeme und deren Software auch häufig in sicherheitskritischen Bereichen eingesetzt werden, ergibt sich aber auch als weitere Verifikationsaufgabe zu zeigen, ob ein Programm gewisse funktionale Eigenschaften besitzt. So sollte eine Steuerungs-Software nicht verklemmen und korrekt auf Anfragen reagieren.

Daneben hat die Mikroarchitektur des Prozessors einen großen Einfluss auf die Abarbeitungsgeschwindigkeit von Instruktionen, d. h. unterschiedliche Mikroarchitekturen können die selbe Instruktionssatzarchitektur implementieren, aber ein unterschiedliches zeitliches Verhalten besitzen. Dies hat Einfluss auf die Ausführungszeiten nicht nur einzelner Instruktionen, sondern auf ganze Programme. Die Überprüfung der Einhaltung der Zeitanforderung ist somit eine Verifikationsaufgabe sowohl auf Block- als auch auf Modulebene.

Auf Systemebene ist die Überprüfung von Eigenschaften der Implementierung noch vordergründiger. Aufgrund der Komplexität von Spezifikation und Implementierung ist eine vollständige Prüfung nicht mehr praktikabel. Das Verhalten auf Systemebene wird oftmals durch ein Modell kommunizierender Prozesse beschrieben. Das in der Implementierung verwendete Strukturmodell ist häufig eine Netzliste aus Prozessoren, Speichern, Bussen und Hardware-Beschleunigern. Bei der Verifikation wird dabei eine Fokussierung auf die Interaktion der Komponenten untereinander durch *Transaktionen* vorgenommen. Neben der Überprüfung des Verhaltens ist aber auch das Einhalten von Zeiteigenschaften von großer Bedeutung.

Das X-Diagramm der Verifikation

Analog zu dem X-Diagramm für den Entwurf (Abb. 1.12) wird nun ein X-Diagramm für die Verifikation entwickelt. Dieses ist in Abb. 1.14 zu sehen. Es dient zur Formulierung der grundlegenden Verifikationsaufgaben bei der Entwicklung eingebetteter Systeme.

In Abb. 1.14a) sieht man graphisch dargestellt, dass die Verifikation die Implementierung gegen ihre Spezifikation prüft. Die gleiche Verfeinerung wie im Entwurf in Abb. 1.12b) ist für die Verifikation vorgenommen worden (Abb. 1.14b)). Eine Spezifikation besteht aus einem Verhaltensmodell und Anforderungen. Eine Implementierung besteht aus einer Strukturbeschreibung und den Qualitätsmerkmalen, die für die Implementierung ermittelbar sind. Die Verifikation besteht aus zwei Schritten:

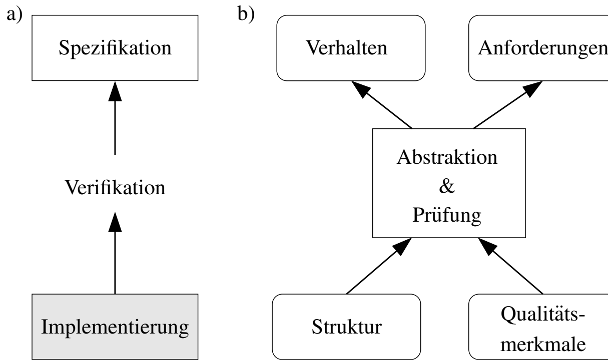


Abb. 1.14. X-Diagramm der Verifikation

Abstraktion und *Prüfung*. Die Abstraktion kann notwendig sein, da im Entwurfsprozess gegenüber dem Verhaltensmodell Verfeinerungsinformationen im Strukturmodell aufgenommen worden sind. Die *Prüfung* hängt von der *Verifikationsaufgabe* ab. Aus dem X-Diagramm lassen sich im Wesentlichen drei grundlegende Verifikationsaufgaben ableiten, die im Folgenden unterschieden werden. Diese sind unabhängig von der Abstraktionsebene definiert:

1. *Äquivalenzprüfung:* Bei der Äquivalenzprüfung wird das Strukturmodell mit dem Verhaltensmodell auf Äquivalenz überprüft, d. h. es wird versucht, die Frage zu beantworten, ob unter allen Umständen das Strukturmodell das selbe Verhalten wie das Verhaltensmodell der Spezifikation aufweist (siehe Abb. 1.15a)).
2. *Prüfung funktionaler Eigenschaften:* In der funktionalen Eigenschaftsprüfung wird das Strukturmodell der Implementierung dahingehend überprüft, ob es die funktionalen Eigenschaften in den Anforderungen erfüllt (siehe Abb. 1.15b)). Dies können z. B. Gefahrlosigkeitseigenschaften oder Lebendigkeitseigenschaften sein.
3. *Prüfung nichtfunktionaler Eigenschaften:* In der nichtfunktionalen Eigenschaftsprüfung werden die Anforderungen an die Qualitätsmerkmale der Implementierung überprüft. Es wird also geprüft, ob die Merkmale den Anforderungen der Spezifikation genügen (siehe Abb. 1.15c)). Die Schwierigkeit besteht darin, dass die Qualitätsmerkmale auf niedrigen Abstraktionsebenen ermittelt werden und durch geeignete Kompositionen von Abschätzungen auf höheren Ebenen durchgeführt werden müssen, wo die Anforderungen spezifiziert sind.

Funktionale und nichtfunktionale Eigenschaften sind typischerweise nicht unabhängig voneinander. So kann beispielsweise die Verletzung einer Zeitanforderung eine Gefahrensituation auslösen, oder anders herum.

Die grundlegenden Verifikationsaufgaben sind noch einmal in Abb. 1.15 graphisch dargestellt. Jede der drei Aufgaben kann auf jeder Abstraktionsebene zum Einsatz kommen. Allerdings bekommen die Eigenschaftsprüfungen auf hohen Ab-

straktionsebenen eine größere Bedeutung, während die Äquivalenzprüfung im Wesentlichen auf tieferen Abstraktionsebenen eingesetzt wird.

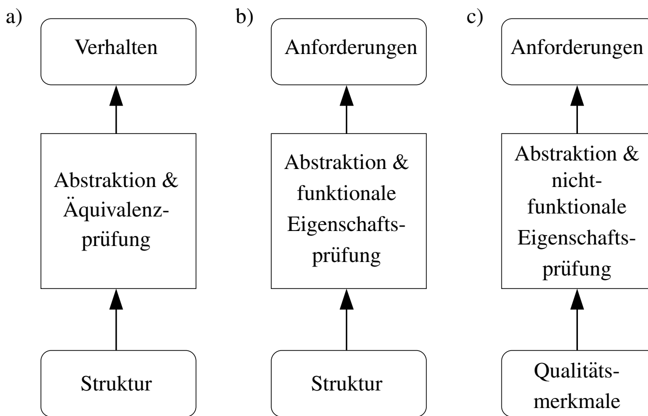


Abb. 1.15. Drei Arten der Prüfung in der Verifikation

1.3 Eine kurze Geschichte der Verifikation

Zur Verifikation wird neben der Spezifikation und der Implementierung auch eine leistungsfähige *Verifikationsmethode* benötigt. Grob lassen sich Verifikationsmethoden in formale und simulative Methoden klassifizieren. Die Entwicklung von Verifikationsmethoden hat eine lange Geschichte. Insbesondere formale Methoden basieren auf mathematischen Beweisverfahren, die teilweise Jahrhunderte vor der Erfindung des Computer entwickelt wurden. Im Gegensatz dazu wurden viele simulative Verfahren erst mit dem Vorhandensein von leistungsfähigen Computern ermöglicht. In diesem Abschnitt wird eine kurze geschichtliche Übersicht zu der Entwicklung von Verifikationsmethoden gegeben. Eine ausführliche Darstellung findet man in [390]. Ein Überblick über die Geschichte der Logik kann [413] entnommen werden.

Der Begriff Verifikation leitet sich vom lateinischen Wort *veritas*, die Wahrheit, ab. Verifikation ist der Nachweis bzw. der Prozess des Nachweisens, dass ein vermuteter oder behaupteter Sachverhalt wahr ist. Zentral für die Verifikation ist der Begriff der *Logik*. Bereits Aristoteles (384 v. Chr. – 322 v. Chr.) entwickelte eine erste formale Logik mit dem Ziel, Gesetze des menschlichen Denkens zu finden. Dabei entwickelte er Schemata zur Repräsentation gültiger Schlüsse, sog. *Syllogismen*. Ein gültiger Schluss ist eine Konfiguration der Form [413]:

$$\frac{\textit{Prämissen}}{\textit{Folgerung}}$$

Hierbei wird ausgehend von einer endlichen Anzahl an *Prämissen* eine *Folgerung* geschlossen. Als Beispiel diene die Folgerung:

$$\frac{\text{Alle } P \text{ sind } Q, a \text{ ist } P}{a \text{ ist } Q}$$

Unabhängig von der konkreten Interpretation von P und Q ist die Folgerung richtig. Man kann z. B. für P das Prädikat *Mensch*, für Q das Prädikat *sterblich* und für a das Individuum *Sokrates* wählen. Hierdurch entsteht eine berühmte Instanz der obigen Folgerung:

$$\frac{\text{Alle } \textit{Menschen} \text{ sind } \textit{sterblich}, \textit{Sokrates} \text{ ist ein } \textit{Mensch}}{\textit{Sokrates} \text{ ist } \textit{sterblich}}$$

Die Richtigkeit dieser Folgerung ist bereits bewiesen.

2000 Jahre später, im Jahr 1686, entwickelte Gottfried Wilhelm Leibniz in seinem Werk *Generales Inquisitiones de Analysi Notionum et Veritatum* (Allgemeine Untersuchungen über die Zerlegung der Begriffe und Wahrheiten) eine erste moderne Logik, aufgebaut in einer mathematischen Sprache, der *Kalkülform*. Nahezu zwei Jahrhunderte später veröffentlicht George Boole 1847 in *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning* das erste algebraische Logikkalkül und begründet darin die heutige *Aussagenlogik*. In der Aussagenlogik werden ausgehend von atomaren Aussagen $A, B, C, \ldots$, welche die Werte wahr (T) oder nicht wahr (F) annehmen können, mit Hilfe der Junktoren $\neg$ (*nicht*), $\vee$ (*oder*), $\wedge$ (*und*) und $\Rightarrow$ (*impliziert*) neue Aussagen gebildet. Ein Kalkül der Aussagenlogik basiert auf einer endlichen Anzahl an *Axiomen*, mit deren Hilfe gültige Aussagen unabhängig vom Wahrheitsgehalt der atomaren Aussagen gefolgert werden können. Beispiele hierfür sind [413]:

$$\frac{A \vee \neg A}{\mathsf{T}} \qquad \frac{A, A \Rightarrow B}{B}$$

Die *Prädikatenlogik erster Ordnung* wird erstmals von Gottlob Frege in seinem Buch *Begriffsschrift – Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens* im Jahr 1879 beschrieben. Die Prädikatenlogik erster Ordnung erweitert die Aussagenlogik um *Funktionen*, *Prädikate* und *Quantoren*. Mit Hilfe mehrstelliger Prädikate oder Relationen werden atomare Aussagen gebildet. Diese können wie in der Aussagenlogik durch Junktoren zu neuen Aussagen zusammengesetzt werden. Zusätzlich kann durch die *Existenz-* ($\exists$) bzw. *Allquantifizierung* ($\forall$) geprüft werden, ob eine Aussage auf *mindestens eine* bzw. auf *alle* Variablen zutrifft. Neue Axiome der Prädikatenlogik, die über die Axiome der Aussagenlogik hinausgehen, sind beispielsweise [413]:

$$\frac{A(y) \Rightarrow \exists x : A(x,y)}{\mathsf{T}} \qquad \frac{A(x)}{\forall x : A(x)}$$

1929 beweist Kurt Gödel die Vollständigkeit der Prädikatenlogik erster Ordnung, d. h. alle logisch gültigen Aussagen sind herleitbar im System der Prädikatenlogik

erster Ordnung. Zwei Jahre später, 1931, veröffentlichte Kurt Gödel in [198] den *Gödelschen Unvollständigkeitssatz*, der Grenzen für formale Systeme aufzeigt. Er besagt beispielsweise, dass in der Arithmetik nicht alle Aussagen formal bewiesen oder widerlegt werden können. Die zentrale Aussage des Satzes lautet: „*Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig*". Obwohl die Aussage des Unvollständigkeitssatzes eine negative ist, ist sie eine der zentralen Erkenntnisse des vergangenen Jahrhunderts und hat nachhaltig die Fragestellungen der Forschung in eine neue und fruchtbare Richtung bewegt.

Im Jahr 1937 veröffentlichte Alan Turing eine grundlegende Arbeit [444], in der er die Ergebnisse Kurt Gödels neu formulierte. Dabei ersetzte er die formalen Systeme Gödels durch eine einfache Maschine, die nach ihm benannte *Turing-Maschine*. Eine Turing-Maschine ist eine hypothetische „Rechenmaschine". Sie besteht aus einem Band, welches unendlich lang ist. Dieses ist in einzelne Felder unterteilt. In einem Feld können Symbole aus einem endlichen Alphabet stehen. Ein Lese-/Schreibkopf steht zu jedem Zeitpunkt genau auf einem Feld. Neben dem Speicher auf dem Band besitzt eine Turing-Maschine noch interne Zustände. In einem Rechenschritt liest die Turing-Maschine das Symbol auf dem Band, ändert seinen internen Zustand, schreibt an die selbe Position auf dem Band ein neues (evtl. identisches) Symbol und bewegt den Lese-/Schreibkopf um eine Position nach links oder rechts, oder bleibt gleich. Die Eingaben zur Berechnung müssen vorher auf das Band geschrieben werden. Ist die Berechnung abgeschlossen, so befindet sich die Turing-Maschine in einem ausgezeichneten Haltezustand und das Ergebnis der Berechnung steht auf dem Band.

Turing konnte beweisen, dass eine Turing-Maschine in der Lage ist, „*jedes vorstellbare mathematische Problem zu lösen, sofern dieses auch durch einen Algorithmus gelöst werden kann*". Hiermit bewies Turing, dass das *Halteproblem* mit Hilfe der Turing-Maschine nicht entscheidbar ist. Die Unentscheidbarkeit des Halteproblems kann wie folgt formuliert werden: *Es gibt keinen Algorithmus A_1, der für einen beliebigen Algorithmus A_2 und beliebige Eingabe E entscheidet, ob A_2 mit der Eingabe E terminiert.*

Turings Werk ist zentral für die Theoretische Informatik und wird manchmal sogar als Geburtsstunde der gesamten Informatik bezeichnet. Dies liegt nicht zuletzt daran, dass Turings Ideen einen wesentlichen Einfluss auf den Bau digitaler Computer ausübte. Die Bedeutung von Turings (und somit Gödels) Arbeiten kann man fassen, wenn man sich vor Augen führt, dass sie die Grenzen der Verifikation aufzeigen. Mit anderen Worten: Es gibt Verifikationsprobleme, die sich nicht beantworten lassen, da das zugrundeliegende Problem nicht entscheidbar ist. Somit haben diese Arbeiten Auswirkungen, die in diesem Buch berücksichtigt werden müssen.

Neben der Aussagenlogik und Prädikatenlogik erster Ordnung gibt es viele weitere für die Verifikation wichtige Logiken. Ein Beispiel für eine ganze Klasse solcher weiteren Logiken sind die sog. *Modallogiken*. Modallogiken erweitern die bisher vorgestellten Logiken um die Konzepte „notwendig" und „möglich ". Die Ideen gehen wiederum auf Aristoteles zurück. Ende der 1950er Jahre veröffentlichte Saul Kripke aufsehenerregende Arbeiten zur Modallogik [268]. Die Bedeutung der Modallogik und der Arbeiten von Kripke wird klar, wenn man sich ihr Anwendungs-

feld, die Überprüfung funktionaler Eigenschaften von digitalen Systemen, vor Augen führt. Dies wird an dem folgenden Beispiel aus [413] verdeutlicht:

Gegeben sei das folgende Programm, welches die Sequenz der Primzahlen ausdruckt:

```
1  a := 2
2  print(a)
3  a := a + 1
4  b := 2
5  if ((b * b) > a) then goto 2
6  if ((a mod b) == 0) then goto 3
7  b := b + 1
8  goto 5
```

Die möglichen Zustände bei der Ausführung des Programms können nun durch das Tripel (l, y_1, y_2) angegeben werden, wobei l die momentane Zeilennummern und y_1 und y_2 die Belegungen der Variablen a bzw. b nach der Berechnung in dieser Zeile darstellen. Somit beschreibt $(8, 3, 3)$ einen möglichen Zustand. Mit Hilfe der Modallogik können nun Aussagen wie „*Es ist notwendig, dass im Zustand* $(2, y_1, y_2)$ *gilt, dass* y_1 *prim ist*“ formuliert werden.

1977 schlägt Pnueli eine Erweiterung von Modallogiken um *temporale* Operatoren vor, so dass auch die zeitliche Ordnung beim Auftreten von Eigenschaften berücksichtigt werden kann [364]. Dabei werden Operatoren verwendet, um anzuzeigen, ob beispielsweise eine Eigenschaft irgendwann oder immer in der Zukunft gilt.

Neben der Entscheidbarkeit von Problemen stellt sich die Frage nach der Effizienz, mit der Verifikationsprobleme lösbar sind. Diese Frage untersuchte Stephen Cook in seinem 1971 veröffentlichten Werk [116]. Auf seinen Überlegungen basiert die sog. *Komplexitätstheorie*. Es stellte sich heraus, dass einige Probleme in Polynomialzeit lösbar sind, d. h. die Laufzeit des Algorithmus zur Lösung kann als Polynom mit Abhängigkeit von den Eingabedaten des Problems abgeschätzt werden. Probleme, die diese Eigenschaft besitzen, gehören zur Komplexitätsklasse $\mathcal{P}$. Daneben gibt es viele Probleme, für die ein solcher Zusammenhang nicht gefunden werden kann, sondern bei denen die Laufzeit vielmehr exponentiell in der Größe der Eingabedaten wächst. Für viele praktische Probleme jedoch gilt, dass, wenn erst mal eine Lösung gefunden ist, diese in polynomieller Zeit auf ihre Gültigkeit überprüft werden kann. Solche Probleme, werden als $\mathcal{NP}$-vollständige Probleme (*nichtdeterministisch polynomial*) bezeichnet. Cook beweist, dass das *Boolesche Erfüllbarkeitsproblem* (engl. *Satisfiability, SAT*) $\mathcal{NP}$-vollständig ist. Das Boolesche Erfüllbarkeitsproblem kann wie folgt formuliert werden: *Gegeben sei eine aussagenlogische Formel. Finde eine Zuweisung der Werte* T *und* F *an die atomaren Aussagen, so dass die Formel zu wahr evaluiert.* Dabei wirft er die bis heute unbeantwortete Frage auf, ob die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ äquivalent sind (siehe Anhang C).

Nachdem die ersten Computer verfügbar waren, entwickelte sich schnell die neue Forschungsrichtung des *automatischen Theorembeweisens*. Der erste automatische Beweis durch eine Maschine gelang im Jahr 1954, wobei bewiesen wurde, dass die Summe zweier gerader Zahlen wieder gerade ist. Im Laufe der Zeit wur-

den die Implementierungen der automatischen Beweistechniken immer weiter verbessert. Im Jahr 1962 publizierten Davis et al. [128] eine neue Beweisprozedur zur Lösung des Booleschen Erfüllbarkeitsproblems. Es handelt sich hierbei um den *DPLL-Algorithmus* (nach deren Entwicklern Martin Davis, Hilary Putnam, George Logemann und Donald W. Loveland), der im Grunde noch heute in nahezu allen Programmen zum Lösen des Booleschen Erfüllbarkeitsproblems, sog. *SAT-Solvern*, implementiert wird.

Mitte der 1990er Jahre gab es große Fortschritte in der Implementierung von SAT-Solvern. Daneben wurden diese Programme auch immer häufiger erfolgreich mit anderen Theorielösern kombiniert [392], mit dem Ziel, entscheidbare Varianten der Prädikatenlogik erster Ordnung auf Erfüllbarkeit zu prüfen. Dabei wird eine Formel nicht hinsichtlich aller möglichen Interpretationen auf Erfüllbarkeit überprüft, sondern lediglich bezüglich einer sog. *Hintergrundtheorie*. Deshalb spricht man auch häufig von Erfüllbarkeit modulo Theorien (engl. *Satisfiability Modulo Theories, SMT*). Untersuchte Theorien sind u. a. Äquivalenz und uninterpretierte Funktionen, Lineare Arithmetik über reelle und ganze Zahlen, Divergenzlogik, Bitvektor-Theorie und Array-Theorie.

Ein großes Interesse an der Anwendung des Theorembeweisens lag schon früh in der *Programmverifikation*. Die frühesten Arbeiten gehen auf Floyd [166] und Hoare [221] zurück. Das zentrale Element des *Hoare-Kalküls* ist das sog. *Hoare-Tripel*, das beschreibt, wie ein Programmteil P den Zustand einer Berechnung verändert. Sei Φ eine Vorbedingung und Ψ eine Nachbedingung. Das Hoare-Tripel

$$\{\Phi\}\, P\, \{\Psi\}$$

besagt, dass wenn die Vorbedingung Φ gilt, dann das Programm P gestartet wird und terminiert, so ist die Nachbedingung Ψ erfüllt. Besonders interessant in diesem Zusammenhang ist die folgende Regel für while-Schleifen

$$\frac{\Phi \Rightarrow \Phi_I, \{\Phi_I \wedge C\}\, P\, \{\Phi_I\}, \Phi_I \wedge \neg C \Rightarrow \Psi}{\{\Phi\}\, \text{while}\ C\ \text{do}\ P\ \text{end}\, \{\Psi\}}$$

Hierbei stellt P den Schleifenrumpf und $\neg C$ die Abbruchbedingung für die Schleife dar. Φ_I ist die *Schleifeninvariante*, da diese vor der Ausführung der Schleife als auch nach deren Ausführung gilt. Ein großes Problem ist allerdings, dass die Bestimmung der Schleifeninvarianten oftmals nicht einfach möglich ist. Eine Möglichkeit hierzu bilden *Fixpunktalgorithmen*.

Fixpunktalgorithmen spielen auch bei den Beweisverfahren für temporale Aussagenlogiken eine entscheidende Rolle. Dabei wird die geforderte funktionale Eigenschaft eines Systems mit Hilfe einer temporallogischen Formel ausgedrückt. Für das System selbst wird ein Modell benötigt. Aus diesem Grund werden die entsprechenden Beweisverfahren auch als *Modellprüfung* bezeichnet. Erste Modellprüfungsansätze sind in den frühen 1980er Jahren entstanden [369, 97, 304, 449].

Diese Modellprüfungsverfahren basierten auf einer expliziten Aufzählung der erreichbaren Zustände. Für Systeme mit kleiner Anzahl an Zuständen, stellt dies kein Problem dar. Allerdings für Systeme, wie sie in der Praxis vorkommen, sind diese

Ansätze meist nicht praktikabel. 1987 erkannte McMillan [74, 316], dass durch eine symbolische Repräsentation des Zustandsraums sehr viel größere Systeme automatisch verifiziert werden können. Die symbolische Repräsentation basierte dabei auf sog. *binären Entscheidungsdiagrammen* (engl. *Binary Decision Diagrams, BDDs*) [62]. Binäre Entscheidungsdiagramme sind Repräsentationen von Booleschen Funktionen und oftmals kompakter als andere Repräsentationen, etwa Funktionstabellen. Weiterhin existieren effiziente Algorithmen zu deren Manipulation. Mit der symbolischen Modellprüfung wurde es möglich, Systeme mit mehr als 10^{120} Zuständen zu verifizieren [72, 73]. Basierend auf den Ergebnissen der symbolischen Modellprüfung entwickelten sich auch schnell neue Verfahren zur *formalen Äquivalenzprüfung* von Hardware [58].

All den formalen Ansätzen gemein ist allerdings, dass diese für reale Systeme sehr viel Speicher benötigen. Glücklicherweise zeigten die 1990er Jahre, wie oben beschrieben, enorme Fortschritte im Entwurf von SAT-Solvern. 1999 stellten Biere et al. eine symbolische Modellprüfung vor, die anstelle von BDDs SAT-Solver verwendet [48, 49]. Die *SAT-basierte Modellprüfung* führt dabei zunächst keine vollständige Verifikation durch, sondern betrachtet das Systeme lediglich über k Zeitschritte. Hierdurch ist es nicht mehr direkt möglich, zu zeigen, dass das System eine geforderte Eigenschaft erfüllt. Statt dessen wird ein „Fehlerfall“ gefunden, in dem die Eigenschaft nicht erfüllt wird.

Mit der Verfügbarkeit von Computern hat sich noch eine weitere Richtung von Verifikationsverfahren entwickelt. Diese Verfahren basieren auf der *Simulation* des Systems. Wie bei der SAT-basierten Modellprüfung kann hierbei lediglich die Anwesenheit von Fehlern gezeigt werden, sofern ein Fehlerfall gefunden wird. Um einen simulativen Beweis für Korrektheit zu führen, wäre es notwendig, das System mit allen möglichen Testfalleingaben zu stimulieren und die Systemantwort zu bewerten. Dies ist bei heutigen Systemen kein realistischer Ansatz. Dennoch haben sich simulative Verifikationsansätze, insbesondere im industriellen Einsatz, etabliert. Ein Grund hierfür liegt in der im Gegensatz zu formalen Methoden einfacheren Handhabung. Für simulative Verifikationsverfahren werden die zu prüfenden Eigenschaften direkt im Quelltext des Programms formuliert. In der Praxis hat sich gezeigt, dass das Einfügen von *Zusicherungen* (engl. *assertions*) direkt in den Quelltext während des Programmierens sehr effizient und effektiv ist, um Fehler frühzeitig zu finden. Bereits eine der ersten Hochsprachen, die Sprache *C* [257], welche in den frühen 1970er Jahren entwickelt wurde, verfügt dafür über spezielle `assert`-Befehle. Auch die in den 1980er entwickelte Hardware-Beschreibungssprache VHDL [20, 233] hat bereits das Konzept von Zusicherungen aufgegriffen.

Allerdings waren die in den sequenziellen Programmiersprachen entwickelten Zusicherungen nicht ausreichend, um zeitliche Zusammenhänge zwischen Annahmen und Zusicherungen zu formulieren. Erst Anfang der 2000er Jahre wurden die Prinzipien temporaler Logiken adaptiert und in Simulationsbibliotheken zur Verfügung gestellt [234, 235]. Als Ergebnis etablierte sich im industriellen Umfeld eine *simulative Modellprüfung*. Durch die formale Spezifikation der funktionalen Eigenschaften kann die simulative Modellprüfung aber auch durch formale

Modellprüfungsverfahren unterstützt werden. Dies wird als *zusicherungsbasierte Eigenschaftsprüfung* (engl. *Assertion-Based Verification, ABV*) bezeichnet.

Zur selben Zeit entstanden auch neue Programmiersprachen, welche die Objektorientierung von sequentiellen Programmiersprachen und die Nebenläufigkeit aus Hardware-Beschreibungssprachen kombinierten. Das Ergebnis sind sog. *Systembeschreibungssprachen*, etwa *SystemC* [207, 236] oder SpecC [172]. Insbesondere SystemC hat sich mittlerweile zu einem Defacto-Standard in der ausführbaren Spezifikation von Hardware/Software-Systemen entwickelt. Simulative Verifikationsansätze für SystemC werden im Rahmen der SCV (engl. *SystemC Verification Library*) unterstützt [424]. Darüber hinaus definiert SystemC-TLM einen Standard zur Beschreibung von sog. *Transaktionsebenenmodellen* [352], welche zunehmend als Strukturmodelle auf Systemebene Verwendung finden.

Große Verbreitung hat SystemC-TLM im Bereich der simulativen Verifikation des Zeitverhaltens. Aufgrund der hohen Abstraktionsebene ist die Simulation deutlich schneller als auf niedrigeren Abstraktionsebenen. Da in SystemC aber die Modellierung von Ressourcenbeschränkungen möglich ist, können die simulierten Zeiten dennoch recht exakt sein. Simulative Ansätze im Bereich der Zeitanalyse haben allerdings das selbe Problem wie simulative Ansätze zur funktionalen Eigenschaftsprüfung: Sie können nicht die Abwesenheit von Fehler beweisen. In der Verifikation des Zeitverhaltens bedeutet dies, dass das beste oder schlechteste Zeitverhalten des Systems per Simulation nicht mit Sicherheit bestimmbar ist. In den vergangenen zehn Jahren wurden aus diesem Grund neue formale Ansätze zur Verifikation des Zeitverhaltens entwickelt, die ganze Systeme, bestehend aus mehreren Prozessoren, berücksichtigen können.

Eine Zusammenfassung der wichtigen geschichtlichen Ereignisse im Bereich der Verifikation des vergangenen und dieses Jahrhunderts findet sich in Abb. 1.16.

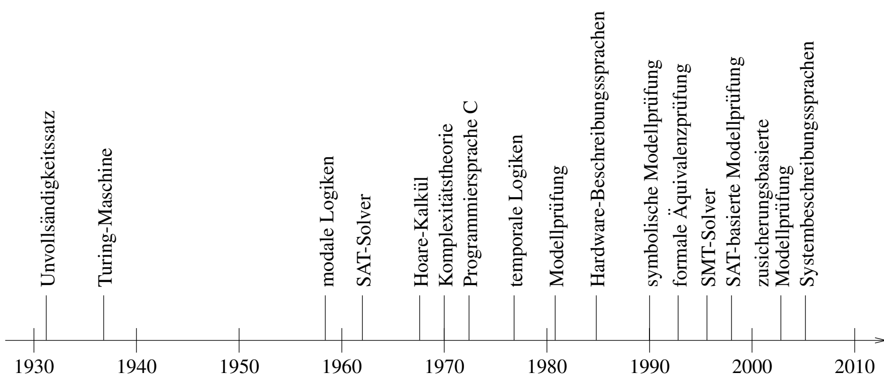


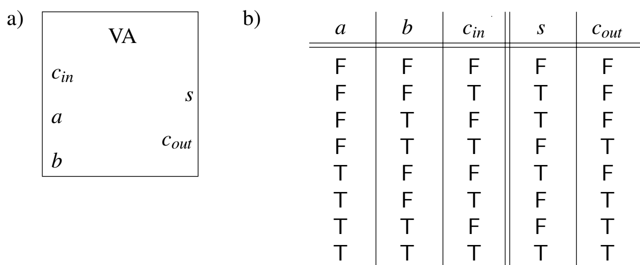
Abb. 1.16. Geschichte der Verifikation

1.4 Beispiele

Zum Abschluss dieses Kapitels werden einige Beispiele zu Äquivalenzprüfung sowie funktionaler und nichtfunktionaler Eigenschaftsprüfung auf unterschiedlichen Abstraktionsebenen präsentiert.

Hardware-Äquivalenzprüfung

Die Äquivalenzprüfung für Hardware wird anhand eines Addierers skizziert. Hardware-Addierer lassen sich aus sog. *Volladdierern* entwickeln. Das Blockschaltbild eines Volladdierer ist in Abb. 1.17a) zu sehen. Der Volladdierer hat drei Eingänge c_{in}, a, b und zwei Ausgänge s und c_{out}, wobei alle diese Signale einzelne Bits sind, d. h. $c_{in}, a, b, s, c_{out} \in \{\mathsf{F}, \mathsf{T}\}$.



a	b	c_{in}	s	c_{out}
F	F	F	F	F
F	F	T	T	F
F	T	F	T	F
F	T	T	F	T
T	F	F	T	F
T	F	T	F	T
T	T	F	F	T
T	T	T	T	T

Abb. 1.17. a) Volladdierer und b) Funktionstabelle

Die Ausgänge lassen sich als Boolesche Funktionen der Eingänge beschreiben:

$$s(c_{in}, a, b) := a \oplus b \oplus c_{in} \tag{1.1}$$

und

$$c_{out}(c_{in}, a, b) := (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in}) \tag{1.2}$$

wobei $\vee$ das logisch Oder, $\wedge$ das logische Und und $\oplus$ das logische Exklusiv-Oder darstellen.

Die Gleichungen (1.1) und (1.2) sind Verhaltensmodelle auf der Logikebene. Ob das Strukturmodell der Implementierung (abstrahiert durch das Blockschaltbild in Abb. 1.17a)) die selben Funktionen implementiert, kann durch Simulation gezeigt werden. Die drei Eingänge c_{in}, a und b können je einen der Werte F oder T annehmen. Dies bedeutet, es gibt acht mögliche Belegungen für die Eingänge des Volladdierers. Legt man jede der möglichen acht Belegungen nacheinander an die Eingänge des Volladdierers an, so erhält man an den Ausgängen die Werte, die in der Funktionstabelle in Abb. 1.17b) zu sehen sind.

Um zu zeigen, dass die Ausgaben in Abb. 1.17b) äquivalent zu der Verhaltensspezifikation in Gleichung (1.1) und (1.2) ist, müssen entweder die Verhaltensmodelle

in eine Funktionstabelle oder die Werte der Funktionstabelle in eine Funktionsbeschreibung umgewandelt werden. Hier wird letztere Möglichkeit gezeigt.

Die Funktionstabelle in Abb. 1.17b) lässt sich durch eine Boolesche Funktion in *disjunktiver Normalform* darstellen. Diese codiert die Einsstellen der entsprechenden Funktion. Für die beiden Funktionen $s(c_{in},a,b)$ und $c_{out}(c_{in},a,b)$ ergibt sich:

$$s(c_{in},a,b) = (\neg a \wedge b \wedge \neg c_{in}) \vee (a \wedge \neg b \wedge \neg c_{in}) \vee (\neg a \wedge \neg b \wedge c_{in}) \vee (a \wedge b \wedge c_{in}) \quad (1.3)$$

und

$$c_{out}(c_{in},a,b) = (a \wedge b \wedge \neg c_{in}) \vee (\neg a \wedge b \wedge c_{in}) \vee (a \wedge \neg b \wedge c_{in}) \vee (a \wedge b \wedge c_{in}) \quad (1.4)$$

Nun muss noch die Äquivalenz zwischen Gleichung (1.1) und Gleichung (1.3) sowie die Äquivalenz zwischen Gleichung (1.2) und Gleichung (1.4) gezeigt werden. Hier wird lediglich die Äquivalenz für die Berechnung der Summenbits gezeigt.

Mit der Definition der Exklusiv-Oder-Funktion $x_1 \oplus x_2 := (\neg x_1 \wedge x_2) \vee (x_1 \vee \neg x_2)$ ergibt sich für Gleichung (1.1):

$$s(c_{in},a,b) = (\neg((\neg a \wedge b) \vee (a \wedge \neg b)) \wedge c_{in}) \vee (((\neg a \wedge b) \vee (a \wedge \neg b)) \wedge \neg c_{in}) \quad (1.5)$$

Gleichung (1.5) kann mittels der *De Morganschen Gesetze* $\neg(x_1 \vee x_2) = \neg x_1 \wedge \neg x_2$ und $\neg(x_1 \wedge x_2) = \neg x_1 \vee \neg x_2$ umgeformt werden:

$$s(c_{in},a,b) = ((a \vee \neg b) \wedge (\neg a \vee b) \wedge c_{in}) \vee (((\neg a \wedge b) \vee (a \wedge \neg b)) \wedge \neg c_{in}) \quad (1.6)$$

Durch Anwendung des Distributivgesetzes $x_1 \wedge (x_2 \vee x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ kann Gleichung (1.6) umgeformt werden zu:

$$s(c_{in},a,b) = (a \wedge b \wedge c_{in}) \vee (\neg a \wedge \neg b \wedge c_{in}) \vee (\neg a \wedge b \wedge \neg c_{in}) \vee (a \wedge \neg b \wedge \neg c_{in}) \quad (1.7)$$

Durch Umsortieren der Terme erhält man Gleichung (1.1). Somit ist die vom Volladdierer implementierte Berechnung des Summenbits äquivalent zur Spezifikation.

Software-Eigenschaftsprüfung

Betrachtet wird ein Software-System bestehend aus zwei Prozessen. Da die Prozesse auf einem einzelnen Prozessor ausgeführt werden, gibt es beschränkte Ressourcen, die nur exklusiv von einem Prozess belegt werden dürfen. Um dies zu realisieren, werden beispielsweise sog. *Semaphore* verwendet. Ein Prozess, der eine Ressource exklusiv belegt, beansprucht zunächst das zugehörige Semaphor. Erhält der Prozess das beanspruchte Semaphor nicht, so muss dieser warten und kann die Ressource nicht belegen. Erhält der Prozess das Semaphor, so kann er die Ressource belegen. Man sagt, er betritt den kritischen Bereich. Nach Verlassen des kritischen Bereichs muss der Prozess das Semaphor wieder freigeben. Ein Zustandsmodell für einen ausführungsbereiten Prozess ist in Abb. 1.18a) dargestellt.

Ein ausführungsbereiter Prozess startet im Zustand N. Sobald dieser einen kritischen Bereich (Zustand K) betreten möchte, muss er zunächst das zugehörige Semaphor anfordern ($s?$) und in den Wartezustand W übergehen. Den kritischen Bereich

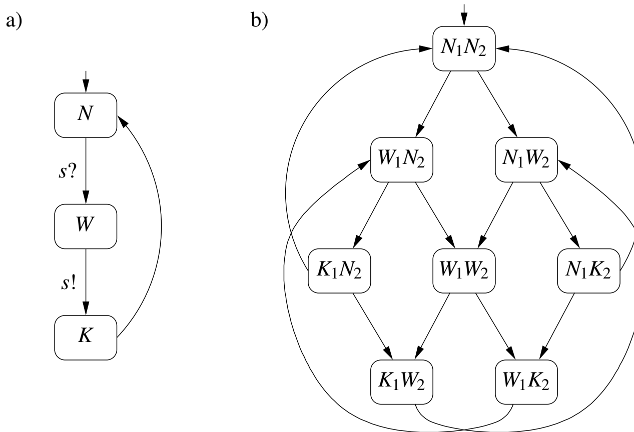


Abb. 1.18. a) Zustandsmodell eines ausführungsbereiten Prozesses und b) Zustandsmodell für zwei Prozesse

K kann der Prozess nach Zuteilung des Semaphors ($s!$) betreten. Den kritischen Bereich verlässt der Prozess, indem er in den Zustand N zurück wechselt. Dabei wird das Semaphor frei gegeben. Die Verwaltung des Semaphors und deren exklusive Zuteilung an Prozesse ist in Abb. 1.18a) nicht dargestellt und wird typischerweise von einem Betriebssystem kontrolliert.

Für die funktionale Eigenschaftsprüfung müssen neben dem System selbst auch die zu überprüfenden Eigenschaften formuliert werden. Eine wichtige Eigenschaft für das oben beschriebene Software-System ist beispielsweise, dass sich niemals zwei Prozesse gleichzeitig im kritischen Bereich befinden. Um solche Eigenschaften automatisiert überprüfen zu können, reicht es allerdings nicht, die Eigenschaft umgangssprachlich auszudrücken. Vielmehr ist es notwendig, die Eigenschaft formal zu beschreiben. Wenn z. B. K_1 und K_2 die kritischen Bereiche zweier Prozesse darstellen, so könnte ein erster Versuch, obige Eigenschaft zu formalisieren, auf Basis der Aussagenlogik erfolgen, d. h. als Formel $\neg(K_1 \wedge K_2)$. Die Aussagenlogik ist allerdings nicht ausreichend expressiv, um auszudrücken, wann diese Eigenschaft gelten soll. So könnte es unterschiedliche Interpretationen geben, wie z. B.: „Diese Eigenschaft gilt irgendwann“ oder „Es ist möglich, dass die Eigenschaft irgendwann gilt“.

Um diese Mehrdeutigkeit aufzulösen, kann beispielsweise die temporale Aussagenlogik zum Einsatz kommen. In dieser kann spezifiziert werden, dass die Aussage $\neg(K_1 \wedge K_2)$ zu jedem Berechnungszeitpunkt (G) und für jede Ausführungsreihenfolge (A) der beiden Prozesse gilt. Die obige Eigenschaft lässt sich somit als

$$\mathrm{AG}\ \neg(K_1 \wedge K_2) \tag{1.8}$$

formulieren.

Ein weiteres Beispiel für eine Eigenschaft wäre: „Eine Anfrage, den kritischen Bereich zu betreten, wird irgendwann erfüllt“. Auch hier ist wiederum die Aussagenlogik nicht mächtig genug, um zeitliche und modale Effekte auszudrücken. Die entsprechende temporallogische Formel lautet:

$$\mathrm{AG}\,(W_1 \Rightarrow \mathrm{AF}\,K_1) \tag{1.9}$$

Diese besagt, dass auf allen Berechnungspfaden (A) und zu einem beliebigen Berechnungszeitpunkt in der Zukunft (F), nachdem Prozess 1 den Wartezustand W_1 eingenommen hat, dieser auch in den kritischen Bereich K_1 wechseln kann. Weiterhin muss diese Bedingung zu jedem Berechnungszeitpunkt (G) und für alle beliebigen Ausführungsreihenfolgen (A) der Prozesse gelten.

Die Überprüfung der Eigenschaften (1.8) und (1.9) kann beispielsweise erfolgen, in dem alle möglichen Ausführungsreihenfolgen der Prozesse betrachtet werden. Für zwei Prozesse ist dies in Abb. 1.18b) dargestellt. Im Anfangszustand N_1N_2 sind beide Prozesse ausführungsbereit. Jeder der beiden Prozesse kann nun das einzige Semaphor im System anfordern, wenn er ausgeführt wird. Hierdurch sind zwei mögliche Folgezustände denkbar, d. h. W_1N_2 oder N_1W_2. Fordert nun der andere Prozess ebenfalls das Semaphor an, so resultiert dies im Zustand W_1W_2. Alternativ kann der jeweils wartende Prozess den kritischen Bereich betreten. Die beiden Folgezustände lauten in diesem Fall K_1N_2 und N_1K_2.

Ohne die weiteren möglichen Zustände in Abb. 1.18b) näher zu beschreiben, sei hier festgestellt, dass ein Zustand K_1K_2, in dem beide Prozesse im kritischen Bereich sind, nicht erreicht wird. Hierfür sorgt das Betriebssystem, welches ein Semaphor immer nur exklusiv vergibt und somit den jeweils anderen Prozess am Eintritt in den kritischen Bereich hindert. Somit ist auch offensichtlich, dass die Eigenschaft (1.8) für dieses System erfüllt ist.

Weiterhin kann man durch genaues Überlegen erkennen, dass die Eigenschaft (1.9) nicht erfüllt ist. Die unendliche Wiederholung des Zyklus $W_1N_2 \rightarrow W_1W_2 \rightarrow W_1K_2 \rightarrow W_1N_2$ repräsentiert eine gültige Ausführungsreihenfolge und stellt somit ein Gegenbeispiel dar, in dem der wartende Prozess 1 niemals den kritischen Bereich betritt. Dies liegt an der unfairen Ablaufplanung der Prozesse, die den Prozess 1 „verhungern“ lässt.

An diesem kleinen Software-System sieht man bereits, wie schwierig es sein kann, die geforderten Eigenschaften korrekt zu formulieren und dass es nicht immer trivial sein muss, zu entscheiden, ob eine Eigenschaft erfüllt ist oder nicht. Ein wesentlicher Grund für diese Schwierigkeit liegt auch in der Größe der Systeme begründet, die heutzutage betrachtet werden. Eine Modellierung möglicher Ausführungsreihenfolgen mit Systemzuständen, wie in Abb. 1.18b), würde schon bei vier und mehr Prozessen nicht mehr auf einer Seite darstellbar sein. Bei heutzutage typischen Systemen mit mehreren hundert Prozessen führt dies zwangsläufig in das sog. *Zustandsexplosionsproblem* (engl. *state explosion*), was auch die Automatisierung vieler Ansätze für solche Systeme verhindert.

Prüfung nichtfunktionaler Systemeigenschaften

Neben der Prüfung funktionaler Eigenschaften ist die Überprüfung nichtfunktionaler Eigenschaften eingebetteter Systeme ein wichtiges Thema. Hier spielt insbesondere die Überprüfung des Zeitverhaltens eine entscheidende Rolle. Die Komplexität dieses Problems wird anhand eines Beispiels auf Systemebene aus [435] beschrieben.

Gegeben sei die Architektur und die Prozessbindung aus Abb. 1.19a). Ein Sensor sendet periodisch Daten an die CPU. Der Prozess p_1, der auf der CPU ausgeführt wird, ist dafür verantwortlich, die Sensordaten in den lokalen Speicher zu speichern. Ein zweiter Prozess p_2 verarbeitet diese Daten und versendet sie über den gemeinsamen Bus an eine I/O-Einheit. Die Ausführungszeit des Prozesses p_2 variiert zwischen der schnellsten Ausführungszeit d_{BC} und langsamsten Ausführungszeit d_{WC}. Auf der I/O-Einheit nimmt Prozess p_3 die Daten zur Weiterverarbeitung entgegen. In diesem Beispiel wird angenommen, dass auf der CPU ein Betriebssystem mit einer prioritätsbasierten, präemptiven Ablaufplanung unter Verwendung statischer Prioritäten läuft. Dabei hat Prozess p_1 eine höhere Priorität als der Prozess p_2. Die maximale Auslastung der CPU ergibt sich für den Fall, dass Prozess p_2 bei der Bearbeitung stets die längste Ausführungszeit benötigt, d. h. d_{WC}.

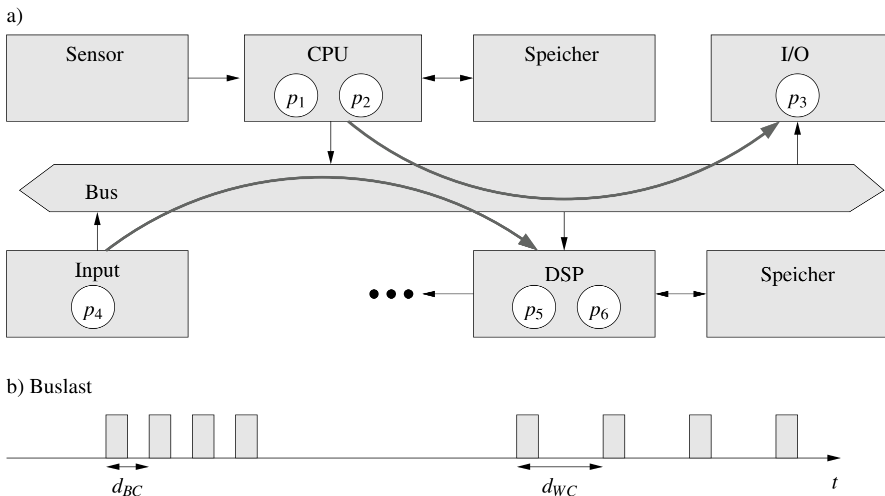


Abb. 1.19. Interferenz zweier Anwendungen auf einem gemeinsam genutzten Bus [435]

Neben der oben beschriebenen Anwendung gibt es eine weitere Anwendung, die in Abb. 1.19a) dargestellt ist. Diese Anwendungen erhält über das Interface Input periodisch Echtzeitdaten, welche durch den Prozess p_4 über den gemeinsamen Bus an den Prozess p_5, der auf dem Digitalen Signal Prozessor (DSP) im System läuft, sendet. Der Prozess p_5 speichert die Daten im lokalen Speicher des DSP. Aus diesem

Speicher liest der Prozess p_6 die Daten periodisch wieder aus und gibt sie an eine weitere lokale Komponente, z. B. die Audiowiedergabeeinheit.

Im Folgenden wird angenommen, dass auf dem Bus eine *First Come, First Served*-Arbitrierung verwendet wird. Unter dieser Annahme sieht man, dass die Datenübertragung von Prozess p_2 zu Prozess p_3 mit der Übertragung der Daten zwischen Prozess p_4 und p_5 interferiert. Interessant hierbei ist der Aspekt, dass eine Ausführung des Prozesses p_2 mit der schnellsten Ausführungszeit d_{BC} zu einer hohen Buslast, eine Ausführung mit der langsamsten Ausführungszeit d_{WC} zu einer geringen Buslast führt (siehe Abb. 1.19b)). Somit geht eine hohe Auslastung der CPU mit einer geringen Buslast einher, sowie eine geringe CPU-Auslastung mit einer hohen Buslast. Entscheidend ist allerdings, dass diese Varianz in der Buslast dazu führen kann, dass es im lokalen Speicher des DSP zu Unter- bzw. Überläufen kommen kann. Allein durch die Verwendung des gemeinsamen Busses ist das Zeitverhalten der beiden ansonsten unabhängigen Anwendungen voneinander abhängig geworden.

1.5 Ausblick

Basierend auf den in diesem Kapitel eingeführten Verifikationsaufgaben werden im Folgenden Ansätze zur Verifikation von digitalen Hardware/Software-Systemen vorgestellt. Zunächst werden modellbasierte Ansätze beschrieben. Die zugrundeliegenden Modelle sind dabei Teil der Spezifikation, weshalb in Kapitel 2 Möglichkeiten zur formalen oder ausführbaren Spezifikation präsentiert werden. Kapitel 3 gibt eine detaillierte Klassifikation von Verifikationsansätzen nach Verifikationsaufgabe, Verifikationsmethode und Verifikationsziel.

Bei den modellbasierten Ansätzen werden zunächst Verfahren zur Äquivalenzprüfung vorgestellt (Kapitel 4). Obwohl diese nur eine Spezialform der allgemeineren Eigenschaftsprüfung (Kapitel 5) darstellen, eignen sie sich besonders gut zur Einführung in die Verifikationsproblematik. Die in diesem Buch vorgestellten modellbasierten Verfahren lassen sich sowohl zur Verifikation von Hardware- als auch von Software-Komponenten einsetzen. Hierbei muss man allerdings die notwendige Abstraktion kritisch betrachten. Mit anderen Worten: Es eignen sich nicht alle modellbasierten Verfahren gleich gut zur Verifikation von Hardware und Software.

Im Anschluss an die Einführung in modellbasierte Verifikationsansätze werden spezielle Verfahren zur Hardware-Verifikation (Kapitel 6), zur Software-Verifikation (Kapitel 7) und Systemverifikation (Kapitel 8) vorgestellt. Anhang A bis C enthält grundlegende Definitionen, Datenstrukturen und Algorithmen.

1.6 Literaturhinweise

Das *Doppeldachmodell* sowie der Entwurfsprozess für digitale Hardware/Software-Systeme ist ausführlich in [426] beschrieben. Neben dem Doppeldachmodell gibt es weitere Modelle, die den Entwurf von eingebetteten Systemen beschreiben. So kann

das Doppeldachmodell als Erweiterung des *Y-Diagramms* nach Gajski und Kuhn [170] mit einer expliziten Trennung von Hardware- und Software-Entwurfsprozess verstanden werden. Dabei verzichtet das Doppeldachmodell auf ein zusätzliches *Layout-Dach*, welches eine physikalische Sicht auf den Entwurf bieten würde. Während traditionell Layout-Informationen auf der Systemebene eine untergeordnete Rolle gespielt haben, werden diese zunehmend auch auf dieser Ebene eingesetzt, um räumliche Effekte wie sog. engl. *hot spots* [477] oder durch Leitungslängen verursachte Verzögerungszeiten [354] zu berücksichtigen.

Das *X-Diagramm* ist ausführlich in [182] beschrieben und gibt einen Überblick über Methoden zur Systemsynthese. Es kombiniert zwei unterschiedliche Aspekte: Die Synthese mit dem Strukturmodell als Ausgabe und die Bewertung einer Implementierung in Form von Qualitätsmaßen. Diese beiden Aspekte wurden bereits früher durch zwei entsprechende Y-Diagramme getrennt behandelt: Der Syntheseaspekt wurde im bereits erwähnten Y-Diagramm nach Gajski und Kuhn [170] vorgestellt und später in eine entsprechend Entwurfsmethodik [171] umgesetzt. Der Bewertungsaspekt wurde erstmals durch Kienhuis et al. in [258] als Y-Diagramm präsentiert.

2

Spezifikation digitaler Systeme

Dieses Kapitel stellt wichtige Ansätze zur formalen und ausführbaren Spezifikation digitaler Hardware/Software-Systeme vor.

2.1 Wie spezifiziert man ein System?

Im Doppeldachmodell in Abb. 1.10 und 1.13 auf Seite 14 bzw. 19 beschreibt das obere Dach die *Spezifikation* des Systems auf unterschiedlichen Abstraktionsebenen. Die Erstellung der Spezifikation auf Systemebene stellt einen zentralen Schritt vor dem Systementwurf und somit der Systemverifikation dar. Typischerweise wird eine Spezifikation anhand von *Anforderungen* (engl. *requirements*) erstellt. Zwei wesentliche Bestandteile einer Spezifikation sind dabei immer eine Beschreibung des gewünschten *Verhaltens* und der geforderten *Eigenschaften* an die Implementierung des Systems.

Definition 2.1.1 (Spezifikation). *Eine* Spezifikation *eines Systems besteht aus einer präzisen Beschreibung des* Verhaltens *des Systems und einer präzisen Beschreibung der geforderten* Eigenschaften *an die Implementierung des Systems.*

Eine Spezifikation beschreibt somit, was ein System unter welchen Bedingungen tun soll. Dabei sollte die Spezifikation möglichst abstrakt sein, also unnötige Details vermeiden, und generell sein, um resistent gegen spätere Änderungen der Anforderungen zu sein. Spezifikationen können dabei informal oder formal sein. Eine *formale Spezifikation* basiert auf Beschreibungen mit klar definierter formaler Semantik. Formale Spezifikationen basieren somit auf formalen Verhaltensmodellen sowie formalen Anforderungsbeschreibungen. Dies kann es ermöglichen, Konsequenzen zu berechnen, die sich aus der Spezifikation ergeben.

Somit stellt sich aber die Frage, wie ein System zu spezifizieren ist? Wie in Abb. 1.3 auf Seite 4 dargestellt, startet der Entwurfsfluss mit einer Idee. Diese wird typischerweise nicht direkt in eine formale Spezifikation umgesetzt. Vielmehr beginnen die meisten Entwicklungen mit einer *informalen Spezifikation*. Diese kann in

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_2, © Springer-Verlag Berlin Heidelberg 2010

Form von natürlicher Sprache, Skizzen oder anderen unvollständigen Beschreibungen mit informaler Semantik erfolgen. Informale Spezifikationen leiden oft darunter, dass sie [272]:

- mehrdeutig und
- unvollständig sowie
- schlecht strukturiert und somit schwer zu analysieren und formalisieren sind.

Um die Qualität eines Systems zu verbessern, ist es somit notwendig, eine informale Spezifikation in eine formale Spezifikation zu überführen. Dies ist sowohl im Entwurf [426] als auch in der Verifikation von Vorteil. Wie kann also eine formale Spezifikation aus einer informalen Beschreibung gewonnen werden, so dass diese *korrekt*, *eindeutig* und *vollständig* ist sowie alle wesentlichen Aspekte abdeckt?

Die Überprüfung, ob eine Spezifikation korrekt ist, wird als *Validierung* bezeichnet. Für formale Spezifikationen können hierzu oftmals Plausibilitätstests oder Konsistenzprüfungen durchgeführt werden. Ist eine Spezifikation *ausführbar*, d. h. basiert diese auf einer ausführbaren Verhaltensbeschreibung, so kann diese zum Zwecke der Validierung simuliert werden. Obwohl einige der in diesem Buch vorgestellten Methoden auch zur Validierung einer Spezifikation Anwendung finden können, ist der Prozess der Validierung nicht Gegenstand dieses Buches. Dennoch sei betont, dass die Validierung der Korrektheit einer Spezifikation entscheidend für die Qualität eines zu entwickelnden Produktes ist. Dies liegt insbesondere daran, dass unentdeckte Spezifikationsfehler zu Produktfehlern führen können, da die Verifikation nur die Korrektheit bezüglich einer Spezifikation zeigen kann.

Die Probleme bei der Erstellung einer formalen Spezifikation aus einer informalen Beschreibung werden im nachfolgenden Beispiel illustriert. Das Beispiel stammt aus [272].

Beispiel 2.1.1. Betrachtet wird das System aus Abb. 2.1.

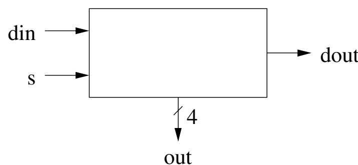


Abb. 2.1. Seriell-Parallel-Wandler [272]

Weiterhin ist die folgende Spezifikation gegeben: *Der Eingang* din *verarbeitet eine Sequenz aus Bits. Der Ausgang* dout *emittiert die selbe Sequenz mit einer Verzögerung von vier Taktzyklen. Der Bus* out *ist vier Bit breit. Falls der Eingang* s *den Wert* F *hat, dann entspricht das 4-Bit Wort an* out *den letzten vier Bit am Eingang* din. *Falls* s *gleich* T *ist, dann ist das Wort an* out *gleich* 0*, d. h.* $[\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F}]$.

Obwohl die Spezifikation sehr detailliert für ein solch kleines System wirkt, ist sie

1. *ungenau*: Enthalten die letzten vier Bit des Eingangs auch das momentane Bit?
2. *unvollständig*: Auf welchen Wert wird der Ausgang dout in den ersten drei Taktzyklen gesetzt?
3. *nicht analysierbar*: Die Spezifikation ist umgangssprachlich und unstrukturiert und somit nicht automatisch verarbeitbar.

Zunächst wird aus der informalen eine formale Spezifikation erstellt. Dabei wird die Zeit als natürliche Zahl $\tau \in \mathbb{N}$ modelliert.

$$\forall \tau \in \mathbb{N} : \mathrm{dout}(\tau) = \mathrm{din}(\tau - 4)$$
$$\mathrm{out}(\tau) = \begin{cases} [\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F}] & \text{falls } \mathrm{s}(\tau) = \mathsf{T} \\ [\mathrm{din}(\tau-4),\mathrm{din}(\tau-3),\mathrm{din}(\tau-2),\mathrm{din}(\tau-1)] & \text{sonst} \end{cases}$$

Betrachtet man diese formale Spezifikation etwas genauer, fällt auf, dass die Werte für dout in den ersten drei Taktzyklen weiterhin unspezifiziert sind. Da hier die Zeit lediglich über die natürlichen Zahlen definiert ist, ist unklar, ob der Wert $\mathrm{din}(-1)$, $\mathrm{din}(-2)$ und $\mathrm{din}(-3)$ definiert vorliegt. Dieses Problem kann mit der folgenden Annahme, dass dout die ersten drei Takt den Wert F trägt, behoben werden.

$$\forall \tau \in \mathbb{N} : \mathrm{dout}(\tau) = \begin{cases} \mathsf{F} & \text{if } \tau < 4 \\ \mathrm{din}(\tau-4) & \text{else} \end{cases}$$
$$\mathrm{out}(\tau) = \begin{cases} [\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F}] & \text{falls } \mathrm{s}(\tau) = \mathsf{T} \\ [\mathrm{din}(\tau-4),\mathrm{din}(\tau-3),\mathrm{din}(\tau-2),\mathrm{din}(\tau-1)] & \text{sonst} \end{cases}$$

Nun ist das Verhalten zwar vollständig spezifiziert, fraglich ist jedoch, ob dieses Verhalten von der informalen Spezifikation beabsichtigt war. Eine alternative formale Spezifikation, die ohne eine zusätzliche Annahme auskommt, könnte wie folgt aussehen.

$$\forall \tau \in \mathbb{N} : \mathrm{dout}(\tau+4) = \mathrm{din}(\tau) \wedge$$
$$\mathrm{out}(\tau+4) = \begin{cases} [\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F}] & \text{falls } \mathrm{s}(\tau) = \mathsf{T} \\ [\mathrm{din}(\tau),\mathrm{din}(\tau+1),\mathrm{din}(\tau+2),\mathrm{din}(\tau+3)] & \text{sonst} \end{cases}$$

Obwohl das obige Beispiel zeigt, dass die Erstellung einer formalen Spezifikation nicht einfach ist, hat diese einen erheblichen Vorteil: Bereits durch die Erstellung einer formalen Spezifikation können viele Fehler im Entwurfsprozess vermieden werden, sogar ohne dass die Spezifikation für die Verifikation verwendet wird.

Jede formale Spezifikation basiert auf ihrem formalen Verhaltensmodell, welches mathematisch fundiert ist. Die Ausdruckskraft eines Verhaltensmodells wird als *Berechnungsmodell* (engl. *Model of Computation, MoC*) bezeichnet. Berechnungsmodelle mit einer hohen Ausdruckskraft sind schwieriger zu analysieren, als Berechnungsmodelle mit einer geringeren Ausdruckskraft. Bestimmte Fragestellungen lassen sich somit nicht für alle Berechnungsmodelle beantworten.

Formale Verhaltensmodelle besitzen oftmals keine *Ausführungssemantik* und sind somit nicht simulierbar. Eine *ausführbare Spezifikation* hingegen besitzt die Eigenschaft, dass diese auf einem ausführbaren Verhaltensmodell basiert. Häufig werden zum Zweck der Erstellung von ausführbaren Verhaltensmodellen Programmiersprachen eingesetzt. Es sollte hier betont werden, dass Programmiersprachen (wie

z. B. C/C++, VHDL/SystemVerilog, SystemC, Esterel) im Allgemeinen verschiedene Berechnungsmodelle repräsentieren können und dass ein Berechnungsmodell in mehreren Sprachen ausgedrückt werden kann.

Ausführbare Verhaltensmodelle besitzen den Vorteil, dass diese eindeutig sind. Kommen Fragen während des Entwurfs des Systems auf, wird die Spezifikation für den fraglichen Fall simuliert und liefert die entsprechende Antwort. Auf der anderen Seite sind ausführbare Verhaltensmodelle häufig überspezifiziert, d. h. diese spezifizieren nicht nur das gewünschte Verhalten der Implementierung, sondern geben bereits Implementierungsentscheidungen vor. Dies kann dazu führen, dass Optimierungen an dem System während des Entwurfs nicht mehr vorgenommen werden können, da diese die Spezifikation verletzen. Somit können suboptimale Implementierungen entstehen. Ein weiterer Nachteil bei der Verwendung von Programmiersprachen zur Spezifikation ist die Detektion eingeschränkter Berechnungsmodelle. Programmiersprachen sind häufig berechnungsuniversell. Welches konkrete Berechnungsmodell in einem Programm gemeint war, lässt sich dann nur schwer oder gar nicht feststellen. Da allerdings formale Verifikationsmethoden auf eingeschränkten Berechnungsmodellen basieren, ist deren Anwendung somit nicht direkt möglich, da die Spezifikation nicht restriktiv genug ist.

Betrachtet man Abb. 1.12 auf Seite 17, so sieht man, dass eine Spezifikation neben dem Verhaltensmodell weitere *Anforderungen* enthält. Während das Verhaltensmodell beschreibt, was ein System tun soll, schränken die Anforderungen die Implementierungsmöglichkeiten ein, d. h. sie stellen Bedingungen an die Qualitätsmerkmale einer Implementierung dar. Wie bei der Spezifikation des Verhaltens, ist es auch bei der Spezifikation der Anforderungen vorteilhaft, diese formal zu erstellen. Anforderungen können in *funktionale* und *nichtfunktionale Anforderungen* eingeteilt werden. Funktionale Anforderungen beschränken *funktionale Eigenschaften* des Systems. Wichtige funktionale Eigenschaften für eingebettete Systeme sind *Gefahrlosigkeits-* (engl. *safety*) und *Lebendigkeitseigenschaften* (engl. *liveness properties*). Gefahrlosigkeit drückt dabei aus, dass ein System niemals einen für sich oder die Umwelt gefährlichen Zustand einnehmen wird. Lebendigkeit beschreibt die Eigenschaft, dass ein System irgendwann in der Zukunft „etwas“ tut.

Nichtfunktionale Anforderungen formulieren Ziele für *nichtfunktionale Eigenschaften* eines eingebetteten Systems. Beispiele für nichtfunktionale Eigenschaften eingebetteter Systeme sind vielfältig. Wichtige Vertreter sind die Reaktionszeit, der Durchsatz, das Gewicht, die Zuverlässigkeit etc. Da eingebettete Systeme für spezielle Aufgaben entwickelt werden, sind nichtfunktionale Anforderungen, wie maximale Antwortzeit, minimale Durchsatz, maximales Gewicht, minimale Zuverlässigkeit etc., in diesem Bereich sehr wichtig. Die Messung oder Abschätzung funktionaler und nichtfunktionaler Eigenschaften erfolgt mittels geeigneter Qualitätsmaße. Obwohl das Einhalten verschiedenster nichtfunktionaler Anforderungen essentiell für eingebettete Systeme ist, und somit alle diese Anforderungen erfüllt sein müssen, wird in dem vorliegenden Buch lediglich die Überprüfung zeitlicher Anforderungen behandelt.

Im Folgenden werden wichtige formale und ausführbare Verhaltensmodelle sowie Ansätze zur Spezifikation von funktionalen und nichtfunktionalen Anforderungen im Bereich eingebetteter Hardware/Software-Systeme vorgestellt.

2.2 Formale Verhaltensmodelle

Eine Möglichkeit, das gewünschte Systemverhalten präzise zu beschreiben, besteht in der Erstellung eines *formalen Verhaltensmodells*. Geeignete Modelle zur Verhaltensmodellierung von digitalen Hardware/Software-Systemen wurden bereits in [426] vorgestellt. Hier folgt eine Zusammenfassung wichtiger formaler Verhaltensmodelle. Weiterhin werden die für dieses Buch notwendigen Erweiterungen vorgestellt.

2.2.1 Petri-Netze

Ein *Petri-Netz* (nach dem Mathematiker C. A. Petri [360] benannt) ist ein formales Verhaltensmodell zur Beschreibung von Systemen mit dem Schwerpunkt der Modellierung von asynchronen, nebenläufigen Vorgängen. Petri-Netze werden sowohl zur Modellierung von Systemen im Hardware-Entwurf (insbesondere asynchroner Schaltungen) als auch im Software-Entwurf eingesetzt. Abbildung 2.2 zeigt ein Beispiel eines *Netzgraphen*, einer graphischen Darstellung eines Petri-Netzes. Ein Petri-Netz besteht aus einem Netzgraphen und einer *Dynamisierungsvorschrift*. Im Weiteren wird allerdings nicht zwischen einem Petri-Netz und seinem Netzgraphen unterschieden. Weiterhin wird im Folgenden lediglich die Klasse der *Stellen-Transitions-Netze* betrachtet. Der Begriff Petri-Netz wird im Folgenden als Synonym für Stellen-Transitions-Netze verwendet. Formal definiert man ein Petri-Netz wie folgt:

Definition 2.2.1 (Petri-Netz). *Ein* Petri-Netz $G(P,T,F,K,W,M_0)$ *ist ein 6-Tupel* (P,T,F,K,W,M_0) *mit* $P \cap T = \emptyset$ *und* $F \subseteq (P \times T) \cup (T \times P)$. *Darin bezeichnet*

- $P = \{p_0, p_1, \ldots, p_{m-1}\}$ *die Menge der* Stellen *(oder* Plätze*),*
- $T = \{t_0, t_1, \ldots, t_{n-1}\}$ *die Menge der* Transitionen.

Beide Mengen zusammen bilden die Menge der Knoten.

- F *heißt* Flussrelation. *Die Elemente von* F *heißen* Kanten.
- $K : P \to \mathbb{N} \cup \{\infty\}$ *(*Kapazitäten *der Stellen – evtl. unbeschränkt),*
- $W : F \to \mathbb{N}$ *(*Kantengewichte *der Kanten),*
- $M_0 : P \to \mathbb{Z}_{\geq 0}$ Anfangsmarkierung, *wobei* $\forall p \in P : M_0(p) \leq K(p)$.

Ein Petri-Netz ist also ein *gerichteter, bipartiter Graph*. In der üblichen Darstellung werden die Stellen als Kreise, die Transitionen als Rechtecke und die Kanten als Pfeile notiert. Im Zusammenhang mit einem Netzknoten x hat man häufig mit zwei bestimmten Mengen von Nachbarknoten zu tun. Dies ist zum einen der *Vorbereich*

$$\bullet x = \{y \mid (y,x) \in F\},$$

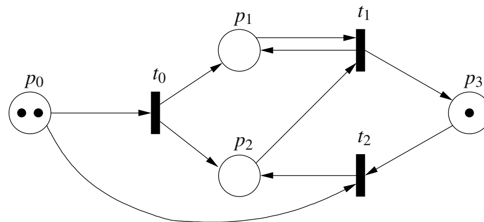
also die Menge aller direkten Vorgänger von x (z. B. *Eingangsstellen*, falls x Transition ist). Zum anderen ist dies der *Nachbereich*

$$x\bullet \;=\; \{y \mid (x,y) \in F\},$$

also die Menge aller direkten Nachfolger von x.

Beispiel 2.2.1. In Abb. 2.2 sind neben einem Netzgraphen ebenfalls die Vor- und Nachbereiche der Transitionen dargestellt sowie die Anfangsmarkierung M_0. Es gibt zehn Kanten ($F = \{f_0, f_1, \ldots, f_9\}$), z. B. ist $f_0 = (p_0, t_0)$. In der graphischen Darstellung wird die Markenzahl $M(p)$ einer Stelle $p \in P$ unter der Markierung M durch $M(p)$ Punkte dargestellt, siehe z. B. Stelle p_0. Die Kapazität einer Stelle bezeichnet eine obere Schranke der Markenzahl dieser Stelle. Die Bedeutung der Kantengewichte wird in der nachfolgenden Definition erläutert. Kapazitäten $\neq \infty$ bzw. Gewichte $\neq 1$ werden an die betreffenden Stellen bzw. Kanten geschrieben. Die Kapazitäten aller Stellen in Abb. 2.2 seien unbeschränkt, die Gewichte aller Kanten $= 1$.

$G = (P, T, F, K, W, M_0)$
$P = \{p_0, p_1, p_2, p_3\}$
$T = \{t_0, t_1, t_2\}$
$F = \{f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9\}$
$f_0 = (p_0, t_0), f_1 = (t_0, p_1), f_2 = (p_0, t_2), f_3 = (t_0, p_2), f_4 = (p_1, t_1),$
$f_5 = (t_1, p_1), f_6 = (p_2, t_1), f_7 = (t_2, p_2), f_8 = (t_1, p_3), f_9 = (p_3, t_2)$
$M_0(p_0) = 2, M_0(p_1) = 0, M_0(p_2) = 0, M_0(p_3) = 1$



$\bullet t_0 = \{p_0\}$ $\qquad t_0\bullet = \{p_1, p_2\}$
$\bullet t_1 = \{p_1, p_2\}$ $\qquad t_1\bullet = \{p_1, p_3\}$
$\bullet t_2 = \{p_0, p_3\}$ $\qquad t_2\bullet = \{p_2\}$

Abb. 2.2. Beispiel eines Petri-Netzes G

Marken sind Attribute von Stellen und zirkulieren im Petri-Netz. Der Zustand eines Netzes wird über die *Markierungsfunktion* $M : P \to \mathbb{Z}_{\geq 0}$ (im Weiteren oft einfach als *Markierung* bezeichnet) definiert, wobei $M(p)$ die Zahl der Marken einer Stelle $p \in P$ darstellt.

Es fehlt nun noch eine *Dynamisierungsvorschrift*, die angibt, unter welchen Umständen Marken im Petri-Netz bewegt werden können. Jeweils zulässige Mar-

kierungswechsel ergeben sich aus der sog. *Schaltregel* (engl. *firing rule*), worin die *Schaltbereitschaft* und das *Schalten* einer Transition wie folgt definiert werden:

Definition 2.2.2 (Schaltbereitschaft). *Eine Transition $t \in T$ eines Petri-Netzes $G(P, T, F, K, W, M_0)$ heißt* schaltbereit *unter der Markierung M, geschrieben $M[t\rangle$, wenn*

1. $\forall p \in \bullet t : M(p) \geq W(p,t)$,
2. $\forall p \in t\bullet \setminus \bullet t : M(p) \leq K(p) - W(t,p)$ *und*
3. $\forall p \in t\bullet \cap \bullet t : M(p) \leq K(p) - W(t,p) + W(p,t)$.

Damit eine Transition *schalten* (oder feuern) kann, muss sie schaltbereit sein. Das Schalten führt zu dem Ergebnis, dass von jeder Eingangsstelle $p \in \bullet t$ genau $W(p,t)$ Marken abgezogen (oder konsumiert) werden und jeweils $W(t,p)$ Marken an Ausgangsstellen $p \in t\bullet$ gelegt (sprich produziert) werden:

Definition 2.2.3 (Schalten einer Transition). *Beim* Schalten einer Transition $t \in T$ *eines Petri-Netzes $G(P, T, F, K, W, M_0)$ von Markierung M auf M', geschrieben $M[t\rangle M'$, wird M' aus M wie folgt gebildet:*

$$M'(p) = \begin{cases} M(p) - W(p,t) & \text{falls } p \in \bullet t \setminus t\bullet \\ M(p) + W(t,p) & \text{falls } p \in t\bullet \setminus \bullet t \\ M(p) - W(p,t) + W(t,p) & \text{falls } p \in t\bullet \cap \bullet t \\ M(p) & \text{sonst} \end{cases}$$

M' heißt *Folgemarkierung* von M unter t. Das Entfernen der Marken der Eingangsstellen und das Produzieren der Marken an den Ausgangsstellen der Transition erfolgt simultan.

Beispiel 2.2.2. In dem Petri-Netz in Abb. 2.2 ist Transition t_2 schaltbereit. Nach Feuern von t_2 ist die neue Markierung M' mit $M'(p_0) = M'(p_2) = 1$ und $M'(p_1) = M'(p_3) = 0$.

Petri-Netze sind zur Modellierung dynamischer, zustandsdiskreter Systeme weit verbreitet. Zunächst werden nur Petri-Netze mit unbeschränkter Kapazität und Einheitsgewichten ($\forall p \in P : K(p) = \infty, \forall f \in F : W(f) = 1$) betrachtet. Mit Petri-Netzen kann man *Sequentialität*, *Nebenläufigkeit*, *Synchronisation* und *Konflikte* modellieren. Ein Konflikt liegt vor, wenn zwei oder mehr Transitionen schaltbereit sind, die mindestens eine gemeinsame Eingangsstelle haben, und das Schalten einer Transition die Schaltbereitschaft einer anderen zerstört.

Definition 2.2.4. *Zwei Transitionen t_i und t_j eines Petri-Netzes mit $\forall p \in P : K(p) = \infty$ sind im* Konflikt, *wenn $M[t_i\rangle$ und $M[t_j\rangle$, aber nicht $M[\{t_i, t_j\}\rangle$. Dabei bezeichne $M[\{t_0, t_1, \ldots, t_{n-1}\}\rangle$ die gleichzeitige, nebenläufige Schaltbereitschaft aller Transitionen $t_0, t_1, \ldots, t_{n-1}$.*

Funktionale Eigenschaften von Petri-Netzen

Im Folgenden werden funktionale Eigenschaften von Petri-Netzen definiert. Diese Eigenschaften erlauben die Einordnung bekannter Modelle als Teilklassen von Petri-Netzen. Die wichtigsten Eigenschaften sind *Sicherheit* und *Lebendigkeit*.

Zu jeder Markierung M eines Petri-Netzes gehört eine *Markierungsklasse*. Dies ist diejenige Menge von Markierungen, die neben M selbst noch alle diejenigen Markierungen enthält, die ausgehend von M durch Schaltfolgen erreichbar sind. Die Markierungsklasse der Anfangsmarkierung M_0 eines Petri-Netzes G heißt auch *Erreichbarkeitsmenge* von G, geschrieben $[M_0\rangle$.

Damit kann die Sicherheit eines Petri-Netzes wie folgt definiert werden:

Definition 2.2.5 (Sicherheit). *Sei $G(P,T,F,K,W,M_0)$ ein Petri-Netz und $B : P \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ eine Abbildung, die jeder Stelle eine* kritische Markenzahl *zuordnet. G heißt B-*sicher *bzw. B-*beschränkt*, wenn*

$$\forall M \in [M_0\rangle, p \in P : M(p) \leq B(p).$$

G heißt beschränkt*, wenn es eine natürliche Zahl b gibt, für die G B-beschränkt ist mit $\forall p \in P :\ B(p) = b$.*

Während Kapazitäten Begrenzungen der Markenzahlen darstellen, die *a priori* gegeben sind, stellt Sicherheit eine Begrenzung *a posteriori* dar, also eine Beobachtung bei Anwendung der Schaltregeln. Ein Petri-Netz ist genau dann beschränkt, wenn seine Erreichbarkeitsmenge endlich ist [32].

Definition 2.2.6. *Eine Transition t eines Petri-Netzes $G(P,T,F,K,W,M_0)$ heißt* tot*, wenn sie unter keiner erreichbaren Markierung schaltbereit ist: $\forall M \in [M_0\rangle : \neg M[t\rangle$.*

Definition 2.2.7. *Eine Transition t eines Petri-Netzes $G(P,T,F,K,W,M_0)$ heißt* aktivierbar*, wenn sie mindestens unter einer Folgemarkierung schaltbereit ist: $\exists M \in [M_0\rangle : M[t\rangle$. Sie heißt* lebendig*, wenn sie unter allen Folgemarkierungen aktivierbar ist: $\forall M \in [M_0\rangle : \exists \widetilde{M} \in [M\rangle : \widetilde{M}[t\rangle$.*

Man beachte, dass in dieser Definition tot nicht das Gegenteil von lebendig ist. Aufbauend auf dem Lebendigkeitsbegriff für Transitionen definiert man die Lebendigkeit eines Petri-Netzes wie folgt:

Definition 2.2.8 (Lebendigkeit). *Ein Petri-Netz $G(P,T,F,K,W,M_0)$ heißt* verklemmungsfrei *oder* schwach lebendig*, wenn es unter keiner Folgemarkierung tot ist: $\forall M \in [M_0\rangle : \exists t \in T : M[t\rangle$. G heiße* lebendig *oder* stark lebendig*, wenn alle seine Transitionen lebendig sind: $\forall t \in T, M \in [M_0\rangle : \exists \widetilde{M} \in [M\rangle : \widetilde{M}[t\rangle$. G heißt* tot *in einer Markierung M, wenn alle seine Transitionen tot sind: $\forall t \in T : \neg M[t\rangle$.*

Eng verbunden mit dem Begriff der toten Transition ist der Begriff der *Verklemmung* (engl. *deadlock*), formal geschrieben als $\exists M \in [M_0\rangle : \forall t \in T : \neg M[t\rangle$. Der Begriff der Verklemmung ist vor allem dann von Bedeutung, wenn das Petri-Netz

eine Verhaltensspezifikation für ein System darstellt, bei dem Verklemmungen als Störfälle einzustufen sind.

Ein Petri-Netz heißt *reversibel*, wenn es aus jeder erreichbaren Markierung wieder in die Anfangsmarkierung $[M_0\rangle$ gelangen kann.

Definition 2.2.9 (Reversibilität). *Ein Petri-Netz $G(P,T,F,K,W,M_0)$ heißt* reversibel*, wenn gilt:*

$$\forall M \in [M_0\rangle : M_0 \in [M\rangle$$

Ein Petri-Netz heißt schließlich *konservativ*, wenn die Anzahl der Marken im Netz unter allen Schaltfolgen von Transitionen konstant ist.

Definition 2.2.10 (Konservativität). *Sei $w : P \to \mathbb{Z}_{\geq 0}$. Ein Petri-Netz $G(P,T,F,K,W,M_0)$ heißt* konservativ*, wenn es für einen strikt positiven Vektor w die Bedingung*

$$\forall M \in [M_0\rangle : w \cdot M = w \cdot M_0$$

erfüllt, wobei $w \cdot M := \sum_{p \in P} w(p) \cdot M(p)$ ist.

Zeitbehaftete Petri-Netze

In den bisher betrachteten Petri-Netzen verbraucht das Schalten von Transitionen keinerlei Zeit. Mit anderen Worten: Schaltfolgen sind lediglich über die Datenabhängigkeiten kausal geordnet.

Zur Bewertung des Zeitverhaltens müssen Verhaltensmodelle um Zeitannotationen erweitert werden. *Zeitbehaftete Petri-Netze* sind Petri-Netze bei denen die Schaltzeitpunkte der Transitionen durch *untere* β_l und *obere Zeitschranken* β_u begrenzt sind.

Definition 2.2.11 (Zeitbehaftete Petri-Netz). *Ein* zeitbehaftetes Petri-Netz $G(P,T,F,K,W,M_0,\mathcal{B})$ *ist ein 7-Tupel $(P,T,F,K,W,M_0,\mathcal{B})$. Darin bezeichnet*

- *(P,T,F,K,W,M_0) ein Petri-Netz nach Definition 2.2.1 und*
- *$\mathcal{B} : T \to \mathbb{T} \times (\mathbb{T} \cup \{\infty\})$ eine Funktion, die jeder Transition $t \in T$ ein geschlossenes $[\beta_l(t), \beta_u(t)]$ bzw. offenes Intervall $[\beta_l(t), \beta_u(t))$ zuweist.*

Die Zeitschranken sind aus der Menge $\mathbb{T}$ bzw. $\mathbb{T} \cup \{\infty\}$. $\mathbb{T}$ ist die Menge aller Zeitpunkte und kann entweder die Menge $\mathbb{Z}_{\geq 0}$, $\mathbb{Q}_{\geq 0}$ oder $\mathbb{R}_{\geq 0}$ sein.

Die Zeitschranken begrenzen die Verzögerung einer Transition t von deren Aktivierung bis zu deren Schalten. Dies bedeutet: Wird eine Transition $t_0 \in T$ zum Zeitpunkt τ_0 nach Definition 2.2.2 schaltbereit, so schaltet diese im Zeitintervall $[\tau_l(t_0), \tau_u(t_0)] := \tau_0 + \mathcal{B}(t_0)$, sofern sie ihre Schaltbereitschaft zuvor nicht verliert. Dies kann beispielsweise passieren, wenn eine Transition zuvor schaltet, die mit t_0 in Konflikt steht. Die Markierungsfunktion $M : P \to \mathbb{Z}_{\geq 0}$ ist somit nicht mehr ausreichend, den Zustand eines zeitbehafteten Petri-Netzes zu beschreiben. Zusätzlich wird eine Zeitstruktur $\mathcal{T} : T \to \mathbb{T} \times (\mathbb{T} \cup \{\infty\})$ benötigt, die jeder Transition $t \in T$ einen frühesten $\tau_l(t)$ und spätestens Startzeitpunkt $\tau_u(t)$ zuordnet.

Definition 2.2.12 (Schaltbereitschaft). *Eine Transition $t_0 \in T$ eines zeitbehafteten Petri-Netzes $G(P,T,F,K,W,M_0,\mathcal{B})$ heißt* schaltbereit *unter der Markierung M und der Zeitstruktur $\mathcal{T}$, geschrieben $(M,\mathcal{T})[t_0\rangle$, wenn*

1. *t_0 schaltbereit unter M nach Definition 2.2.2, d. h. $M[t_0\rangle$, und*
2. *$\forall t \in T : \tau_l(t_0) \leq \tau_u(t)$ ist.*

Mit anderen Worten: Die Transition t_0 muss genügend Marken im Vorbereich und genügend Kapazität im Nachbereich besitzen, um schalten zu können; Zusätzlich muss der Schaltzeitpunkt $\tau(t_0)$ von t_0 so wählbar sein, dass keine andere Transition ihre eigenen Zeitschranken verletzt.

Das Schalten einer Transition $t_0 \in T$ bewirkt eine Zustandsänderung, d. h. sowohl eine Änderung der Markierung als auch der Zeitstruktur. Dies wird als $(M,\mathcal{T})[t_0\rangle(M',\mathcal{T}')$ geschrieben, wobei M' die Folgemarkierung nach Definition 2.2.3 und $\mathcal{T}'$ die aktualisierte Zeitstruktur ist. Um diese Änderung zu formulieren, müssen zunächst diejenigen Transitionen $t \in T$ identifiziert werden, die durch Schalten der Transition t_0 neu aktiviert werden. Dies sind diejenigen Transitionen $t \in T$, die nach dem Schalten der Transition t_0 schaltbereit sind ($M'[t\rangle$), dies aber vor dem Schalten von t_0 nicht waren ($\neg M[t\rangle$). Zusätzlich kann die Transition t_0 neu aktiviert werden. Somit erfüllt eine *neuaktivierte Transition* folgende Bedingung:

$$M'[t\rangle \wedge (\neg M[t\rangle \vee (t = t_0))$$

Die aktualisierte Zeitstruktur $\mathcal{T}'$ nach Schalten der Transition t_0 zum Zeitpunkt $\tau(t_0)$ ergibt sich zu:

$$\tau_l'(t) := \begin{cases} \tau(t_0) + \beta_l(t) & \text{falls } M'[t\rangle \wedge (\neg M[t\rangle \vee (t = t_0)) \\ \max\{0, \tau_l(t) - \tau(t_0)\} & \text{falls } M'[t\rangle \wedge M[t\rangle \wedge (t \neq t_0) \\ \infty & \text{sonst} \end{cases} \tag{2.1}$$

und

$$\tau_u'(t) := \begin{cases} \tau(t_0) + \beta_u(t) & \text{falls } M'[t\rangle \wedge (\neg M[t\rangle \vee (t = t_0)) \\ \max\{0, \tau_u(t) - \tau(t_0)\} & \text{falls } M'[t\rangle \wedge M[t\rangle \wedge (t \neq t_0) \\ \infty & \text{sonst} \end{cases} \tag{2.2}$$

Mit anderen Worten: Für alle neuaktivierten Transitionen werden die aktualisierte Zeitstruktur auf die unteren und oberen Zeitschranken relativ zum Zeitpunkt $\tau(t_0)$ gesetzt. Alle nicht aktivierten Transitionen erhalten als untere und obere Schranke den Wert ∞, da diese nicht schalten können. Für diejenigen Transitionen, die vor dem Schalten von t_0 schaltbereit waren und dies danach weiterhin sind, werden die Zeitschranken angepasst, indem die Differenz der gegeben Zeitschranke zum Schaltzeitpunkt $\tau(t_0)$ ermittelt wird. Ist diese Differenz negativ, muss die entsprechende Schranke auf null gesetzt werden, da ein Schalten in der Vergangenheit nicht möglich ist.

Beispiel 2.2.3. Gegeben ist das zeitbehaftete Petri-Netz in Abb. 2.3. Dieses besteht aus drei Transitionen t_0, t_1 und t_2 und drei Stellen p_0, p_1 und p_2. Die Zeitschranken

der Transitionen sind an den Transitionen als Intervall annotiert. Die Anfangsmarkierung M_0 lautet $M_0(p_0) = M_0(p_2) = 1$ und $M_0(p_1) = 0$. Die Zeitstruktur ergibt sich zu $\mathcal{T}(t_0) = [0,4]$, $\mathcal{T}(t_1) = [\infty,\infty)$ und $\mathcal{T}(t_2) = [4,6]$.

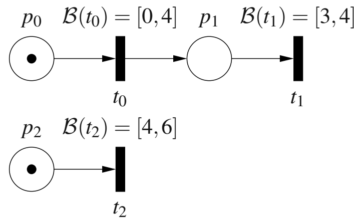


Abb. 2.3. Zeitbehaftetes Petri-Netz [88]

Unter dieser Anfangsmarkierung sind die Transitionen t_0 und t_2 schaltbereit. Unter der Annahme, dass t_2 zuerst schaltet, gibt es nur einen einzigen möglichen Schaltzeitpunkt $\tau(t_2) = 4$. Dieser entspricht der unteren Zeitschranke der Transition t_2. Ein späteres Schalten würde dazu führen, dass t_0 seine Zeitbeschränkungen verletzt. Durch Schalten der Transition t_2 ergibt sich die Folgemarkierung $M_0[t_2\rangle M_1$ zu $M_1(p_0) = 1$ und $M_1(p_1) = M_1(p_2) = 0$. Die aktualisierte Zeitstruktur sieht dann wie folgt aus: $\mathcal{T}(t_0) = [4,4]$, $\mathcal{T}(t_1) = \mathcal{T}(t_2) = [\infty,\infty)$. Lediglich t_0 ist schaltbereit. Als einziger Schaltzeitpunkt kommt $\tau(t_0) = 4$ in Frage. Die Folgemarkierung $M_1[t_0\rangle M_2$ ergibt sich zu $M_2(p_0) = M_2(p_2) = 0$ und $M_2(p_1) = 1$. Die Zeitstruktur wird aktualisiert zu: $\mathcal{T}(t_1) = [7,8]$, $\mathcal{T}(t_0) = \mathcal{T}(t_2) = [\infty,\infty)$. Nun ist lediglich Transition t_2 schaltbereit. Das Schalten von t_2 kann zu einem beliebigen Zeitpunkt innerhalb des Intervalls $\mathcal{T}(t_1)$ erfolgen. Danach ist keine Transition mehr schaltbereit und das Petri-Netz ist verklemmt.

2.2.2 Endliche Automaten

Definition 2.2.13 (Endlicher Automat). *Ein* nichtdeterministischer endlicher Automat *(engl.* Nondeterministic Finite Automaton, NFA*) ist ein* 6*-Tupel* (I,O,S,R,f,g). *Darin bezeichnet*

- I *das* Eingabealphabet*,*
- O *das* Ausgabealphabet*.*
- S *eine endliche nichtleere Menge von* Zuständen*,*
- $R \subseteq S$ *die* Menge der Anfangszustände*,*
- $f \subseteq (S \times I \times S)$ *die* Übergangsrelation *und*
- $g \subseteq (S \times I \times O)$ *die* Ausgaberelation*.*

Ein endlicher Automat heißt *deterministisch* (engl. *Deterministic Finite Automaton, DFA*), wenn $|R| = 1$ und f sowie g Funktionen sind mit $f : (S \times I) \to S$ und

$g : (S \times I) \rightarrow O$. Die Funktionen f und g werden dann als *Übergangsfunktion* bzw. *Ausgabefunktion* bezeichnet. Ein DFA heißt *vollständig spezifiziert*, wenn f und g Funktionen sind, also für alle Elemente aus $S \times I$ definiert sind.

Ein *Zustandsdiagramm*, siehe z. B. in Abb. 2.4a), ist eine graphische Repräsentation eines endlichen Automaten in Form eines gerichteten Graphen $G(V,E)$, in dem jeder Knoten $v \in V$ einen Zustand $s \in S$ und jede Kante $e \in E$ einen *Zustandsübergang* repräsentiert. Eine Kante wird mit einem Paar (i,o) in der Schreibweise *i/o* benannt, das den Zustandsübergang charakterisiert: Befindet sich der Automat in dem *aktuellen Zustand s* und das entsprechende Eingabeereignis $i \in I$ tritt ein, dann erfolgt ein Zustandsübergang in den *Folgezustand* s', und das Ausgabeereignis $o \in O$ wird erzeugt. Man schreibt auch $s \xrightarrow{i/o} s'$. Bei DFAs heißt ein Zustand $s \in S$ unter einer Eingabe *i stabil*, wenn für den Folgezustand $f(s,i)$ gilt: $f(s,i) = s$. Graphisch bedeutet dies, es existiert eine Schleife im Zustandsdiagramm.

Beispiel 2.2.4. In Abb. 2.4a) ist ein Zustandsdiagramm mit $|S| = 3$ Zuständen dargestellt. Es handelt sich um einen DFA. Das Eingabealphabet ist $\{i_0, i_1, i_2\}$, das Ausgabealphabet enthält die Ereignisse o_0, o_1, o_2, o_3, o_4. Die Menge der Zustände ist $\{s_0, s_1, s_2\}$. Der Anfangszustand heißt s_0. Der Zustand s_0 ist stabil unter der Eingabe i_0, s_1 ist stabil unter i_1 und s_2 ist stabil unter i_2.

Zustandsdiagramme können auch mit Petri-Netzen dargestellt werden. Petri-Netze mit der Eigenschaft, dass jede Transition genau eine Eingangsstelle und genau eine Ausgangsstelle besitzt und diese mit genau einer Marke auf einer der Stellen anfangs markiert sind, sind äquivalent mit dem Modell des nichtdeterministischen endlichen Automaten.

Definition 2.2.14 (NFA als Petri-Netz). *Ein* nichtdeterministischer endlicher Automat *ist ein Petri-Netz* $G(P,T,F,K,W,M_0)$ *mit den Eigenschaften:*

1. $\forall t \in T : |\bullet t| = |t \bullet| = 1$,
2. $\sum_{p \in P} M_0(p) = 1$,
3. $\forall p \in P : K(p) = 1$,
4. $\forall f \in F : W(f) = 1$,
5. *jeder Transition ist ein Paar in der Schreibweise Eingabeereignis/Ausgabeereignis zugewiesen, wobei das Eingabeereignis eine zusätzliche Bedingung an die Schaltbereitschaft der Transition beschreibt.*

Assoziiert man mit jeder Stelle des Petri-Netzes einen Zustand, dann ist ein Petri-Netz nach Definition 2.2.14 offensichtlich konservativ und damit die Menge erreichbarer Markierungen (= Zustandsraum) endlich. Es gibt genau einen Anfangszustand (es gibt genau eine Stelle, die eine Marke besitzt). Dieser ist bei einem deterministischen Automaten sogar eindeutig. Ein Zustandsdiagramm (siehe Abb. 2.4a)) konvertiert man in ein Petri-Netz, indem man jeden Zustand durch eine Stelle und jeden Zustandsübergang durch eine entsprechende stellenverbindende Transition modelliert (siehe Abb. 2.4b)). Zustandsdiagramme lassen sich somit als interpretierte Petri-Netze auffassen.

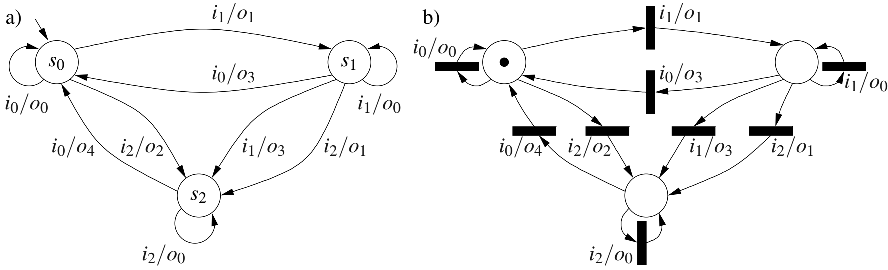


Abb. 2.4. a) Zustandsdiagramm und b) äquivalentes Petri-Netz

Zeitbehaftete Automaten

Die Erweiterung um Zeitaspekte in Petri-Netzen kann auch für endliche Automaten übernommen werden. Das Ergebnis sind sog. *zeitbehaftete Automaten*. Diese basieren auf einer Menge von Zeitvariablen C (engl. *clocks*). Der momentane Wert einer Zeitvariablen $c \in C$ ist $\nu(c) \in \mathbb{T}$. Entsprechend beschreibt die Funktion ν die momentanen Werte aller Zeitvariablen. $\mathbb{T}$ bezeichnet eine Menge der Zeitpunkte. Die betrachteten Zeitpunkte bei zeitbehafteten Automaten sind typischerweise $\mathbb{T} = \mathbb{R}_{\geq 0}$.

Die Werte der Zeitvariablen können beschränkt werden. Die entsprechenden Beschränkungen werden mit $\beta \in \mathcal{B}(C)$ bezeichnet und sind konjunktive Formeln über atomare Beschränkungen der Form $c \sim \gamma$ mit $c \in C$, $\gamma \in \mathbb{Q}$ und den Relationen $\sim \in \{\leq, <, =, >, \geq\}$. Die Beschränkungen $\mathcal{B}(C)$ werden verwendet, um die Zeitpunkte für Zustandsübergänge bzw. die Verweildauer in Zuständen einzuschränken. Hierbei handelt es sich um lokale Beschränkungen.

Definition 2.2.15 (Zeitbehafteter Automat). *Ein* zeitbehafteter Automat *ist ein* 6-*Tupel* $(I, S, s_0, C, \text{inv}, f)$. *Darin bezeichnet*

- I *das* Eingabealphabet,
- S *eine endliche nichtleere Menge von* Zuständen,
- $s_0 \in S$ *den* Anfangszustand,
- C *eine endliche Menge an* Zeitvariablen,
- $\text{inv} : S \to \mathcal{B}(C)$ *eine Funktion, die jedem Zustand eine* Invariante *zuweist und*
- $f \subseteq (S \times \mathcal{B}(C) \times I \times 2^C \times S)$ *die* Übergangsrelation.

Ein Zustandsübergang $(s, \beta, i, \mathcal{R}, s') \in f$ ist der Form $s \xrightarrow{\beta, i, \mathcal{R}} s'$ und beschreibt einen Übergang von Zustand s in den Zustand s', wenn die Beschränkungen in β erfüllt sind und Eingabe i eintritt. In diesem Fall werden die Zeitvariablen $c \in \mathcal{R}$ auf null gesetzt.

Wie im Fall der zeitbehafteten Petri-Netze, ist der Zustand eines zeitbehafteten Automaten jetzt nicht mehr allein durch den momentanen Zustand $s \in S$ des Automaten definiert, sondern es müssen zusätzlich die Werte $\nu(c)$ der Zeitvariablen $c \in C$ berücksichtigt werden. Die Ausführungssemantik eines zeitbehafteten Automaten ergibt sich aus zwei Arten von Zustandsänderungen:

1. Zustandsänderung aufgrund eines Zeitfortschritts $\delta \in \mathbb{T}$:

$$(s, \nu) \xrightarrow{\delta \in \mathbb{T}} (s, \nu') \text{ mit } \forall c \in C : \nu'(c) := \nu(c) + \delta$$

   Dabei muss für alle $0 \leq \delta' \leq \delta$ und alle $c \in C$ gelten, dass $\nu(c) + \delta'$ die Zeitinvariante $\mathrm{inv}(s)$ erfüllt, geschrieben als $\nu(c) + \delta' \models \mathrm{inv}(s)$.
2. Zustandsänderung aufgrund einer Eingabe $i \in I$, die den Zustandsübergang $s \xrightarrow{\beta, i, \mathcal{R}} s'$ anstößt:

$$(s, \nu) \xrightarrow{i \in I} (s', \nu') \text{ mit } \nu'(c) := \begin{cases} 0 & \text{für } c \in \mathcal{R} \\ \nu(c) & \text{sonst} \end{cases}$$

   Dabei muss gelten, dass ν die Zeitbeschränkungen β erfüllt, geschrieben als $\nu \models \beta$, und dass $\nu' \models \mathrm{inv}(s')$.

Die Zustandsänderungen können in beliebiger Reihenfolge auftreten.

Beispiel 2.2.5. Gegeben ist der zeitbehaftete Automat in Abb. 2.5 mit zwei Zeitvariablen c_0 und c_1. An den Zustandsübergängen sind Eingaben, die Zeitbeschränkungen und die (evtl. leere) Menge an zurückzusetzenden Zeitvariablen annotiert. Die Abwesenheit von Zeitbeschränkungen ist durch $-$ gekennzeichnet. Weist die Funktion inv einem Zustand eine Invariante zu, so ist diese in der unteren Hälfte des Zustands dargestellt. Die Zeitvariable c_0 wird z. B. jedes Mal zurückgesetzt, wenn der Zustandsübergang von s_0 nach s_1 durch die Eingabe i_0 ausgelöst wird. Die Invariante $c_0 < 1$ in den Zuständen s_1 und s_2 verlangt, dass die Eingaben i_1 und i_2 innerhalb von einer Zeiteinheit nach der Eingabe i_0 auftreten. Beim Zustandsübergang von s_1 nach s_2 wird die Zeitvariable c_1 zurückgesetzt. Die Zeitbeschränkung $c_1 > 2$ an dem Übergang von Zustand s_3 nach Zustand s_0 sorgt dafür, dass die Eingaben i_1 und i_3 mindestens zwei Zeiteinheiten auseinanderliegen.

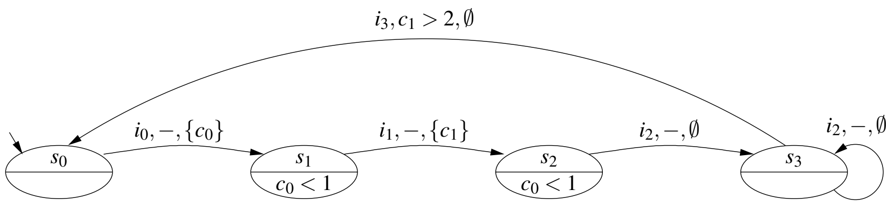


Abb. 2.5. Zeitbehafteter Automat mit zwei Zeitvariablen [7]

Mit $\mathbb{T} := \mathbb{R}_{\geq 0}$ ist eine beispielhafte Ausführungssequenz:

$$\begin{aligned} &(s_0; 0; 0) \xrightarrow{1,2} (s_0; 1,2; 1,2) \xrightarrow{i_0} (s_1; 0; 1,2) \xrightarrow{0,7} (s_1; 0,7; 1,9) \\ &\xrightarrow{i_1} (s_2; 0,7; 0) \xrightarrow{0,1} (s_2; 0,8; 0,1) \xrightarrow{0,1} (s_2; 0,9; 0,2) \end{aligned}$$

Man beachte die Additivität des Zeitfortschritts, d. h.

$$(s,v) \xrightarrow{\delta_1} (s,v') \wedge (s,v') \xrightarrow{\delta_2} (s,v'') \Longrightarrow (s,v) \xrightarrow{\delta_1+\delta_2} (s,v'')$$

2.2.3 Datenflussgraphen

Datenflussgraphen sind gerichtete Graphen. Die Knotenmenge stellt üblicherweise eine Menge von *Aktivitäten* oder *Aktoren* dar, die Daten verarbeiten. Die Kanten repräsentieren den gerichteten *Datenfluss*. In einem Datenflussgraphen werden die Berechnungen allein durch die Verfügbarkeit von Daten gesteuert.

Beispiel 2.2.6. Abbildung 2.6 zeigt den Datenflussgraphen zur Lösung einer Differentialgleichung nach der Euler-Methode. Die Differentialgleichung ist in der Form $y'' + 3xy' + 3y = 0$ gegeben und soll im Intervall $[x_0, a]$ mit der Schrittweite dx und Anfangswerten $y(x_0) = y$ und $y'(x_0) = u$ numerisch gelöst werden. Dabei wird für ein Anfangswertproblem der Form $y' = f(x, y(x))$ mit $y(x_0) = y_0$ die Näherungsformel $y(x+dx) \approx y(x) + dx\, f(x, y(x))$ eingesetzt, um, beginnend mit x_0, die Werte von $y(x_0 + i\,dx)$, $i = 1, 2, \ldots$, sukzessiv zu bestimmen. Im Beispiel lautet die Differentialgleichung $y'' = f(x, y(x), y'(x)) = -3xy'(x) - 3y(x)$. Die zweifache Anwendung der Euler-Methode liefert hier die Lösung: $y'(x+dx) = y'(x) + dx\,(-3xy'(x) - 3y(x))$ und $y(x+dx) = y(x) + dx\, y'(x)$. Neben dem eigentlichen Datenflussgraphen sind hier auch Ein- und Ausgangsdaten (gestrichelt) dargestellt.

Die Kommunikationsregel eines Datenflussgraphen entspricht der Kommunikationsregel desjenigen Petri-Netzes, das man erhält, wenn man die Knoten als Transitionen modelliert und die Kanten als mit den Transitionen verbundene Stellen auffasst. Eingangskanten von Knoten ohne Vorgänger werden durch Stellen ohne Vorbereich ersetzt, Ausgangskanten von Knoten ohne Nachfolger werden durch Stellen ohne Nachbereich ersetzt.

Markierte Graphen

Markierte Graphen [113] sind Datenflussgraphen mit speziellen Eigenschaften. Ein Beispiel eines markierten Graphen ist in Abb. 2.7a) dargestellt. Ein markierter Graph lässt sich ebenfalls als Petri-Netz beschreiben, in dem jede Stelle genau eine Transition im Vorbereich und genau eine Transition im Nachbereich hat.

Definition 2.2.16 (Markierter Graph). *Ein* markierter Graph *entspricht einem Petri-Netz $G(P, T, F, K, W, M_0)$ mit den Eigenschaften:*

1. $\forall p \in P : |\bullet p| = |p \bullet| = 1$
2. $\forall p \in P :\ K(p) = \infty$
3. $\forall f \in F :\ W(f) = 1$

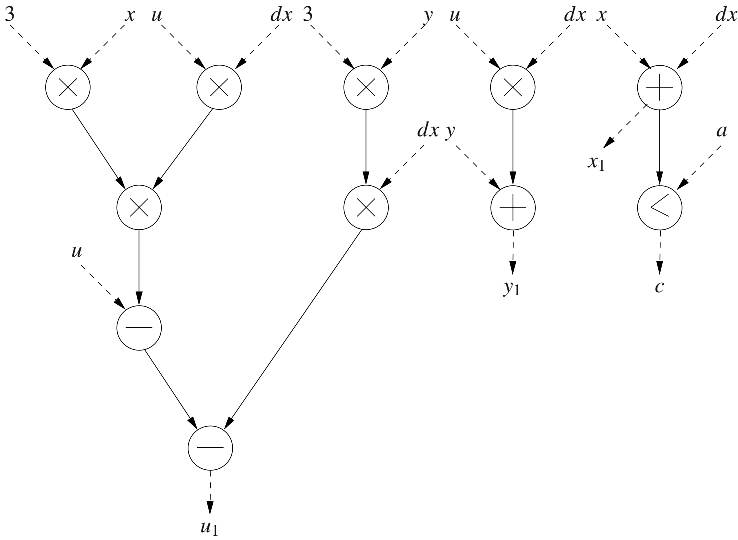


Abb. 2.6. Datenflussgraph zur Lösung einer Differentialgleichung nach der Euler-Methode

Diese Definition ist nur korrekt unter der Annahme, dass die Konsumption und Produktion von Daten aus Stellen in der Reihenfolge der Ankunft erfolgt. Obwohl das Modell des Petri-Netzes keine Information über Ankunftsreihenfolge oder Ankunftszeitpunkte der Marken besitzt, wird bei allen im Folgenden vorgestellten Datenflussgraphen implizit von dieser Annahme Gebrauch gemacht.

Sieht man von der Markierung ab, so sind markierte Graphen dual zu Zustandsautomaten. Dies wird z. B. an dem in Abb. 2.7 dargestellten Beispiel deutlich. Knoten im markierten Graphen entsprechen Transitionen im Petri-Netz und Kanten im markierten Graphen entsprechen Stellen im Petri-Netz.

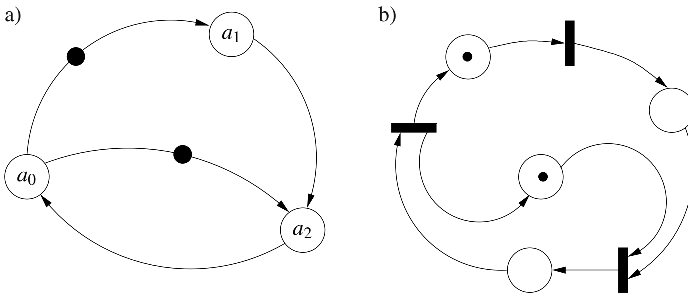


Abb. 2.7. a) Modell eines markierten Graphen und b) Darstellung als Petri-Netz

Die übliche Interpretation dieser Klasse von Netzen ist die Assoziation von Aktivitäten (z. B. Prozesse, Operationen) mit Transitionen und die Assoziation von Stellen mit einem Datenpuffer mit der Semantik eines FIFO-Speichers (engl. *First In, First Out*). Markierte Graphen besitzen eine geringere Ausdruckskraft als allgemeine Petri-Netze: So können Verzweigungen nicht dargestellt werden. Sie sind somit konfliktfrei und können damit nur deterministisches Verhalten beschreiben.

Synchrone Datenflussgraphen (SDF)

Interessant für Anwendungen im Bereich der Signalverarbeitung sind Systeme, deren Teilsysteme bestimmten Teilaufgaben gewidmet sind und diese dabei mit mehreren unterschiedlichen Datenraten arbeiten. Zur Modellierung von Anwendungen der digitalen Signalverarbeitung definieren Lee und Messerschmitt [296] das Modell des *synchronen Datenflussgraphen*, im Folgenden SDF-Modell (engl. *Synchonous Data Flow*) genannt. Ein SDF-Graph ist ein um die Eigenschaft, dass die Gewichte von eins verschieden sein können, erweiterter markierter Graph. SDF-Graphen sind bzgl. des Kommunikationsmodells auch äquivalent zu einer Teilklasse von Petri-Netzen:

Definition 2.2.17 (SDF-Graph). *Ein* SDF-Graph *(synchroner Datenflussgraph) entspricht einem Petri-Netz $G(P,T,F,K,W,M_0)$ mit den Eigenschaften:*

1. $\forall p \in P : |\bullet p| = |p \bullet| = 1$
2. $\forall p \in P : K(p) = \infty$
3. *Jeder Eingangsstelle p einer Transition t ist eine Zahl $cons(p,t) := W(p,t) \in \mathbb{N}$ zugewiesen*
4. *Jeder Ausgangsstelle p einer Transition ist eine Zahl $prod(t,p) := W(t,p) \in \mathbb{N}$ zugewiesen*

Die Zahlen *cons* und *prod* werden als *Konsumptions-* bzw. *Produktionsrate* bezeichnet und werden in SDF-Graphen mit den Kanten assoziiert. Abbildung 2.8 zeigt ein Beispiel eines SDF-Graphen. Die Gewichte *prod* werden als Zahlen am Anfangsknoten einer Kante, die Gewichte *cons* als Zahlen am Endknoten einer Kante dargestellt.

Im Folgenden werden Erweiterungen des SDF-Modells betrachtet.

Zyklostatischer Datenfluss

Eine Erweiterung des SDF-Modells von Engels und Bilsen [153] erlaubt, dass jeder Knoten v_i im Graphen eine Menge von P_i Konsumptionsraten $cons(v_j,v_i)_k$ und Produktionsraten $prod(v_i,v_j)_k$ mit $k = 0,\ldots,P_i - 1$ besitzen kann. Die Zahlen gelten für genau eine Feuerung und sind alle P_i Iterationen von Aktorfeuerungen zyklisch abwechselnd gültig. Durch diese Erweiterung können z. B. bestimmte zustandsabhängige Aktoren (siehe z. B. in Abb. 2.9) dargestellt werden.

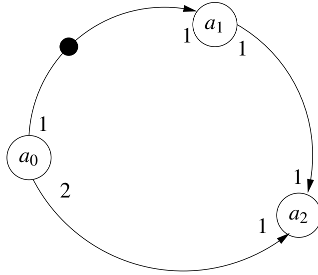
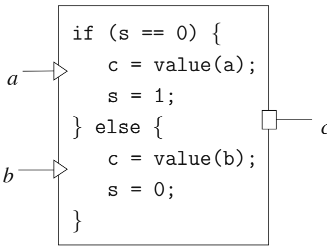


Abb. 2.8. Ein SDF-Graph. Marken sind durch schwarze Punkte dargestellt.

Beispiel 2.2.7. Gegeben ist die Struktur eines Multiplexers gemäß Abb. 2.9a), der die Eingänge a und b besitzt und die Funktion erfüllen soll, abwechselnd Daten von Eingang a bzw. Eingang b an den Ausgang c zu kopieren. In der funktionalen Beschreibung erreicht man dies beispielsweise durch ein Zustandsbit s, das sich zyklisch von 0 auf 1 ändert, um den nächsten zu selektierenden Eingang zu wählen.



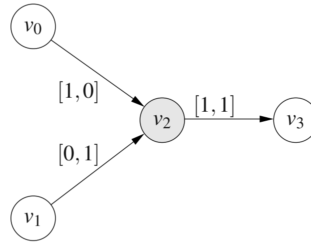


Abb. 2.9. Darstellung eines Multiplexers mit zwei Eingängen a) als Pseudocode und b) als zyklostatisches Datenflussmodell

Abbildung 2.9b) zeigt eine Darstellung mit einem zyklostatischen Aktor v_2 mit Periodizität $P_2 = 2$. Der Knoten besitzt damit zwei Zustände. Im Zustand $s = 0$ ist der Knoten feuerbereit, wenn an dem Eingang a mindestens ein Datum anliegt. Beim Feuern wird nun nur vom Eingang a ein Datum konsumiert und an den Ausgang kopiert. Dann wechselt der Knoten in den Zustand $s = 1$. Nun ist der Knoten feuerbereit, wenn mindestens ein Datum an Eingang b liegt. Diese beiden Zustände des Aktors wiederholen sich zyklisch. Die Konsumptions- und Produktionsraten für Aktor v_2 sind in folgender Tabelle zusammengefasst:

k	$cons(v_0, v_2)$	$cons(v_1, v_2)$	$prod(v_2, v_3)$
0	1	0	1
1	0	1	1

Engels et al. zeigten [153], wie sich die für SDF-Graphen interessanten Eigenschaften, insbesondere Existenz von Verklemmungen, periodische Planbarkeit und Beschränktheit, auch auf zyklostatischen Datenflussgraphen (engl. *Cyclo-Static Data Flow, CSDF*) ermitteln lassen.

Dynamische Datenflussmodelle

Die wesentlichen Einschränkungen bisher vorgestellter Datenflussmodelle sind, dass sich keine allgemeinen Kontrollstrukturen, beispielsweise datenabhängige Schleifen oder IF-THEN-ELSE-Konstrukte, darstellen lassen. Es können hier nicht alle bekannten, wichtigen Datenflussmodelle behandelt werden. Statt dessen wird gezeigt, dass bereits geringfügige Modellerweiterungen der bisher vorgestellten Modelle zu einer Mächtigkeitserweiterung auf *Turing-Äquivalenz* führen. Das hat zur Folge, dass die Analyse von Eigenschaften wie Beschränktheit unentscheidbar wird.

Betrachtet wird die folgende Erweiterung des SDF-Modells von Buck [70]: Buck nennt alle Knoten, die eine konstante (bekannte) Anzahl von Daten konsumieren und produzieren, *reguläre Aktoren*, und Datenflussgraphen, die nur reguläre Aktoren besitzen, *reguläre Datenflussgraphen*. Dazu gehören markierte Graphen und SDF-Graphen. Bei *dynamischen Aktoren* muss nun die Anzahl von Daten, die entlang einer Kante konsumiert bzw. produziert werden, nicht mehr konstant sein. In der Regel hängen diese Zahlen von den Werten der Eingangsdaten ab. Besitzt ein Modell solche Aktoren, so handelt es sich um einen *dynamischen Datenflussgraphen*.

Buck [70] erweiterte das SDF-Modell um zwei dynamische Aktoren SWITCH und SELECT, und nennt die Klasse von Datenflussgraphen, die aus einer beliebigen Kombination von SDF-Knoten und SWITCH- und SELECT-Knoten bestehen, BDF (engl. *Boolean-controlled Data Flow*). Gegenüber regulären Aktoren kann bei den SWITCH- und SELECT-Aktoren die Anzahl der konsumierten bzw. produzierten Daten eine zweiwertige Funktion des Wertes, einer sog. *Steuerungsmarke* (engl. *control token*), sein. Das Verhalten wird durch einen Steuereingang bestimmt, der in jeder Ausführung genau ein Datum, die Steuerungsmarke, konsumiert. Buck zeigte, dass das um diese Aktoren erweiterte SDF-Modell Turing-äquivalent (berechnungsuniversell) ist. Jedoch lässt sich zeigen, dass sich gewisse Teilgraphen eines BDF-Graphen durch *Clustering* zusammenfassen lassen und dadurch zumindest auf Teilgraphen statische Analyseverfahren wie auf SDF-Graphen angewendet werden können.

Alle hier betrachteten Datenflussmodelle besitzen die Eigenschaft, *deterministisch* zu sein, d. h., dass bei beliebigen Ausführungsreihenfolgen der Knoten die Sequenz der produzierten Daten jeweils die gleiche ist. Dies ist darin begründet, dass alle hier vorgestellten Modelle Spezialfälle sog. *Kahn-Prozessnetzwerke* [250] sind, für die Kahn Determinismus bewiesen hat. In einem Kahn-Prozessnetzwerk (KPN) sind Lesezugriffe auf Kommunikationskanäle, die ebenfalls eine FIFO-Semantik besitzen, blockierend. Schreibzugriffe sind aufgrund der unbegrenzt großen Kanäle immer nichtblockierend. Diesen Sachverhalt stellt Lee in [295] dar, wo er die Zusammenhänge von Datenflussgraphen zu Prozesskalkülen und Datenflusssprachen

(engl. auch *stream oriented languages*) erläutert. Zu letzteren zählen beispielsweise die Sprachen ESTEREL [44], LUSTRE [212], Lucid [19] und SIGNAL [38].

2.2.4 Heterogene Modelle

Die bisher vorgestellten Modelle haben den Vorteil, dass sie eine bestimmte Eigenschaft bzw. Sichtweise eines Systems gut beschreiben. Während solche spezifischen Modelle zwar leichter zu analysieren sind, leiden sie unter ihrer eingeschränkten *Ausdruckskraft*. Um ein komplexes System zu beschreiben, benutzt man also im Allgemeinen *heterogene Modelle*.

Die Natur der Anwendungsgebiete von Hardware/Software-Systemen fordert, dass sowohl *Kontrollfluss* als auch *Datenfluss* in einem Modell dargestellt werden können. Die bisher vorgestellten Modelle eignen sich entweder zur Modellierung von datenflussdominanten oder zur Modellierung von kontrollflussdominanten Systemen. Heterogene Graphenmodelle, die beides kombiniert darstellen können, sind beispielsweise sog. *Kontroll-Datenflussgraphen* (engl. *Control/Data Flow Graphs, CDFGs*).

Da es unterschiedliche Möglichkeiten der Definition und zahlreiche Variationen von CDFGs gibt, werden an dieser Stelle nur die wesentlichen Prinzipien nichtformal vorgestellt.

Kontroll-Datenflussgraphen (CDFGs)

Offensichtlich kann ein Datenflussgraph keine *Kontrollstrukturen* wie z. B. *Verzweigungen* und *Iterationen* (z. B. Schleifenkonstrukte) modellieren. Dazu dienen sog. *Kontrollflussgraphen* (engl. *Control Flow Graphs, CFGs*). Ein Kontrollflussgraph ist ein gerichteter Graph, in dem den Knoten Berechnungen entsprechen und Kanten Nachfolgerrelationen in einem (sequentiellen) Kontrollfluss ausdrücken, nicht aber Datenabhängigkeiten. Besitzt ein Knoten mehrere Nachfolger, so handelt es sich um einen *Verzweigungsknoten*. Der von einem Verzweigungsknoten ausgehende Kontrollfluss ist *alternativ*, d. h. es wird exakt ein Nachfolgerzweig durchlaufen. Die Auswahl eines Zweigs ist abhängig von Booleschen Ausdrücken, die man üblicherweise an die Ausgangskanten eines Verzweigungsknotens schreibt.

Beispiel 2.2.8. Abbildung 2.10a) zeigt einen Ausschnitt eines C-Programms. Die Aufgabe des Programms sei an dieser Stelle vernachlässigt. Der zugehörige Kontrollflussgraph ist in Abb. 2.10b) dargestellt: Üblicherweise assoziiert man mit jedem Knoten eine C-Anweisung (durch Zeilennummern gekennzeichnet). Bei Knoten, die mehr als eine Ausgangskante besitzen (Verzweigungsknoten), ist der Kontrollfluss *alternativ*. Entsprechende Verzweigungsbedingungen werden an die Ausgangskanten geschrieben. Ein Schleifenkonstrukt lässt sich als eine Verzweigung mit Test auf die Abbruchbedingung der Iteration modellieren.

Ein Kontrollflussgraph kann offensichtlich nicht die Datenabhängigkeiten der Berechnungen darstellen. Das Modell von *Kontroll-Datenflussgraphen* ist ein he-

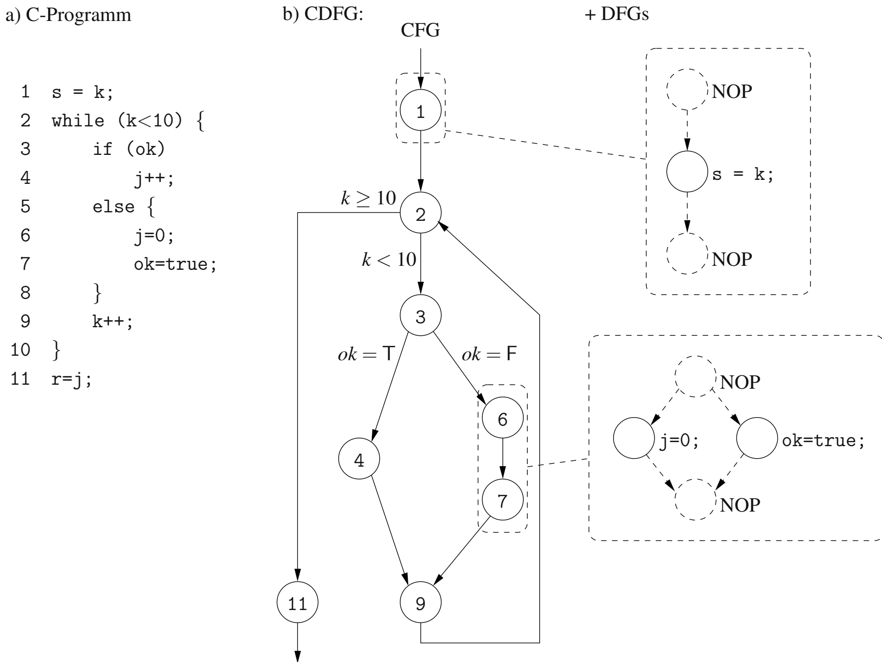


Abb. 2.10. a) Ausschnitt eines C-Programms und b) CDFG. Der CFG bildet mit den Datenflussgraphen (DFGs) einen CDFG.

terogenes Modell, das eine Aufteilung des Verhaltensmodells in kontrollflussdominante und datenflussdominante Anteile vornimmt und damit beide Aspekte in einem Modell vereinigen kann.

Beispiel 2.2.9. Gegeben ist der Ausschnitt aus einem C-Programm in Abb. 2.10a). Nun kann man z. B. mit jeder Anweisung einen Datenflussgraphen assoziieren (siehe Abb. 2.10b), der die Berechnung der entsprechenden Anweisung modelliert. Die gestrichelten Knoten (NOP-Operationen) modellieren einen einheitlichen Eintritts- bzw. Austrittspunkt eines Datenflussgraphen (DFGs).

Schließlich zeigt Abb. 2.10b) einen weiteren relevanten Aspekt der Definition eines CDFG-Modells: In dem Modell in Abb. 2.10a) besteht kein Grund, die beiden Anweisungen in Zeile 6 und 7 sequentiell auszuführen. Bei der Analyse wird daher ein DFG pro *Grundblock* (engl. *basic block*) erzeugt. Ein Grundblock ist eine Folge fortlaufender Anweisungen, in die der Kontrollfluss am Anfang eintritt und die er am Ende verlässt, ohne dass er, außer am Ende, verzweigt. Im CFG gibt es dann pro Grundblock nur einen Knoten. Die Datenflussgraphen bestimmt man durch *Datenflussanalyse* (siehe z. B. [4, 343]).

FunState

FunState [433, 417] ist ein Akronym für engl. *Functions driven by State machines*, also Funktionen, die durch Zustandsmaschinen (endliche Automaten) gesteuert bzw. aktiviert werden. Ein nichthierarchisches FunState-Modell besteht dabei aus einem *transformativen* und einem *reaktiven* Teil. Der transformative Teil wird durch ein Netzwerk ähnlich einem Petri-Netz modelliert, der reaktive Teil durch einen endlichen Automaten.

Definition 2.2.18 (FunState-Modell [433]). *Ein* FunState-Modell *besteht aus einem Netzwerk N und einem endlichen Automaten M. Das Netzwerk $N = (F,S,E)$ besteht aus einer Menge an Speicherelementen $s \in S$, einer Menge von Funktionen $f \in F$ und einer Menge an gerichteten Kanten $e \in E \subseteq (F \times S) \cup (S \times F)$.*

Im Gegensatz zu Transitionen in Petri-Netzen erfolgt die Aktivierung der Funktionen $f \in F$ im FunState-Modell nicht eigenständig durch das Vorhandensein von Marken auf den Speicherplätzen, sondern durch Zustandsübergänge im endlichen Automaten M. Hierbei können die Bedingungen an den Zustandsübergängen von M die Anzahl der Marken in einem Speicherelement oder aber den mit einer Marke assoziierten Wert berücksichtigen. Die Speicherelemente in einem FunState-Modell können FIFO-Semantik besitzen bzw. Register sein. Im Folgenden wird davon ausgegangen, dass alle Speicherelemente FIFO-Semantik besitzen.

Beispiel 2.2.10. Ein Beispiel eines FunState-Modells ist in Abb. 2.11 zu sehen. Der obere Teil stellt das Netzwerk N dar, welches aus drei Funktionen und zwei Speicherelementen besteht. Die Anzahl der Marken in einem Speicherelement s wird mit $s\# \in \mathbb{Z}_{\geq 0}$ bezeichnet. Die Anzahl der anfänglichen Marken wird mit $s\#_0$ bezeichnet. In Abb. 2.11 ist die Anzahl der anfänglichen Marken $s_0\#_0 = s_1\#_0 = 1$. Die Menge der Funktionen F ist gegeben durch $F = \{f_0, f_1, f_2\}$. Der untere Teil zeigt den Automaten M, welcher zwei Zustände besitzt. Die Zustandsübergänge sind mit Bedingungen und Aktionen beschriftet. Zum Beispiel kann der Übergang von z_1 nach z_0 erfolgen, sofern das Speicherelement s_1 mindestens eine Marke enthält. In diesem Fall wird die Funktion f_0 als Aktion ausgeführt.

Die Funktionen $f \in F$ in einem FunState-Modell sind eindeutig benannt und verarbeiten bei ihrer Ausführung Marken. Allen Ein- und Ausgängen von Funktionen sind Werte $c_i \in \mathbb{Z}_{\geq 0}$ bzw. $p_i \in \mathbb{Z}_{\geq 0}$ zugeordnet, die der Anzahl der konsumierten bzw. produzierten Marken nach Ausführung der Funktion entsprechen.

Die Ausführungssemantik des FunState-Modells ist in fünf Phasen aufgeteilt:

1. Initialisierung: Der aktuelle Zustand des endlichen Automaten M wird auf den Anfangszustand gesetzt und die Speicherelemente $s \in S$ im Netzwerk werden mit den anfänglichen Marken vorbelegt.
2. Prädikatenevaluierung: Alle Prädikate von aus dem aktuellen Zustand ausgehenden Transitionen des endlichen Automaten M werden evaluiert.
3. Fortschrittsprüfung: Ist kein Prädikat erfüllt, wird die Ausführung gestoppt.

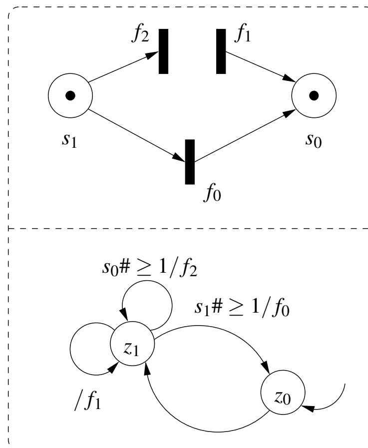


Abb. 2.11. FunState-Modell

4. Transitionsausführung: Ein Zustandsübergang aus dem aktuellen Zustand, dessen Prädikat erfüllt ist, wird nichtdeterministisch ausgewählt, und die annotierte Funktion aktiviert.
5. Funktionsfeuerung: Die aktivierte Funktion f wird ausgeführt. Hierbei konsumiert bzw. produziert f entsprechend der mit den Ein- und Ausgängen assoziierten Werte c_i und p_i Marken von/auf den verbundenen Speicherelementen. Die Ausführung wird mit Schritt 2. fortgesetzt.

Die Besonderheit des FunState-Modells liegt darin begründet, dass es allein durch Einschränkung des endlichen Automaten viele der zuvor eingeführten Modelle abbilden kann.

2.3 Ausführbare Verhaltensmodelle

Ausführbare Spezifikationen basieren auf einem ausführbaren Verhaltensmodell. Diese können mit Hilfe von Programmiersprachen erstellt werden. Programmiersprachen bieten den Vorteil, dass diese häufig mehrere Aspekte, wie z. B. datenfluss- und kontrollflussdominante Modelle, gleichzeitig darstellen können. Grundsätzlich unterscheidet man zwei Arten von Programmiersprachen: *imperative* und *deklarative*. Imperative Programmiersprachen, wie beispielsweise C und Pascal, besitzen ein Ausführungsmodell, in dem Anweisungen in der Reihenfolge ausgeführt werden, in der sie im Programmtext erscheinen (*control-driven*). LISP und PROLOG hingegen sind deklarative Sprachen. Für diese Sprachen ist charakteristisch, dass sie keine explizite Ausführungsreihenfolge spezifizieren. Statt dessen wird das Ziel der Berechnung durch eine Menge von Funktionen oder logische Regeln ausgedrückt (*demand-driven*).

Basierend auf C/C++ sind in den letzten Jahren sog. *Systembeschreibungssprachen* entwickelt worden. Diese erweitern imperative Programmiersprachen um die Konzepte von Nebenläufigkeit, Kommunikation und Synchronisation. Im Folgenden wird *SystemC* als wichtiger Vertreter von Systembeschreibungssprachen zur Erstellung ausführbarer Spezifikationen für Hardware/Software-Systeme vorgestellt. Im Anschluss wird eine Erweiterung von SystemC mit dem Namen *SysteMoC* präsentiert. SysteMoC ermöglicht die Modellierung von FunState-Modellen in SystemC und bietet vielseitige Synthese- und Analysemöglichkeiten.

2.3.1 SystemC

SystemC ist eine Systembeschreibungssprache und ein Defacto-Standard zur Modellierung von Hardware/Software-Systemen [207]. SystemC unterstützt dabei verschiedene Abstraktionsebenen in der Modellierung. Technisch gesehen ist SystemC eine C++-Klassenbibliothek und erweitert C++ um die Aspekte

- Nebenläufigkeit, Kommunikation und Synchronisation,
- ereignisgesteuerte Simulation,
- Zeitannotationen sowie
- spezielle Datentypen für die Hardware-Modellierung.

Der SystemC-Sprachumfang ist im Dokument IEEE Std 1666 standardisiert [236]. Eine Referenzimplementierung von SystemC wird von der Open SystemC Initiative (OSCI) unter www.systemc.org angeboten.

Modellierung mit SystemC

Im Folgenden werden einige wichtige Modellierungskonzepte vorgestellt.

SystemC-Module

Ähnlich wie in Hardware-Beschreibungssprachen stellt SystemC Sprachkonstrukte zur Deklaration von Modulen zur Verfügung. Ein *SystemC-Modul* kapselt somit die Verhaltensbeschreibung in einer Hardware- oder einer Software-Komponente. Jedes SystemC-Modul muss von der Basisklasse `sc_module` abgeleitet werden. SystemC-Module sollen Daten lediglich über Kanäle austauschen. Der Zugriff auf einen angeschlossenen Kanal erfolgt über sog. *Ports*. Das Verhalten des Moduls wird entweder durch Komposition aus anderen Modulen oder durch nebenläufige Prozesse beschrieben.

Beispiel 2.3.1. Das folgende SystemC-Modul implementiert eine einfache Komponente, welche verifiziert werden soll (engl. *System Under Verification, SUV*). Es liest lediglich Daten von einem Eingangsport und kopiert diese unverändert auf den Ausgangsport.

```
1    class SUV: public sc_module {
2      public:
3        sc_fifo_in<double>  in;
4        sc_fifo_out<double> out;
5        double data;
6
7      private:
8        void pass() {
9          while(true) {
10           data = in.read();
11           std::cout << "suv:  " << data << std::endl;
12           out.write(data);
13         }
14       }
15
16     public:
17       SC_HAS_PROCESS(SUV);
18
19       SUV(sc_module_name name) : sc_module(name) {
20         SC_THREAD(pass);
21      }
22   };
```

In Zeile 1 wird das SystemC-Modul `SUV` deklariert, welches von der Basisklasse `sc_module` abgeleitet ist. In den Zeilen 3-5 erfolgt die Deklaration von Ports und Variablen. Die Eingangs- und Ausgangsports `in` und `out` werden später bei der Instantiierung des Moduls mit FIFO-Kanälen verbunden. Das eigentliche Verhalten der Komponente ist in Zeile 8-14 in der Methode `pass` definiert. Hier wird ein Datum vom Eingangsport `in` gelesen, auf der Standardausgabe für Debugzwecke ausgegeben und anschließend unverändert auf den Ausgangsport `out` geschrieben.

Dass es sich bei dem Modul `SUV` nicht um eine rein strukturelle Beschreibung handelt, sondern das Verhalten in Form eines Prozesses implementiert ist, wird durch das Makro in Zeile 17 angezeigt. Um welchen Prozess es sich dabei handelt wird im Konstruktor in Zeile 20 durch das Makro `SC_THREAD` bezeichnet. Dieses zeigt auch an, dass es sich um einen Thread handelt (siehe unten). Bei der Definition des Konstruktors wird erwartet, dass der Konstruktor der Basisklasse mit einem Argument, das den Namen des Moduls als Zeichenkette repräsentiert, aufgerufen wird.

Ports und Interfaces

Wie in Beispiel 2.3.1 zu sehen ist, werden *Ports* als Objekte in einem `sc_module` definiert. Durch die Verwendung des C++-Schlüsselworts `public` können Ports von anderen Objekten außerhalb der Klasse verwendet werden. In dem Beispiel wurden dabei vordefinierte Porttypen `sc_fifo_in` und `sc_fifo_out` bei der Instantiierung des Moduls verwendet. Diese können mit *SystemC-FIFO-Kanälen* verbunden werden. Neben diesen Porttypen stellt SystemC weitere vordefinierte Porttypen

zur Verfügung. Im Bereich der Hardware-Modellierung besitzen insbesondere Ports für *Signale* eine besondere Bedeutung. Dabei wird zwischen Eingangs-, Ausgangs- und Ein-/Ausgangsports unterschieden. Die entsprechenden Klassen lauten `sc_in`, `sc_out` und `sc_inout`. Alle Porttypen sind als Template-Klassen implementiert, wobei der Template-Parameter den zu transportierenden Datentyp festlegt. Dieser kann ein beliebiger Standard-C++-, SystemC- oder benutzerdefinierter Datentyp sein. In dem Beispiel 2.3.1 wurde als Datentyp `double` verwendet.

Neben der Verwendung vordefinierter Porttypen ist es ebenfalls möglich, eigene Porttypen zu definieren. Dies erfolgt durch Ableitung von der Basisklasse `sc_port`. Diese Klasse ist wiederum eine Template-Klasse, die als Template-Parameter ein Interface vom Typ `sc_interface` erhält. Dieses Interface deklariert lediglich die Methoden zum Kanalzugriff (z. B. `read()` und `write()`). Die eigentliche Implementierung dieser Methoden erfolgt im *Kanal*.

Beispiel 2.3.2. Die folgende Port-Definition ist äquivalent zu `sc_in`:

```
sc_port < sc_signal_in_if < bool > >
```

Hierbei ist `sc_signal_in_if` eine vordefiniertes Interface, das von `sc_interface` abgeleitet ist.

Prozesse

SystemC unterscheidet im Wesentlichen zwei Typen von *Prozessen*: *Threads* und *Methoden*. Die Gemeinsamkeiten dieser beiden Prozesstypen sind, dass beide durch sequentiell abzuarbeitende Anweisungen beschrieben werden und dass sie nebenläufig ausgeführt werden können. Prozesse werden dabei nicht unterbrochen, d. h. sie werden so lange ausgeführt, bis sie die Kontrolle an den Simulator zurückgeben. Die Aktivierung von Prozessen erfolgt in SystemC analog zu VHDL über Sensitivitätslisten. Sowohl Threads als auch Methoden können sowohl statische als auch dynamische Sensitivitätslisten besitzen. Der wesentliche Unterschied zwischen den beiden Prozesstypen ist, dass Methoden vom Typ `SC_METHOD` nicht unterbrochen werden dürfen, d. h. sie laufen immer von Beginn bis zum Ende durch. An dieser Stelle wird die Methode beendet und eventuell entsprechend der Sensitivitätsliste in Zukunft wieder aktiviert.

Im Gegensatz dazu werden SystemC-Threads vom Typ `SC_THREAD` in der Regel nicht beendet. Im Beispiel 2.3.1 wird deshalb eine `while(true)`-Schleife verwendet. Damit andere Prozesse nicht verhungern, muss jeder Prozess irgendwann die Kontrolle an den Simulator zurückgeben. Hierzu blockiert sich ein Thread an Ereignissen, wie z. B. einem Taktsignal oder wie im Beispiel 2.3.1 an den Lesezugriffen an einem FIFO-Kanal. Ist dieser FIFO-Kanal leer, wird der Thread solange blockiert, bis ein neues Datum in den Kanal geschrieben wurde.

SystemC-Methoden werden oft zur Modellierung von Hardware-Komponenten verwenden. Bei der Modellierung auf Systemebene, auf der Hardware- und Software-Komponenten interagieren, werden, obwohl ihres Geschwindigkeitsnachteils, häufig SystemC-Threads bevorzugt.

Kanäle

Kanäle werden für die Kommunikation zwischen Modulen verwendet. Sie können dabei so primitiv wie ein Signal oder so komplex wie ein ganzer Bus oder ein sog. engl. *Network on Chip* (*NoC*) sein. SystemC stellt vordefinierte Kanaltypen für Signale (`sc_signal`), FIFO-Kanäle (`sc_fifo`), Semaphore (`sc_semaphore`) etc. zur Verfügung. Daneben erlaubt SystemC die Definition eigener Kanaltypen. Hierfür muss der Kanal von `sc_interface` abgeleitete Lese- und Schreib-Interfaces implementieren.

Die Möglichkeit zur Definition eigener Kanäle erlaubt es, in SystemC unterschiedliche Abstraktionsebenen zu unterstützen. Dies unterscheidet SystemC deutlich von Hardware-Beschreibungssprachen wie VHDL.

Komposition

Neben der Definition eines SystemC-Moduls durch nebenläufige Prozesse, kann ein Modul auch strukturell durch Komposition anderer SystemC-Module definiert werden. Dies sei an einem Beispiel für eine simulative Verifikationsumgebung verdeutlicht.

Beispiel 2.3.3. Die Verifikationsumgebung ist durch das Modul `TestBench` definiert:

```
 1    class TestBench : public sc_module {
 2      private:
 3        Generator generator;
 4        SUV       suv;
 5        Monitor   monitor;
 6
 7      public:
 8        sc_fifo<double> f1;
 9        sc_fifo<double> f2;
10
11        TestBench(sc_module_name name, int count)
12          : sc_module(name),
13            generator("generator", count),
14            suv("suv"),
15            monitor("monitor"),
16            f1(2),
17            f2(2) {
18          generator.out(f1);
19          suv.in(f1);
20          suv.out(f2);
21          monitor.in(f2);
22        }
23    };
```

Die Verifikationsumgebung enthält neben dem Modul SUV, welches verifiziert werden soll, noch die Module Generator und Monitor. Das Generator-Modul ist dafür verantwortlich, Stimuli für die Simulation zur Verfügung zu stellen, während das Monitor-Modul die Ausgaben des SUV erfasst und gegebenenfalls auswertet. Die Instantiierung der drei Module erfolgt in den Zeilen 13-15. Neben diesen drei Modulen enthält die Verifikationsumgebung noch zwei FIFO-Kanäle, um die Daten zwischen den Modulen zu transportieren (Zeilen 8 und 9).

Die Initialisierung der Kanäle erfolgt im Konstruktor der Verifikationsumgebung (Zeilen 16 und 17). Jedes Modul wird mit einem Namen initialisiert. Das Generator-Modul erhält darüber hinaus noch die Anzahl der zu erzeugenden Stimuli als Konstruktorparameter. Die FIFO-Kanäle erhalten als Konstruktorparameter die Größe des Kanals, d. h. jeder FIFO-Kanal f1 und f2 kann zwei Daten vom Typ double speichern. Weiterhin werden im Konstruktor der Verifikationsumgebung die Module und Kanäle miteinander verbunden. So schreibt beispielsweise das Generator-Modul über seinen Port out auf den Kanal f1 (Zeile 18). Aus dem selben Kanal liest das SUV-Modul über den Eingangsport in.

Simulation von SystemC-Modellen

Zur Simulation muss das SystemC-Modell mit einem C++-Compiler compiliert werden und ausgeführt werden. Für die Simulation der Verifikationsumgebung aus Beispiel 2.3.3 müssen zunächst noch die Module Generator und Monitor implementiert werden. Der folgende Quelltext zeigt die beiden Implementierungen:

```
 1   class Generator: public sc_module {
 2     public:
 3       sc_fifo_out<double> out;
 4
 5     private:
 6       const int max;
 7       void produce() {
 8         for (int i=1; i<=max; i++) {
 9           out.write(i);
10           std::cout << "generator: " << i << std::endl;
11         }
12       }
13
14     public:
15       SC_HAS_PROCESS(Generator);
16
17       Generator(sc_module_name name, int count)
18         : sc_module(name), max(count) {
19         SC_THREAD(produce);
20       }
21   };
```

Das `Generator`-Modul erzeugt insgesamt `max` Stimuli, die es auf den Ausgabeport `out` schreibt. Um die Beschreibung möglichst einfach zu halten, wurde hierbei eine `for`-Schleife verwendet.

```
1    class Monitor: public sc_module {
2     public:
3       sc_fifo_in<double> in;
4
5     private:
6       void consume(void) {
7         while(true) {
8           const double data = in.read();
9           std::cout << "monitor: " << data << std::endl;
10        }
11      }
12
13    public:
14      SC_HAS_PROCESS(Monitor);
15
16      Monitor(sc_module_name name) : sc_module(name) {
17        SC_THREAD(consume);
18      }
19   };
```

Das `Monitor`-Modul liest einfach die Daten vom Eingangsport `in` und gibt diese auf der Standardausgabe aus.

Beispiel 2.3.4. Das Hauptprogramm für die Simulation von fünf Stimuli könnte wie folgt aussehen:

```
1    int sc_main (int argc, char **argv) {
2      TestBench testbench("testbench", 5);
3      sc_start();
4      return 0;
5    }
```

In Zeile 2 wird zunächst die Verifikationsumgebung instantiiert. `sc_start()` startet die Simulation. Liegen keine weiteren Ereignisse in der Simulation vor, wird die Simulation beendet.

Die Ausgabe bei der Simulation sieht dann wie folgt aus:

```
generator: 1
generator: 2
suv:       1
suv:       2
generator: 3
generator: 4
```

```
monitor:   1
monitor:   2
suv:       3
suv:       4
generator: 5
monitor:   3
monitor:   4
suv:       5
monitor:   5
```

Man sieht, dass der ereignisgesteuerte Simulator der SystemC-Referenzimplementierung versucht, ein Modul möglichst lange auszuführen. In diesem Beispiel bedeutet dies, dass zu Beginn das einzige ausführbare Modul, das `Generator`-Modul, zweimal hintereinander ausgeführt wird. Danach ist der FIFO-Kanal `f1` allerdings voll, weshalb der Simulator das nächste ausführbare Modul auswählt. In diesem Fall ist lediglich das `SUV`-Modul ausführbar. Nachdem auch dieses zweimal ausgeführt wurde, ist der FIFO-Kanal `f2` gefüllt, was die weitere Ausführung des `SUV`-Moduls blockiert.

Aus diesem Grund schaltet der SystemC-Simulator zu einem anderen ausführbaren Modul. In diesem Beispiel handelt es sich wieder um das `Generator`-Modul. Alternativ hätte der Simulator auch das `Monitor`-Modul wählen können. Wiederum führt der Simulator das `Generator`-Modul zweimal aus. Anschließend ist lediglich das `Monitor`-Modul ausführbar, welches nun die beiden zuerst produzierten Daten konsumiert usw. Schließlich, nachdem alle fünf Stimuli produziert, weitergegeben und konsumiert sind, beendet sich die Simulation. Man beachte, dass die Reihenfolge der Ausführung auch hätte anders aussehen können, da der SystemC-Standard [236] keine Ablaufplanungsstrategie für den Simulator vorgibt.

Zeitmodellierung in SystemC

SystemC stellt Primitiven zur Zeitmodellierung zur Verfügung. Das zentrale Konstrukt hierfür ist die `wait()`-Anweisung. Der `wait()`-Anweisung kann als Argument entweder ein *SystemC-Ereignis* oder eine Zeitangabe übergeben werden. So führt die Anweisung `wait(10, SC_NS)` z. B. dazu, dass der aktuelle Prozess blockiert wird und erst nach der Simulation von 10*ns* fortgesetzt wird.

Beispiel 2.3.5. Das SystemC-Modell des Generators mit Zeit kann zum Beispiel so modelliert werden, dass der Thread `produce` um eine `wait`-Anweisung erweitert wird:

```
1    void produce() {
2      for (int i=1; i<=max; i++) {
3        out.write(i);
4        wait(10, SC_NS);
5      }
6    }
```

D. h. nach jedem Schreiben eines Stimuli wird 10*ns* gewartet, bis der nächste Stimulus generiert wird. Weiterhin kann der Thread `pass` des Moduls `SUV` aus Beispiel 2.3.3 mit einer Verzögerungszeit von 5*ns* modelliert werden:

```
1     void pass(void) {
2       while(true) {
3         data = in.read();
4         wait(5, SC_NS);
5         out.write(data);
6       }
7     }
```

Um den zeitlichen Verlauf der Daten aufzuzeichnen, bietet es sich an, Signalverläufe (engl. *traces*) in der Simulation aufzuzeichnen. Dies kann z. B. dadurch erfolgen, dass der Konstruktor des `SUV`-Moduls um die folgenden Anweisungen erweitert wird:

```
1     sc_set_time_resolution(1, SC_NS);
2     sc_trace_file *tf;
3     tf = sc_create_vcd_trace_file("trace");
4     sc_trace(tf, generator.i, "stimuli");
5     sc_trace(tf, suv.data, "data");
6     sc_trace(tf, monitor.data, "response");
```

Die Anweisung in Zeile 1 setzt fest, mit welcher zeitlichen Genauigkeit Daten im Trace aufgezeichnet werden. In Zeile 2 und 3 wird die Datei zum Speichern der Signalverläufe deklariert und initialisiert. In den Zeilen 4 bis 6 wird schließlich festgelegt, welche Daten im Trace aufgezeichnet werden. Das Ergebnis der Simulation ist in Abb. 2.12 zu sehen.

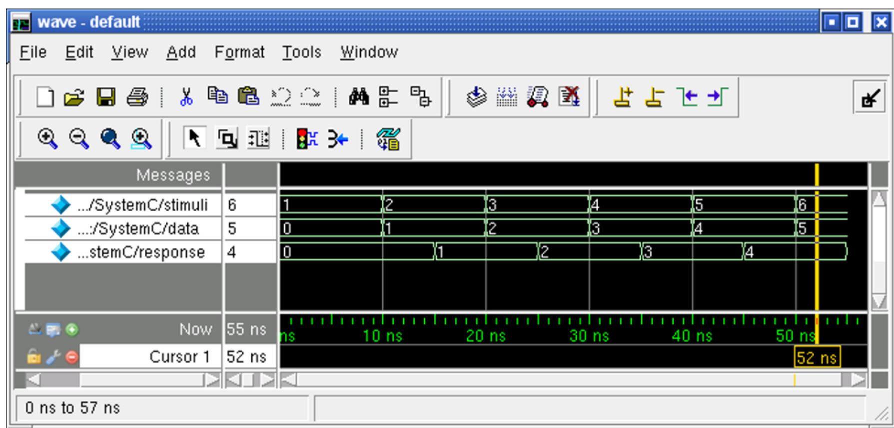


Abb. 2.12. Signalverläufe der SystemC-Simulation

2.3.2 SysteMoC

In heutigen Entwurfsumgebungen im industriellen Umfeld ist es wünschenswert, zu Beginn des Entwurfes ein ausführbares Modell des zu entwickelnden Systems zu haben. Da ausführbare Verhaltensmodelle, etwa in SystemC-Beschreibungen, zwar eindeutig, aber oftmals nicht automatisch analysierbar sind, verwendet das Entwurfswerkzeug *SystemCoDesigner* [215, 254] eine Implementierung des FunState-Modells in SystemC mit dem Namen SysteMoC [157]. SysteMoC unterstützt den aktororientierten Modellierungsansatz [3, 297]. Eine SysteMoC-Beschreibung besteht aus einem Netzwerk kommunizierender Aktoren, welche jeweils wie in FunState aus einer Menge von Funktionen und einem endlichen Automaten bestehen. Die SysteMoC-Implementierung setzt die Ausführungssemantik für FunState-Modelle um. Die SysteMoC-Bibliothek erlaubt es, ein ausführbares Modell zu schreiben und dieses Modell automatisch auf Basis des FunState-Modells zu analysieren.

Definition 2.3.1 (SysteMoC-Modell). *Ein* SysteMoC-Modell *ist ein 4-Tupel* (N,M,I,O)*, wobei N einen Netzgraph, M einen endlichen Automaten und I und O endliche Mengen an Ein- bzw. Ausgängen darstellen. Der Netzgraph N ist wiederum ein 3-Tupel* $N = (A,C,E)$*, wobei A eine endliche Menge von SysteMoC-Aktoren (SysteMoC-Modellen), C eine endliche Menge an Kanälen und E eine endliche Menge an gerichteten Kanten* $E \subseteq A.O \times (C \cup O) \cup (C \cup I) \times A.I$ *ist. Die Menge der Kanäle ist partitioniert in Kanäle Q vom Typ Queues mit FIFO-Semantik und Register R, d. h.* $C = Q \cup R$ *und* $Q \cap R = \emptyset$*. Dabei ist mit jedem Ein- bzw. Ausgang eines FunState-Aktors genau eine Kante* $e \in E$ *verbunden. Bei Queues* $q \in Q$ *gilt, dass es höchstens eine eingehende und eine ausgehende Kante* $e \in E$ *gibt.*

Ein SysteMoC-Modell ist ein Netzgraph bestehend aus kommunizierenden Aktoren. Aktoren wiederum können SysteMoC-Modelle, oder, wie im FunState-Modell, Funktionen sein. Eine Funktion besitzt die Eigenschaft, dass sie irgendwann terminiert. Funktionen werden als *Aktionen* bezeichnet und können dabei auf Daten in angeschlossenen Kanälen lesend und schreibend zugreifen. Funktionen, die den momentanen Zustand des endlichen Automaten oder Daten auf Kanälen nicht verändern, werden als *Wächterfunktionen* (engl. *guards*) bezeichnet. Wächterfunktionen dürfen also nur lesend auf angeschlossene Kanäle zugreifen, wobei die gelesenen Daten nicht konsumiert werden, d. h. eine Wächterfunktion darf den Zustand des Modells nicht ändern.

Die Ausführung eines Aktors wird von dem endlichen Automaten M gesteuert. Hierzu sind die Zustandsübergänge mit Bedingungen und Aktionen attributiert. Ein Zustandsübergang kann durchgeführt werden, wenn die assoziierten Bedingungen erfüllt sind. Bedingungen bestehen aus *Konsumptions-* und *Produktionsraten* sowie Wächterfunktionen. Sind die Bedingungen erfüllt, wird die assoziierte Aktion ausgeführt. Erst nach Beendigung der Aktion wird der neue Zustand in M eingenommen und die Daten auf den Kanälen aktualisiert. Ein bedingter Zustandsübergang kann ausgeführt werden, wenn:

- Mindestens so viele Marken in jedem Eingangskanal verfügbar sind, wie in der jeweiligen Konsumptionsrate spezifiziert ist,

- Mindestens so viele freie Speicherplätze in jedem Ausgangskanal verfügbar sind, wie in der jeweiligen Produktionsrate spezifiziert ist, und
- jede Wächterfunktion den Wert $\top$ (wahr) zurück gibt.

Kanäle können entweder Queues Q mit FIFO-Semantik oder Register R sein, wobei die Kanäle eine unendliche Kapazität besitzen können. Register werden verwendet, um einen internen Zustand in Aktoren zu speichern. FIFO-Kanäle werden verwendet, um Daten zwischen Aktoren auszutauschen. Die Anzahl verfügbarer Daten (sog. *Marken*) in einem Kanal c ist durch $c\#$, die freie Kapazität mit $c\%$ gegeben. Der Wert einer Marke im Kanal c kann mit dem Ausdruck $c\$_i$ abgefragt werden, wobei i die Position der Marke im Kanal c angibt. Für manche Kanäle ist es notwendig, dass diese mit Daten initialisiert werden. Die Anzahl der sog. *Anfangsmarken* eines Kanals c ist durch $c\#_0$ gegeben. Anhand eines Beispiels wird die Modellierung mit SysteMoC gezeigt.

Beispiel 2.3.6. Das folgende Beispiel berechnet iterativ näherungsweise die Quadratwurzel einer Gleitkommazahl, die von der Quelle `src` erzeugt wird. Der SysteMoC-Aktor `sqrloop` entscheidet hierbei, wann das Ergebnis genau genug berechnet wurde. Der Aktor `approx` führt die eigentliche Näherung durch. Der Aktor `dup` dupliziert die vom `approx`-Aktor berechneten Werte. Das Resultat wird schließlich zur Senke `sink` gesendet. Abbildung 2.13 zeigt den Netzgraphen dieser kommunizierenden SysteMoC-Aktoren. Der Zugriff auf Speicherelemente erfolgt in SysteMoC über *Ports*.

```
1   class SqrRoot : public smoc_graph {
2     private:
3       Src      src;
4       SqrLoop  sqrloop;
5       Approx   approx;
6       Dup      dup;
7       Sink     sink;
8     public:
9       SqrRoot(sc_module_name name) : smoc_graph(name),
10        src("A0", 50), sqrloop("A1"), approx("A2"),
11        dup("A3"), sink("A4") {
12        connectNodePorts(src.o0,    sqrloop.i0);
13        connectNodePorts(sqrloop.o1, approx.i1);
14        connectNodePorts(approx.o4,  dup.i4,
15          smoc_fifo<double>(1));
16        connectNodePorts(dup.o5,     approx.i5,
17          smoc_fifo<double>() << 2 );
18        connectNodePorts(dup.o3,     sqrloop.i3);
19        connectNodePorts(sqrloop.o2, sink.i2);
20      }
21  };
```

Ein SysteMoC-Netzgraph wird immer von der Klasse `smoc_graph` abgeleitet.

Die einzelnen Aktoren werden bei der Konstruktion des Netzgraphen initialisiert und verbunden. Die Verbindungen werden hierbei mit Hilfe der Funktion

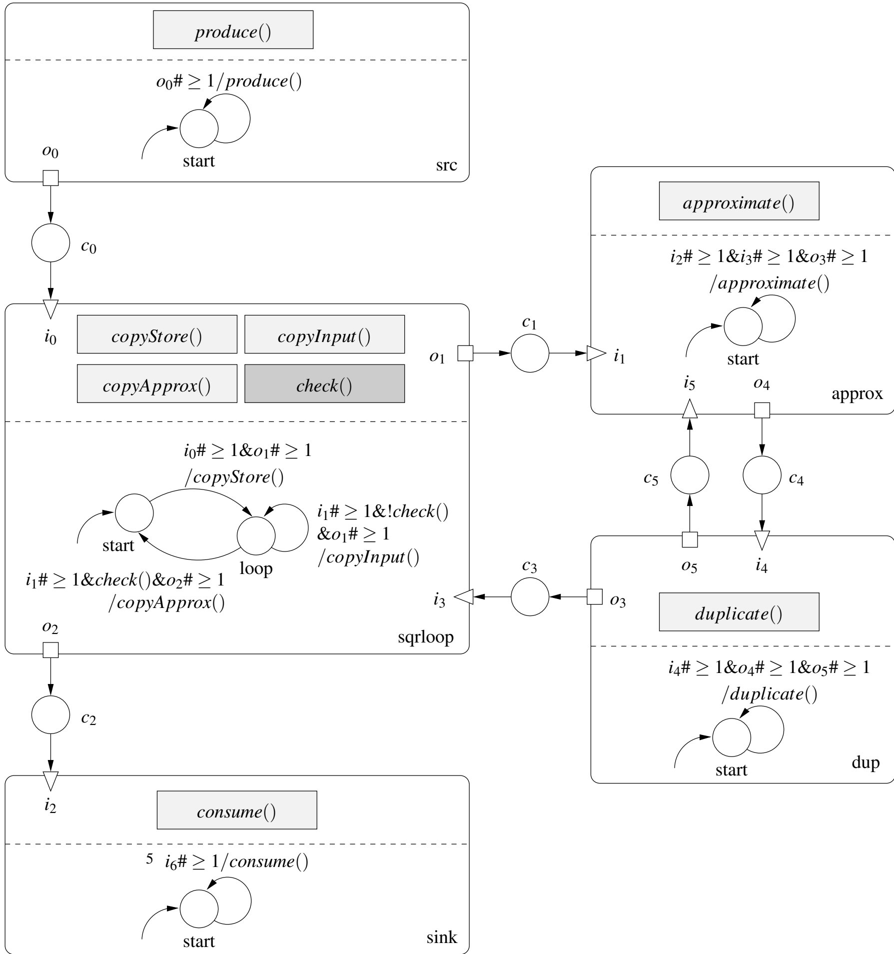


Abb. 2.13. Netzgraph einer SysteMoC-Beschreibung zur näherungsweisen Berechnung der Quadratwurzel

`connectNodePorts` durchgeführt. Als Argumente erhält diese Funktion die beiden zu verbindenden Ports der Aktoren. Standardmäßig erfolgt die Verbindung über einen unbeschränkten FIFO-Kanal (`smoc_fifo`), wobei der Datentyp aus den Portdeklarationen bestimmt wird. Optional kann jedoch ein FIFO-Kanal beschränkter Größe und mit benutzerdefiniertem Datentyp übergeben werden (siehe z. B. die Verbindung von `approx.o4` und `dup.i4`). Schließlich ist es auch möglich, Anfangsmarken auf die FIFO-Kanäle zu legen (siehe Verbindung zwischen `dup.o5` und `approx.i5`).

Als Beispiel für eine SysteMoC-Aktor-Beschreibung wird hier der `SqrLoop`-Aktor ausgewählt. Wie bereits in Abb. 2.13 gezeigt, besteht der `SqrLoop`-Aktor aus

drei unterschiedlichen Aktionen (copyStore, copyInput und copyApprox) und einem endlichen Automaten M mit zwei Zuständen (start und stop). Mit Hilfe der Funktion copyStore wird ein neuer Wert von dem FIFO-Kanal der Quelle src gelesen und in der Variablen tmp_i0 gespeichert. Weiterhin wird dieser Wert auf den Ausgangsport geschrieben, welcher mit dem approx-Aktor verbunden ist. Die Funktion copyInput kopiert für den approx-Aktor den gespeicherten Wert aus der Variablen tmp_i0. Schließlich kopiert die Funktion copyApprox das Ergebnis vom dup-Aktor zum sink-Aktor. Die Aktionen copyStore, copyInput und copyApprox können auf die angeschlossenen Kanäle zugreifen. Die Verbindungen zu den Ports sind nicht eingezeichnet.

Die Zustandsübergänge werden im Konstruktor des Aktors definiert, wobei im Gegensatz zu FunState auch geprüft werden kann, ob ein Speicherelement genügend freie Speicherplätze zum Schreiben besitzt, sollten diese beschränkt sein. So kann beispielsweise vom Zustand start in den Zustand loop gewechselt werden, sofern von der Quelle src mindestens ein Datum zur Verfügung gestellt wurde (i0(1)) und ein Datum in den FIFO-Kanälen zum approx-Aktor geschrieben werden kann (o1(1)). Bei diesem Zustandsübergang wird die Funktion copyStore ausgeführt. Neben der Anzahl der Marken bzw. freien Speicherplätze können auch sog. *Wächterfunktionen* (engl. *guards*), Funktionen mit einem Booleschen Rückgabewert, die den Zustand eines SysteMoC-Modells nicht ändern, als Bedingung berücksichtigt werden. Die Wächterfunktionen können somit die mit Marken assoziierten Werte beziehungsweise einen internen Aktorzustand überprüfen. Eine Wächterfunktion im SqrRoot-Aktor ist die Funktion check, welcher die Genauigkeit des Ergebnisses überprüft. Ist das Ergebnis zu ungenau, verbleibt der Aktor im Zustand loop und wartet auf eine neue (genauere) Approximation. Ist jedoch die Abweichung der Berechnung von der tatsächlichen Quadratwurzel kleiner als eine vorgegebene Schranke, so wird dies als Ergebnis an den Aktor sink übertragen. Der interne Aktorzustand ist in einem Register gespeichert, welches nicht im Bild dargestellt.

Es folgt der SysteMoC-Quelltext des SqrRoot-Aktors:

```
1   class SqrLoop : public smoc_actor {
2     public:
3       smoc_port_in<double>  i0, i3;
4       smoc_port_out<double> o1, o2;
5     private:
6       double tmp_i0;
7       void copyStore()  {
8         tmp_i0 = i0[0];
9         o1[0] = tmp_i0;
10      }
11      void copyInput()  { o1[0] = tmp_i0;          }
12      void copyApprox() { o2[0] = i3[0];           }
13      bool check() const {
14        return fabs(tmp_i0 - i3[0]*i3[0]) < 0.0001;
15      }
16
17      smoc_firing_state start;
```

```
18        smoc_firing_state loop;
19      public:
20        SqrLoop(sc_module_name name) :
21          smoc_actor( name, start ) {
22          start =
23            i0(1)                              >>
24            o1(1)                              >>
25            CALL(SqrLoop::copyStore)           >> loop;
26          loop  =
27            (i3(1) &&  GUARD(SqrLoop::check))  >>
28            o2(1)                              >>
29            CALL(SqrLoop::copyApprox)          >> start
30          | (i3(1) && !GUARD(SqrLoop::check))  >>
31            o1(1)                              >>
32            CALL(SqrLoop::copyInput)           >> loop;
33        }
34    };
```

2.4 Formale Spezifikation funktionaler Anforderungen

Neben dem Verhaltensmodell enthält die Spezifikation von Systemen stets Anforderungen an die Implementierung. Diese Anforderungen können sich dabei sowohl auf *funktionale Eigenschaften* als auch auf *nichtfunktionale Eigenschaften* beziehen. Entsprechend spricht man von *funktionalen* bzw. *nichtfunktionalen Anforderungen*.

Erste Beispiele für funktionale Eigenschaften wurden bereits im Abschnitt 2.2.1 im Zusammenhang mit Petri-Netzen diskutiert. Funktionale Eigenschaften sind dabei häufig sog. *Gefahrlosigkeits-* oder *Lebendigkeitseigenschaften*. Ein Beispiel für eine Gefahrlosigkeitseigenschaft ist, dass ein gegebenes Petri-Netz beschränkt ist. Ist das Petri-Netz beschränkt, so ist sichergestellt, dass keine Pufferüberläufe zu Datenverlusten mit eventuell katastrophalen Folgen führen. Ein Beispiel für eine Lebendigkeitseigenschaft ist die Reversibilität eines Petri-Netzes. Ist diese gegeben, so wird das modellierte System immer verfügbar sein, also die geplante Funktionalität, eventuell in gemindertem Umfang, erbringen.

In diesem Abschnitt werden wichtige Ansätze zur Formulierung von Anforderungen an die funktionalen Eigenschaften präsentiert. Im folgenden Abschnitt werden zunächst die grundlegenden Konzepte zur formalen Spezifikation funktionaler Anforderungen diskutiert. Dabei wird eine Einschränkung auf Ansätze vorgenommen, die eine automatische Verifikation erlauben. Als zwei wichtige Vertreter werden die Logiken *CTL* (engl. *Computation Tree Logic*) und *LTL* (engl. *Linear Time propositional Logic*) vorgestellt. Die zugehörigen Verifikationsmethoden werden in Kapitel 5 präsentiert. Eine Verallgemeinerung von LTL und CTL wird unter dem Namen CTL* diskutiert. Zum Abschluss dieses Abschnittes wird noch auf die Sprache *PSL* (engl. *Property Specification Language*) eingegangen, die von vielen Werkzeugen als Eingabemöglichkeit temporallogischer Formeln unterstützt wird.

2.4.1 Temporale Strukturen

Die in Abschnitt 2.2.1 aufgelisteten Eigenschaften für Petri-Netze können als Gefahrlosigkeits- und Lebendigkeitseigenschaften aufgefasst werden. Diese unterscheiden sich typischerweise in ihrer Formulierung:

- *Gefahrlosigkeitseigenschaften* besagen, dass etwas Schlimmes *niemals* passiert.
- *Lebendigkeitseigenschaften* besagen, dass etwas Gutes *irgendwann* eintritt.

In dieser Formulierung kann man bereits an den Wörtern *niemals* und *irgendwann* erkennen, dass zeitliche Aspekte bei der Formulierung von funktionalen Anforderungen eine Rolle spielen. Zeitliche Aspekte können in *Aussagenlogik* (siehe Anhang A) nicht formuliert werden. Hierfür ist eine Erweiterung um *temporale Operatoren* notwendig. Hierdurch entsteht eine *temporale Aussagenlogik*.

Ein wesentlicher Unterschied zwischen Aussagenlogik und temporaler Aussagenlogik besteht in der Auswertung von Formeln. Während aussagenlogische Formeln konstante Werte repräsentieren, können dies im Fall von temporallogischen Formeln Wertsequenzen sein, welche die funktionalen Eigenschaften eines System beschreiben. Diese Sequenzen werden in einer sog. *temporalen Struktur* repräsentiert. Aus historischen Gründen werden temporale Strukturen auch häufig als *Kripke-Strukturen* bezeichnet [269].

Definition 2.4.1 (Temporale Struktur). *Eine* temporale Struktur *M ist ein Tripel (S,R,L), wobei S eine endliche Menge an Zuständen, $R \subseteq S \times S$ eine Übergangsrelation und $L : S \to 2^V$ eine Markierungsfunktion ist. V ist eine Menge an aussagenlogischen Variablen.*

Die Übergangsrelation R ist total, d. h. $\forall s \in S : \exists s' \in S : (s,s') \in R$. Eine temporale Struktur kann als Graph visualisiert werden, wobei die Zustände als Knoten und die Übergangsrelation als Kanten repräsentiert werden. Die Knoten werden zusätzlich mit einer Teilmenge von Variablen V entsprechend der Markierungsfunktion beschriftet.

Beispiel 2.4.1. Ein Beispiel für eine temporale Struktur M ist in Abb. 2.14a) zu sehen. M besteht aus den drei Zuständen s_0, s_1 und s_2. Insgesamt sind fünf Zustandsübergänge definiert, d. h. $R = \{(s_0,s_1),(s_0,s_2),(s_1,s_1),(s_2,s_0),(s_2,s_1)\}$. Die Menge der Variablen V enthält die drei Elemente a, b und c. Die Markierungsfunktion L liefert $L(s_0) = \{a,b\}$, $L(s_1) = \{c\}$ und $L(s_2) = \{b,c\}$.

Definiert man einen Anfangszustand, können die durch die temporale Struktur repräsentierten Sequenzen in einem unendlichen Graphen dargestellt werden. Da diese Sequenzen typischerweise mit Berechnungen in einem System assoziiert werden, wird dieser Graph als *Berechnungsbaum* (engl. *computation tree*) bezeichnet. Für die temporale Struktur aus Abb. 2.14a) und dem Anfangszustand s_0 ist der Berechnungsbaum in Abb. 2.14b) zu sehen. Man beachte, dass die durch die temporale Struktur repräsentierten Sequenzen unendlich lang sind, weshalb nur ein Ausschnitt aus dem Berechnungsbaum gezeichnet werden kann.

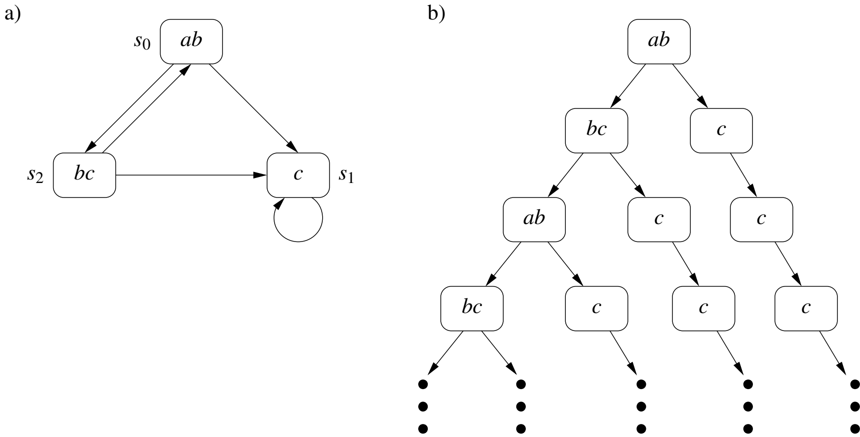


Abb. 2.14. a) Temporale Struktur und b) Berechnungsbaum [272]

Pfade in einer temporalen Struktur beschreiben Sequenzen. Ein Pfad in einer temporalen Struktur M lässt sich formal definieren.

Definition 2.4.2 (Pfad). *Ein* Pfad $\tilde{s}$ *in einer temporalen Struktur* $M = (S,R,L)$ *ist eine unendliche Sequenz* $\tilde{s} := \langle s_0, s_1, \dots \rangle$ *an Zuständen* $s \in S$*, wobei* $\forall i \geq 0 : (s_i, s_{i+1}) \in R$ *gilt. Ein* Suffix $\tilde{s}^i$ *eines Pfades* $\tilde{s} = \langle s_0, s_1, \dots, s_i, s_{i+1}, \dots \rangle$ *ist definiert durch* $\tilde{s}^i := \langle s_i, s_{i+1}, \dots \rangle$*. Ein* Präfix ${}^i\tilde{s}$ *eines Pfades* $\tilde{s} = \langle s_0, s_1, \dots, s_i, s_{i+1}, \dots \rangle$ *ist definiert durch* ${}^i\tilde{s} := \langle s_0, s_1, \dots, s_{i+1}, s_i \rangle$.

Temporale Strukturen sind eng verwandt mit endlichen Automaten (siehe Definition 2.2.13 auf Seite 47). Intuitiv ist eine temporale Struktur ein endlicher Automat ohne Ausgabe. Erfasst man die Eingabe in jedem Zustand einer temporalen Struktur, können endliche Automaten (ohne Ausgabe) in eine äquivalente temporale Struktur umgewandelt werden.

Die Umwandlung eines gegeben endlichen Automaten ohne Ausgabe in eine temporale Struktur erfolgt, indem jeder Zustand einzeln bearbeitet wird. Dies geschieht in vier Schritten:

1. Alle eingehenden Zustandsübergänge werden anhand ihrer Eingabesymbole partitioniert.
2. Für jede Partition wird ein Zustand in der temporalen Struktur erzeugt.
3. Jeder neue Zustand wird mit dem Namen des Zustands des endlichen Automaten und dem Eingabesymbols markiert.
4. Entsprechend der partitionierten Zielzustände werde Zustandsübergänge in der temporalen Struktur eingefügt.

Beispiel 2.4.2. Gegeben ist der endliche Automat in Abb. 2.15a) mit drei Zuständen s_0, s_1 und s_2. Abbildung 2.15b) zeigt die äquivalente temporale Struktur. Zustand s_0 besitzt lediglich einen eingehenden Zustandsübergang mit Eingabesymbol i_0. Aus

diesem Grund gibt es in der temporalen Struktur auch nur einen Zustand der mit s_0i_0 markiert ist.

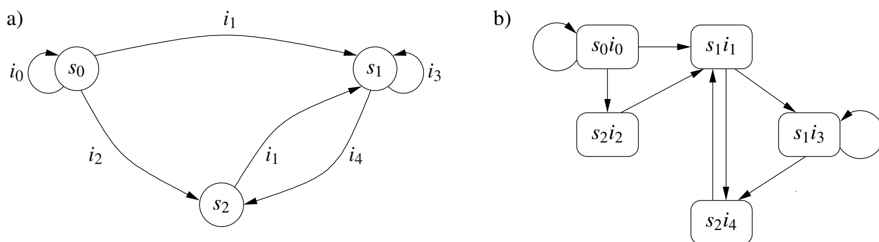


Abb. 2.15. a) endlicher Automat und b) äquivalente temporale Struktur

Zustand s_1 hingegen besitzt drei eingehende Zustandsübergänge mit zwei unterschiedlichen Symbolen, weshalb in der temporalen Struktur zwei Zustände angelegt und mit Eingabesymbolen markiert werden. Schließlich besitzt Zustand s_2 zwei eingehende Zustandsübergänge mit zwei unterschiedlichen Symbolen. Es werden entsprechend zwei Zustände in der temporalen Struktur angelegt.

2.4.2 Temporale Aussagenlogik

Temporale Strukturen repräsentieren Sequenzen. Diese beschreiben funktionale Eigenschaften eines Systems und werden in der Spezifikation als funktionale Anforderungen verwendet. Temporale Strukturen lassen sich mittels temporallogischer Formeln beschreiben. Im Folgenden werden zwei wichtige Vertreter sog. *temporaler Aussagenlogiken* und deren Relation zueinander diskutiert.

Lineare temporale Aussagenlogik (LTL)

Handelt es sich bei der Übergangsrelation R einer temporalen Struktur in Definition 2.4.1 um eine Funktion, so besitzt jeder Zustand $s \in S$ der temporalen Struktur genau einen Nachfolgezustand. In diesem Fall repräsentiert die temporale Struktur genau einen unendlich langen Pfad. Da die Berechnungssequenz nicht verzweigen kann, spricht man auch von einem linearen Zeitmodell. Im Folgenden wird die Erweiterung der Aussagenlogik (siehe Anhang A) um temporale Operatoren für lineare Zeitmodelle präsentiert. Das Ergebnis ist eine *lineare temporale Aussagenlogik* (engl. *Linear Time propositional Logic, LTL*). Im Anschluss wird die Annahme eines linearen Zeitmodells verworfen. Somit können Sequenzen „verzweigen“ (siehe Abb. 2.14b)). In diesem Fall spricht man von einem verzweigenden Zeitmodell (engl. *branching time*).

LTL-Formeln werden mittels atomarer Aussagen (aussagenlogische Variablen), logischen Operatoren und *temporalen Operatoren* gebildet. Zwei grundlegende temporale Operatoren existieren hierbei: X und U. Diese werden als *Next-* und *Until-Operatoren* bezeichnet. Der Next-Operator ist ein unärer, der Until-Operator ein binärer Operator. Ein Zustandsübergang in einer temporalen Struktur wird als *Zeitschritt* bezeichnet. Die Formel X φ sagt aus, dass φ im nächsten Zeitschritt gilt. Die Formel φ U ψ besagt, dass φ solange gilt, bis ψ wahr wird. Implizit gilt, dass ψ irgendwann gilt.

Definition 2.4.3 (Syntax von LTL). *Sei V die Menge an aussagenlogischen Variablen (atomaren Formeln). Eine* LTL-Formel *wird rekursiv wie folgt definiert:*

- *Atomare Formeln $\varphi \in V$ sind LTL-Formeln.*
- *Seien φ und ψ LTL-Formeln, so sind auch $\neg\varphi$ und $\varphi \vee \psi$ LTL-Formeln.*
- *Seien φ und ψ LTL-Formeln, so sind auch* X φ *und* φ U ψ *LTL-Formeln.*

Weitere logische Verknüpfungen können aus $\neg$ und $\vee$ gebildet werden, z. B. $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$. Neben den temporalen Operatoren X und U gibt es weitere temporale Operatoren: Der *Globally-Operator* G φ besagt, dass φ für alle Zeiten gilt. Der *Finally-Operator* F φ sagt aus, dass φ irgendwann in der Zukunft gilt. Dabei schließt die Zukunft die Gegenwart mit ein. Schließlich besagt der *Release-Operator* φ R ψ, dass ψ solange gilt, bis φ das erste Mal wahr wird (eventuell für immer, falls φ niemals gilt, wodurch sich R von dem Operator U unterscheidet). Diese temporalen Operatoren können mit Hilfe des Until-Operators definiert werden:

$$\begin{aligned} \mathrm{F}\,\varphi &:= \top\,\mathrm{U}\,\varphi \\ \mathrm{G}\,\varphi &:= \neg\mathrm{F}\,\neg\varphi \\ \varphi\,\mathrm{R}\,\psi &:= \neg(\neg\varphi\,\mathrm{U}\,\neg\psi) \end{aligned}$$

Die Semantik von LTL-Formeln wird mit Hilfe eines unendlichen Pfad $\tilde{s}$ in einer temporalen Struktur definiert:

Definition 2.4.4 (Semantik von LTL-Formeln). *Sei φ eine LTL-Formel, so bedeutet $M, \tilde{s} \models \varphi$, dass φ entlang des Pfades $\tilde{s} = \langle s_0, s_1, \dots \rangle$ der linearen temporalen Struktur M mit Anfangszustand s_0 gilt. Seien φ und ψ zwei LTL-Formeln, dann kann $\models$ wie folgt definiert werden:*

$$\begin{array}{ll} M,\tilde{s} \models \top & \Leftrightarrow \textit{gilt immer} \\ M,\tilde{s} \models \varphi & \Leftrightarrow \varphi \in L(s_0),\ \textit{falls}\ \varphi \in V \\ M,\tilde{s} \models \neg\varphi & \Leftrightarrow M,\tilde{s} \not\models \varphi \\ M,\tilde{s} \models \varphi \vee \psi & \Leftrightarrow M,\tilde{s} \models \varphi\ \textit{oder}\ M,\tilde{s} \models \psi \\ M,\tilde{s} \models \mathrm{X}\,\varphi & \Leftrightarrow M,\tilde{s}^1 \models \varphi \\ M,\tilde{s} \models \varphi\,\mathrm{U}\,\psi & \Leftrightarrow \exists k \geq 0 : M,\tilde{s}^k \models \psi \wedge \forall 0 \leq j < k : M,\tilde{s}^j \models \varphi \end{array}$$

Man beachte, dass die Semantik von LTL-Formeln bezüglich eines Pfades definiert ist, d. h. das zugrundeliegende Zeitmodell einer LTL-Formel ist linear. Die Semantik der temporalen Operatoren in LTL ist in Abb. 2.16 dargestellt.

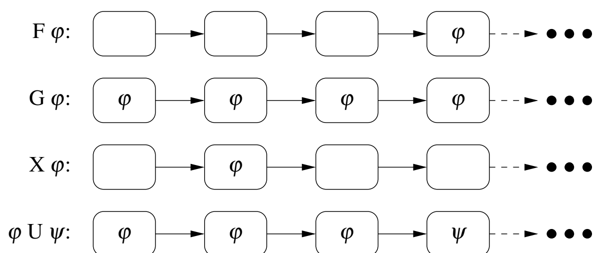


Abb. 2.16. Semantik der temporalen Operatoren in LTL [272]

Definition 2.4.5 (Gültigkeit und Erfüllbarkeit von LTL-Formeln). *Eine LTL-Formel φ heißt* gültig*, falls für alle linearen temporalen Strukturen M gilt, dass $M, \tilde{s} \models \varphi$. Sie heißt* erfüllbar*, falls eine temporale Struktur M existiert, so dass $M, \tilde{s} \models \varphi$. Jede Struktur M, für die gilt, dass $M, \tilde{s} \models \varphi$, heißt* Modell *von φ.*

Eine LTL-Formel ist in ihrer *Normalform* gegeben, wenn Negationen lediglich vor aussagenlogischen Formeln auftreten. So ist beispielsweise $\neg\mathrm{F}\,\varphi$ nicht in Normalform, da die Negation vor dem temporalen Operator F auftritt. Die äquivalente LTL-Formel $\mathrm{G}\,\neg\varphi$ hingegen ist in Normalform. Eine LTL-Formel kann immer in eine äquivalente LTL-Formel in Normalform gebracht werden, indem die Negationen hinter die temporalen Operatoren gebracht werden. Die wichtigsten Äquivalenzen hierfür lauten:

$$\begin{aligned}
\mathrm{X}\,\varphi &= \neg\mathrm{X}\,\neg\varphi \\
\mathrm{G}\,\varphi &= \neg\mathrm{F}\,\neg\varphi \\
\varphi\,\mathrm{U}\,\psi &= \neg(\neg\psi\,\mathrm{R}\,\neg\varphi) \\
\mathrm{F}\,\varphi &= \top\,\mathrm{U}\,\varphi
\end{aligned}$$

Verzweigende temporale Aussagenlogik (CTL)

Nach der Einführung von LTL wird nun die Annahme verworfen, dass die Übergangsrelation *R* der temporalen Struktur *M* eine Funktion sein muss. Als Konsequenz können die repräsentierten Sequenzen „verzweigen“und man erhält dann ein verzweigendes Zeitmodell (engl. *branching time*). In einem verzweigenden Zeitmodell ist es allerdings nicht mehr ausreichend, lediglich temporale Aspekte, wie in einem linearen Zeitmodell, zu berücksichtigen. Vielmehr sind auch die Umstände (Modalitäten) von Bedeutung, für welchen Zweig eine Formel gelten soll. Deshalb werden neben temporalen Operatoren auch *modallogische Operatoren* benötigt.

Zwei modallogische Operatoren werden unterschieden: E und A. Der Operator E beschreibt die *Möglichkeit*, der Operator A die *Notwendigkeit*, dass die folgende Formel gültig wird. Übertragen auf das verzweigende Zeitmodell bedeutet dies, dass zur Erfüllung einer Formel, der der Operator E voransteht, ein Pfad existiert, auf dem die Formel erfüllt. Im Fall einer Formel, der der Operator A voransteht, muss die folgende Formel auf allen Pfaden gültig werden. Aus diesem Grund werden die

modallogischen Operatoren auch als *Pfadquantoren* bezeichnet. Der Pfadquantor E wird als *existenzieller Pfadquantor* (auch Existenzquantor) bezeichnet, da dieser die Existenz mindestens eines Pfades fordert, der die folgende Formel erfüllt. Der Pfadquantor A wird als *universeller Pfadquantor* (auch Allquantor) bezeichnet, da dieser verlangt, dass alle Pfade die folgende Formel erfüllen.

Die beiden Pfadquantoren E und A sind integraler Bestandteil von *CTL* (engl. *Computation Tree Logic*) und eine Voraussetzung für eine temporale Aussagenlogik für verzweigende Zeitmodelle.

Definition 2.4.6 (Syntax von CTL). *Sei V die Menge der aussagenlogischen Variablen (atomaren Formeln). Eine* CTL-Formel *wird rekursiv wie folgt definiert:*

- *Atomare Formeln $\varphi \in V$ sind CTL-Formeln.*
- *Seien φ und ψ CTL-Formeln, so sind auch $\neg\varphi$ und $\varphi \vee \psi$ CTL-Formeln.*
- *Seien φ und ψ CTL-Formeln, so sind auch* $\mathrm{EX}\ \varphi$, $\mathrm{E}\ \varphi\ \mathrm{U}\ \psi$ *und* $\mathrm{EG}\ \varphi$ *CTL-Formeln.*

Wie im Fall von LTL, können auch bei CTL weitere logische und temporale Operatoren abgeleitet werden. Man beachte, dass in CTL jedem temporalen Operator ein Pfadquantor vorangestellt wird. Berücksichtigt man die Pfadquantoren E und A sowie die temporalen Operatoren X, F, G und U, so gibt es in CTL acht temporallogische Operatoren EX, AX, EF, AF, EG, AG, EU und AU. Die in Definition 2.4.6 verwendeten Operatoren EX, EG und EU bilden jedoch eine vollständige Basis, d. h. die anderen temporallogischen Operatoren können aus ihnen abgeleitet werden.

Definition 2.4.7 (Semantik von CTL-Formeln). *Sei φ eine CTL-Formel, so bedeutet $M,s \models \varphi$, dass φ im Zustand s der temporalen Struktur M gilt. Seien φ und ψ zwei CTL-Formeln, dann kann $\models$ wie folgt definiert werden:*

$$
\begin{aligned}
M,s \models \mathsf{T} \quad & \Leftrightarrow \textit{gilt immer}\\
M,s \models \varphi \quad & \Leftrightarrow \varphi \in L(s)\ \textit{falls}\ \varphi \in V\\
M,s \models \neg\varphi \quad & \Leftrightarrow M,s \not\models \varphi\\
M,s \models \varphi \vee \psi \quad & \Leftrightarrow M,s \models \varphi\ \textit{oder}\ M,s \models \psi\\
M,s \models \mathrm{EX}\ \varphi \quad & \Leftrightarrow \exists\langle s_0, s_1, \ldots\rangle, (s = s_0) : M, s_1 \models \varphi\\
M,s \models \mathrm{EG}\ \varphi \quad & \Leftrightarrow \exists\langle s_0, s_1, \ldots\rangle, (s = s_0) : \forall k \geq 0 : M, s_k \models \varphi\\
M,s \models \mathrm{E}\ \varphi\ \mathrm{U}\ \psi \quad & \Leftrightarrow \exists\langle s_0, s_1, \ldots\rangle, (s = s_0) :\\
& \qquad \exists k \geq 0 : M, s_k \models \psi \wedge \forall 0 \leq j < k : M, s_j \models \varphi
\end{aligned}
$$

Abbildung 2.17 zeigt die Semantik der acht temporallogischen CTL-Operatoren.

Definition 2.4.8 (Gültigkeit und Erfüllbarkeit von CTL-Formeln). *Eine CTL-Formel φ heißt* gültig, *falls für alle temporalen Strukturen M und für alle Zustände s gilt, dass $M,s \models \varphi$. Ist eine temporale Struktur M gegeben, so heißt sie* gültig in *M, falls für alle Zustände s von M gilt, dass $M,s \models \varphi$. Sie heißt* erfüllbar, *falls eine temporale Struktur M mit Zustand s existiert, so dass $M,s \models \varphi$.*

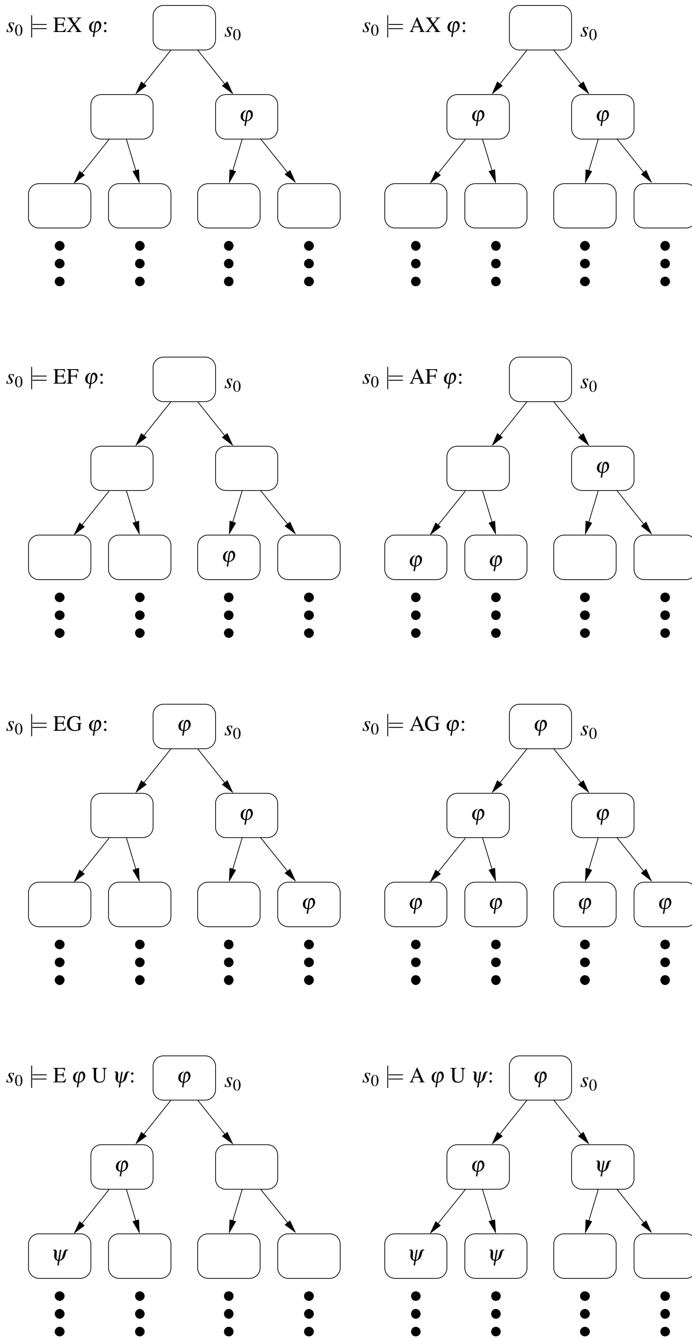


Abb. 2.17. Semantik der acht temporallogischen Operatoren in CTL [272]

Eine CTL-Formel ist in *Normalform*, wenn die Negation nur für aussagenlogische Formeln verwendet wird. Entsprechend ist $\neg\text{AX}\ \varphi$ nicht in Normalform, da die Negation vor AX steht. Hingegen ist die CTL-Formel $\text{EX}\ \neg\varphi$ in Normalform und äquivalent zu $\neg\text{AX}\ \varphi$. Eine CTL-Formel kann stets mit den folgenden Äquivalenzen in Normalform überführt werden:

$$\begin{aligned}
\text{AX}\ \varphi &= \neg\text{EX}\ \neg\varphi \\
\text{AG}\ \varphi &= \neg\text{EF}\ \neg\varphi \\
\text{A}\ \varphi\ \text{U}\ \psi &= \neg(\text{E}\ \neg\psi\ \text{U}\ \neg\varphi \wedge \neg\psi) \wedge \neg\text{EG}\ \neg\psi \\
\text{AF}\ \varphi &= \text{A}\ \top\ \text{U}\ \varphi \\
\text{EF}\ \varphi &= \text{E}\ \top\ \text{U}\ \varphi
\end{aligned}$$

CTL*

Das lineare Zeitmodell, welches LTL zugrundeliegt, ist eine Spezialfall des verzweigenden Zeitmodells. Aus diesem Grund ist es möglich, LTL auch auf einem verzweigenden Zeitmodell zu definieren. Hierzu wird jede LTL-Formel implizit allquantifiziert. In LTL muss nicht jedem temporalen Operator ein Pfadquantor vorangestellt werden, während dies in CTL-Formeln notwendig ist. Deshalb gibt es LTL-Formeln, die sich nicht in CTL ausdrücken lassen und umgekehrt.

Beispiel 2.4.3. Gegeben ist die LTL-Formel $\varphi := \text{FG}\ p$. Wird diese in ein verzweigendes Zeitmodell eingebettet, erhält man $\varphi = \text{AF G}\ p$. Eine temporale Struktur, die ein Modell für φ darstellt, ist in Abb. 2.18 zu sehen. Die LTL-Formel φ ist allerdings nicht äquivalent zu der CTL-Formel $\psi = \text{AF AG}\ p$.

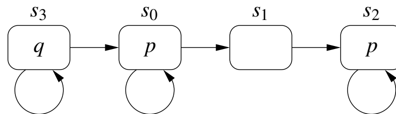


Abb. 2.18. Temporale Struktur, die ein Modell für AF G p ist [457]

Der Grund hierfür ist, dass LTL-Formeln mit individuellen Pfaden assoziiert werden. Auf jedem unendlichen Pfad beginnend im Zustand s_0 kann der Zustand s_2 erreicht werden. Sobald s_2 erreicht ist, gilt p in jedem Folgezustand. Die CTL-Formel $\psi := \text{AF AG}p$ verlangt allerdings, dass auf jedem Pfad, beginnend in s_0, ein Zustand erreicht werden kann, in dem AG p gilt. Der einzige Zustand, in dem AG p gilt, ist Zustand s_2. Allerdings muss nicht jeder Pfad zwangsläufig zum Zustand s_2 führen. Vielmehr gibt es einen Pfad, der beliebig lange in Zustand s_0 verweilt. Genau dieser Pfad ist ein Gegenbeispiel für die Gültigkeit von AF AG p.

Andererseits gibt es CTL-Formeln, die sich nicht in LTL formulieren lassen. Ein Beispiel hierfür ist: AG EF p. Diese Formel besagt, dass auf allen Pfaden, von jedem Zustand aus, ein Pfad existiert, auf dem irgendwann p gilt. Dies lässt sich nicht in LTL ausdrücken. Schließlich gibt es Formeln, die sich weder in LTL noch in CTL ausdrücken lassen. Ein solches Beispiel kann aus den beiden vorherigen Beispielen konstruiert werden:

$$\mathrm{AF\,G}\, p \vee \mathrm{AG\,EF}\, p$$

Eine temporale Aussagenlogik, die sowohl LTL-Formeln, CTL-Formeln als auch deren Kombination ausdrücken kann, ist CTL*. Der Zusammenhang zwischen LTL, CTL und CTL* ist in Abb. 2.19 zu sehen.

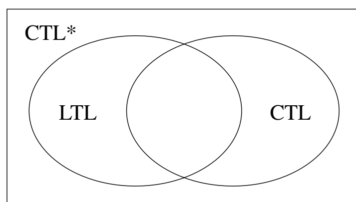


Abb. 2.19. Zusammenhang zwischen LTL, CTL und CTL* [457]

CTL* lässt sich formal definieren (siehe auch [272]):

Definition 2.4.9 (Syntax von CTL*). *Sei V die Menge aussagenlogischer Variablen (atomare Formeln). Jede* Zustandsformel *ist eine zulässige CTL*-Formel die* Pfadformeln *enthalten kann. Dabei sind Zustandsformeln und Pfadformeln wie folgt definiert:*

- *Zustandsformeln:*
  - *Falls $\varphi \in V$, dann ist φ eine Zustandsformel.*
  - *Falls φ und ψ Zustandsformeln sind, so sind $\neg\varphi$ und $\varphi \vee \psi$ ebenfalls Zustandsformeln.*
  - *Falls φ eine Pfadformel ist, dann ist* E φ *eine Zustandsformel.*
- *Pfadformeln:*
  - *Falls φ eine Zustandsformel ist, dann ist φ ebenfalls eine Pfadformel.*
  - *Falls φ und ψ Pfadformeln sind, dann sind ebenfalls $\neg\varphi$, $\varphi \vee \psi$,* X φ *und* φ U ψ *Pfadformeln.*

Die Syntax von CTL* besteht also aus zwei Formelklassen: Zustandsformeln und Pfadformeln. Zustandsformeln können in einem Zustand der temporalen Struktur gültig sein. Die Gültigkeit von Pfadformeln hängt von einem Pfad in einer temporalen Struktur ab. Der Zusammenhang zwischen Zustands- und Pfadformeln ist in Abb. 2.20 dargestellt.

Definition 2.4.10 (Semantik von CTL*). *Sei φ eine Zustandsformel. Dann besagt $M, s \models \varphi$, dass φ im Zustand s der temporalen Struktur M gilt. Sei ψ eine Pfadformel.*

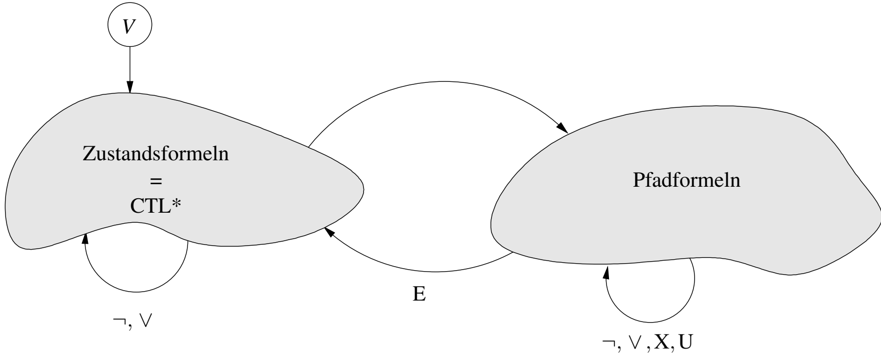


Abb. 2.20. Zustands- und Pfadformeln in CTL* [272]

Dann besagt $M, \tilde{s} \models \psi$*, dass* ψ *entlang des Pfades* $\tilde{s} = \langle s_0, s_1, \ldots \rangle$ *der temporalen Struktur M gilt.*

Gegeben seien zwei Zustandsformeln φ_1 *und* φ_2 *sowie zwei Pfadformeln* ψ_1 *und* ψ_2*. Die Relation* $\models$ *kann wie folgt definiert werden:*

- *Zustandsformeln:*

$$\begin{array}{ll} M, s \models \varphi_1 & \Leftrightarrow \varphi_1 \in L(s), \textit{ falls } \varphi_1 \in V \\ M, s \models \neg\varphi_1 & \Leftrightarrow M, s \not\models \varphi_1 \\ M, s \models \varphi_1 \vee \varphi_2 & \Leftrightarrow M, s \models \varphi_1 \textit{ oder } M, s \models \varphi_2 \\ M, s \models \mathrm{E}\ \psi_1 & \Leftrightarrow \exists \tilde{s} = \langle s_0, s_1, \ldots \rangle, (s = s_0) : M, \tilde{s} \models \psi_1 \end{array}$$

- *Pfadformeln:*

$$\begin{array}{ll} M, \tilde{s} \models \varphi_1 & \Leftrightarrow M, s_1 \models \varphi_1 \\ M, \tilde{s} \models \neg\psi_1 & \Leftrightarrow M, \tilde{s} \not\models \psi_1 \\ M, \tilde{s} \models \psi_1 \vee \psi_2 & \Leftrightarrow M, \tilde{s} \models \psi_1 \textit{ oder } M, \tilde{s} \models \psi_2 \\ M, \tilde{s} \models \mathrm{X}\ \psi_1 & \Leftrightarrow M, \tilde{s}^1 \models \psi_1 \\ M, \tilde{s} \models \psi_1\ \mathrm{U}\ \psi_2 & \Leftrightarrow \exists k \geq 0 : M, \tilde{s}^k \models \psi_2 \wedge \forall 0 \leq j < k : M, \tilde{s}^j \models \psi_1 \end{array}$$

Definition 2.4.11 (Gültigkeit und Erfüllbarkeit von CTL*-Formeln). *Eine Zustandsformel* φ *(eine Pfadformel* ψ*) heißt* gültig*, falls für alle temporalen Strukturen M und für alle Zustände s (alle Pfade* $\tilde{s}$*) in M gilt* $M, s \models \varphi$ *(*$M, \tilde{s} \models \psi$*). Wenn eine temporale Struktur M gegeben ist, dann heißt die Formel* φ *(*ψ*)* gültig in *M, falls für alle Zustände s (alle Pfade* $\tilde{s}$*) von M gilt* $M, s \models \varphi$ *(*$M, \tilde{s} \models \psi$*).*

Eine Formel φ *(*ψ*) heißt* erfüllbar*, falls eine temporale Struktur M mit Zustand s (Pfad* $\tilde{s}$*) existiert, so dass* $M, s \models \varphi$ *(*$M, \tilde{s} \models \psi$*). Wenn eine temporale Struktur M gegeben ist, dann heißt die Formel* φ *(*ψ*)* erfüllbar in *M, falls es einen Zustand s (einen Pfad* $\tilde{s}$*) in M gibt, so dass* $M, s \models \varphi$ *(*$M, \tilde{s} \models \psi$*).*

Wichtige Eigenschaften

In diesem Buch sind insbesondere die folgenden funktionalen Eigenschaften von Interesse:

- Die *Erreichbarkeitseigenschaft* kann in temporaler Aussagenlogik als

  $$\text{EF}\ p$$

  formuliert werden. Sie besagt, dass ein Zustand, der mit p markiert ist, vom momentanen Zustand aus erreichbar ist.
- Die *Gefahrlosigkeitseigenschaft* kann in temporaler Aussagenlogik als

  $$\text{AG}\ p$$

  formuliert werden, wobei $\neg p$ „etwas Schlimmes“ ist. Die Eigenschaft besagt, dass auf allen Pfaden (A) jeder Zustand (G) mit p markiert ist.
- Die *Lebendigkeitseigenschaft* kann in temporaler Aussagenlogik als

  $$\text{AG AF}\ p$$

  formuliert werden. Sie besagt, dass auf allen Pfaden (A) aus allen Zuständen (G) wiederum auf allen Pfaden (A) irgendwann ein Zustand (F) angetroffen wird, der mit p markiert ist.

Gegenbeispiele für Gefahrlosigkeitseigenschaften bestehen aus einem Pfad endlicher Länge, während Gegenbeispiele für Lebendigkeitseigenschaften aus unendlichen Pfaden bestehen. Letztere enthalten dann Schleifen, in denen die geforderte Eigenschaft nicht eintritt.

2.4.3 Die Zusicherungssprache PSL

Neben der formalen Syntax und Semantik temporallogischer Formeln wird hier auch die Zusicherungssprache *PSL* (engl. *Property Specification Language*), die Eingabemöglichkeit von funktionalen Anforderungen bei vielen Verifikationswerkzeugen, präsentiert. PSL unterscheidet bei der Formulierung temporallogischer Formeln drei Ebenen:

1. Die *Boolesche Ebene*, welche aus aussagenlogischen Variablen und Booleschen Ausdrücken besteht.
2. Die *temporale Ebene*, welche den zeitlichen Zusammenhang zwischen den Booleschen Ausdrücken beschreibt.
3. Die *Verifikationsebene*, welche angibt, wie die Eigenschaft während der Verifikation zu verwenden ist.

PSL kann in die sog. *Foundation Language* (FL) und die sog. *Optional Branching Extension* (OBE) unterteilt werden [234]. PSL FL ist eine Mischung aus LTL und erweiterten regulären Ausdrücken. Somit kann mit PSL FL über einzelne Berechnungspfade argumentiert werden. Zur Formulierung von CTL-Formeln wird PSL OBE verwendet. Da PSL oft in Werkzeugen zur Simulation zum Einsatz kommt, wird im Folgenden die hierzu konzipierte PSL FL betrachtet.

Boolesche Ebene

Auf der Booleschen Ebene wird eine funktionale Eigenschaft anhand von Variablen und deren logischen Verknüpfungen beschrieben. Bei den Variablen in PSL handelt es sich um Variablen oder Boolesche Ausdrücke aus dem (ausführbaren) Modell.

Beispiel 2.4.4. Verwendet die Implementierung zwei Boolesche Variablen x_1 und x_2, so wird der gegenseitige Ausschluss dieser Variablen in PSL als $!(x_1 \,\&\, x_2)$ geschrieben [111]. Dabei beschreibt & die Konjunktion und ! die Negation.

Temporale Ebene

Die zeitlichen Relationen zwischen Booleschen Ausdrücken werden auf der temporalen Ebene beschrieben. Hierbei werden sog. *sequentiell erweiterte reguläre Ausdrücke* (engl. *Sequential Extended Regular Expression, SERE*) und *Eigenschaften* (engl. *properties*) unterschieden.

Sequentiell erweiterte reguläre Ausdrücke

Sequentiell erweiterte reguläre Ausdrücke (SEREs) werden verwendet, um temporale Sequenzen von Ereignissen Boolescher Ausdrücke zu beschreiben.

Definition 2.4.12 (SERE). *Seien b ein Boolescher Ausdruck und r_1 und r_2 SEREs, so sind die folgenden Ausdrücke ebenfalls* SEREs*:*

- $[*0]$ *ist die leere Sequenz, diese ist äquivalent zu dem ε-Symbol in regulären Ausdrücken.*
- *b ist eine SERE.*
- $\{r_1\}$ *ist eine SERE. Die geschweiften Klammern sind äquivalent zu den runden Klammern in regulären Ausdrücken.*
- $r_1 \mid r_2$ *ist die Disjunktion von r_1 und r_2.*
- $r_1[*]$ *ist das keinmalige bis beliebig oftmalige Auftreten von r_1.*
- $r_1 \,;\, r_2$ *ist die* Konkatenation *von r_1 und r_2, also r_2 folgt direkt r_1.*
- $r_1 : r_2$ *ist die* Fusion *von r_1 und r_2, d. h. der letzte Boolesche Ausdruck in r_1 und der erste Boolesche Ausdruck in r_2 überlappen sich und müssen gleichzeitig wahr sein.*
- $r_1 \,\&\&\, r_2$ *wird als* Längendurchschnitt *bezeichnet. $r_1 \,\&\&\, r_2$ ist wahr, wenn sowohl r_1 und r_2 wahr sind und beide gleichzeitig starten und enden.*

Neben den oben definierten Operatoren für SEREs wurden abkürzende Schreibweisen eingeführt, die allerdings nicht die Ausdruckskraft von PSL vergrößern [55]. Im Folgenden seien b ein Boolescher Ausdruck, r eine SERE, $i, j, k, l \in \mathbb{N}$ mit $i \geq j$, $l \geq k$ und $l, k > 0$. Die folgenden abkürzenden Schreibweisen können dann in PSL verwendet werden:

$$
\begin{aligned}
r[+] &:= r\,;\, r[*] \\
r[*k] &:= \underbrace{r\,;\, r\,;\, \ldots\,;\, r}_{k\text{-fach}} \\
r[*i:j] &:= r[*i] \mid \cdots \mid r[*j] \\
b[\text{->}] &:= \{(!b)[*]\,;\, b\} \\
b[\text{->}k] &:= \{b[\text{->}]\}[*k] \\
b[\text{->}k:l] &:= \{b[\text{->}]\}[*k:l]
\end{aligned}
$$

Die Operatoren $[*k]$ und $[*i:j]$ werden als *Abzählungswiederholung* und *Bereichswiederholung* bezeichnet. Der Operator [->] wird auch als *GOTO-Operator* bezeichnet, da er das erste Auftreten von b beschreibt.

PSL-Eigenschaften

SEREs können als Sequenzen in PSL und damit als Eigenschaften verwendet werden. Hierfür werden SEREs in geschweiften Klammern geschrieben, z. B. $\{r\}$ für die SERE r. Allerdings müssen auch die Zeitschritte für die SERE definiert werden, also wann der Wechsel zwischen Symbolen auftreten muss. Zeitschritte in Sequenzen werden über Ereignisse identifiziert. Ereignisse können dabei aus den Signalen oder aus Werteänderungen von Variablen in dem untersuchten Modell extrahiert werden.

Beispiel 2.4.5. Häufig besitzt eine Implementierung ein Taktsignal *clk*. Ein geeignetes Ereignis zur Identifikation von Zeitschritten ist dann die positive Taktflanke, was in PSL durch posedge *clk* ausgedrückt wird. Die PSL-Eigenschaft, dass sich die Variablen b_1 und b_2 in jedem Zeitschritt, vorgegeben durch das Taktsignal, ausschließen, wird dann wie folgt formuliert:

$$!(b_1 \;\&\; b_2)@(\text{posedge } clk)$$

Neben SEREs als Sequenzen unterstützt PSL weiterhin alle Standard LTL-Operatoren. Allerdings wird PSL oft in simulativen Verifikationsmethoden eingesetzt (siehe auch Abschnitt 5.2.3) und nicht alle Kombinationen von Eigenschaften eignen sich hierfür. Aus diesem Grund definiert PSL eine *einfache Teilmenge* der Foundation Language, die sich für die Simulation eignet. PSL-FL wird dabei derart eingeschränkt, dass die Negation von SEREs nicht erlaubt ist. Die resultierende Sprache wird mit PSL-FL$^-$ bezeichnet. *PSL-Eigenschaften* werden im Folgenden basierend auf PSL-FL$^-$ beschrieben [55].

Seien b_1 und b_2 zwei Boolesche Ausdrücke. Weiterhin sei r eine SERE. Schließlich seien $i,j,k,l \in \mathbb{N}$ mit $i \geq j$, $l \geq k$ und $l,k > 0$. PSL-Eigenschaften lassen sich wie folgt auf Basis der PSL-FL$^-$ definieren:

- b_1 und b_2 sind PSL-Eigenschaften.
- $\{r\}$ ist eine PSL-Eigenschaft.

Dabei wird davon ausgegangen, dass b_1 und b_2 bzw. $\{r\}$ in der Simulation auftritt. Ist dies nicht der Fall, wird dies als Fehler bewertet. Allerdings ist diese Bedingung eine schwache Bedingung für SEREs, d. h. falls die Simulation endet, bevor die gesamte Sequenz erkannt werden konnte, gilt die Eigenschaft als erfüllt. Eine Eigenschaft kann durch den !-Operator zu einer *starken* Bedingung gemacht werden:

- $\{r\}$! ist eine PSL-Eigenschaft.

In diesem Fall muss die komplette Sequenz vor Beendigung der Simulation erkannt werden.

Sei im Folgenden p eine PSL-Eigenschaft. Die geforderte Erfüllung einer PSL-Eigenschaft kann durch den abort-Operator bei Auftreten einer Booleschen Bedingung b_1 aufgehoben werden:

- p abort b_1 ist eine PSL-Eigenschaft.
- $\{r\}$ |-> p ist eine PSL-Eigenschaft.
- $\{r\}$ |=> p ist eine PSL-Eigenschaft.

Die Operatoren |-> und |=> werden als überlappender oder nichtüberlappender *Suffiximplikations-Operator* bezeichnet. Bei dem überlappenden Operator muss bei jedem Auftreten der Sequenz $\{r\}$ die PSL-Eigenschaft beginnend mit der letzten Booleschen Bedingung in r erfüllt sein. Bei Verwendung des nichtüberlappenden Operators muss p irgendwann nach Auftreten der Sequenz $\{r\}$ erfüllt sein.

In der einfachen Teilmenge von PSL ist die Anwendung des Negations- und Äquivalenzoperators auf PSL-Eigenschaften untersagt. Lediglich Boolesche Ausdrücke dürfen negiert (!) oder auf Äquivalenz (<->) getestet werden:

- !b_1 ist eine PSL-Eigenschaft.
- b_1 <-> b_2 ist eine PSL-Eigenschaft.

Bei der Disjunktion (||) und der Implikation (->) ist lediglich eine PSL-Eigenschaft p als Operand zugelassen, der andere Operand muss ein Boolescher Ausdruck sein. Darüber hinaus gilt bei der Implikation, dass die Bedingung ein Boolescher Ausdruck ist, d. h.:

- b_1 || p ist eine PSL-Eigenschaft, wobei entweder $b_1 = \mathsf{T}$ sein muss oder p erfüllt sein muss, falls $b_1 = \mathsf{F}$ ist.
- b_1 -> p ist eine PSL-Eigenschaft, wobei sobald b_1 eintritt, p erfüllt sein muss.

Zwei PSL-Eigenschaften p_1 und p_2 können konjunktiv verknüpft werden:

- p_1 && p_2 ist eine PSL-Eigenschaft, wobei sowohl p_1 als auch p_2 erfüllt sein muss.

Der LTL-Operator G wird durch den PSL always-Operator realisiert. Für die Negation steht der never-Operator zur Verfügung, dieser kann allerdings lediglich auf Sequenzen $\{r\}$ angewendet werden.

- always p ist eine PSL-Eigenschaft.
- never $\{r\}$ ist eine PSL-Eigenschaft.
- next p ist eine PSL-Eigenschaft.

- next! p ist eine PSL-Eigenschaft.

Der next-Operator entspricht dem LTL-Operator X. Dieser ist allerdings wiederum ein schwacher Operator, d. h. sollte die Simulation nach Aktivierung der Eigenschaft beendet werden, ohne dass es einen weiteren Zeitschritt gibt, gilt die Eigenschaft als erfüllt. Durch Verwendung des next!-Operators, kann dies zu einer starken Bedingung gemacht werden, d. h. der nächste Zeitschritt muss simuliert werden ansonsten gilt die Eigenschaft als verletzt.

Der LTL-Operator U ist in PSL als until-Operator realisiert:

- p until b_1 ist eine PSL-Eigenschaft, wobei p erfüllt sein muss bis b_1 auftritt.
- p until_ b_1 ist eine PSL-Eigenschaft, wobei p erfüllt sein muss bis b_1 auftritt. Außerdem muss p noch während des Auftretens von b_1 erfüllt sein.
- p until! b_1 ist eine PSL-Eigenschaft, wobei p erfüllt sein muss bis b_1 auftritt und b_1 muss in der Simulation auftreten.
- p until!_ b_1 ist eine PSL-Eigenschaft, wobei p erfüllt sein muss bis b_1 auftritt und b_1 muss in der Simulation auftreten. Außerdem muss p noch während des Auftretens von b_1 erfüllt sein.
- b_1 before b_2 ist eine PSL-Eigenschaft, wobei b_1 vor b_2 auftritt.
- b_1 before! b_2 ist eine PSL-Eigenschaft, wobei b_1 vor b_2 auftritt und b_2 muss in der Simulation auftreten.
- b_1 before_ b_2 ist eine PSL-Eigenschaft, wobei b_1 vor b_2 auftritt und auch noch während des Auftretens von b_2 gilt.
- b_1 before!_ b_2 ist eine PSL-Eigenschaft, wobei b_1 vor b_2 auftritt und b_2 muss in der Simulation auftreten. Weiterhin muss b_1 noch während des Auftretens von b_2 gelten.

Der LTL-Operator F ist in PSL nur als starker eventually!-Operator verfügbar.

- eventually! $\{r\}$ ist eine PSL-Eigenschaft, wobei die Sequenz $\{r\}$ bis zum Ende der Simulation auftreten muss.

Zur Erhöhung der Wiederverwendbarkeit, erlaubt PSL die Definition von Eigenschafts- und Sequenzdeklarationen mit Argumenten. Die Eigenschaften und Sequenzen können dann an unterschiedlichen Stellen im Quelltext mit verschiedenen Parametern instantiiert werden.

Beispiel 2.4.6. Die Eigenschaft, dass sich die Variablen x_1 und x_2 gegenseitig ausschließen, lässt sich dann wie folgt formulieren:

$$\text{property } mutex(\text{boolean } clk, x_1, x_2) \;=\; \text{always } !(x_1 \;\&\; x_2)@(\text{posedge } clk)$$

Verifikationsebene

Auf der Verifikationsebene stehen verschiedene Anweisungen zur Verfügung, die dem verwendeten Verifikationswerkzeug mitteilen, wie eine bestimmte Eigenschaft zu verwenden ist. Dabei wird unterschieden, ob eine Eigenschaft erfüllt sein muss,

d. h. dass die Eigenschaft eine *Zusicherung* ist, oder diese erfüllt sein sollte, d. h. dass es sich um eine *Annahme* handelt. Eine Zusicherung wird mit der assert-Anweisung ausgedrückt, während eine Annahme durch die assume-Anweisung gekennzeichnet wird. Für simulative Verifikationsmethoden können diese Anweisungen zur Generierung gesteuerter, zufälliger Testfälle verwendet werden (siehe Abschnitt 3.3).

Weiterhin kann dem Verifikationswerkzeug mitgeteilt werden, welche Sequenzen von Testfalleingaben während der Verifikation zu berücksichtigen sind oder ausgeschlossen werden sollen. Ersteres wird als *Überdeckung* bezeichnet und durch die cover-Anweisung dargestellt. Letzteres wird als *Restriktion* bezeichnet und mittels der restrict-Anweisung angezeigt.

Beispiel 2.4.7. Die Überprüfung der Zusicherung, dass x_1 und x_2 sich gegenseitig ausschließen, kann man somit mit der PSL-Anweisung

$$\text{assert always } !(x_1 \& x_2)\,@\,(\text{posedge } clk);$$

erreichen. Unter Verwendung der Eigenschaftsdeklaration aus Beispiel 2.4.6 kann dies auch als

$$\text{assert } mutex(clk, x_1, x_2);$$

formuliert werden.

2.5 Formale Spezifikation nichtfunktionaler Anforderungen

Eine Anforderung an funktionale Eigenschaften beschreibt, welche Aktion ein System in der Lage sein sollte, zu erbringen. Dabei werden physikalische Randbedingungen der Implementierung nicht näher berücksichtigt. Neben diesen funktionalen Eigenschaften spielen im Bereich eingebetteter Systeme allerdings auch die *nichtfunktionalen Eigenschaften* eine zentrale Rolle. Typische nichtfunktionale Eigenschaften eines Systems sind dabei das Zeitverhalten, die Leistungsaufnahme, der Flächenbedarf etc. Von diesen Eigenschaften ist das Zeitverhalten besonders wichtig, da eingebettete Systeme eng mit ihrer Umgebung interagieren, wobei sicher gestellt sein muss, dass das System schnell genug arbeitet. Die Bedeutung von „schnell genug“ hängt dabei davon ab, welche Aufgabe ein eingebettetes System übernimmt. Handelt es sich beispielsweise um eine Steuerung, so wird das Hauptaugenmerk auf der Antwortzeit liegen. Bei einem MP3-Player hingegen interessiert, ob dieser den notwendigen Durchsatz für eine unterbrechungsfreie Wiedergabe erzielt. Darüber hinaus ist es aber auch denkbar, dass unterschiedliche Verhalten im System implementiert sind, wovon einige eine garantierte Antwortzeit, andere einen garantierten Durchsatz benötigen.

Neben der Tatsache, dass das zu ermittelnde Qualitätsmaß für die Bewertung des Zeitverhaltens nicht offensichtlich ist, eignet sich eine Formulierung wie „schnell genug“ wenig zur quantitativen Bewertung des Zeitverhaltens. Wie schon bei der Spezifikation funktionaler Anforderungen, muss es auch im Fall der nichtfunktionalen Anforderungen möglich sein, diese vollständig, eindeutig und konsistent zu

formulieren. Im Folgenden wird für diesen Zweck eine Erweiterung von CTL vorgestellt. Diese ist besonders gut geeignet, um quantitative Zeitanforderungen für ein gegebenes Verhalten der Implementierung zu spezifizieren. Genauer gesagt wird das Verhalten des Systems, wie im Fall der temporalen Aussagenlogik, als temporale Struktur beschrieben. Allerdings wird die temporale Struktur mit quantitativen Zeitattributen markiert. Zeitliche Anforderungen können dann als Erreichbarkeitseigenschaften zwischen Zuständen bei gegeben Zeitschranken (untere und obere) verstanden werden. Die Vorstellung ist, dass sowohl der Start einer Funktionalität als auch deren Beendigung als Zustand des Systems beschreibbar ist. Zeitabschätzungen zwischen diesen Zuständen entsprechen dann Ende-zu-Ende-Latenzen, die auch für die Quantifizierung des Durchsatzes geeignet sind. In späteren Kapiteln werden spezielle Prüfverfahren für das Zeitverhalten auf diesem Modell vorgestellt.

Die in diesem Abschnitt betrachtete Erweiterung von CTL trägt den Namen *TCTL* (engl. *Timed Computation Tree Logic*). In TCTL werden die temporalen Operatoren (mit Ausnahme des X-Operators) mit quantitativen *Zeitschranken* versehen. Zeitschranken werden als Prädikate über Zeitvariablen definiert. Die Menge $\mathbb{T}$ aller Zeitpunkte kann entweder eine diskrete ($\mathbb{T} = \mathbb{Z}_{\geq 0}$) oder eine kontinuierliche Zeit ($\mathbb{T} = \mathbb{R}_{\geq 0}$) beschreiben. Wenn nicht anders definiert, gilt im Folgenden $\mathbb{T} := \mathbb{Z}_{\geq 0}$. Die Syntax von TCTL lässt sich formal definieren:

Definition 2.5.1 (Syntax von TCTL). *V sei die Menge der aussagenlogischen Variablen (atomaren Formeln). Eine* TCTL-Formel *wird rekursiv wie folgt definiert:*

- *Atomare Formeln $\varphi \in V$ sind TCTL-Formeln.*
- *Seien φ und ψ TCTL-Formeln, so sind auch $\neg\varphi$ und $\varphi \vee \psi$ TCTL-Formeln.*
- *Seien φ und ψ TCTL-Formeln, so sind auch* $\mathrm{EX}\ \varphi$, $\mathrm{E}\ \varphi\ \mathrm{U}_{\sim\gamma}\ \psi$ *und* $\mathrm{A}\ \varphi\ \mathrm{U}_{\sim\gamma}\ \psi$ *TCTL-Formeln.*

Dabei ist $\sim \gamma$ eine sog. Zeitschranke *mit* $\sim \in \{<, \leq, =, \geq, >\}$ *und* $\gamma \in \mathbb{T}$.

Zur Definition der Semantik von TCTL-Formeln muss zunächst die Definition einer temporalen Struktur ebenfalls um Zeitannotationen erweitert werden. Im Folgenden wird von einer Erweiterung der Form ausgegangen, dass ein Zustandsübergang in der temporalen Struktur Zeit benötigt. Diese *Verzögerungszeit* liegt in einem Zeitintervall $\Delta_i(\mathbb{T})$. Die Menge der Intervalle über $\mathbb{T}$ sei mit $\Delta(\mathbb{T})$ bezeichnet. Ein Intervall $\Delta_i(\mathbb{T}) \in \Delta(\mathbb{T})$ kann entweder geschlossen ($\Delta_i(\mathbb{T}) = [\delta_l, \delta_u]$) oder offen ($\Delta_i(\mathbb{T}) = [\delta_l, \delta_u)$) sein. Die untere Grenze δ_l eines Intervalls gibt die minimale Verzögerungszeit eines Zustandsübergangs an, während die obere Grenze δ_u die maximale Verzögerungszeit angibt.

Definition 2.5.2 (Zeitbehaftete temporale Struktur, TTS). *Eine* zeitbehaftete temporale Struktur *(engl.* Timed Temporal Structure, TTS*) M ist ein Tripel (S, R, L), wobei S eine endliche Menge an Zuständen, $R \subseteq R \times \Delta(\mathbb{T}) \times R$ eine zeitbehaftete Übergangsrelation und $L : S \rightarrow 2^V$ eine Markierungsfunktion ist. V ist die Menge der aussagenlogischen Variablen.*

Eine offensichtliche Beschränkung einer TTS ist die ausschließliche Verwendung von Zeitintervallen, deren oberen und unteren Grenzen aufeinander fallen, d. h. die Verzögerungszeit ist ein *exakter Wert*.

Beispiel 2.5.1. Gegeben ist die TTS in Abb. 2.21 aus [312]. Die TTS besteht aus drei Zuständen s_0, s_1 und s_2. Die Zustandsübergänge sind mit den Verzögerungszeiten beschriftet. Man sieht, dass es zwischen zwei Zuständen mehrere Übergänge mit unterschiedlichen Verzögerungszeiten geben kann. Die Bedeutung eines Zustandsübergangs (s, δ, s') ist, dass der Übergang von Zustand s nach Zustand s' exakt δ Zeiteinheiten benötigt. Dabei beschreibt δ die Verzögerungszeit des Zustandsübergangs.

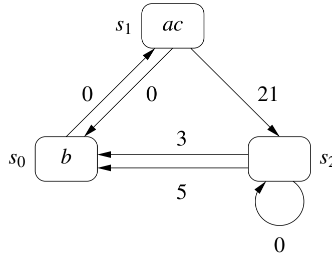


Abb. 2.21. Zeitbehaftete temporale Struktur [312]

Da mehrere Übergänge zwischen zwei Zuständen existieren können, werden Übergänge im Folgenden als $s \xrightarrow{\delta} s'$ geschrieben. Pfade $\tilde{s}$ in einer TTS werden somit als $\tilde{s} = s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} s_2 \xrightarrow{\delta_2} \ldots$ geschrieben. Der *Präfix* ${}^i\tilde{s}$ eines Pfades $\tilde{s}$ der Länge i ist gegeben als ${}^i\tilde{s} := s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{i-1}} s_i$. Die *Länge* eines Präfixes ist gegeben durch $\text{size}({}^i\tilde{s}) = i$. Die *Latenz Λ eines Präfixes* ${}^i\tilde{s}$ ist:

$$\Lambda({}^i\tilde{s}) = \delta_0 + \delta_1 + \cdots + \delta_{i-1}$$

Handelt es sich bei den Verzögerungszeiten δ_i um Intervalle, so ist die Latenz Λ ebenfalls als Intervall gegeben und beschreibt die minimale und maximale Latenz des Präfixes.

Definition 2.5.3 (Semantik von TCTL-Formeln). *Sei φ eine TCTL-Formel, so bedeutet $M, s \models_\tau \varphi$, dass φ im Zustand s der TTS M gilt. Seien φ und ψ zwei TCTL-Formeln, dann kann $\models_\tau$ wie folgt definiert werden:*

$$
\begin{array}{lll}
M,s \models_\tau \mathsf{T} & \Leftrightarrow & \textit{gilt immer} \\
M,s \models_\tau \varphi & \Leftrightarrow & \varphi \in L(s) \textit{ falls } \varphi \in V \\
M,s \models_\tau \neg\varphi & \Leftrightarrow & M,s \not\models_\tau \varphi \\
M,s \models_\tau \varphi \vee \psi & \Leftrightarrow & M,s \models_\tau \varphi \textit{ oder } M,s \models_\tau \psi \\
M,s \models_\tau \mathrm{EX}\,\varphi & \Leftrightarrow & \exists\langle s_0, s_1, \ldots\rangle, (s = s_0) : M, s_1 \models_\tau \varphi \\
M,s \models_\tau \mathrm{E}\,\varphi\,\mathrm{U}_{\sim\gamma}\,\psi & \Leftrightarrow & \exists\langle s_0, s_1, \ldots\rangle, (s = s_0) : \exists k \geq 0 : \\
 & & \quad (\Lambda({}^k\tilde{s}) \sim \gamma) \wedge (M, s_k \models \psi) \wedge (\forall 0 \leq j < k : M, s_j \models \varphi) \\
M,s \models_\tau \mathrm{A}\,\varphi\,\mathrm{U}_{\sim\gamma}\,\psi & \Leftrightarrow & \forall\langle s_0, s_1, \ldots\rangle, (s = s_0) : \exists k \geq 0 : \\
 & & \quad (\Lambda({}^k\tilde{s}) \sim \gamma) \wedge (M, s_k \models \psi) \wedge (\forall 0 \leq j < k : M, s_j \models \varphi)
\end{array}
$$

Wie in CTL können in TCTL weitere Operatoren als Abkürzung abgeleitet werden, wie beispielsweise $\mathrm{AX}\ \varphi = \neg\mathrm{EX}\ \neg\varphi$, $\mathrm{EF}_{\sim\gamma}\ \varphi = \mathrm{E}\ \top\ \mathrm{U}_{\sim\gamma}\ \varphi$, $\mathrm{AF}_{\sim\gamma}\ \varphi = \mathrm{A}\ \top\ \mathrm{U}_{\sim\gamma}\ \varphi$, $\mathrm{EG}_{\sim\gamma}\ \varphi = \neg\mathrm{AF}_{\sim\gamma}\ \neg\varphi$ und $\mathrm{AG}_{\sim\gamma}\ \varphi = \neg\mathrm{EF}_{\sim\gamma}\ \neg\varphi$. Weiterhin können die Standard CTL-Operatoren U, F und G aus den TCTL-Operatoren abgleitet werden als $\mathrm{U}_{\geq 0}$ etc. Schließlich gelten die folgenden Zusammenhänge [287]:

$$\mathrm{A}\ \varphi\ \mathrm{U}_{\leq\gamma}\ \psi = \mathrm{AF}_{\leq\gamma}\ \psi \wedge \neg\mathrm{E}\ (\neg\psi)\ \mathrm{U}\ (\neg\varphi \wedge \neg\psi)$$

und

$$\mathrm{A}\ \varphi\ \mathrm{U}_{\geq\gamma}\ \psi = \mathrm{AG}_{<\gamma}\ (\varphi \wedge \mathrm{A}\ \varphi\ \mathrm{U}_{>0}\ \psi)$$

2.6 Literaturhinweise

Das Thema formale Spezifikation ist ausführlich in [272] und [255] behandelt.

Eine Einführung in die Welt der Petri-Netze gibt der Übersichtsartikel von Murata [338]. Als weiterführende Literatur empfiehlt sich das Studium der Bücher von Peterson [359], Baumgarten [32] sowie von Reisig [373, 374]. Die Bücher von Starke [409] und Girault und Valk [189] widmen sich der Analyse von Petri-Netz-Modellen.

Der Begriff des SDF-Modells stammt von Lee [296] und ist eines der am weitesten verbreiteten Grundmodelle zur Darstellung und zum Entwurf von signalverarbeitenden Systemen. SDF-Graphen können mit anderen Modellen im Ptolemy-Entwurfsystem [69], mittlerweile in der Java-basierten Version II [147], kombiniert werden. In Ptolemy II sind die Aktoren in Java geschrieben und bietet die Unterstützung für unterschiedliche Datenflussmodelle sowie deren Kopplung.

Die Spezifikation von Hardware/Software-Systemen stellt aufgrund der Heterogenität der Komponenten ein großes Problem dar, denn bekannte Modelle und Entwurfssprachen sind stark auf einen Anwendungsbereich (z. B. kontrollflussdominant oder datenflussdominant) bzw. entweder auf die Beschreibung von Hardware (z. B. VHDL [20, 233], SystemVerilog [422, 42]) oder die Beschreibung von Software (z. B. C, C++ [421]) zugeschnitten. Die Sprache SystemC [207, 47, 236] stellt hier einen interessanten Kompromiss dar, da sowohl die Datentypen aus C bzw. C++ als auch die für den Hardware-Entwurf benötigten Datentypen (insbesondere Fixpunktzahlen) unterstützt werden. Außerdem ist die Modellierung von Nebenläufigkeit durch das Modulkonzept vorhanden. Modelle können schließlich in SystemC zeitbehaftet sein oder nicht.

Die *Unified Modeling Language (UML)* [169, 445] stellt letztlich eigentlich keine Sprache, sondern eine Sammlung von unterschiedlichen Modellen zusammen, die darüber hinaus vornehmlich nur den Entwicklungsprozess von reinen Software-Produkten unterstützt haben. Ab der Version 2.0 [13] wird die UML auch zunehmend interessanter für den Entwurf eingebetteter Hardware/Software-Systeme. Von der Intention her stellt UML eine Initiative dar, die vor allem in der Anfangsphase den Entwicklungsprozess von Software standardisieren und damit vereinfachen soll. Zu den wichtigsten der 13 in der Version 2.0 unterstützten Modelle gehören

Sequenzdiagramme, eine Variante von hierarchischen endlichen Automaten, Aktivitätsdiagramme, die ähnlich wie eine Klasse von Petri-Netzen aufgebaut sind, Klassendiagramme, Kommunikationsdiagramme (zur Darstellung von Klassen und Botschaften, die zwischen Objekten der Klassen transferiert werden) sowie sog. *Use Case*-Diagramme. Das große Problem von UML ist jedoch die Semantik und Konsistenzprüfung, wenn mehrere unterschiedliche Modelle gleichzeitig in einem Entwicklungsprojekt verwendet werden.

Bei der Spezifikation von funktionalen Anforderungen spielen temporale Aussagenlogiken eine zentrale Rolle. Eine Übersicht über temporale Aussagenlogiken und temporale Prädikatenlogik erster Ordnung findet man in [148]. Erste Ansätze einer temporalen Aussagenlogik sind in [364] für lineare Strukturen beschrieben. CTL ist in [37] und [97] diskutiert. Da das lineare Zeitmodell ein Spezialfall des verzweigenden Zeitmodells ist, wurde in [286, 150] LTL auf einem verzweigenden Zeitmodell definiert indem alle Pfadformeln impliziert allquantifiziert werden. Hierdurch werden LTL und CTL vergleichbar. Ein zentrales Ergebnis ist allerdings, dass es LTL-Formeln gibt, die nicht in CTL ausdrückbar sind und umgekehrt [286, 149].

Zur Formulierung funktionaler Anforderungen als temporallogische Formel werden zunehmend sog. *Zusicherungssprachen* (engl. *assertion languages*) verwendet. Die beiden wichtigsten Vertreter sind System Verilog Assertions (SVA) [235] und die in Abschnitt 2.4.3 beschriebene Property Specification Language (PSL) [234]. Darüber hinaus werden Bibliotheken angeboten, die wichtige Anforderungen in einer parametrierbaren Form anbieten. Neben dem Vorteil, dass viele Anforderungen nicht mehr geschrieben werden müssen, müssen diese auch nicht mehr auf etwaige Fehler getestet werden. Eine weit verbreitete Anforderungsbibliothek ist die Open Verification Library (OVL) [351].

Erweiterungen von temporalen Logiken um Zeitaspekte sind in [151, 11, 8, 12] beschrieben. Viele Erweiterungen basieren auf CTL und werden als engl. *Timed CTL* (*TCTL*) bezeichnet. Quantitative Zeitschranken können dabei entweder an die temporalen Operatoren annotiert oder über Zeitvariablen in den Formeln beschrieben werden. Der Ansatz über Zeitvariablen ist sehr expressiv aber häufig auch schwer zu verifizieren. Die Semantik von TCTL wird über zeitbehaftete temporale Strukturen (TTS) definiert. Unterschiedliche TTS werden in der Literatur betrachtet. In [287] werden Zustandsübergänge im TTS mit Intervallen annotiert, wobei die untere Schranke die minimale Verzögerungszeit und die obere Schranke die maximale Verzögerungszeit beim Zustandsübergang beschreibt. Eine Beschränkung dieses Modells auf exakte Verzögerungszeiten wird in [152] und [12] vorgestellt. Eine Beschränkung der exakten Verzögerungszeiten auf die Werte 0 und 1 ist in [288] diskutiert. Schließlich beschreibt [151] ein TTS mit konstanten Verzögerungszeiten der Größe 1.

Zeitbehaftete temporale Logiken werden zur Überprüfung von zeitlichen Eigenschaften auf zeitbehafteten Verhaltens- bzw. Strukturmodellen verwendet. Bei den Verhaltensmodellen gibt es zu nahezu jedem zeitfreien Modell auch ein entsprechendes zeitbehaftetes Modell. Bei den zeitbehafteten Petri-Netzen gibt es im Wesentlichen zwei Klassen, die sog. engl. *time Petri nets* [326] und die sog. engl. *timed Petri nets* [370]. Bei *time Petri nets* sind die Schaltzeitpunkte der Transitionen durch Inter-

valle begrenzt, während bei *timed Petri nets* die Transitionen so schnell wie möglich schalten . Bei letzteren wird Zeit mit Stellen oder Transitionen assoziiert. Zeitbehaftete Automaten wurden in [9, 10] auf Basis von Büchi-Automaten mit Erweiterungen um Zeitvariablen eingeführt. Dabei werden die akzeptierenden Zustände verwendet, um einen Fortschritt in den Zeitvariablen der temporallogischen Formeln zu erzwingen. Eine vereinfachte Form zeitbehafteter Automaten, wie sie auch in diesem Buch verwendet werden, wurden erstmals in [218] unter dem Namen engl. *timed safety automata* vorgestellt. In *timed safety automata* wird der Fortschritt der Zeit durch lokale Invarianten in den Zuständen formuliert.

3 Verifikation

Nach der Anforderungsanalyse und Durchführung des Entwurfs (siehe [426]) werden die Syntheseergebnisse verifiziert. Dies ist notwendig, da, insbesondere im Bereich eingebetteter Systeme, Implementierungen diversen Optimierungen unterworfen werden. Dies kann einerseits auch manuelle und somit fehleranfällige Verfeinerungen einschließen. Andererseits sind Synthesewerkzeuge typischerweise selbst nicht verifiziert, was bedeutet, dass nicht sichergestellt ist, dass das Syntheseergebnis korrekt ist.

In diesem Kapitel werden nun grundlegende Fragestellungen zur Durchführung der Verifikation diskutiert, wobei diese unabhängig von der Implementierungsform, also Hardware, Software oder Hardware/Software, und der gewählten Abstraktionsebene vorgestellt werden. Dabei wird eine Unterscheidung in *Verifikationsaufgabe*, *Verifikationsziel* und *Verifikationsmethodik* vorgenommen. Die Verifikationsaufgabe beschreibt dabei, welche Art der Prüfung durchgeführt werden soll. Das Verifikationsziel definiert, was der erwartete Ausgang der Prüfung ist, d. h. ob die Prüfung positiv oder negativ sein soll. Im ersten Fall wird versucht, einen Beweis zu führen, während im zweiten Fall versucht wird, ein Gegenbeispiel zu finden. Schließlich beschreibt die Verifikationsmethodik, welches Verfahren zum Einsatz kommt, wobei prinzipiell simulative und formale Methoden zu unterscheiden sind. Verifikationsaufgabe, -ziel und -methode sind eng miteinander verzahnt. So können Beweise häufig lediglich sinnvoll mit formalen Methoden erzielt werden.

3.1 Verifikationsaufgabe, -ziel und -methode

Beim Entwurf eingebetteter Systeme wird eine Spezifikation in eine Implementierung abgebildet. Die Aufgabe der Verifikation ist es, zu überprüfen, ob dieser Syntheseschritt korrekt verlaufen ist. Hierbei können unterschiedliche Fragestellungen von Interesse sein. Bevor ein System verifiziert werden kann, ist es also notwendig, die *Verifikationsaufgabe* festzulegen. In Kapitel 1 wurden bereits die grundlegenden Verifikationsaufgaben im Bereich eingebetteter Systeme identifiziert: Zum einen kann eine *Äquivalenzprüfung* zwischen Spezifikation und Implementierung durchgeführt

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_3, © Springer-Verlag Berlin Heidelberg 2010

werden. Hierbei wird überprüft, ob das Strukturmodell der Implementierung und das Verhaltensmodell der Spezifikation die gleiche Funktionalität repräsentieren. Die *Eigenschaftsprüfungen* sind eine weitere Klasse von Verifikationsaufgaben. Dabei wird zwischen *funktionaler* und *nichtfunktionaler* Eigenschaftsprüfung unterschieden. Bei der funktionalen Eigenschaftsprüfung wird überprüft, ob das Strukturmodell der Implementierung die funktionalen Anforderungen der Spezifikation erfüllt. Bei der nichtfunktionalen Eigenschaftsprüfung wird überprüft, ob die ermittelten Qualitätsmerkmale die nichtfunktionalen Anforderungen der Spezifikation erfüllen. Die drei Verifikationsaufgaben wurden bereits graphisch in Abb. 1.15 erfasst und sind nochmals in Abb. 3.1 dargestellt.

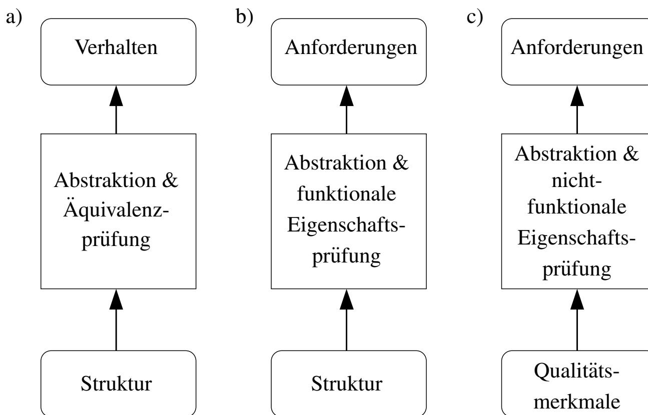


Abb. 3.1. a) Äquivalenzprüfung, b) funktionale und c) nichtfunktionale Eigenschaftsprüfung

Die Grenzen zwischen den drei Verifikationsaufgaben sind fließend. Die Äquivalenzprüfung setzt voraus, dass die Spezifikation vollständig ist, also jedes mögliche Verhalten des Systems spezifiziert. Bei der funktionalen Eigenschaftsprüfung ist dies nicht zwangsläufig notwendig. Vielmehr kann durch Auswahl besonders interessanter Eigenschaften eine unvollständige Spezifikation entstehen. Theoretisch ist es aber denkbar, dass man eine funktionale Eigenschaftsprüfung mit einer vollständigen Spezifikation durchführt. In diesem Fall wird die Verifikationsaufgabe „Äquivalenzprüfung" als Eigenschaftsprüfung formuliert und das Verhaltensmodell der Spezifikation liegt vollständig als Menge funktionaler Anforderungen vor. Somit ist die Äquivalenzprüfung lediglich ein Spezialfall der funktionalen Eigenschaftsprüfung. Dennoch ist eine Unterscheidung sinnvoll. Zum einen bilden Verfahren, die historisch gesehen im Kontext der Äquivalenzprüfung entstanden sind, ein wesentliches Fundament der funktionalen Eigenschaftsprüfung. Zum anderen spielt die Äquivalenzprüfung in der Praxis nach wie vor eine zentrale Rolle. Dies liegt darin begründet, dass Systeme typischerweise evolutionär entstehen, d. h. durch Erweiterung oder Optimierung bestehender Systeme. Das Originalsystem dient in diesem Fall

als Referenz, also als Spezifikation, zur Überprüfung des abgeleiteten Systems, und ist per definitionem bereits vollständig. Berücksichtigt man nun, dass sowohl das Aufstellen vollständiger Spezifikationen als auch das Prüfen aller möglichen Verhaltensweisen eines Systems zeitaufwendig ist, ergibt sich, dass die Äquivalenzprüfung lediglich auf sehr kleine Systeme bzw. Teilsysteme anwendbar ist. Aus diesem Grund wird Äquivalenzprüfung häufig nur für Teilsysteme auf niedrigen Abstraktionsebenen durchgeführt. Auf der Systemebene spielt diese nahezu keine Rolle mehr. Außerdem bietet die Eigenschaftsprüfung die Möglichkeit, wertvolle Verifikationszeit in die Überprüfung interessanter Eigenschaften zu investieren.

Auch die Prüfungen funktionaler und nichtfunktionaler Eigenschaften können nicht immer losgelöst voneinander betrachtet werden. Insbesondere können verschiedene Zustände eines Systems bei der Abarbeitung der selben Funktionen zu unterschiedlichem Zeitverhalten führen. Um beispielsweise zu erkennen, ob das Ausführen einer Funktion kritisch für die Antwortzeit eines Systems ist, müssen somit alle möglichen Systemzustände berücksichtigt werden, aus denen die Funktion aufgerufen werden kann. Andererseits können unterschiedliche Ausführungszeiten der Funktionen im System zu unterschiedlichem Verhalten führen. Als Beispiel sei hier lediglich der Timeout genannt. Erwartet also eine Komponente im System eine Antwort innerhalb eines garantierten Zeitintervalls und bleibt diese aufgrund einer zu langsamen Komponente aus, kann das System zunächst mit der Fehlerbehandlung beschäftigt sein, bevor wieder die eigentliche Grundfunktionalität bereit gestellt wird. Da Fehlerbehandlung aber auch Bestandteil des Verhaltens des Systems ist, sieht man hier das Zusammenspiel zwischen funktionalen und nichtfunktionalen Eigenschaften.

3.1.1 Verifikationsziel

Unabhängig von der Verifikationsaufgabe kann das *Verifikationsziel* formuliert werden. Dabei werden prinzipiell zwei Arten unterschieden: Die *Falsifikation* und der *Beweis*. Die Verifikationsaufgabe beschriebt eine Annahme, z. B. das System besitzt eine Eigenschaft oder zwei Systeme sind äquivalent. Bei Falsifikation wird versucht, die Annahme durch ein *Gegenbeispiel* zu widerlegen, während bei dem Beweis zu zeigen versucht wird, dass keine Eingabesequenz existiert, welche die Annahme widerlegt. Im Folgenden werden Falsifikation und Beweis näher betrachtet.

Falsifikation

Falsifikation beschreibt das Ziel, dass die Annahme aus der Verifikationsaufgabe widerlegt wird. Im Fall der Äquivalenzprüfung soll etwa gezeigt werden, dass Spezifikation und Implementierung *nicht* äquivalent sind. Bei der funktionalen und nichtfunktionalen Eigenschaftsprüfung soll gezeigt werden, dass eine geforderte Eigenschaft *nicht* unter allen Umständen gültig ist. Unabhängig von der Verifikationsaufgabe ist die Ausgabe bei einer erfolgreichen Falsifikation ein Gegenbeispiel, welches zeigt, dass die geforderte Äquivalenz oder die geforderte Eigenschaft nicht erfüllt ist.

Der Vorteil der Falsifikation besteht darin, dass bei einem Erfolg nicht alle möglichen Eingaben und alle möglichen Zustände betrachtet werden müssen. Ist ein Gegenbeispiel gefunden, kann dieses als *Zeuge* verwendet werden, dass die Annahme nicht unter allen Umständen gültig ist. Aus diesem Grund eignen sich simulative Verifikationsmethoden gut für die Falsifikation. Der Nachteil ist allerdings, dass es extrem schwer sein kann, überhaupt ein Gegenbeispiel zu finden. Allerdings spielt die Falsifikation in der Praxis eine zentrale Rolle. Dies liegt daran, dass Entwickler zunächst daran interessiert sind, Entwurfsfehler möglichst schnell aufzudecken. Indem nacheinander unterschiedliche Eigenschaften einer Komponente überprüft werden, bekommt der Entwickler ein größeres Vertrauen in die Implementierung. Dabei ist es nicht entscheidend, einen vollständigen Beweis durchzuführen. Allerdings kann dieses Vertrauen auch trügerisch sein.

Beweis

Ist das Verifikationsziel hingegen ein Beweis der Annahme, müssen alle möglichen Eingabesequenzen eines Systems betrachtet werden. Allgemein lassen sich Beweise auf verschiedene Arten führen, z. B. deduktive Beweise, Reduktion auf Definitionen, der Widerspruchsbeweis, induktive Beweise (siehe auch [223]). Im Rahmen dieses Buches sind insbesondere automatische Verfahren und somit auch automatische Beweisverfahren von Interesse. Die meisten automatisierten Beweisverfahren ziehen den gesamten Zustandsraum eines Systems in Betracht. Für reale Systeme kann dieser allerdings sehr groß werden. Dieses inhärente Problem macht automatisierte Beweise sehr schwer. Insbesondere zeigen sich im Allgemeinen extrem lange Laufzeiten.

Abstraktionsverfeinerung

Eine spezielle Art des automatisierten Beweises ist die sog. *Abstraktionsverfeinerung*. Da der Zustandsraum sehr groß sein kann, wird bei der Abstraktionsverfeinerung zunächst versucht, eine Abstraktion des Systems zu erstellen, was eine Verkleinerung des Zustandsraums zur Folge hat. Diese Abstraktion dient anschließend als Modell des Systems für einen Beweis. Ist die gewählte Abstraktion geeignet und gelingt der Beweis auf dem abstrakten Modell, gilt das Ergebnis auch für das Originalsystem. Eine Abstraktion ist geeignet, wenn es sich um eine sog. *Überapproximation* handelt. Eine Überapproximation enthält das gesamte Verhalten und eventuell zusätzliches Verhalten des Originalsystems. Zeigt die Überapproximation keine Fehler, kann auch das Originalsystem keine Fehler enthalten. Gelingt der Beweis hingegen auf der Überapproximation nicht, muss geprüft werden, ob dass generierte Gegenbeispiel eventuell aus dem zusätzlichen Verhalten resultiert.

Das gefundene *Gegenbeispiel* muss somit nicht zwangsläufig zulässig für das Originalsystem sein, d. h. es existiert eventuell keine Testfalleingabe, die dieses Gegenbeispiel im Originalsystem reproduzieren kann. In diesem Fall muss die gewählte Abstraktion verfeinert werden und der Beweis auf dem verfeinerten Modell erneut durchgeführt werden. Ist das gefundene Gegenbeispiel hingegen *zulässig*, so gilt der

Beweis auf dem Originalsystem auch als gescheitert. Dies ist nochmals in Abb. 3.2 dargestellt.

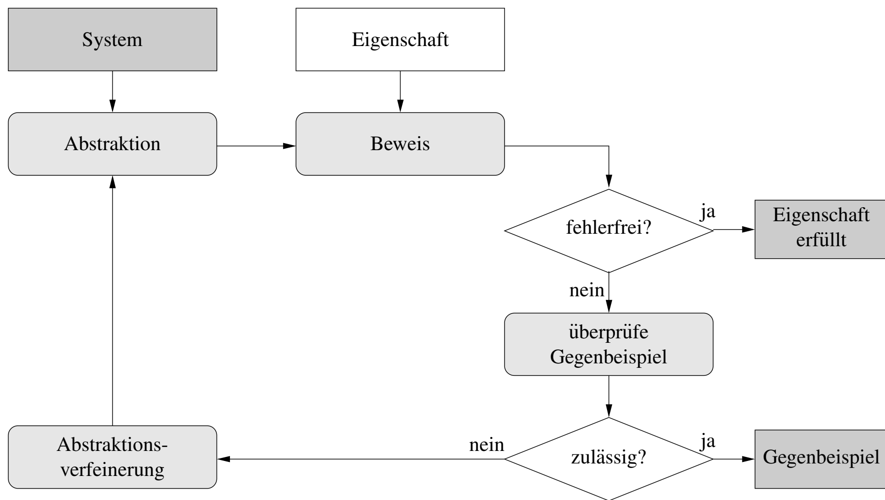


Abb. 3.2. Abstraktionsverfeinerung

3.1.2 Verifikationsmethode

Das Verifikationsziel hat starken Einfluss auf die zu verwendende Verifikationsmethode. Für Beweise werden Methoden benötigt, die *vollständig* sind, d. h. die in der Lage sind, den gesamten Zustandsraum des betrachteten Modells zu analysieren. Entsprechend eignen sich *unvollständige Verifikationsmethoden* lediglich für die Falsifikation, da aufgrund der Unvollständigkeit ein Beweis der Abwesenheit von Fehlern überhaupt nicht möglich ist.

Es können drei grundlegende Verifikationsmethoden unterschieden werden:

1. *Formale Verifikationsmethoden* basieren auf mathematischen Berechnungsmodellen und Beweisverfahren. Formale Verifikationsmethoden besitzen die Eigenschaft, dass sie *vollständig* bezüglich der verwendeten Spezifikation und Abstraktion sind. Dies bedeutet, dass durch formale Verifikationsmethoden die Abwesenheit von Fehlern in einer Abstraktion der Implementierung gezeigt werden kann. Somit lassen sich formale Methoden einsetzen, um das Verifikationsziel eines Beweises zu erreichen. Diese Vollständigkeit begrenzt aber gleichzeitig deren Einsatzfähigkeit. Ganze Systeme lassen sich nur schwer bzw. überhaupt nicht mit formalen Methoden verifizieren. Der Einsatz von formalen Methoden begrenzt sich somit heutzutage auf einzelne Teilsysteme. Die Anwendung formaler Verifikationsmethoden verlangt oftmals ein ausgeprägtes Expertenwissen,

weshalb die Verbreitung dieser Methoden geringer ist als die der simulativen Methoden.

2. *Simulative Verifikationsmethoden* basieren auf ausführbaren Modellen, sog. *Simulationsmodellen*. Diese werden mit ausgewählten Testfällen simuliert, d. h., dass das ausführbare Modell mit ausgewählten Stimuli angeregt und das Ergebnis (die Ausgabe des Simulationsmodells) mit dem erwarteten Ergebnis verglichen wird. Bei heutigen Systemkomplexitäten ist es allerdings nicht möglich, alle möglichen Szenarien, auf die ein System später reagieren soll, zu simulieren. Aus diesem Grund ist die Simulation im Allgemeinen eine *unvollständige Verifikationsmethode*. Dies bedeutet, dass im Allgemeinen lediglich die Anwesenheit von Fehlern, aber nicht deren Abwesenheit in einer Implementierung gezeigt werden kann. Somit lassen sich simulative Verifikationsmethoden lediglich für das Verifikationsziel der Falsifikation einsetzen. Auf der anderen Seite ist der Verzicht auf Vollständigkeit der Grund dafür, dass simulative Methoden auch auf ganze Systeme statt lediglich einzelne Teilsysteme anwendbar sind. Schließlich sei aber auch erwähnt, dass eine Simulation, die sehr viele Implementierungsdetails vernachlässigt, ungenaue Ergebnisse liefert. Wird allerdings der Detaillierungsgrad erhöht, können Simulationen sehr zeitintensiv werden, und verlieren damit ihren Vorteil gegenüber anderen Methoden.
3. *Prototypische Implementierungen* können diesen Geschwindigkeitsnachteil teilweise aufheben, indem diese mit einer ähnlichen Geschwindigkeit wie das endgültige System betrieben werden können. Beispielsweise werden heutzutage sog. *FPGAs* (engl. *Field Programmable Gate Arrays*) eingesetzt, um Hardware-Schaltungen zu prototypisieren. Da es sich hierbei um programmierbare Hardware handelt, kann das System durch *Konfiguration* prototypisiert werden, anstatt einen Chip zu fertigen. Da sich prototypische Implementierungen auch in realen Szenarien testen lassen, ist es auch möglich, nichtfunktionale Eigenschaften, wie z. B. Latenz, Durchsatz und Leistungsaufnahme, viel exakter abzuschätzen, als dies mit formalen oder simulativen Methoden möglich ist. Der Aufwand zur Erstellung einer prototypischen Implementierung ist vergleichbar mit der tatsächlichen Realisierung des Systems. Die größte Zeitersparnis liegt in der Herstellung des Systems. Somit sind die Verifikationsergebnisse basierend auf prototypischen Implementierungen auch erst sehr spät im Entwurfsfluss verfügbar. Aus diesem Grund wird die prototypische Implementierung hier nicht weiter betrachtet.

Formale und simulative Verifikationsmethoden stellen Extrema bezüglich ihrer Vollständigkeit und Anwendbarkeit dar. Während formale Methoden oftmals nur auf Teilsysteme oder sehr stark abstrahierte Systeme anwendbar sind, können simulative Verfahren ganze Systeme betrachten. Auf der anderen Seite sind formale Methoden vollständig und können die Abwesenheit von Fehlern zeigen, d. h. die Korrektheit der Implementierung beweisen, während simulative Verfahren hierzu nicht geeignet sind. Sie dienen vielmehr dazu das Vertrauen in eine Implementierung zu erhöhen. Zu beachten ist aber stets, dass die Vollständigkeit formaler Methoden lediglich für den gewählten Detaillierungsgrad und der zugrundeliegenden Spezifikation gilt. Sind

wesentliche Aspekte für das Verifikationsziel in der verwendeten Abstraktion von der Implementierung oder in der Spezifikation nicht enthalten, können diese auch nicht berücksichtigt werden.

Neben reinen simulativen und formalen Ansätzen, gibt es auch eine Vielzahl von *hybriden Verifikationsmethoden*. Hybride Verfahren basieren auf simulativen und formalen Methoden und versuchen in einer geschickten Art und Weise, die in einer Methode erzielten Teilergebnisse in die jeweils andere Methode zu übertragen. Das Ziel ist die Erhöhung der Vollständigkeit simulativer bzw. die Erhöhung der Anwendbarkeit formaler Verfahren. Zusammenhänge zwischen Verifikationsmethoden bezüglich Vollständigkeit und Anwendbarkeit sind in Abb. 3.3 dargestellt.

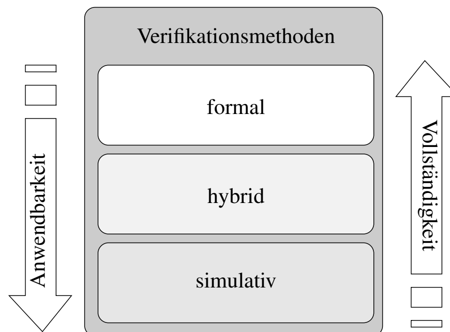


Abb. 3.3. Vollständigkeit bzw. Anwendbarkeit simulativer, formaler als auch hybrider Verifikationsmethoden

Im Folgenden werden simulative und formale Verifikationsmethoden sowie hybride Ansätze in Anlehnung an [472] vorgestellt.

Simulation

Die Simulation einer Implementierung zielt darauf ab, zu überprüfen, ob das System auf eine Stimulation mit einem oder mehreren *Testfalleingaben* korrekt reagiert. Eine Testfalleingabe ist dabei als eine Sequenz von Eingaben gegeben, die an die primären Eingänge des Systems angelegt werden. Die Reaktion des Systems wird an den primären Ausgängen gemessen. Um eine Simulation durchführen zu können, müssen im Wesentlichen vier Aufgaben durchgeführt werden:

1. Die *Generierung* von Testfällen bestehend aus Testfalleingaben und erwarteten Ausgaben,
2. die *Stimulation* des Systems mit den Testfalleingaben in der Simulation,
3. die Überprüfung, ob die Systemreaktion korrekt ist, d. h., ob diese der erwarteten Ausgabe entspricht und
4. die Erhebung von Statistiken zur Ermittlung der Verifikationsvollständigkeit.

Der erste und die beiden letzten Aufgaben werden innerhalb einer sog. *Testbench* durchgeführt. Die Stimulation des Systems übernimmt ein *Simulator*.

Eingebettete Systeme sind stets für eine spezielle Aufgabe entwickelt und optimiert. Dabei interagieren diese Systeme mit ihrer Umgebung. Für die simulative Verifikation eines Systems wird die Umgebung durch eine sog. *Testbench* modelliert. Diese erzeugt Eingaben für das zu prüfende System, welches als engl. *System Under Verification* (SUV) bezeichnet wird. Dies ist in Abb. 3.4 dargestellt. Die Hauptbestandteile einer Testbench sind dabei die Testfallgenerierung, die Überprüfungsstrategie und die Bewertung der Verifikationsvollständigkeit. Diese werden im Folgenden näher betrachtet.

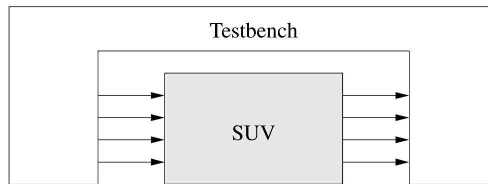


Abb. 3.4. Testbench und SUV (engl. *System Under Verification*)

Testfallgenerierung

Methoden zur Testfallgenerierung können grob in zwei Klassen eingeteilt werden: *gerichtete* und *zufällige*. Man spricht in diesem Zusammenhang auch von *gerichteten* und *zufälligen Testfällen*. Gerichtete Testfälle werden häufig manuell erstellt. Ziel ist es dabei, ein bestimmtes Verhalten zu überprüfen. Aus diesem Grund sind gerichtete Testfälle oftmals auch auf das SUV abgestimmt und können somit nur schwer für andere Modelle oder andere Abstraktionen des SUV wiederverwendet werden. Im Gegensatz dazu ist die zufällige Testfallgenerierung darauf ausgelegt, wiederverwendbar zu sein. Wie der Name bereits sagt, erfolgt die Generierung der Testfälle zufällig. Die Generierung unterliegt allerdings zusätzlichen Einschränkungen, weshalb man auch von einer *gesteuerten zufälligen Simulation* (engl. *constrained random simulation*) spricht. Diese Einschränkungen können während der Simulation angepasst werden, um die Überprüfung unterschiedlicher Verhaltensweisen des SUV zu erlauben.

Gerichtete und zufällige Testfallgenerierung sind so alt wie die Simulation selbst. Allerdings ist die Bestimmung der Einschränkungen in der zufälligen Generierung sehr rechenintensiv, weshalb die zufällige Testfallgenerierung erst in den letzten Jahren mit der Verfügbarkeit neuer Lösungsstrategien an Bedeutung gewonnen hat. In der Praxis werden heutzutage typischerweise zunächst gerichtete Testfälle eingesetzt und anschließend ein große Anzahl zufälliger Testfälle simuliert.

Überprüfungsstrategien

Um zu überprüfen, ob die Reaktion des SUV auf die Stimulation mit einer Testfalleingabe korrekt reagiert, müssen die Ausgaben des Systems beobachtet und bewertet werden. Es wird somit eine *Überprüfungsstrategie* benötigt. Dabei gibt es prinzipiell zwei Ansätze:

- Neben dem eigentlichen SUV wird auch ein *Referenzmodell* mit den Testfalleingaben stimuliert. Die Ausgabe des Referenzmodells wird als korrekt angenommen, weshalb die Ausgabe des SUV mit dieser verglichen wird. Der Nachteil ist, dass ein Referenzmodell verfügbar sein muss. Dieses kann entweder auf der selben oder auf einer anderen Abstraktionsebene vorliegen. Aus diesem Grund wird die Überprüfung auf Basis eines Referenzmodells oftmals in der Äquivalenzprüfung eingesetzt, wo andere Modelle verfügbar sind. Ein Beispiel hierfür sind sog. *Regressionstests*, die bei kleinen inkrementellen Änderungen in der Implementierung zum Einsatz kommen. Die Ausgabe der aktuellen Version wird dabei mit der (als korrekt angenommenen) Ausgabe der vorherigen Version eines SUV verglichen.
- Zur Überprüfung können auch sog. *Monitore* eingesetzt werden. Monitore haben die Aufgabe, auf Basis der Eingaben und Ausgaben des SUV, die Verletzung von Zusicherungen zu erkennen. Monitore können automatisch aus Zusicherungen generiert werden. Der Vorteil gegenüber der Verwendung eines Referenzmodells ist, dass bestimmte Eigenschaften des SUV geprüft werden können und somit die Entwicklung eines Referenzmodells, welches ebenfalls verifiziert werden muss, entfällt. Sollen allerdings alle Eigenschaften des Verhaltensmodells der Spezifikation verifiziert werden, ist der Aufwand zur Erstellung der Zusicherungen im Allgemeinen genau so hoch wie die Entwicklung eines Referenzmodells.

Simulative Verifikationsverfahren eignen sich aufgrund ihrer Unvollständigkeit lediglich zur Falsifikation. Somit ist es also das Ziel der Überprüfungsstrategien, ein von der Spezifikation abweichendes Verhalten zu detektieren. Vorsicht ist aber geboten, wenn die Überprüfung einen Fehler anzeigt, da die Überprüfungsstrategie selbst oftmals nicht verifiziert wurde. Mit anderen Worten: Der Fehler muss nicht zwangsläufig im SUV liegen. Es ist ebenfalls möglich, dass der Fehler in der Überprüfung (dem Monitor, dem Testfall oder dem Referenzmodell) vorliegt.

Verifikationsvollständigkeit

Die Unvollständigkeit simulativer Verifikationsverfahren macht es notwendig, den erzielten Grad an Verifikationsvollständigkeit zu bestimmen. Die Verifikationsvollständigkeit, die durch simulative Verifikation erzielt wird, gibt Aufschluss darüber, wie gut ein System überprüft wurde. Die Vollständigkeit wird typischerweise als *Überdeckungsmaß* ausgedrückt. Dabei werden zwei Arten von Überdeckungsmaßen unterschieden: *strukturorientierte* und *funktionsorientierte*.

Die strukturorientierten Überdeckungsmaße beziehen sich dabei auf das Strukturmodell der Implementierung. Liegt das Strukturmodell in einer Programmier-,

Hardware-Beschreibungs- oder Systembeschreibungssprache vor, können strukturorientierte Überdeckungsmaße auf einen Anteil der Quelltextzeilen abgebildet werden. Beispiele für strukturorientierte Überdeckungsmaße auf Quelltextebene sind die *Anweisungs-* oder *Zweigüberdeckung*. Bei diesen wird der Anteil an ausgeführten Anweisungen bzw. Grundblöcken ins Verhältnis zu allen vorhandenen Anweisungen bzw. Grundblöcken gesetzt. Da Strukturmodelle endlich sind, kann während der simulativen Verifikation auch eine 100%-ige strukturorientierte Überdeckung erzielt werden. Dies ist für reale Systeme allerdings im Allgemeinen nicht durchführbar. Auch sagt eine strukturorientierte Überdeckung wenig darüber aus, wie viel von dem eigentlichen Verhalten des Systems verifiziert wurde.

Dies kann mit funktionsorientierten Überdeckungsmaßen bewertet werden. Funktionsorientierte Überdeckungsmaße beziehen sich auf das Verhaltensmodell oder die funktionalen Anforderungen der Spezifikation. Ist das Verhaltensmodell beispielsweise ein endlicher Automat, kann die relative Anzahl an simulierten Zustandsübergängen als funktionsorientiertes Überdeckungsmaß dienen. Werden Zusicherungen zur Formulierung funktionaler Anforderungen verwendet, kann die Anzahl geprüfter Zusicherungen als Maß dienen. In diesem Fall spricht man auch von einer *zusicherungsbasierten Eigenschaftsprüfung* (engl. *assertion-based verification*). Eine 100%-ige funktionsorientierte Überdeckung garantiert allerdings keine 100%-ige strukturorientierte Überdeckung, da das Strukturmodell beispielsweise nicht erreichbaren Quelltext enthalten kann.

Simulatoren

Simulatoren für diskrete Systeme können entweder *zyklenbasiert*, *ereignisgesteuert* oder *hybrid* arbeiten. Zyklenbasierte Simulatoren tasten Signale in äquidistanten Abständen ab. Im Bereich digitaler Systeme werden die Abtastzeitpunkte typischerweise von den Flanken eines Taktsignals bestimmt. Somit eignet sich die zyklenbasierte Simulation besonders gut zur Simulation synchroner Schaltungen. Dabei wird der neue Inhalt eines Registers als Funktion der Ausgänge anderer Register bestimmt. Die eigentlichen Signallaufzeiten werden dabei nicht simuliert. Hierdurch kann die Simulation auch komplexer Systeme effizient durchgeführt werden. Allerdings kann das exakte Zeitverhalten in Gegenwart asynchroner Eingaben nicht simuliert werden.

Ereignisgesteuerte Simulatoren verrichten im Wesentlichen abwechselnd zwei unterschiedliche Aufgaben: Zum einen erzeugen sie Ereignisse für bestimmte Zeitpunkte in der Zukunft, zum anderen simulieren sie Änderungen im System, wobei neue Ereignisse entstehen. Ein Ereignis ist dabei etwa die Änderung eines Signals. Dadurch, dass der Simulator die Abarbeitung von Ereignissen zeitlich plant, eignen sich ereignisgesteuerte Simulationen zur Modellierung von Verzögerungszeiten und zur Simulation asynchroner Eingaben. Die Effizienz von ereignisgesteuerten Simulatoren ergibt sich aus der Beobachtung, dass während einer Zustandsänderung im SUV nur relativ wenige Signaländerungen betrachtet werden müssen. Der Nachteil ist allerdings, dass Datenabhängigkeiten analysiert werden müssen, um die Ereignisse in der richtigen Reihenfolge zu planen. Weiterhin sind viele Signaländerungen transient, d. h. sie haben keinen Einfluss auf die Zustandsänderungen. Heutige

Simulatoren kombinieren zyklenbasierte und ereignisgesteuerte Simulation. Dabei wird der zyklenbasierte Ansatz zur Simulation der synchronen Teilsysteme eingesetzt. Der ereignisgesteuerte Ansatz dient zur Simulation asynchroner Ereignisse.

Aufgrund der Vielzahl an zu simulierenden Ereignissen, die sich aus der Komplexität heutiger Systeme ergibt, ist es notwendig, die Simulation effizient durchzuführen. Da oftmals Teilsysteme bereits als Hardware/Software-Implementierung vorliegen, ist eine Möglichkeit der Simulationsbeschleunigung dadurch gegeben, diese Teilsysteme nicht zu simulieren, sondern deren Implementierung direkt mit der Simulation zu koppeln. Man spricht dabei auch von einer engl. *hardware-in-the-loop simulation*. Liegen Teilsysteme jedoch nicht bereits als Implementierung vor, können diese prototypisiert werden. Die Simulationsbeschleunigung durch Prototypisierung kann soweit getrieben werden, dass das Gesamtsystem prototypisiert wird. Dabei ist es sogar denkbar, die Testbench ebenfalls zu prototypisieren. Eine extreme Ausprägung simulativer Verifikationsmethoden ist somit die Prototypisierung des Gesamtsystems.

Formale Methoden

Im Gegensatz zu simulativen Verifikationsmethoden bieten formale Methoden eine 100%-ige Verifikationsvollständigkeit bezüglich der zu prüfenden Eigenschaften und der gewählten Abstraktionsebene. Diese Vollständigkeit erfordert aber auch eine enorme Rechenleistung, weshalb formale Methoden im Wesentlichen für kleine oder stark abstrahierte Systeme einsetzbar sind. Im Folgenden werden einige wichtige formale Verifikationsmethoden kurz diskutiert.

Symbolische Simulation

Eine formale Verifikationsmethode, die an simulative Verifikationsmethoden angelehnt ist, ist die sog. *symbolische Simulation*. In der symbolischen Simulation werden anstelle von konkreten Eingabewerten Symbole verwendet, die alle möglichen (oder eine Teilmenge der) Eingabewerte repräsentieren. Die Simulation selbst kann zyklenbasiert oder ereignisorientiert erfolgen. Die Ergebnisse von Funktionsberechnungen auf den Symbolen ergeben allerdings ebenfalls keine konkreten Werte, sondern werden in *Ausdrücke mit Symbolen* transformiert. Diese Ausdrücke dienen dann als Eingabe für folgende Funktionsberechnungen, bis die Ergebnisse zu den Ausgängen übertragen sind.

Da durch die Verwendung von Symbolen alle möglichen Eingabewerte vollständig repräsentiert werden, ist die symbolische Simulation eine formale Methode. Auf der anderen Seite führt diese Vollständigkeit, wie bei allen anderen formalen Methoden, zu einer sog. *Zustandsraumexplosion*. Dies bedeutet bei der symbolischen Simulation, dass die berechneten Ausdrücke so komplex werden, dass sie nicht mehr effizient transformiert werden können. Etliche Optimierungen wurden in der Vergangenheit für die symbolische Simulation vorgeschlagen. Besonders interessant ist dabei die Verwendung von uninterpretierten Funktionen, welche die Fortpflanzung symbolischer Ausdrücke beschränkt, indem Funktionen als sog. *Black-Boxes* betrachtet werden [200].

Modellprüfung

Modellprüfung ist ein Verfahren zur funktionalen Eigenschaftsprüfung, welches auf einem endlichen Zustandsmodell des Systems arbeitet. Die zu prüfenden Eigenschaften werden als temporallogische Formeln formuliert [369, 97]. Diese Formeln beschreiben Zustandsmengen, die durch Fixpunktberechnung bestimmt werden. Ebenfalls durch Fixpunktberechnung lässt sich die Menge aller erreichbaren Zustände eines Modells bestimmen. Indem geprüft wird, ob die Zustandsmenge, die durch die Formel beschrieben wird, in der Menge erreichbarer Zustände enthalten ist, wird die Gültigkeit einer temporallogischen Formel gezeigt.

Modellprüfung kann entweder *explizit*, durch Aufzählen des Zustandsraums, oder *implizit*, durch eine symbolische Repräsentation des Zustandsraums, sein. Letztere wird auch als *symbolische Modellprüfung* bezeichnet. Implizite Modellprüfungsverfahren können, aufgrund der effizienteren Repräsentation und Manipulation von Zustandsmengen, größere Zustandsräume behandeln [74].

Im letzten Jahrzehnt wurden Modellprüfungsverfahren durch den Einsatz von SAT-Solvern (siehe Anhang C.2) stark optimiert [49]. SAT-basierte Modellprüfung stellt allerdings im Allgemeinen keine vollständige Verifikationsmethode dar, da die Anzahl der betrachteten Zustandsübergänge beschränkt ist. Somit ist das Verfahren vergleichbar mit einer erschöpfenden Simulation, welche auf die selbe Anzahl von Zustandsübergängen begrenzt ist. Mittels Induktion kann allerdings für SAT-basierte Modellprüfung Verifikationsvollständigkeit erzielt werden. Daneben kann eine unbeschränkte Modellprüfung durch Kombination von SAT-Solvern und binären Entscheidungsdiagrammen (siehe Anhang B.2) [321, 209] oder durch Anwendung spezieller Approximationstechniken [320] erreicht werden.

Sprachinklusion

Bei der Prüfung auf Sprachinklusion werden sowohl das System als auch die zu prüfende Eigenschaft als endlicher Automat repräsentiert. Jeder Automat spezifiziert eine Sprache. Die Verifikationsmethode basierend auf Sprachinklusion überprüft, ob die durch den Eigenschaftsautomaten beschriebene Sprache die Sprache des Systemautomaten enthält.

Die Überprüfung auf Sprachinklusion und Modellprüfung sind verwandte Verfahren. Insbesondere ist die Modellprüfung von LTL-Formeln (siehe Abschnitt 2.4.2) als Sprachinklusionsproblem formuliert. Beide Verfahren, Modellprüfung und Prüfung auf Sprachinklusion, sind automatisierbar und sind vollständig bezüglich der gewählten Abstraktion und der betrachteten funktionalen Eigenschaften. Diese Vollständigkeit birgt allerdings den Nachteil, dass beide Verfahren von der Zustandsraumexplosion betroffen sind.

Theorembeweiser

Im Gegensatz zu Modellprüfung und Überprüfung auf Sprachinklusion arbeiten Theorembeweiser im Allgemeinen nicht automatisiert. Für den Einsatz von Theorembeweisern werden sowohl die zu prüfende Eigenschaft als auch das zu überprüfende System als Formel repräsentiert. Die Erfüllbarkeit der Eigenschaft wird

überprüft, indem versucht wird, die Formel der Eigenschaft aus der Systemrepräsentation mittels Axiomen abzuleiten. Der eigentliche Beweis erfolgt oftmals nicht direkt aus den Axiomen, sondern mittels zusätzlicher Definitionen und Lemmas, basierend auf den zugrundeliegenden Axiomen. Nahezu alle Theorembeweiser arbeiten interaktiv, d. h. sie erfordern Hinweise durch Eingaben eines menschlichen Benutzers, weshalb interaktive Theorembeweiser auch als *Beweisassistenten* bezeichnet werden.

Hybride Methoden

Die Klassen der simulativen und formalen Verifikationsmethoden haben jeweils ihre Vor- und Nachteile. Durch Kombination von beiden Ansätzen wird versucht, die Vorteile von beiden zu nutzen, und dabei die Nachteile zu umgehen. Es gibt eine Vielzahl an Möglichkeiten, simulative und formale Methoden zu kombinieren. Hier werden lediglich zwei Möglichkeiten vorgestellt.

Ein einfaches Vorgehen bei der Kombination von simulativen und formalen Methoden ist, diese ineinander zu verschachteln. Beispielsweise können zunächst einige gerichtete und zufällige Testfälle simuliert werden, um triviale Fehler zu beseitigen. Anschließend können mittels formaler Methoden weitere Fehler erkannt und beseitigt werden, bis die formalen Methoden aufgrund ihres Ressourcenbedarfs unrentabel werden. In einem abschließend Schritt wird ausgiebige zufällige Simulation angestoßen.

Eine andere Form der Kombination erhält man, wenn man eine zufällige Simulation verwendet, um Zustände zu erreichen, die als neue Anfangszustände für die formale Verifikation dienen. Diese Zustände können dabei so gewählt werden, dass sie in der Nähe von Zuständen liegen, die durch die zu prüfende Eigenschaft spezifiziert werden. Dabei wird die Nähe zwischen zwei Zuständen meist über deren Codierung berechnet, z. B. der Hamming-Distanz [473, 469].

3.2 Beobachtbarkeit und Steuerbarkeit

Eine offensichtliche Unterscheidung zwischen simulativen und formalen Verifikationsmethoden liegt in der Verwendung von Testfällen. Simulative Verfahren benötigen Testfalleingaben, um eine Simulation durchzuführen, während formale Verfahren diese nicht verlangen. Das Vorgehen bei simulativen Verfahren ist somit eingabegetrieben, d. h. es werden zunächst Stimuli definiert, mit denen das System angeregt wird. Aus der Spezifikation werden zu diesen Stimuli *Referenzergebnisse* abgeleitet bzw. *Monitore* generiert, die in der Simulation zur Überprüfung der Ergebnisse verwendet werden.

Beim Einsatz formaler Verifikationsmethoden wird diese Vorgehensweise umgekehrt: Hierbei wird zunächst das gewünschte (Ausgabe-)Verhalten des Systems festgelegt und formalisiert. Die Verifikationsmethodik hat dann die Aufgabe, zu überprüfen, ob dieses gewünschte Verhalten durch die Implementierung erfüllt wird

oder nicht. Dabei werden sämtliche Eingaben berücksichtigt, d. h. formale Verifikationsmethoden sind vollständig. Bei der Verwendung formaler Verifikationsmethoden denkt der Entwickler in keinster Weise an irgendwelche Stimuli. Dieser Ansatz fällt vielen Entwicklern in der Praxis schwerer als gerichtet Testfalleingaben für die simulative Verifikation zu erstellen.

Vergleicht man simulative und formale Verifikationsmethoden, so erkennt man, dass die Simulation eines einzelnen Testfalls wie ein Punkt in einem möglichen *Eingaberaum* betrachtet werden kann. Dabei stellt der Eingaberaum die Menge aller möglichen Testfalleingaben dar. Somit kann simulative Verifikation als ein Abtasten des Eingaberaums angesehen werden. Solange einzelne Punkte aber nicht simuliert worden sind, bleibt die Möglichkeit, dass ein folgender, noch nicht simulierter, Testfall fehlschlägt. Betrachtet man formale Verifikation als Abtastung des Eingaberaums, so stellt man fest, dass aufgrund der Vollständigkeit formaler Methoden, bei deren Anwendung alle möglichen Punkte im Eingaberaum betrachtet werden. Andererseits kann man Simulation als ein Abtasten des *Ausgaberaums* ansehen. Der Ausgaberaum ist dabei die Menge aller Systemausgaben auf die Stimulation mit den Testfalleingaben aus dem Eingaberaum. Funktionale Eigenschaften, wie sie in der formalen Verifikation eingesetzt werden, beschreiben in der Regel eine Menge an Punkten im Ausgaberaum. Dies ist in Abb. 3.5 dargestellt.

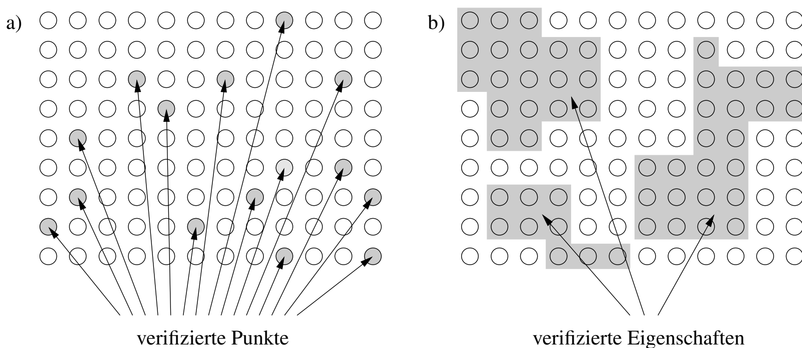


Abb. 3.5. Abtastung des Ausgaberaums in a) simulativer und b) formaler Verifikation

Die Vollständigkeit von formalen Verifikationsmethoden führt oftmals dazu, dass fälschlicherweise angenommen wird, dass eine formal überprüfte Implementierung frei von Fehlern ist. Betrachtet man wiederum Abb. 3.5b), so sieht man, dass formale Verifikationsmethoden lediglich vollständig bezüglich der formulierten Eigenschaften sind. Um eine Implementierung vollständig zu überprüfen, müssen die formulierten Eigenschaften vollständig sein, d. h. alle Punkte im Ausgaberaum abdecken. Aufgrund von Speicher- und Laufzeitbeschränkungen ist es in der Praxis allerdings im Allgemeinen nicht möglich, alle Eigenschaften zu spezifizieren und zu verifizieren. Somit wird oftmals eine Implementierung lediglich gegen eine partielle Spezi-

fikation geprüft. Alle nicht überprüften Eigenschaften lassen Raum für Fehler in der Implementierung.

Daneben gibt es weitere Faktoren, warum die Anwendung formaler Verifikationsmethoden keine Garantie für fehlerfreie Implementierungen sein kann [284]:

- Fehler können bereits in der *Spezifikation* vorhanden sein. Diese Fehler können nicht während der Verifikation aufgedeckt werden, da die Spezifikation als Referenz und somit als fehlerfrei gilt. Das Auffinden von Spezifikationsfehlern wird als *Validierung* bezeichnet.
- Benutzerfehler bei der *Bedienung* von Verifikationswerkzeugen können dazu führen, dass Fehler in einer Implementierung übersehen werden.
- Weiterhin können *Verifikationswerkzeuge* selbst Fehler enthalten, da diese z. B. nicht verifiziert wurden.
- Schließlich kann das Modell der Implementierung zu *abstrakt* sein, um Fehler sichtbar zu machen. Mit anderen Worten: Ein Fehler kann in der Implementierung aufgrund der gewählten Abstraktion nicht mehr im verwendeten Modell enthalten sein.

Der letzte Punkt drückt aus, dass formale Verifikationsmethoden neben den formulierten Eigenschaften auch nur bezüglich der gewählten *Abstraktion* vollständig sind. Dies gilt uneingeschränkt auch für simulative Verifikationsmethoden und wird daher genauer betrachtet: Um einen Fehler in dem SUV erkennen zu können, muss der Fehler sowohl *aktivierbar* als auch *übertragbar* sein. Die Aktivierung eines Fehlers beschreibt die Anregung des SUV mit geeigneten Stimuli, so dass das fehlerhafte Verhalten auftritt. Daneben muss die Belegung der primären Eingänge des SUV so gewählt werden, dass der Fehler an einen primären Ausgang übertragen wird. Lässt sich ein Fehler nicht aktivieren oder nicht übertragen, so ist er nicht *steuerbar* bzw. nicht *beobachtbar*, und somit auch nicht überprüfbar.

Ob ein Fehler in der Implementierung, der nicht steuerbar oder nicht beobachtbar ist, überhaupt ein Fehler ist, ist wohl eher ein philosophisches Problem und wird hier nicht näher betrachtet.

Abstraktion

Allerdings wird der Effekt der Abstraktion auf die Beobachtbarkeit und Steuerbarkeit betrachtet, d. h. es werden steuerbare und beobachtbare Fehler in der Implementierung betrachtet, die durch die gewählte Abstraktion des Modells in der Verifikation entweder Steuerbarkeit, Beobachtbarkeit oder beides verlieren.

Abstraktion ist ein notwendiges Hilfsmittel, um die speicher- und rechenintensiven Verifikationsmethoden auf heutige Systeme überhaupt anwenden zu können. Dabei sollte die Abstraktion stets so gewählt werden, dass alle interessanten (also die zu überprüfenden) Eigenschaften aus der Implementierung erhalten bleiben. Typische Abstraktionen, die durchgeführt werden, sind [325]:

- *Strukturelle Abstraktion*: Hierbei werden interne Details der Struktur einer Implementierung versteckt. Im Extremfall bedeutet dies, dass das System als ein

monolithischer Block erscheint und somit keinerlei Nebenläufigkeit mehr erkennen lässt. Dies hat zur Folge, dass nicht mehr alle möglichen Verschränkungen in der Ausführung potentiell parallel arbeitender Teilsysteme in der Verifikation berücksichtigt werden können. Man beschränkt sich also auf genau eine Ausführungsreihenfolge. Ist diese fehlerfrei, ist dies noch keine Garantie für die Fehlerfreiheit anderer Ausführungsreihenfolgen.
- *Funktionale Abstraktion*: Hierbei wird Teilfunktionalität, wie z. B. die Ausnahmebehandlung eines Systems, versteckt. Somit kann dann zwar die Grundfunktionalität auf einem deutlich kleineren System getestet werden. Allerdings können Fehler in der Ausnahmebehandlung nicht mehr erkannt werden.
- *Datenabstraktion*: Bei der Datenabstraktion werden Datentypen, die in der Implementierung Verwendung finden, durch abstrakte Datentypen in der Verifikation ersetzt. Dies können beispielsweise ganze Zahlen anstelle von Bitvektoren oder Fließpunktzahlen anstelle von Fixpunktdarstellungen sein. Bei einer solchen Abstraktion können evtl. Bereichsüber- bzw. Bereichsunterschreitungen nicht mehr erkannt werden. Auf der anderen Seite werden rechenintensive Überprüfungen der Originaldefinitionsbereiche verhindert.
- *Zeitabstraktion*: Schließlich lässt sich Zeit mit unterschiedlicher Granularität repräsentieren. Auf Logikebene beispielsweise als Gatterlaufzeiten. In einem synchronen Entwurf werden typischerweise Taktzyklen als Zeitbasis verwendet. Auf höheren Abstraktionsebenen sind Verzögerungszeiten auf Grundblockebene, Synchronisationen zwischen Prozessen oder sogar Zeitintervalle gängige Zeitabstraktionen.

Die hier vorgestellten Abstraktionen sind orthogonal zueinander und lassen sich miteinander kombinieren. Insbesondere können den Abstraktionsebenen im Doppeldachmodell spezielle Ausprägungen der oben genannten Abstraktionen zugeordnet werden (siehe [426]).

Betrachtet man, wie oben beschrieben, jeden möglichen Testfall als einen Punkt im Eingabebereich, so schränkt Abstraktion den steuerbaren Eingabebereich ein. Analog kann man jedem Testfall einen Punkt im Ausgabebereich zuordnen. Durch Abstraktion wird der beobachtbare Ausgabebereich eingeschränkt. Zusammenfassend kann man sagen, dass die Aktivierung eines Fehler im *steuerbaren Eingabebereich* als auch im *beobachtbaren Ausgabebereich* des SUV liegen muss, so dass dieser überprüfbar wird. Dies ist in Abb. 3.6 dargestellt.

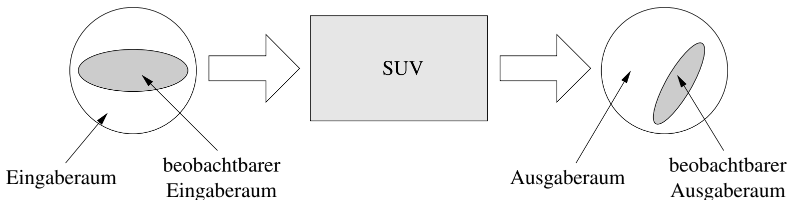


Abb. 3.6. Einfluss der Abstraktion auf die Beobachtbarkeit und Steuerbarkeit von Fehlern

Zusicherungsbasierte Eigenschaftsprüfung

In den letzten Jahren hat sich eine neue hybride Verifikationsmethodik etabliert, die als *zusicherungsbasierte Eigenschaftsprüfung* (engl. *assertion-based verification*) bezeichnet wird. Ziel der zusicherungsbasierten Eigenschaftsprüfung ist es, simulative und formale Verifikation einfacher zu kombinieren. Grundlage hierzu bilden *Zusicherungen* (engl. *assertions*), die in einer formalen Sprache definiert werden. Zusicherungen lassen sich in simulativen sowie formalen Verifikationsmethoden verwenden. Die Verwendung von Zusicherungen in der Simulation bietet zwei wesentliche Vorteile:

1. Eine einzelne Zusicherung kann, wie in der formalen Verifikation, mehrere Punkte im Ausgaberaum abdecken.
2. Zusicherungen sind eng mit der Implementierung gekoppelt und können interne Signale überwachen.

Insbesondere der letzte Punkt ist im Zusammenhang mit beobachtbaren ud steuerbaren Fehlern wichtig, da Zusicherungen in der Simulation einen von außen sichtbaren Zustand besitzen, d. h. die Verletzung einer Zusicherung wird immer angezeigt. Es stellt sich somit nicht mehr die Frage, wie ein Fehler an die primären Ausgänge eines Systems übertragen werden kann. Somit kann durch die Verwendung von Zusicherungen die Beobachtbarkeit von Fehlern eines Systems stark erhöht werden. Weiterhin wird die Fehlerlokalisierung vereinfacht, da nicht nur Ausgaben des Systems überprüft werden können.

3.3 Gesteuerte zufällige Simulation

Zufällige Simulation verspricht eine hohe Wiederverwendbarkeit und einen hohen Grad an Automatisierung. Aus diesem Grund ist zufällige Simulation heutzutage eine wichtige Verifikationsmethode, unabhängig von der Implementierungsform (Hardware, Software oder Hardware/Software) und der gewählten Abstraktionsebene. Wegen der zentralen Bedeutung werden die Prinzipien der zufälligen Simulation im Folgenden näher betrachtet. Die Diskussion erfolgt in Anlehnung an [472].

Eine Simulation mit rein zufällig generierten Testfällen würde dazu führen, dass sehr häufig uninteressante Aspekte eines Systems betrachtet werden. Dies liegt daran, dass große Bereiche von Eingaben zu einem ähnlichen, für die Verifikation keinen Mehrwert bringenden Verhalten führen. Hierin liegt der Grund, die zufällige Generierung von Testfällen einzuschränken. Im Englischen spricht man auch von einer *constrained random simulation*. Es handelt sich also um eine *gesteuerte zufällige Simulation*.

Die Steuerung der Simulation erfolgt anhand der Spezifikation. Dies umfasst sowohl die *Beschränkungen* an die zu generierenden Testfälle als auch die Ermittlung der bisher erzielten Verifikationsvollständigkeit. Da die Vollständigkeit permanent während der Simulation bestimmt wird, können die Beschränkungen an die Testfälle dynamisch angepasst werden. Die Generierung der Testfalleingaben unter Berücksichtigung der Beschränkungen erfolgt dann aber zufällig.

Beschränkung der Testfälle

Beschränkungen und Zusicherungen sind zwei Seiten der selben Medaille. Beide sind formale Spezifikationen von Eigenschaften. Zusicherungen sind diejenigen *Eigenschaften*, die es während der Verifikation zu überprüfen gilt. Beschränkungen hingegen sind *Bedingungen*, die während der Verifikation erfüllt sein müssen. Bei der gesteuerten, zufälligen Simulation geben Beschränkungen also an, welche Kombinationen von Eingaben wann erzeugt werden dürfen.

Prinzipiell können zwei Arten von Beschränkungen unterschieden werden:

1. *Umgebungsbeschränkungen* spezifizieren die Protokolle der Schnittstellen des Systems mit seiner Umgebung.
2. *Testvorschriften* sind diejenigen Beschränkungen, die verwendet werden, um die Simulation zu interessanten Testfällen zu führen.

Mit anderen Worten: Man kann sagen, dass gesteuerte zufällige Simulation nur dann sinnvoll ist, wenn diese sowohl die Randbedingungen der Umgebung berücksichtigt als auch in der Lage ist, interessantere Testfälle unter Berücksichtigung von Testvorschriften zu generieren. Dies ist in Abb. 3.7 dargestellt. Der Eingaberaum wird durch die Umgebungsbeschränkungen auf einen akzeptierten Eingaberaum reduziert. Testfalleingaben aus dem akzeptierten Eingaberaum können zur Stimulation des SUV verwendet werden. Man erkennt allerdings, dass es kleine Bereiche am Rand gibt, die eventuell interessanter für die Verifikation sein könnten. In diesen Bereichen einen Testfall zufällig zu generieren ist relativ unwahrscheinlich, da eine viel größere Anzahl gültiger Testfälle außerhalb des interessierenden Bereichs existiert. Durch die Verwendung zusätzlicher Testvorschriften kann man allerdings erreichen, dass Testfälle genau aus diesen Randbereichen erzeugt werden.

Einhalten der Beschränkungen

Liegen alle Beschränkungen, inklusive Umgebungsbeschränkungen und Testvorschriften, für einen neuen Simulationsschritt vor, so muss eine neue Belegung für die Eingänge des SUV gefunden werden, die alle Beschränkungen erfüllt. Dies wird auch als Lösen eines sog. *Beschränkungs-Erfüllbarkeitsproblem* (engl. *Constraint Satisfaction Problem, CSP*) bezeichnet. Ein CSP besteht dabei aus:

- einer Menge an Variablen,
- einer nichtleeren Wertemenge für jede Variable,
- einer Menge an Beschränkungen, welche die Wertebelegungen für die Variablen beschränken, und
- optional einer Kostenfunktion, welche die Qualität einer erfüllenden Wertebelegung bestimmt.

Das Lösen eines CSP besteht in der Suche einer Wertebelegung aller Variablen, so dass alle Beschränkungen gleichzeitig erfüllt sind, oder es wird gezeigt, dass keine solche Belegung existiert. Abhängig von der Art der Definitionsbereiche und der Art der Beschränkungen kann das allgemeine CSP in Klassen unterteilt werden, für die spezielle Lösungsansätze existieren:

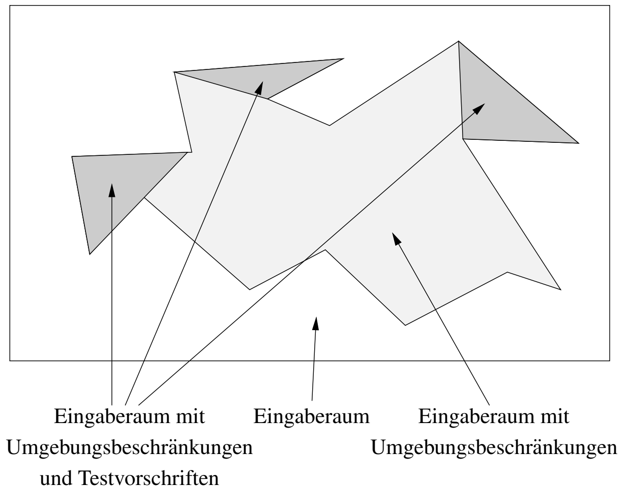


Abb. 3.7. Einfluss von Umgebungsbeschränkungen und Testvorschriften [472]

- *Binäre Entscheidungsdiagramme* (engl. *Binary Decision Diagrams, BDDs*) sind graphenbasierte Datenstrukturen zur Repräsentation Boolescher Funktionen. Lassen sich also die Beschränkungen als Boolesche Funktionen darstellen, kann das CSP durch Konstruktion eines BDDs gelöst werden, das alle erfüllenden Variablenbelegungen beschreibt.
- *SAT-Solver* und *automatische Testfallgenerierung* (siehe auch Abschnitt 6.1.2) sind Suchverfahren für Belegungen aussagenlogischer Formeln. Der Unterschied zwischen beiden Verfahren besteht darin, dass SAT-Solver auf einer Menge von Klauseln operieren, während die automatische Testfallgenerierung auf Schaltungsbeschreibungen arbeitet. Allerdings lassen sich die Schaltungsbeschreibungen in eine Klauselmenge und umgekehrt transformieren. Im Gegensatz zu BDDs wird aber bei diesen Suchverfahren immer nur eine erfüllende Variablenbelegung generiert.
- *Lineare Programmierung* löst ein lineares Ungleichungssystem für reellwertige Variablen mit linearer Zielfunktion. Eine Spezialform ist die sog. *ganzzahlige lineare Programmierung* (engl. *Integer Linear Programming, ILP*), bei der die Variablen ganzzahlig sein müssen. Dies kann weiter auf die Definitionsmenge $\{0,1\}$ eingeschränkt werden, hierdurch entstehen sog. *0-1-ILPs*.
- Die engl. *Constraint Logic Programming* (CLP) verbindet CSP und Logikprogrammierung. Somit kann CLP sowohl lineare als auch nichtlinear Beschränkungen sowie aussagenlogische als auch temporallogische Formeln lösen. Das eigentliche Lösen erfolgt dabei mittels SAT-Solvern, (I)LP-Solvern oder Intervallarithmetik.

Der erste Ansatz mittels binärer Entscheidungsdiagramme wird als *Offline*-Ansatz zum Lösen von CSPs bezeichnet, da bereits nach Konstruktion des Entschei-

dungsdiagramms alle Lösungen bekannt sind, und somit die Testfälle generiert werden können. Die restlichen drei Verfahren werden im Gegensatz dazu als *Online*-Ansätze bezeichnet, da diese während der Simulation für jede neue Testfalleingabe immer wieder nach neuen Lösungen gefragt werden.

In beiden Fällen stellt die Unerfüllbarkeit der Beschränkungen ein schwieriges Problem dar. Tritt dieses Problem auf, kann die Verifikation nicht fortgesetzt werden. Es muss zunächst eine Diagnose der Beschränkungen durchgeführt werden, um festzustellen, ob ein Fehler bei der Erstellung der Umgebungsbeschränkungen oder bei der Erstellung der Testvorschriften vorliegt.

Die Randomisierung

Durch das Lösen des CSP ist sichergestellt, dass lediglich gültige Testfalleingaben generiert werden. Daneben stellt sich allerdings die Frage, wie auch *gute* Testfalleingaben erzeugt werden können?

Eine Minimalanforderung an gute Testfälle sollte bei einer zufälligen Simulation sein, dass jeder gültige Testfall mit der selben Wahrscheinlichkeit ausgewählt werden kann. Diese Annahme gilt allerdings nur solange, wie jeder Testfall auch den gleichen Beitrag zu dem gewählten Vollständigkeitskriterium beiträgt. Dies ist offensichtlich nicht allgemeingültig. Zum einen hat das zu prüfende System ein spezielles Ein-/Ausgabeverhalten, zum anderen betonen verschiedene Überdeckungsmaße auch unterschiedliche Testfälle. Vor diesem Hintergrund ist es besser, von einer gewichteten Verteilung für die Häufigkeit von Testfällen auszugehen und speziell für diese Verteilung Fairness bei der Auswahl der Testfälle zu schaffen. Als Beispiel sei hier lediglich die Verifikation von Einheiten für arithmetische Berechnungen genannt. Bei diesen ist es oft vorteilhaft, minimale und maximale Werte bei der Variablenbelegung stärker zu gewichten, da diese erfahrungsgemäß eher Fehler provozieren.

Neben der Fairness bei der Erzeugung von Testfällen muss allerdings berücksichtigt werden, dass die Simulationsläufe nachvollziehbar und reproduzierbar sein müssen. Ohne Kenntnis der fehlererzeugenden Testfalleingabe könnte sonst die Fehlerlokalisierung unangemessen aufwendig werden. Durch die Verwendung von Pseudo-Randomisierungsalgorithmen wird nahezu eine Gleichverteilung bei der Testfallauswahl erreicht und die Reproduzierbarkeit durch Speicherung des Anfangszustands des Zufallszahlengenerators garantiert.

Das Lösen der Beschränkungen und die zufällige Generierung von Testfällen sind eng miteinander gekoppelt. Um hier den Aufwand bei der Erstellung von Testbenches zu minimieren, bietet es sich an, auf existierende Lösungen zurückzugreifen. Die wichtigsten Vertreter in diesem Bereich sind SystemVerilogs Random Constraint language (SVRC), SystemC Verification Library (SCV) und die Testbenchsprache e.

4

Äquivalenzprüfung

Aufgabe der Äquivalenzprüfung ist es, zu zeigen, dass das Verhaltensmodell einer Spezifikation und das Strukturmodell einer Implementierung die gleiche Funktion repräsentieren (siehe Abb. 4.1). Hierbei kann eine Prüfung zwischen zwei Beschreibungen auf der selben oder unterschiedlichen Abstraktionsebenen erfolgen. Obwohl auf vielen Abstraktionsebenen die Synthese mittlerweile automatisiert abläuft [426], ist eine Äquivalenzprüfung häufig notwendig, da die verwendeten Synthesewerkzeuge nicht zwangsläufig fehlerfrei funktionieren. Zum anderen wird Äquivalenzprüfung eingesetzt, um zu zeigen, dass das Verhalten zweier Implementierungen identisch ist. In diesem Fall dient eine der Implementierungen als Spezifikation (Referenz).

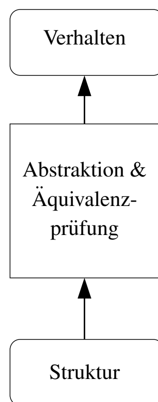


Abb. 4.1. Äquivalenzprüfung

Da bei der Äquivalenzprüfung jedes mögliche Verhalten sowohl der Spezifikation als auch der Implementierung zu überprüfen ist, sind Verfahren zur Äquivalenzprüfung sehr rechenintensiv. Deshalb wird die Äquivalenzprüfung typischerweise

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_4, © Springer-Verlag Berlin Heidelberg 2010

lediglich auf kleine und mittelgroße Systeme angewendet. Sie ist daher vorwiegend auf tieferen Abstraktionsebenen zu finden.

Abbildung 4.2 visualisiert die Äquivalenzprüfung von zwei Implementierungen. In diesem Fall wurde jede Implementierung von unterschiedlichen Entwicklern basierend auf der selben Spezifikation entworfen. Ein solches Vorgehen findet man beispielsweise in der Luft- und Raumfahrttechnik, um eine redundante Auslegung von Systemen zu erreichen. Ein Problem, welches hierbei adressiert wird, ist, dass unterschiedliche Entwickler die selbe Spezifikation verschieden umsetzen. So lässt sich in einem Vergleich der Implementierungen ein Fehler detektieren. Ist also eine Spezifikation unvollständig oder mehrdeutig, ist die Chance größer, dass mindestens ein Entwickler Teile des System richtig implementiert.

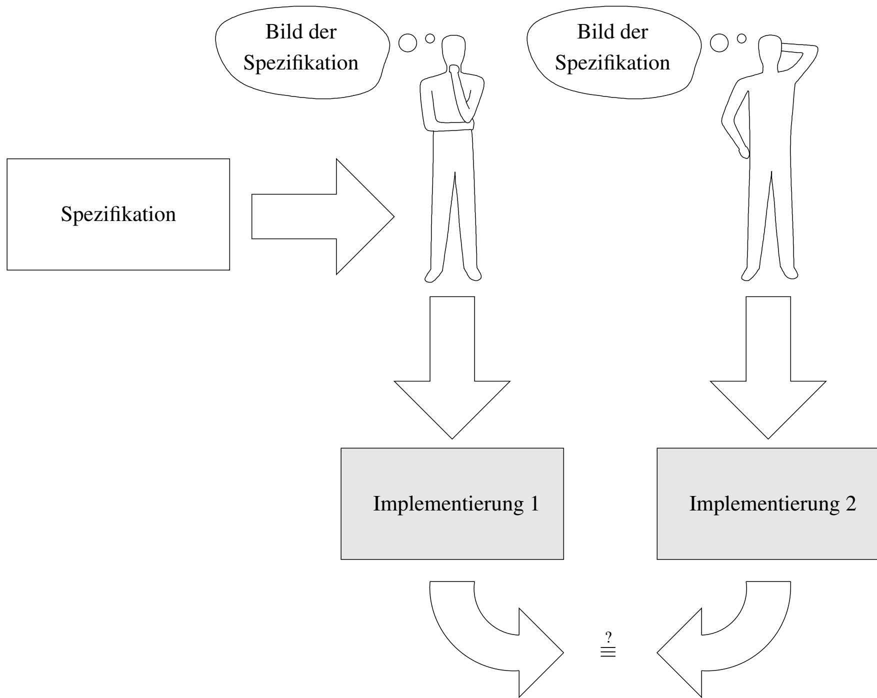


Abb. 4.2. Äquivalenzprüfung [305]

Zur Klassifikation von Äquivalenzprüfungsverfahren können sowohl die angewandten Methoden als auch die betrachteten Systeme dienen. Die Methoden werden als *implizit*, *explizit* oder *strukturell* bezeichnet. Implizite Äquivalenzprüfung setzt stets ein formales Modell voraus, während explizite Verfahren sowohl formal also auch simulativ sein können. Strukturelle Verfahren verwenden implizite oder explizite Methoden. Bei Systemen unterscheidet man zwischen Systemen ohne Speicher

und Systemen mit Speicher. Verfahren zur Äquivalenzprüfung, die auf Systeme ohne Speicher angewendet werden, nennt man *kombinatorisch*. Äquivalenzprüfungsverfahren für Systeme mit Speicher werden als *sequentiell* bezeichnet. Auf wichtige Methoden zur Äquivalenzprüfung, klassifiziert nach impliziten, expliziten, sequentiellen und strukturellen Methoden, wird im Folgenden näher eingegangen.

4.1 Implizite Äquivalenzprüfung

Implizite Verfahren zur Äquivalenzprüfung basieren auf der Idee, dass (eingeschränkte) Funktionen *eindeutig* repräsentiert werden können. Besitzen zwei Funktionen in einer solchen Repräsentation den selben Repräsentanten, so sind sie äquivalent. Dieses Vorgehen wird *implizite Äquivalenzprüfung* genannt, da die eigentliche Verifikationsaufgabe, die Äquivalenz zu zeigen, nicht explizit ausgedrückt wird. Im Folgenden wird die implizite, kombinatorische Äquivalenzprüfung anhand von Polynomfunktionen eingeführt. In den Abschnitten 6.1.1 und 6.2 werden spezielle Methoden zur impliziten, kombinatorischen Äquivalenzprüfung von Hardware diskutiert.

4.1.1 Kanonische Funktionsrepräsentationen

Die implizite, kombinatorische Äquivalenzprüfung auf Basis eindeutiger Funktionsrepräsentationen ist eine formale Verifikationsmethode. Sie ist vollständig und somit für das Verifikationsziel eines Beweises geeignet. Um implizite Verfahren zur Äquivalenzprüfung entwickeln zu können, muss zunächst der Begriff der Repräsentation von Funktionen genauer definiert werden. Formal kann dies wie folgt formuliert werden (siehe auch Anhang A.2): Sei $(\mathcal{R},\phi)$ eine Repräsentation der betrachteten Funktionen $f \in \mathcal{F}$ mit $\phi : \mathcal{R} \rightarrow \mathcal{F}$. Dann heißt $(\mathcal{R},\phi)$ *vollständig*, falls ϕ surjektiv ist, d. h. für jedes Element $f \in \mathcal{F}$ existiert ein $r \in \mathcal{R}$, so dass $\phi(r) = f$. $(\mathcal{R},\phi)$ heißt *eindeutig*, falls ϕ injektiv ist, d. h. jede Funktion $f \in \mathcal{F}$ ist höchstens einmal Funktionswert. Falls ϕ bijektiv ist, also surjektiv und injektiv ist, dann ist $(\mathcal{R},\phi)$ eine *kanonische Repräsentation* der Funktionen $f \in \mathcal{F}$. Sei $\phi(r) = f$, dann sagt man, dass r ein *Repräsentant* von f ist und ϕ ist die *Interpretation*. Man beachte, dass für kanonische Repräsentationen von Funktionen, die Repräsentanten einer Funktion stets eindeutig sind.

Beispiel 4.1.1. Viele Berechnungen, die in Systemen durchgeführt werden, lassen sich als *Polynome* beschreiben. Als Beispiel sei hier die diskrete Faltung $\mathrm{conv}_{(x_1,x_2,N)}$ von zwei Signalen x_1 und x_2 in einem endlichen Zeitfenster von N Schritten genannt, die sich wie folgt berechnen lässt:

$$\mathrm{conv}_{(x_1,x_2,N)}(i) := \sum_{k:=0}^{N-1} x_1(k) \cdot x_2(i-k)$$

Für $N = 4$ ist das Blockschaltbild in Abb. 4.3 dargestellt. Die aktuellen Signale $x_1(3)$ und $x_2(3)$ liegen zusammen mit den drei vorherigen Signalen $x_i(2), x_i(1)$ und $x_i(0)$ als Eingänge des Systems an. Die vier neusten Ergebnisse $y(4),\ldots,y(0)$ sind Ausgänge des Systems.

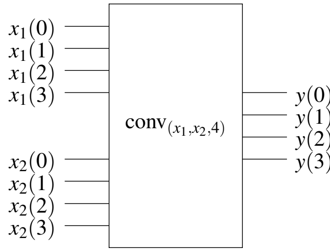


Abb. 4.3. Diskrete Faltung [92]

Allgemein ist ein Polynom ein Ausdruck, der anhand einer oder mehrerer Variablen und Konstanten nur unter Verwendung von Addition (+), Subtraktion (−), Multiplikation (·) und konstanten positiven ganzzahligen Exponenten gebildet wird. Im Fall von n Variablen kann das n-variate Polynom $p(x_1,\ldots,x_n)$ mit $x_1,\ldots,x_n \in \mathbb{R}$ wie folgt dargestellt werden:

$$p(x_1,\ldots,x_n) = \sum_{i:=0}^{m} p_i(x_1,\ldots,x_{n-1}) \cdot x_n^i \tag{4.1}$$

Die individuellen Polynome $p_i(x_1,\ldots,x_{n-1})$ können dabei als Koeffizienten der Variable x_n angesehen werden. Diese Repräsentation wird als engl. *sparse recursive representation* bezeichnet und stellt nur Polynome ungleich null dar. Sie ist für eine gegebene Variablenordnung kanonisch.

Um eine Funktion $f(x_1,\ldots,x_n)$ in die Form von Gleichung (4.1) zu bringen, kann die *Taylor-Reihen-Entwicklung* verwendet werden. Für eine Funktion $f(x)$ mit einer Variablen x ist die Taylor-Reihen-Entwicklung um den Punkt $x = a$ wie folgt definiert:

$$\begin{aligned} f(x) &:= \sum_{k:=0}^{\infty} \frac{1}{k!}(x-a)^k \cdot f^{(k)}(a) \\ &= f^{(0)}(a) + x \cdot f^{(1)}(a) + \frac{1}{2}x^2 \cdot f^{(2)}(a) + \ldots \end{aligned} \tag{4.2}$$

Hierzu muss die Funktion f stetig sein und alle Ableitungen nach den Variablen $x_1,\ldots,x_n$ in $x_i = a$ existieren. Hierbei beschreibt $f^{(k)}(a)$ die k-te Ableitung der Funktion f ausgewertet an der Position $x = a$. Im Fall von n Variablen sieht die Taylor-Reihen-Entwicklung wie folgt aus:

$$\begin{aligned} f(x_1,\ldots,x_n) := f(x_1,\ldots,x_{n-1},a_n) &+ \left.\frac{\partial f(x_1,\ldots,x_n)}{\partial x_n}\right|_{x_n:=a_n} \cdot (x_n - a_n) \\ &+ \frac{1}{2} \left.\frac{\partial^2 f(x_1,\ldots,x_n)}{\partial x_n^2}\right|_{x_n:=a_n} \cdot (x_n - a_n)^2 \\ &+ \ldots \end{aligned}$$

Durch sukzessive Taylor-Reihen-Entwicklung nach allen Variablen in einer gegebenen Reihenfolge, der sog. *Variablenordnung*, erhält man somit eine *Polynomfunktion* in der Form von Gleichung (4.1).

Beispiel 4.1.2. Das folgende Beispiel stammt aus [92]. Gegeben ist die folgende Polynomfunktion:

$$f(x_1,x_2,x_3) := (x_1+x_2)(x_1+2\cdot x_3) = x_1^2 + x_1\cdot(x_2+2\cdot x_3)+2\cdot x_2\cdot x_3$$

Für die Variablenordnung $x_1 < x_2 < x_3$ ergibt die Taylor-Reihen-Entwicklung für $a_1 = a_2 = a_3 := 0$:

$$\begin{aligned} f(x_1,x_2,x_3) &= x_1^2 + x_1\cdot(x_2+2\cdot x_3)+2\cdot x_2\cdot x_3 \\ &= f(0,x_2,x_3) + x_1 \cdot \left.\frac{\partial f(x_1,x_2,x_3)}{\partial x_1}\right|_{x_1:=0} + \ldots \\ &= 2\cdot x_2\cdot x_3 + x_1\cdot(x_2+2\cdot x_3)+x_1^2 \end{aligned}$$

Der konstante Term ergibt sich durch Einsetzen von null für x_1 und resultiert in $2\cdot x_2\cdot x_3$. Für die erste Ableitung nach x_1 ergibt sich $2\cdot x_1 + x_2 + 2\cdot x_3$. Durch die Substitution $x_1 := 0$ resultiert der Koeffizient $x_2 + 2\cdot x_3$. Für die zweite Ableitung nach x_1 ergibt sich $\frac{1}{2}\cdot 2$ und somit der Koeffizient 1.

Im nächsten Schritt erfolgt die Taylor-Reihen-Entwicklung dieser Koeffizienten nach x_2 für $a_2 := 0$. Da der Koeffizient für die zweite Ableitung nach x_1 bereits eine Konstante ist, muss dieser nicht weiter berücksichtigt werden. Zunächst wird $f(0,x_2,x_3)$ betrachtet:

$$\begin{aligned} f(0,x_2,x_3) &= 2\cdot x_2\cdot x_3 \\ &= f(0,0,x_3) + x_2 \cdot \left.\frac{\partial f(0,x_2,x_3)}{\partial x_2}\right|_{x_2:=0} + \ldots \\ &= 0 + x_2\cdot 2\cdot x_3 \end{aligned}$$

Der konstante Koeffizient ergibt sich durch $x_2 := 0$, wobei dieser zu null evaluiert. Der Koeffizient für die erste Ableitung liefert $2\cdot x_3$. Alle weiteren Ableitungen sind null. Als nächstes wird der Koeffizient für die erste Ableitung nach x_1 betrachtet, also $x_2 + 2\cdot x_3$:

$$\begin{aligned} \left.\frac{\partial f(x_1,x_2,x_3)}{\partial x_1}\right|_{x_1:=0} &= x_2 + 2\cdot x_3 \\ &= 2\cdot x_3 + 1\cdot x_2 \end{aligned}$$

Aus beiden Rechnungen bleibt lediglich ein nichtkonstanter Term übrig ($2\cdot x_3$). Da dieser bereits als Polynom vorliegt, erübrigt sich die Taylor-Reihen-Entwicklung. Das resultierende Polynom für die Funktion f lautet somit

$$f(x_1,x_2,x_3) = x_1^2 + x_1\cdot x_2 + 2\cdot x_1\cdot x_3 + 2\cdot x_2\cdot x_3\,.$$

4.1.2 Taylor-Expansions-Diagramme

Endliche Taylor-Reihen lassen sich mit Hilfe sog. *Taylor-Expansions-Diagramme* (engl. *Taylor Expansion Diagram, TED*) darstellen [92]. Hierbei wird davon ausgegangen, dass die Taylor-Reihen-Entwicklung um den Nullpunkt $a_1 = \cdots = a_n := 0$ erfolgt. Ein TED ist wie folgt definiert:

Definition 4.1.1 (TED). *Ein* TED *ist ein gerichteter azyklischer Graph $G = (V,E)$ mit Knotenmenge V und Kantenmenge E sowie den folgenden Eigenschaften:*

- *G ist ein Baum,*
- *V ist eine Partition von* Terminalknoten V_T *und* Nichtterminalknoten V_N,
- *eine Funktion* index : $V_\mathrm{N} \to \{1,\ldots,n\}$ *weist jedem Nichtterminalknoten $v \in V_\mathrm{N}$ eine Variable $x_{\mathrm{index}(v)} \in \{x_1,\ldots,x_n\}$ zu,*
- *für jeden Nichtterminalknoten $v \in V_\mathrm{N}$ weist eine Funktion* child : $V_\mathrm{N} \times \mathbb{N} \to V$ *dem Knoten v seine Nachfolger zu*
- *eine Funktion* weight : $V_\mathrm{N} \times \mathbb{N} \to \mathbb{R}$ *weist jeder ausgehenden Kante eines Nichtterminalknotens einen Wert zu.*
- *eine Funktion* value : $V_\mathrm{T} \to \mathbb{R}$ *weist jedem Terminalknoten einen Wert zu.*

Die *Interpretation* eines TED erfolgt rekursiv. Die Polynomfunktion $f_v = \phi(v)$, die durch den Knoten v repräsentiert wird, ergibt sich zu:

$$\phi(v) := \begin{cases} \mathrm{value}(v) & \text{falls } v \in V_\mathrm{T} \\ \sum_{i:=1}^{k} \mathrm{weight}(v,i) \cdot \phi(\mathrm{child}(v,i)) \cdot x_{\mathrm{index}(v)}^{i-1} & \text{falls } v \in V_\mathrm{N} \end{cases}$$

Beispiel 4.1.3. Als Fortsetzung zu Beispiel 4.1.2 soll das TED für die Polynomfunktion $f(x_1,x_2,x_3) = (x_1 + x_2)(x_1 + 2 \cdot x_3)$ mit der Variablenordnung $x_1 < x_2 < x_3$ entwickelt werden. Die Konstruktion ist in Abb. 4.4 zu sehen. Die Taylor-Reihen-Entwicklung nach x_1 ist in Abb. 4.4a) dargestellt. Der Quellknoten ist entsprechend mit x_1 beschriftet. Kanten, die als durchgezogene Linien dargestellt sind, zeigen auf den konstanten Koeffizienten. Gestrichelte Kanten zeigen auf den Koeffizienten für die erste Ableitung. Gepunktete Kanten auf Koeffizienten für die zweite Ableitung. Da die zweite Ableitung nach x_1 die Konstante 1 ergibt, zeigt die gepunktete Kante auf den entsprechenden Terminalknoten. Alle weiteren Koeffizienten in diesem Beispiel sind null und werden nicht im TED dargestellt.

Abbildung 4.4b) zeigt den zweiten Schritt bei der Taylor-Reihen-Entwicklung, die Entwicklung nach x_2. In diesem Fall sind zwei Koeffizienten unabhängig von allen weiteren Variablen, also Konstanten, und somit zeigen die zugehörigen Kanten auf Terminalknoten. Weiterhin kann man sehen, dass zwei Koeffizienten den selben Ausdruck ergeben ($2 \cdot x_3$). Somit muss nur dieser Ausdruck weiter entwickelt werden. Das Ergebnis der Taylor-Reihen-Entwicklung ist in Abb. 4.4c) zu sehen.

4.1.3 Reduktion und Normalisierung von TEDs

Häufig ist es möglich, die Größe eines Taylor-Expansions-Diagramm zu reduzieren. Die Größe eines TED G ist die Anzahl seiner Knoten $\mathrm{size}(G) := |V|$. Analog

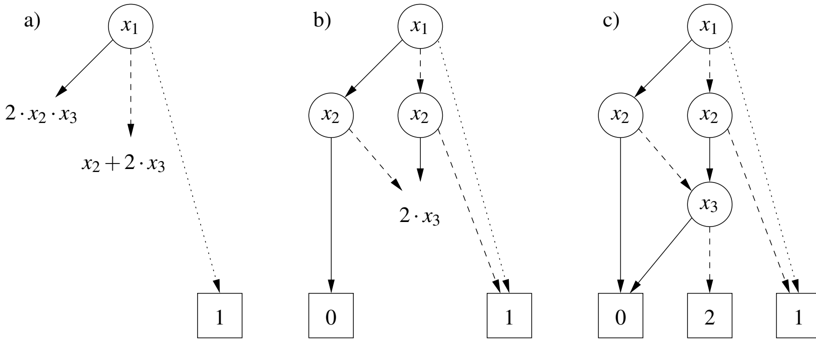


Abb. 4.4. Konstruktion eines TED für die Polynomfunktion $(x_1 + x_2)(x_1 + 2 \cdot x_3)$ [92]

zu OBDDs (siehe Anhang B.2) können TEDs reduziert werden, indem redundante Knoten eliminiert und isomorphe Teilgraphen verschmolzen werden. Für redundante Knoten gilt, dass diese aus dem Graphen entfernt werden können und alle ihre eingehenden Kanten auf ihren Nachfolger umgelenkt werden können. Die repräsentierte Polynomfunktion bleibt dabei unverändert. Hieraus ergibt sich für TEDs die folgende *Eliminationsregel*:

Definition 4.1.2 (Redundanter TED-Knoten). *Ein TED-Knoten v heißt* redundant*, wenn außer dem ersten alle seine Nachfolger der Terminalknoten mit dem Wert* 0 *sind, d. h.* $\forall k > 1 : \text{value}(\text{child}(v,k)) = 0$ *und* $\text{value}(\text{child}(v,1)) \neq 0$.

Redundante TED-Knoten haben also lediglich einen konstanten Koeffizienten. Da dieser nicht mit der Variablen des Knotens gewichtet wird, ist der Knoten redundant und kann entfernt werden. Für den Fall, dass der erste Nachfolger $\text{child}(v,1)$ ebenfalls auf den Terminalknoten mit dem Wert null zeigt, ist die repräsentierte Funktion die Konstante 0. Auch in diesem Fall darf v entfernt werden und alle eingehenden Kanten auf den Terminalknoten mit dem Wert 0 umgelenkt werden. Aus der obigen Argumentation folgt ebenfalls, dass der Terminalknoten mit dem Wert 0 nicht benötigt wird und alle eingehenden Kanten gelöscht werden können, ohne die repräsentierte Funktion zu verändern. Der Terminalknoten mit dem Wert 0 wird dann ebenfalls gelöscht.

Neben redundanten Knoten, kann ein TED auch redundante Teilgraphen enthalten. Existieren zwei Knoten v und v' im TED mit identischer Interpretation, d. h. $\phi(v) = \phi(v')$, so ist der Teilgraph mit Quellknoten v' *redundant*. Solche Teilgraphen sind *isomorph* (siehe Anhang B.2). Umgangssprachlich bedeutet dies, dass beide Teilgraphen mit Quellknoten v und v' die selbe Struktur und die selben Attribute besitzen. Es existiert also eine Eins-zu-Eins-Abbildung zwischen den Knoten- und Kantenmengen der Teilgraphen, so dass die Adjazenz von Knoten, die Kantenbeschriftung und die Werte der Terminalknoten erhalten bleiben. Existieren isomorphe Teilgraphen in einem TED, können alle bis auf einen dieser Teilgraphen gelöscht werden und alle eingehenden Kanten in den Quellknoten werden auf den einen ver-

bleibenden Teilgraphen umgelenkt. Dieses Vorgehen wird auch als *Verschmelzungsregel* bezeichnet. Die Anwendung der Eliminationsregel und Verschmelzungsregel sind in Abb. 4.5 dargestellt.

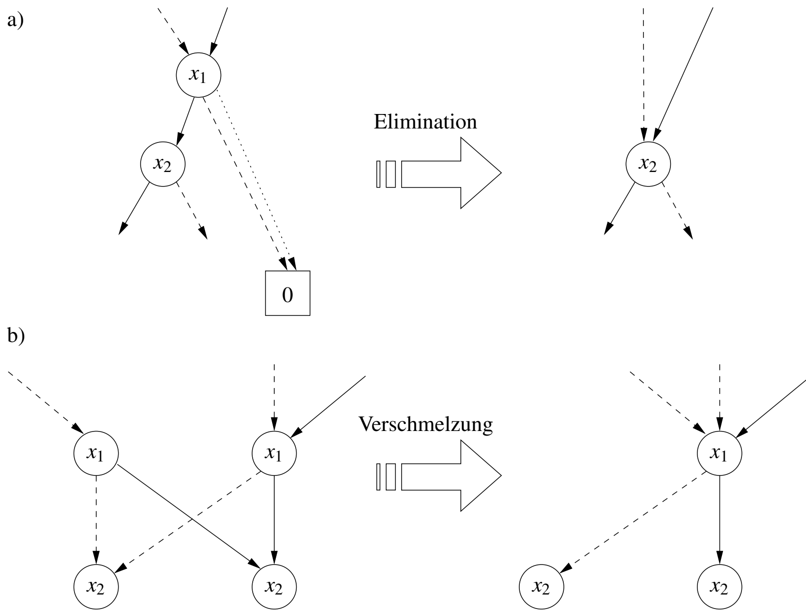


Abb. 4.5. a) Eliminationsregel und b) Verschmelzungsregeln zur Reduktion von TEDs

Definition 4.1.3 (Reduziertes TED). *Ein Taylor-Expansions-Diagramm* $G = (V,E)$ *heißt* reduziert*, sofern es keine redundanten Knoten und keine zwei unterschiedlichen Knoten* $v, v' \in V$ *enthält, so dass* $\phi(v) = \phi(v')$ *ist.*

Mit anderen Worten: Es sind alle Knoten $v \in V$ einzigartig. Ein TED G heißt weiterhin *geordnet*, falls auf jedem Pfad in G vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten.

Offensichtlich können nichtreduzierte TEDs keine kanonische Repräsentation für Polynomfunktionen darstellen. Um zu einer kanonischen Repräsentation zu gelangen, ist neben der Reduktion aber auch eine *Normalisierung* durchzuführen. Im Folgenden werden lediglich Polynomfunktionen mit ganzzahligen Koeffizienten betrachtet. Die Normalisierung beginnt mit dem Verschieben der von null verschiedenen Werte $\text{value}(v')$ von Terminalknoten $v' \in V_\text{T}$ auf die Kanten $e \in \{e' = (v,v') \in E \mid v' \in V_\text{T}\}$. Diese werden zu Kantengewichten $\text{weight}(v,i)$ und gehen damit multiplikativ in die Interpretation des TED ein. Nach diesem Schritt existieren lediglich Terminalknoten $v' \in V_\text{T}$ mit den Werten $\text{value}(v') = 0$ und $\text{value}(v') = 1$. Terminalknoten mit dem Wert null und deren eingehende Kanten können gelöscht werden, da

diese ebenfalls multiplikativ in die Interpretation eingehen. Existieren mehrere Terminalknoten mit dem selben Wert, können diese verschmolzen werden, da es sich um isomorphe Teilgraphen handelt. Es existiert anschließend lediglich ein Terminalknoten v' mit dem Wert $\text{value}(v') = 1$.

In weiteren Schritten können die Kantengewichte in Abhängigkeit von ihrem Wert weiter in Richtung Quellknoten propagiert werden. Anschließend werden isomorphe Teilgraphen verschmolzen. Dies wird anhand eines Beispiels erläutert [92].

Beispiel 4.1.4. Gegeben ist die Polynomfunktion $f(x_1,x_2,x_3) := x_1^2 \cdot (x_2+x_3) + 5 \cdot x_1 \cdot (x_2+x_3) + 6 \cdot (x_2+x_3)$. Das zugehörige TED mit Variablenordnung $x_1 < x_2 < x_3$ ist in Abb. 4.6a) dargestellt. Das Ergebnis nach dem ersten Schritt der Normalisierung ist in Abb. 4.6b) zu sehen. In diesem Schritt wurde der Terminalknoten mit dem Wert 0 gelöscht. Anschließend werden die Kantengewichte in Richtung Quellknoten des TED propagiert. Hierbei kann lediglich der größte gemeinsame Teiler der Gewichte aller ausgehenden Kanten aus einem Knoten v als Faktor über v auf die eingehenden Kanten in v propagiert werden. Hierzu werden Kantengewichte der eingehenden kanten mit dem Faktor multipliziert und die Kantengewichte der ausgehenden Kanten entsprechend dividiert.

In Abb. 4.6c) sieht man, dass die Kantengewichte 5 und 6 bis zu den ausgehenden Kanten des Quellknoten propagiert werden können. Nach diesem Schritt erkennt man, dass alle drei Teilgraphen mit Quellknoten x_2 isomorph sind, und so der TED mittels Verschmelzungsregel weiter reduziert werden kann (Abb. 4.6d)).

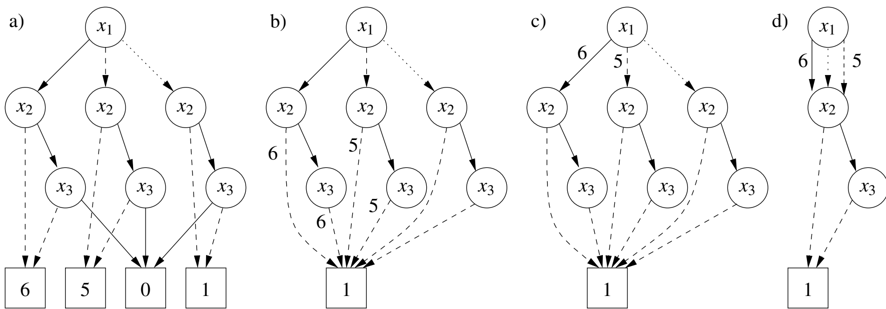


Abb. 4.6. Normalisierung des TED für $x_1^2 \cdot (x_2+x_3) + 5 \cdot x_1 \cdot (x_2+x_3) + 6 \cdot (x_2+x_3)$ [92]

Definition 4.1.4 (Normalisiertes TED). *Ein reduziertes, geordnetes TED $G = (V,E)$ heißt* normalisiert*, wenn*

- *die Kantengewichte auf allen ausgehenden Kanten eines Knotens $v \in V$ teilerfremd sind,*
- *kein Terminalknoten und kein Kantengewicht den Wert* 0 *hat und*
- *es genau einen Terminalknoten gibt und dieser den Wert* 1 *hat.*

Um die Teilerfremdheit zu garantieren, wird eine zusätzliche eingehende Kante auf den Quellknoten eines TED mit zusätzlichem Kantengewicht gezeichnet.

4.1.4 Kanonizität von TEDs

Normalisierte, reduzierte, geordnete Taylor-Expansions-Diagramme sind eine kanonische Repräsentation von Polynomfunktionen, die durch eine endliche Taylor-Reihe dargestellt werden können. Um dies zu zeigen, wird zunächst das *Taylorsche Theorem* wiederholt.

Theorem 4.1.1 (Taylorsches Theorem). *Sei $f(x)$ eine Polynomfunktion mit Definitionsbereich $\mathbb{R}$ und $a \in \mathbb{R}$. Dann existiert genau eine Taylor-Reihe entwickelt nach $x := a$ und diese kann nach Gleichung (4.2) bestimmt werden.*

Dieses Theorem besagt, dass eine Taylor-Reihe entwickelt im Punkt a eindeutig ist. Der Beweis des Theorems basiert darauf, dass die Ableitungen einer Polynomfunktion in einem Punkt eindeutig sind. Mit diesem Theorem und den Eigenschaften von normalisierten, reduzierten und geordneten TEDs kann die Kanonizität dieser Repräsentation von Polynomfunktionen bewiesen werden [92].

Theorem 4.1.2 (Kanonizität von TEDs). *Für* jede *multivariate Polynomfunktion f mit ganzzahligen Koeffizienten existiert ein* eindeutiges *normalisiertes, reduziertes und geordnetes Taylor-Expansions-Diagramm. Jedes andere geordnete TED mit gleicher Variablenordnung, welches f repräsentiert, enthält mehr Knoten.*

Mit anderen Worten: Ein normalisiertes, reduziertes und geordnetes TED ist *minimal* und *kanonisch*.

Beweis. Der Beweis des Theorem folgt den Beweisen für ROBDDs [62]. Im Folgenden werden die Eigenschaften *Eindeutigkeit*, *Kanonizität* und *Minimalität* bewiesen.

Eindeutigkeit: Ein reduziertes TED besitzt per definitionem weder redundante Knoten noch isomorphe Teilgraphen. Weiterhin werden nach der Normalisierung alle gemeinsamen Teilausdrücke durch die weitere Anwendung der Verschmelzungsregel von Teilfunktionen gemeinsam verwendet. Durch das Taylorsche Theorem gilt, dass alle Knoten in einem normalisierten, reduzierten und geordneten TED eindeutig und notwendig sind. Somit ist das normalisierte, geordnete und reduzierte TED selbst ein eindeutiger Repräsentant der dargestellten Polynomfunktion.

Kanonizität: Im Folgenden wird gezeigt, dass die rekursive Evaluierung der einzelnen Terme der Taylor-Reihen-Entwicklung eindeutig durch die Nichtterminalknoten des TED repräsentiert sind. Es gilt, dass für Polynomfunktionen die Taylor-Reihen-Entwicklung in einem gegebene Punkt endlich und entsprechend dem Taylorschen Theorem eindeutig ist. Weiterhin entspricht jeder Term der Taylor-Reihe den sukzessiven Ableitungen der Funktion im gegebenen Punkt. Diese Ableitungen in dem gegebenen Punkt sind eindeutig. Jeder Knoten im TED repräsentiert eindeutig die (Teil-)Funktion, die in diesem Knoten berechnet wird, da die Knoten den rekursiv berechneten Ableitungen entsprechen. Da jeder Knoten in einem normalisierten, reduzierten und geordneten TED bedeutend ist und eindeutig die berechnete

Teilfunktion repräsentiert, sind TEDs kanonische Repräsentationen von Polynomfunktionen.

Minimalität: Der Beweis der Minimalität eines normalisierten und reduzierten TED bezüglich einer gegeben Variablenordnung wird per Widerspruch bewiesen. Sei G ein normalisiertes, reduziertes und geordnetes TED, welches (kanonisch) die Polynomfunktion f repräsentiert. Angenommen es existiert ein anderes TED G', welches die selbe Variablenordnung wie G verwendet, und ebenfalls die Polynomfunktion f repräsentiert, aber kleiner ist, d. h. $\text{size}(G') := |V'| < \text{size}(G) := |V|$. Dies bedeutet, dass aus G Knoten entfernt werden müssten, ohne die repräsentierte Funktion zu verändern. In anderen Worten G kann auf G' reduziert werden. Aufgrund der Definition eines normalisierten, reduzierten und geordneten TEDs ist dies nicht möglich, da G keinerlei Redundanz enthält. Die Wiederverwendung gemeinsamer Teilausdrücke ist bereits durch die Reduktion garantiert. Dies bedeutet, dass G' entweder mindestens die selbe Größe wie G hat oder nicht die Funktion f repräsentiert, was die gemachte Annahme widerlegt. Somit ist G eine *minimale* und *kanonische* Repräsentation von f. □

Im Folgenden wird der Begriff TED stets im Sinne eines normalisierten, reduzierten und geordneten TED verwendet. TEDs sind eine kanonische Repräsentation für Funktionen, die eine endliche Taylor-Reihe besitzen. Insbesondere sind dies Polynomfunktionen mit einem endlichen Grad. Für solche Funktionen führt die sukzessive Ableitung zu einer endlichen Anzahl an Termen. Neben diesen Einschränkungen können TEDs auch keine Relationen ausdrücken, wie z. B. $x_1 < x_2$ oder $x_1 = x_2$. Relationen sind durch Diskontinuitäten charakterisiert und somit nicht differenzierbar.

4.1.5 Implizite Äquivalenzprüfung mit TEDs

Die implizite Äquivalenzprüfung mit Taylor-Expansions-Diagrammen wird anhand eines Beispiels verdeutlicht.

Beispiel 4.1.5. Gegeben ist die Funktion f:

$$\begin{aligned} f(x_1,x_2,x_3,x_4) := {} & 36 \cdot x_2^2 \cdot x_3^2 \\ & -3 \cdot x_3 \cdot (x_1 \cdot (8 \cdot x_1 \cdot x_4 - 3 \cdot x_1 \cdot x_3 + 2 \cdot x_2 \cdot (16 \cdot x_4 - 6 \cdot x_3))) \\ & -4 \cdot x_4 \cdot \left(x_2^2 \cdot (24 \cdot x_3 - 16 \cdot x_4) - 4 \cdot x_1 \cdot (4 \cdot x_2 \cdot x_4 + x_1 \cdot x_4)\right) \end{aligned}$$

Es soll die Äquivalenz mit Funktion g geprüft werden:

$$g(x_1,x_2,x_3,x_4) := (x_1 + 2 \cdot x_2)^2 \cdot (3 \cdot x_3 - 4 \cdot x_4)^2$$

Die Äquivalenzprüfung erfolgt implizit, indem für beide Funktionen ein normalisiertes, reduziertes und geordnetes TED mit der selben Variablenordnung $x_1 < x_2 < x_3 < x_4$ erstellt wird.

Zunächst wird das TED für die Funktion f bestimmt. Die Konstruktion ist in Abb. 4.7 dargestellt. Abbildung 4.7a) zeigt das Ergebnis nach der Taylor-Reihen-Entwicklung nach der Variablen x_1. Hierbei ergeben sich die folgenden Werte:

$$
\begin{aligned}
f\big|_{x_1:=0} &= 36 \cdot x_2^2 \cdot x_3^2 - 4 \cdot x_4 \cdot \left(x_2^2 \cdot (24 \cdot x_3 - 16 \cdot x_4)\right) \\
\left.\frac{\partial f}{\partial x_1}\right|_{x_1:=0} &= -3 \cdot x_3 \cdot (2 \cdot x_2 \, (16 \cdot x_4 - 6 \cdot x_3)) - 4 \cdot x_4 \cdot (-16 \cdot x_2 \cdot x_4) \\
\frac{1}{2} \left.\frac{\partial^2 f}{\partial x_1^2}\right|_{x_1:=0} &= -3 \cdot x_3 \cdot (8 \cdot x_4 - 3 \cdot x_3) + 4 \cdot x_4 \cdot (4 \cdot x_4)
\end{aligned}
$$

Abbildung 4.7b) zeigt das TED nach der Taylor-Reihen-Entwicklung nach x_2, wobei sich folgende Werte ergeben:

$$
\begin{aligned}
\frac{1}{2} \left.\frac{\partial^2 f}{\partial^2 x_2}\right|_{\substack{x_1:=0 \\ x_2:=0}} &= 36 \cdot x_3^2 - 4 \cdot x_4 \cdot (24 \cdot x_3 - 16 \cdot x_4) \\
\left.\frac{\partial^2 f}{\partial x_1 \partial x_2}\right|_{\substack{x_1:=0 \\ x_2:=0}} &= -3 \cdot x_3 \cdot (32 \cdot x_4 - 12 \cdot x_3) - 4 \cdot x_4 \cdot (-16 \cdot x_4) \\
\frac{1}{2} \left.\frac{\partial^2 f}{\partial x_1^2}\right|_{\substack{x_1:=0 \\ x_2:=0}} &= -3 \cdot x_3 \cdot (8 \cdot x_4 - 3 \cdot x_3) + 4 \cdot x_4 \cdot (4 \cdot x_4)
\end{aligned}
$$

Abbildung 4.7c) zeigt das TED nach der Taylor-Reihen-Entwicklung nach x_3, wobei sich folgende Werte ergeben:

$$
\begin{aligned}
\frac{1}{2} \left.\frac{\partial^2 f}{\partial^2 x_2}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= 64 \cdot x_4^2 \\
\frac{1}{2} \left.\frac{\partial^3 f}{\partial^2 x_2 \partial x_3}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= -96 \cdot x_4 \\
\frac{1}{4} \left.\frac{\partial^4 f}{\partial^2 x_2 \partial x_3^2}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= 36
\end{aligned}
$$

$$
\begin{aligned}
\left.\frac{\partial^2 f}{\partial x_1 \partial x_2}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= 64 \cdot x_4^2 \\
\left.\frac{\partial^3 f}{\partial x_1 \partial x_2 \partial x_3}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= -96 \cdot x_4 \\
\frac{1}{2} \left.\frac{\partial^4 f}{\partial x_1 \partial x_2 \partial x_3^2}\right|_{\substack{x_1:=0 \\ x_2:=0 \\ x_3:=0}} &= 36
\end{aligned}
$$

$$\frac{1}{2}\left.\frac{\partial^2 f}{\partial x_1^2}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = 16\cdot x_4^2$$

$$\frac{1}{2}\left.\frac{\partial^3 f}{\partial x_1^2 \partial x_3}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = -24\cdot x_4$$

$$\frac{1}{4}\left.\frac{\partial^4 f}{\partial x_1^2 \partial x_3^2}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = 9$$

Eine Taylor-Reihen-Entwicklung ist in diesem Schritt nicht mehr notwendig, da bereits alle Koeffizienten als Taylor-Reihe vorliegen. Das geordnete TED ist in Abb. 4.7d) zu sehen. Das normalisierte, reduzierte und geordnete TED ist in Abb. 4.8 dargestellt.

Als nächstes wird für die Funktion g das TED mit der selben Variablenordnung konstruiert. Im ersten Schritt ergeben sich die folgenden Werte:

$$g|_{x_1:=0} = 4\cdot x_2^2\cdot(3\cdot x_3 - 4\cdot x_4)^2$$

$$\left.\frac{\partial g}{\partial x_1}\right|_{x_1:=0} = 4\cdot x_2\cdot(3\cdot x_3 - 4\cdot x_4)^2$$

$$\frac{1}{2}\left.\frac{\partial^2 g}{\partial x_1^2}\right|_{x_1:=0} = (3\cdot x_3 - 4\cdot x_4)^2$$

Im nächsten Schritt erfolgt die Entwicklung nach x_2 mit folgenden Ergebnissen:

$$\frac{1}{2}\left.\frac{\partial^2 g}{\partial^2 x_2}\right|_{\substack{x_1:=0\\x_2:=0}} = 4\cdot(3\cdot x_3 - 4\cdot x_4)^2$$

$$\left.\frac{\partial^2 g}{\partial x_1 \partial x_2}\right|_{\substack{x_1:=0\\x_2:=0}} = 4\cdot(3\cdot x_3 - 4\cdot x_4)^2$$

$$\frac{1}{2}\left.\frac{\partial^2 g}{\partial x_1^2}\right|_{\substack{x_1:=0\\x_2:=0}} = (3\cdot x_3 - 4\cdot x_4)^2$$

Man sieht, dass sich die ersten beiden Koeffizienten vom letzten lediglich durch den Faktor 4 unterschieden. Aus diesem Grund wird hier lediglich der letzte Faktor nach x_3 entwickelt. Als Ergebnis ergibt sich:

$$\frac{1}{2}\left.\frac{\partial^2 g}{\partial x_1^2}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = 16\cdot x_4^2$$

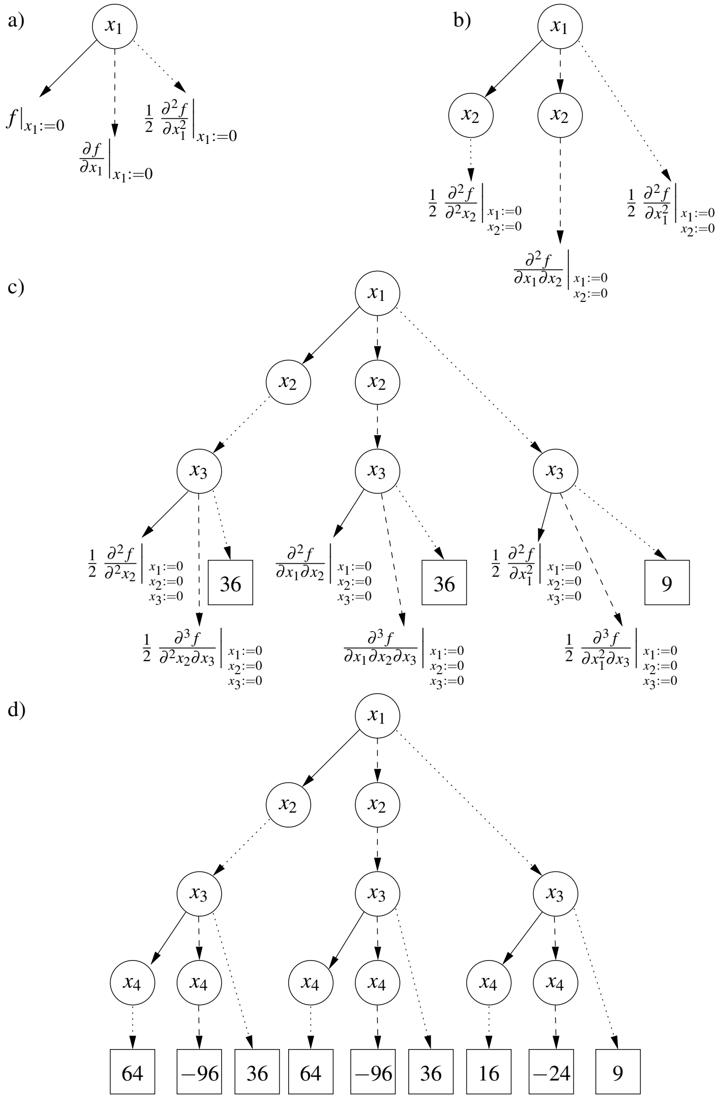


Abb. 4.7. Konstruktion des TED für $f(x_1,x_2,x_3,x_4)$

$$\frac{1}{2}\left.\frac{\partial^3 g}{\partial x_1^2 \partial x_3}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = -24 \cdot x_4$$

$$\frac{1}{4}\left.\frac{\partial^4 g}{\partial x_1^2 \partial x_3^2}\right|_{\substack{x_1:=0\\x_2:=0\\x_3:=0}} = 9$$

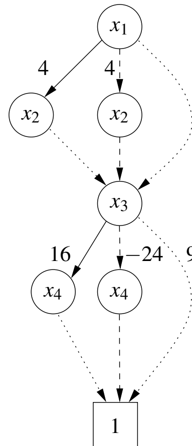


Abb. 4.8. Normalisiertes, reduziertes und geordnetes TED für $f(x_1,x_2,x_3,x_4)$

Durch Vergleich, mit Abb. 4.7 sieht man, dass die entwickelten Koeffizienten beider Funktionen identisch sind. Durch die Normalisierung ergibt sich somit für die Funktion g das selbe TED, wie in Abb. 4.8. Da die Funktionen f und g durch das selbe normalisierte, reduzierte und geordnete TED repräsentiert werden, sind die Funktionen in der Tat äquivalent.

4.2 Explizite Äquivalenzprüfung

Die implizite Äquivalenzprüfung zweier Funktionen f und g, behandelt in Abschnitt 4.1, wird formal als $f \stackrel{?}{\equiv} g$ geschrieben. Diese Prüfung auf Äquivalenz kann auch wie folgt umgeformt werden:

$$(f - g) \stackrel{?}{\equiv} 0$$

Hierbei handelt es sich um das sog. *Null-Äquivalenzproblem*. Technisch lässt sich dieses Problem lösen, indem beide Funktionen mit der selben Belegung der Variablen x stimuliert werden und die Ergebnisse beider Funktionen, y_f und y_g, verglichen werden. Somit kann die Subtraktion als Vergleich der Ergebnisse interpretiert werden. Dies ist in Abb. 4.9 zu sehen.

Da die Verifikationsaufgabe, die Äquivalenzprüfung, in Form des Vergleichers mit im Modell enthalten ist, handelt es sich hierbei um eine *explizite Äquivalenzprüfung*. Verfahren zur expliziten Äquivalenzprüfung können entweder simulativ oder formal sowie eine Mischform von beidem sein. Formale Methoden zur expliziten Äquivalenzprüfung können Verifikationsvollständigkeit bezüglich der gewählten

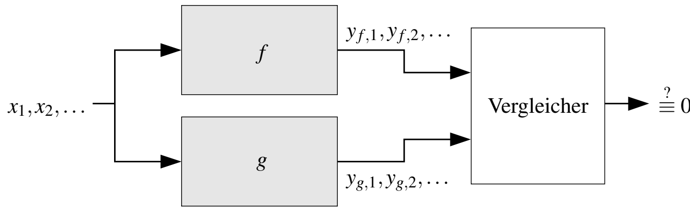


Abb. 4.9. Explizite Äquivalenzprüfung

Abstraktion garantieren. Simulative Verfahren sind inhärent unvollständig, weshalb diese meistens mit dem Ziel der Falsifikation gestartet werden. Es wird also versucht, eine Variablenbelegung zu finden, die zu unterschiedlichen Funktionsergebnissen für f und g führen.

Eine simulative, explizite Äquivalenzprüfung setzt voraus, dass für beide Systeme eine ausführbare Verhaltensbeschreibung vorliegt. Dies impliziert, dass die simulative Äquivalenzprüfung häufig zum Vergleich von zwei oder mehr Implementierungen verwendet wird. Diese Implementierungen können entweder von unterschiedlichen Entwicklern unabhängig erstellt worden sein oder durch Optimierungsschritte entstehen.

Im Fall von unterschiedlichen Entwicklern ist es die Intention, diversitäre Systeme zu erstellen, d. h. unterschiedliche Implementierungen der selben Spezifikation zu entwickeln. Werden unterschiedliche Implementierungen nach dem Verfahren in Abb. 4.9 auf Äquivalenz überprüft, können die Implementierungen entweder identische oder unterschiedliche Ergebnisse liefern. Liefern die beiden Implementierungen unterschiedliche Ergebnisse, so ist mindestens ein Ergebnis inkorrekt. Da bei einer expliziten Äquivalenzprüfung nicht auf den internen Aufbau der Systeme geachtet wird, spricht man von einem sog. *Ende-zu-Ende-Test* (engl. *back-to-back test*).

Beim Ende-zu-Ende-Test geht man zunächst davon aus, dass identische Ergebnisse auch die Korrektheit der Berechnung implizieren. Somit ist es das Ziel, unterschiedliche Ergebnisse bei der Simulation von verschiedenen Implementierungen aufzudecken. Beim Ende-zu-Ende-Test handelt es sich somit um eine *diversifizierende Verifikationsmethode*. Allerdings kann die Identität von Ergebnissen auch zufällig sein, z. B. dann, wenn zwei Entwickler den selben Fehler aufgrund einer fehlerhaften Spezifikation implementieren. In diesem Fall wird das Ergebnis der Äquivalenzprüfung als *falschpositiv* (engl. *false positive*) bezeichnet, da implizit angenommen wird, dass die Übereinstimmung der Ergebnisse auf ein korrektes Verhalten schließen lässt, was allerdings falsch ist. Die diversitären Versionen des Systems enthalten ausschließlich die selben Fehler. Dies ist in Abb. 4.10 dargestellt.

Um die Anzahl der falschpositiven Ergebnisse zu minimieren, ist es sinnvoll, dass die Entwickler der unterschiedlichen Implementierungen keinen Kontakt haben und somit die Möglichkeit zur gemeinsamen Interpretation der Spezifikation genommen ist. Der Ende-zu-Ende-Test kann unterschiedlich ausgeführt werden. Zum einen können manuell erstellte, gerichtete Testfälle verwendet werden, um die Äquivalenz

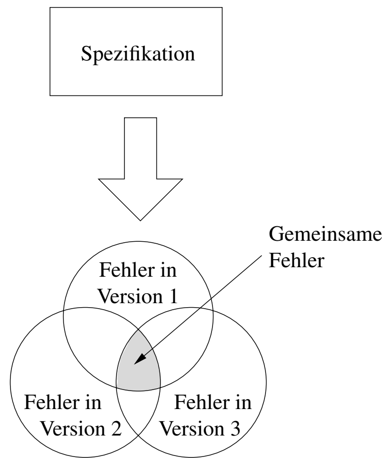


Abb. 4.10. Grund für falschpositive Ergebnisse in diversifizierenden Verifikationsmethoden [305]

der Implementierungen in ihrer Grundfunktionalität zu zeigen, zum anderen können zufällige Testfälle verwendet werden, um nicht anderweitig berücksichtigte Szenarien zu überprüfen (siehe auch Abschnitt 3.3).

4.2.1 Regressionstest

Eine spezielle Form einer diversifizierenden, simulativen Verifikationsmethode ist der sog. *Regressionstest*. Während der Ende-zu-Ende-Test dazu dient, die Äquivalenz von Implementierungen zu prüfen, die von verschiedenen Entwicklern anhand der selben Spezifikation entwickelt wurden, dient der Regressionstest dazu, zu prüfen, ob bei der Optimierung oder Weiterentwicklung einer Implementierung bereits korrektes Verhalten zerstört wurde. Man geht also davon aus, dass zuvor fehlerfrei durchlaufene Testfälle nach einer Modifikation der Implementierung nicht grundsätzlich weiterhin fehlerfrei abgearbeitet werden. Ob allerdings alle zuvor durchlaufenen Testfälle potentiell durch die Modifikation betroffen sein können, muss im Einzelfall entschieden werden. Das Besondere am Regressionstest ist allerdings, dass nicht direkt gegen eine Spezifikation, sondern gegen eine Vorgängerversion der Implementierung geprüft wird.

Ein Regressionstest besteht also aus der Wiederholung einer Simulation mit identischen Testfalleingaben, wobei lediglich die zu prüfende Implementierung ausgetauscht wurde. Hierbei ist darauf zu achten, dass die zu prüfende Implementierung in dem gleichen Anfangszustand versetzt wird, wie die Vorgängerversion. Schließlich erfolgt ein Vergleich der durch die Simulation erzeugten Ausgabe mit der Ausgabe aus der Prüfung der Vorgängerversion. Das Prinzip des Regressionstests ist in Abb. 4.11 zu sehen.

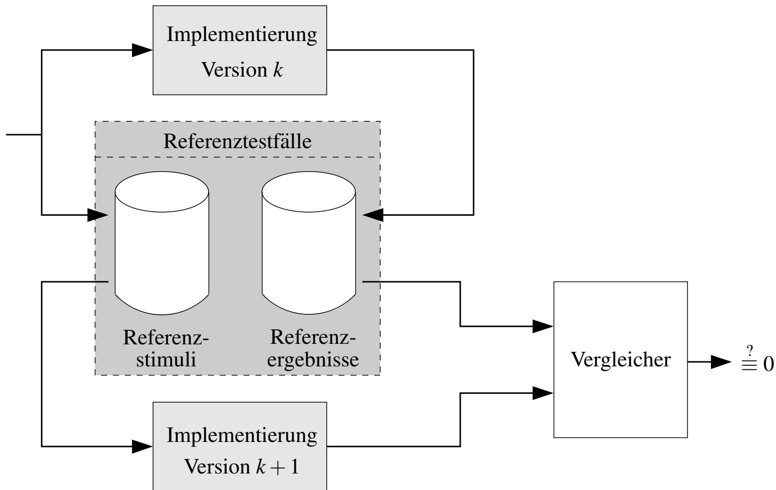


Abb. 4.11. Prinzip des Regressionstests [305]

Zunächst werden in einem Simulationslauf sowohl die verwendeten Testfalleingaben als auch die erzeugten Ausgaben einer Version k der Implementierung aufgezeichnet und gespeichert. Diese Daten werden als *Referenztestfälle*, aufgeteilt in *Referenzstimuli* und *Referenzergebnisse*, bezeichnet. Liegt eine Version $k+1$ der Implementierung vor, wird diese mit den Referenzstimuli stimuliert und die erzeugten Ergebnisse mit den Referenzergebnissen verglichen.

Treten Diskrepanzen zwischen erzeugten Ergebnissen der Version $k+1$ und den Referenzergebnissen auf, muss bewertet werden, ob es sich hierbei um eine gewünschte oder unerwünschte Abweichung handelt. Unerwünschte Ergebnisse lassen auf einen Fehler in der Implementierung schließen, was weiter überprüft werden muss. Dies ist notwendig, da es ebenfalls denkbar ist, dass das Referenzergebnis fehlerhaft war. Handelt es sich jedoch um eine gewünschte Abweichung von dem Referenzergebnis, etwa durch detailliertere Ausgaben, so wird das Referenzergebnis durch das neu erzeugte Ergebnis ersetzt.

Während Ende-zu-Ende-Tests auf einen konkreten Anwendungsfall eingeschränkt sind, stellen Regressionstests heutzutage einen Mindeststandard in der Systementwicklung dar. Dies liegt im Wesentlichen an dem hohen Automatisierungsgrad, der bei Regressionstest erreicht werden kann. Auf der anderen Seite bieten Ende-zu-Ende-Tests die Möglichkeit, die Qualität der Spezifikation zu erhöhen, da durch die mehrfache Implementierung Mehrdeutigkeiten und Unvollständigkeit in der Spezifikation mit einer höheren Wahrscheinlichkeit aufgedeckt werden. Allerdings kommt diese Qualitätssteigerung zu dem Preis der mehrfachen Entwicklung.

4.2.2 Bereichstest

Im Zusammenhang mit diversifizierenden Verifikationsmethoden stellt sich die Frage, wie mit der größten Wahrscheinlichkeit aber minimierten Simulationsaufwand Fehler aufgedeckt werden können. Das Ziel ist es, die meist große Anzahl an Testfällen zu reduzieren und nur wichtige Szenarien zu simulieren. Hierzu muss man entscheiden, welche Testfälle wichtig und welche eher unwichtig sind. Da hierbei typischerweise eine große Menge an Testfällen ausgeschlossen wird, muss man bei der Auswahl der Testfälle sehr sorgfältig vorgehen. Die wesentlichen Kriterien zur Auswahl sind dabei, dass:

- potentielle Fehler zuverlässig erkannt und
- die Testfälle möglichst frei von Redundanz sind.

Während ersteres darauf abzielt, effektive Testfälle auszuwählen, zielt letzteres darauf ab, die Effizienz der Simulation der Testfälle zu erhöhen.

Ein weit akzeptiertes Vorgehen bei der Auswahl von Testfällen ist die sog. *Äquivalenzklassenbildung*. Hierbei werden Testfälle in sog. *Äquivalenzklassen* eingeteilt.

Definition 4.2.1 (Äquivalenzklasse). *Eine* Äquivalenzklasse *von Testfällen ist eine Teilmenge aller möglichen Testfälle, bei denen das erwartete Verhalten durch Stimulation mit Testfalleingaben aus dieser Äquivalenzklasse* ähnlich *ist.*

Man sieht bereits an dieser Definition, dass es sich bei dem Begriff der *Äquivalenzklasse* um ein Konzept handelt, da es viele mögliche Interpretationen für die Ähnlichkeit von Verhalten des Systems gibt. Dies ist maßgeblich von der gewählten Abstraktionsebene abhängig. Prinzipiell werden zwei Arten von Äquivalenzklassenbildung unterschieden, die *funktionsorientierte Äquivalenzklassenbildung* und die *strukturorientierte Äquivalenzklassenbildung*. Bei der funktionsorientierten Äquivalenzklassenbildung werden die Äquivalenzklassen anhand der Spezifikation erstellt. Im Gegensatz dazu basiert die strukturorientierte Äquivalenzklassenbildung auf der Implementierung. Weiterhin beachte man bei Definition 4.2.1, dass lediglich von einem erwarteten Verhalten gesprochen wird. Resultiert aus den Simulationen mit zwei Elementen aus der selben Äquivalenzklasse unterschiedliches Verhalten, muss entweder die Äquivalenzklassenbildung oder die Implementierung des Systems fehlerhaft sein.

Beispiel 4.2.1. Als einfaches Beispiel wird zunächst ein System mit einem einzelnen nummerischen Eingang betrachtet. Die Spezifikation gibt vor, dass das System ausschließlich positive Werte als Eingabe gestattet. Somit können zwei Äquivalenzklassen von Testfällen gebildet werden: Solche mit Stimuli durch positive Zahlen und solche mit Stimuli durch nichtpositive Zahlen. Die Eingabe von positiven Zahlen stellen den Normalfall für das System dar, weshalb die zugehörige Äquivalenzklasse als *gültig* bezeichnet wird. Testfälle mit Eingaben von nichtpositiven Zahlen bilden entsprechend die *ungültige Äquivalenzklasse*. Dies weißt darauf hin, dass Testfalleingaben aus dieser Äquivalenzklasse ungültig sind und somit vermutlich zu einem Fehlverhalten führen.

Zur Auswahl der Testfälle können auf Basis der Äquivalenzklassen die folgenden Richtlinien gegeben werden: Falls die gültige Äquivalenzklasse einen Bereich von Werten repräsentiert, so können jeweils zwei ungültige Äquivalenzklassen gebildet werden. Ist z. B. die gültige Äquivalenzklasse durch den Bereich $1 \leq x \leq 10$ gegeben, so lauten die zugehörigen ungültigen Äquivalenzklassen $x < 1$ und $x > 10$. Ist die gültige Äquivalenzklasse allerdings eine Menge an Werten, so wird in der Regel lediglich eine ungültige Äquivalenzklasse gebildet. Sei z. B. $x \in \{rot, blau\}$ eine gültige Äquivalenzklasse, so kann eine ungültige Äquivalenzklasse wie folgt gebildet werden: $x \in \{allerFarben\} \setminus \{rot, blau\}$. Ist eine gültige Äquivalenzklasse durch eine Bedingung gegeben, z. B. $x > 10$, so kann eine ungültige Äquivalenzklasse wie folgt aussehen: $x \leq 10$.

Häufig werden Äquivalenzklassen aber nicht anhand der Eingänge, sondern für Ausgangsbedingungen eines Systems gebildet. Hierzu werden zunächst die Ausgangsbedingungen aufgestellt. Anhand dieser werden, in einem folgenden Schritt, die Äquivalenzklassen für die Testfalleingaben ermittelt, welche die gewünschten Ausgangsbedingungen produzieren. Die Regeln zur Erzeugung der gültigen und ungültigen Äquivalenzklassen können in diesem Fall direkt übertragen werden. Unter der Annahme, dass das System die Funktion $y = f(x)$ implementiert und die Ausgangsbedingung $1 \leq y \leq 10$ lautet, können eine gültige Äquivalenzklasse $x \in \{x \mid 1 \leq f(x) \leq 10\}$ und zwei ungültige Äquivalenzklassen $x \in \{x \mid f(x) < 1\}$ und $x \in \{x \mid f(x) > 10\}$ gebildet werden.

4.2.3 Pfadbereichstest

Durch Bildung von Äquivalenzklassen ist das Problem der Auswahl sinnvoller Testfälle allerdings noch nicht gelöst. Hierzu wurden eine Reihe von Verfahren entwickelt, die sich unter dem Begriff *Bereichstest* (engl. *domain testing*) zusammenfassen lassen. Das strukturorientierte Verfahren mit dem Namen *Pfadbereichstest* [462] geht davon aus, dass Fehler in einem System in zwei Klassen eingeteilt werden können:

1. *Bereichsfehler* führen zur Ausführung eines falschen Kontrollpfads.
2. *Berechnungsfehler* entstehen, wenn zwar der richtige Kontrollpfad ausgeführt wird, das berechnete Ergebnis allerdings falsch ist.

Der Pfadbereichstest zielt auf die Erkennung von Bereichsfehlern. Allerdings ist hierdurch nicht ausgeschlossen, dass ebenfalls Berechnungsfehler erkannt werden können. Das prinzipielle Vorgehen zur Erstellung eines Pfadbereichstests ist das wiederholte bilden von Äquivalenzklassen anhand des Kontrollflusses eines Systems. Dies wird anhand eines Beispiels aus [305] verdeutlicht.

Beispiel 4.2.2. Gegeben ist die folgende C-Funktion zur Bestimmung der betragsmäßig größeren Zahl zweier reeller Zahlen.

```
 1 void minMaxBetrag(double &min, double &max) {
 2   if (min < 0) {
 3     min *= -1;
 4   }
 5   if (max < 0) {
 6     max *= -1;
 7   }
 8   if (min > max) {
 9     double tmp = min;
10     min = max;
11     max = tmp;
12   }
13 }
```

Hierzu wird zunächst der Betrag der Variable `min`, anschließend der Betrag der Variablen `max` bestimmt. In einem anschließenden Schritt wird der größere Betrag beider Zahlen bestimmt. Der größere Betrag ist am Ende in der Variablen `max`, der kleinere Betrag in der Variablen `min` gespeichert.

Zur Auswahl der Testfälle beim Pfadbereichstest stützt man sich auf die Annahme, dass Fehler am häufigsten an Bereichsgrenzen auftreten. Um auf dieser Annahme Testfälle zu generieren, werden zunächst Eingabebedingungen für die unterschiedlichen Programmpfade der C-Funktion aufgestellt. Die Eingabebedingungen definieren Teilbereiche der Eingabewerte, durch welche die Ausführung eines bestimmten Programmpfads stimuliert wird. Ist einer dieser Teilbereiche leer, so kann der zugehörige Pfad nicht ausgeführt werden.

Sind die Eingabebedingungen bekannt, kann jedem Pfad p_i ein Teilbereich $d(p_i)$ der Eingabewerte, der sog. *Pfadbereich*, und die berechnete Funktion $c(p_i)$, die sog. *Pfadberechnung*, zugeordnet werden. Mit diesem Wissen kann das Verhalten eines Systems durch eine potentiell unendlich große Anzahl an Paaren $(d(p_i), c(p_i))$ repräsentiert werden.

Beispiel 4.2.3. Für das Programm aus Beispiel 4.2.2 ist der zugehörige Kontroll-Datenflussgraph in Abb. 4.12 dargestellt. Insgesamt sind acht unterschiedliche Pfade im Kontrollflussgraphen möglich. Somit lässt sich das Programm aus Beispiel 4.2.2 durch folgende Paare aus Eingangsbedingungen und Berechnungen für die acht Pfade darstellen, wobei die Pfadbereiche durch Prädikate beschrieben werden:

$p_1 = (v_1, v_2, v_3, v_4, v_5, v_6, v_7)$:

$$d(p_1) : (min_{in} < 0) \wedge (max_{in} < 0) \wedge (-max_{in} < -min_{in})$$
$$c(p_1) : min_{out} := -max_{in};\ max_{out} := -min_{in}$$

$p_2 = (v_1, v_3, v_4, v_5, v_6, v_7)$:

$$d(p_2) : (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (-max_{in} < min_{in})$$
$$c(p_2) : min_{out} := -max_{in};\ max_{out} := min_{in}$$

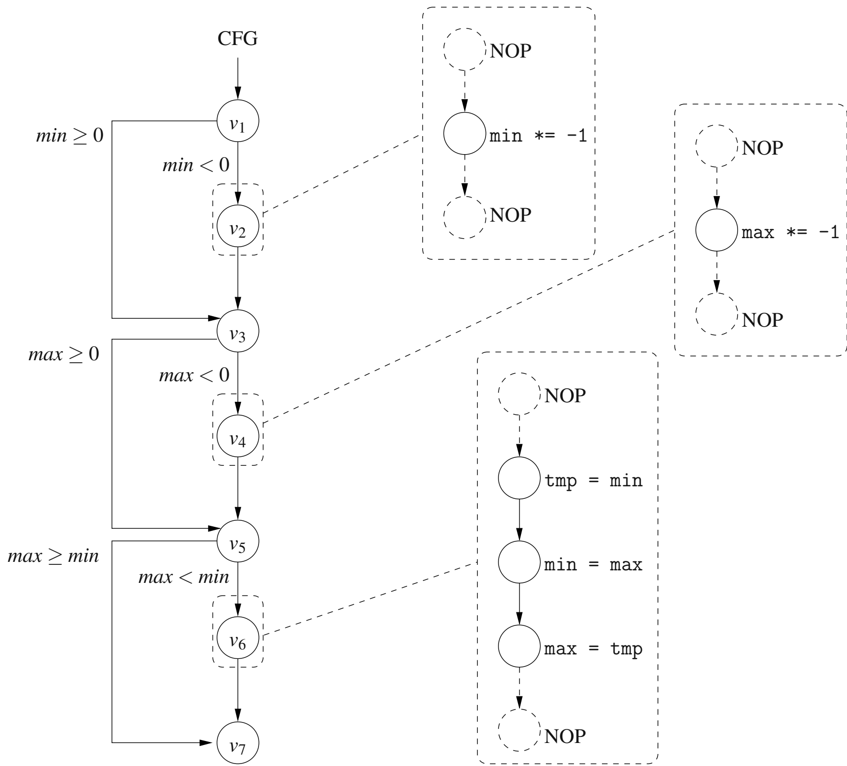


Abb. 4.12. Kontroll–Datenflussgraph für die Funktion `minMaxBetrag` aus Beispiel 4.2.2

$p_3 = (v_1, v_2, v_3, v_5, v_6, v_7)$:

$$d(p_3) : (min_{in} < 0) \wedge (max_{in} \geq 0) \wedge (max_{in} < -min_{in})$$
$$c(p_3) : min_{out} := max_{in};\; max_{out} := -min_{in}$$

$p_4 = (v_1, v_3, v_5, v_6, v_7)$:

$$d(p_4) : (min_{in} \geq 0) \wedge (max_{in} \geq 0) \wedge (max_{in} < min_{in})$$
$$c(p_4) : min_{out} := max_{in};\; max_{out} := min_{in}$$

$p_5 = (v_1, v_2, v_3, v_4, v_5, v_7)$:

$$d(p_5) : (min_{in} < 0) \wedge (max_{in} < 0) \wedge (-min_{in} \leq -max_{in})$$
$$c(p_5) : min_{out} := -min_{in};\; max_{out} := -max_{in}$$

$p_6 = (v_1, v_3, v_4, v_5, v_7)$:

$$d(p_6) : (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (min_{in} \leq -max_{in})$$
$$c(p_6) : min_{out} := min_{in};\; max_{out} := -max_{in}$$

$p_7 = (v_1, v_2, v_3, v_5, v_7)$:

$$d(p_7) : (min_{in} < 0) \wedge (max_{in} \geq 0) \wedge (-min_{in} \leq max_{in})$$
$$c(p_7) : min_{out} := -min_{in};\ max_{out} := max_{in}$$

$p_8 = (v_1, v_3, v_5, v_7)$:

$$d(p_8) : (min_{in} \geq 0) \wedge (max_{in} \geq 0) \wedge (min_{in} \leq max_{in})$$
$$c(p_8) : min_{out} := min_{in};\ max_{out} := max_{in}$$

Dabei bezeichnet min_{in} und max_{in} jeweils die Eingangsbelegung der Variablen `min` und `max`. Entsprechend bezeichnet min_{out} und max_{out} die jeweilige Belegung der Variablen `min` und `max` beim Verlassen der Funktion `minMaxBetrag`.

Zur Auswahl der Testfälle werden in [462] sog. *On-* und *Off-Testfälle* definiert. Eingaben von On-Testfälle besitzen die Eigenschaft, dass sie Werte, die direkt auf einer Bereichsgrenze liegen, stimulieren. Eingaben von Off-Testfälle hingegen stimulieren Werte, die ein ε von dieser Bereichsgrenze entfernt liegen. Hierbei müssen zwei Fälle unterschiedenen werden:

1. Ist die betrachtete Bereichsgrenze eine geschlossene Grenze, so liegt der On-Testfall innerhalb dieses Bereiches. Der Off-Testfall wird ein ε entfernt von der Bereichsgrenze gelegt, also außerhalb des Bereichs. Der Off-Testfall gehört somit zu einem Nachbarbereich.
2. Ist die betrachtete Bereichsgrenze hingegen eine offene Grenze, so liegt der On-Testfall bereits in einem Nachbarbereich. Der Off-Testfall wird dann ein ε entfernt von der Bereichsgrenze, in den betrachteten Bereich gelegt.

Beispiel 4.2.4. Als Fortsetzung für die Beispiele 4.2.2 und 4.2.3 werden Testfalleingaben für die Pfadbereiche $d(p_i)$ für alle acht Kontrollflusspfade p_i des Programms `minMaxBetrag` erstellt. Hierzu bietet es sich an, die Pfadbereiche graphisch darzustellen (siehe Abb. 4.13). Pfadbereichsgrenzen sind durch die min_{in}- und max_{in}-Achse sowie die beiden Winkelhalbierenden gegeben. Ob eine Bereichsgrenze offen oder geschlossen ist, ist jeweils durch einen kleinen Pfeil gekennzeichnet. Hierbei zeigt der Pfeil an der Grenze immer in den für die Grenze geschlossenen Bereich.

Als Beispiel seien die Bereichsgrenzen des Bereichs $d(p_6) : (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (min_{in} \leq -max_{in})$ betrachtet. Sowohl die angrenzende max_{in}-Achse als auch die angrenzende Winkelhalbierende gehören zu diesem Bereich. Um die Pfadbereichsgrenze, die durch die Winkelhalbierende gegeben ist, zu überprüfen, wählt man nun drei Testfalleingaben, so dass diese eine On-Off-On-Folge ergeben, d. h. man wählt zunächst einen Testfall direkt auf der Winkelhalbierenden, dann einen um einen ε entfernten im Bereich $d(p_2)$ liegenden, und anschließend wieder einen Testfall direkt auf der Winkelhalbierenden. Dies ist in Abb. 4.13 für die Testfälle mit Eingabe (x_1, x_2, x_3) gegeben.

Führt man nun eine Simulation für einen Pfadbereich $d(p_i)$ entsprechend mit den On-Off-On-Testfällen durch, kann es zu einem nicht erwarteten Ergebnis kommen.

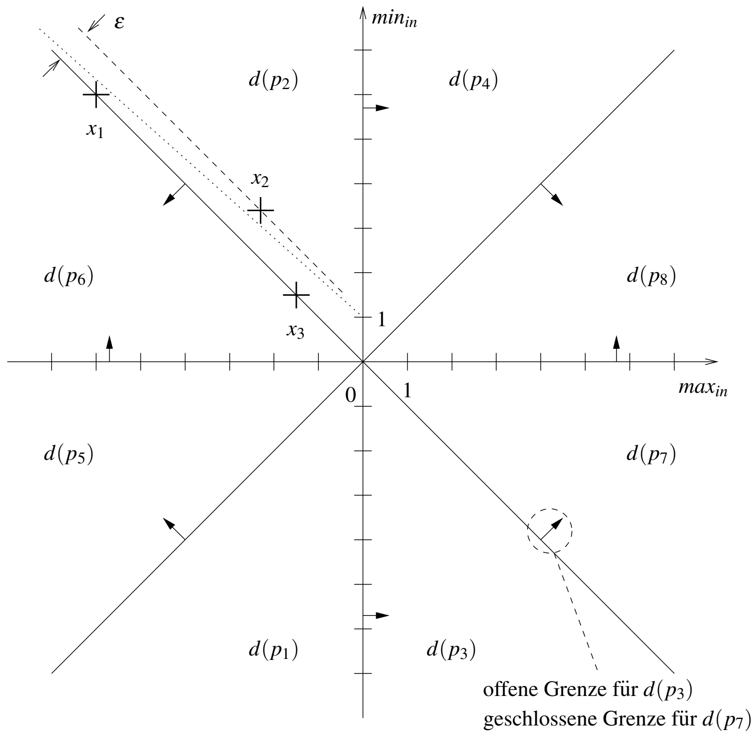


Abb. 4.13. Pfadbereiche des Programms `minMaxBetrag` aus Beispiel 4.2.2 [305]

Ob dieses unerwartete Ergebnis auf einen Fehler schließen lässt, muss zunächst geprüft werden. Handelt es sich um ein Fehlverhalten, so können drei Arten von Fehler unterschieden werden:

1. Wird ein Bereichsfehler entdeckt, so muss der Pfadbereich $d(p_i)$ angepasst werden. Es wird eine sog. *Pfadbereichsanpassung* vorgenommen. Dies ist in Abb. 4.14a) für den Fall, dass ein On-Testfall fehlschlägt, in Abb. 4.14b) für den Fall, dass der Off-Testfall fehlschlägt, dargestellt. Mögliche neue Bereichsgrenzen sind als gepunktete Linien gezeichnet.
2. Wird hingegen ein Berechnungsfehler erkannt, d. h. das tatsächliche Ergebnis passt nicht zu dem erwarteten Ergebnis, so muss die Berechnung $c(p_i)$ dieses Pfades korrigiert werden. Es wird eine sog. *Pfadberechnungsanpassung* vorgenommen.
3. Wird das Fehlen eines Pfades p_j erkannt, so muss ein neues Paar $(d(p_j), c(p_j))$ gebildet werden, wobei $d(p_j)$ ein Teilbereich des vorherigen Pfadbereichs $d(p_i)$ ist. Es wird eine sog. *Pfadergänzung* vorgenommen.

Man beachte, dass Anpassungen im Allgemeinen nicht einer einzelnen Klasse dieser Fehler zugeordnet werden können. So kann eine Änderung in einer Variablenbele-

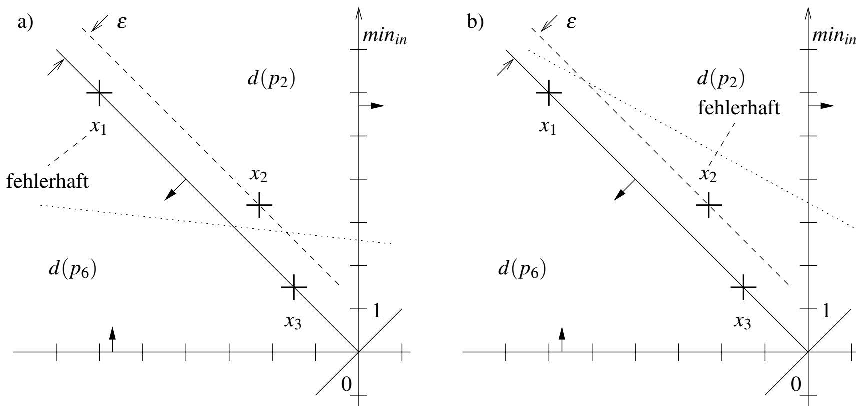


Abb. 4.14. Anpassung der Pfadbereichsgrenze [305]

gung sowohl den Pfadbereich als auch die Pfadberechnung ändern. Insbesondere ist jede Pfadergänzung auch eine Pfadbereichsanpassung.

Falls jedoch alle drei Testfalleingaben das erwartete Ergebnis liefern, gibt es zwei Möglichkeiten:

1. Entweder ist die Bereichsgrenze korrekt oder
2. sie weicht im Dreieck der drei Testfälle maximal um das gewählte ε ab. Ein solches Beispiel ist in Abb. 4.13 als gepunktete Linie dargestellt.

In den meisten Fällen wird es nicht möglich sein, alle Pfade eines Programms mit Testfällen zu simulieren, da dies zu aufwendig wäre. Allerdings lässt sich der Grad der erzielten Verifikationsvollständigkeit beim Pfadbereichstest direkt mit der Pfadüberdeckung messen und somit eine Aussage über den Grad der Vollständigkeit der Verifikation treffen.

4.2.4 Fehleroffenbarende Unterbereiche

Eine Erweiterung des Pfadbereichstests ist in [461] vorgestellt. Neben der Konstruktion der Pfadbereiche anhand der Implementierung, wird zusätzlich die Spezifikation zu Rate gezogen. Hierdurch soll es z. B. ermöglicht werden, fehlende Pfade in der Implementierung zu erkennen. Hierzu werden sog. *fehleroffenbarende Unterbereiche* konstruiert. Dabei werden zunächst die Pfadbereiche nach dem oben beschriebenen Verfahren bestimmt. In einem zweiten Schritt werden Äquivalenzklassen aus der Spezifikation abgeleitet. Pfadbereiche und Äquivalenzklassen werden paarweise geschnitten. Als Ergebnis erhält man die fehleroffenbarenden Unterbereiche, die dadurch gekennzeichnet sind, dass sie nur Testfalleingaben enthalten, die laut Spezifikation zu einem gleichen Verhalten führen und die auch vom System gleichartig verarbeitet werden. Dies wird an folgendem Beispiel aus [305] gezeigt:

Beispiel 4.2.5. Für das Programm `minMaxBetrag` aus Beispiel 4.2.2 wurde in Beispiel 4.2.3 gezeigt, dass die Implementierung acht verschiedene Kontrollpfade enthält. Aus der Spezifikation können drei Äquivalenzklasse $d(s_1), d(s_2)$ und $d(s_3)$ abgeleitet werden:

s_1:

$$\begin{aligned} d(s_1) &: |min_{in}| < |max_{in}| \\ c(s_1) &: min_{out} := |min_{in}|; max_{out} := |max_{in}| \end{aligned}$$

s_2:

$$\begin{aligned} d(s_2) &: |min_{in}| > |max_{in}| \\ c(s_2) &: min_{out} := |max_{in}|; max_{out} := |min_{in}| \end{aligned}$$

s_3:

$$\begin{aligned} d(s_3) &: |min_{in}| = |max_{in}| \\ c(s_3) &: min_{out} = max_{out} := |min_{in}| \end{aligned}$$

Während die Äquivalenzklasse $d(s_2)$ die Pfadbereiche $d(p_1), \ldots, d(p_4)$ umfasst, kann eine eindeutige Zuordnung für die Äquivalenzklassen $d(s_1)$ und $d(s_3)$ nicht gemacht werden. So gilt z. B., dass $d(p_5)$ zum Teil in $d(s_1)$ und zum Teil in $d(s_3)$ enthalten ist, aber weder $d(s_1)$ noch $d(s_3)$ vollständig in $d(p_5)$ enthalten ist. Durch die paarweise Schnittmengenbildung können die fehleroffenbarenden Unterbereiche identifiziert werden:

$$\begin{aligned}
d_1 = d(s_1) \cap d(p_5) &= (min_{in} < 0) \wedge (max_{in} < 0) \wedge (-min_{in} < -max_{in}) \\
d_2 = d(s_1) \cap d(p_6) &= (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (min_{in} < -max_{in}) \\
d_3 = d(s_1) \cap d(p_7) &= (min_{in} < 0) \wedge (max_{in} \geq 0) \wedge (-min_{in} < max_{in}) \\
d_4 = d(s_1) \cap d(p_8) &= (min_{in} \geq 0) \wedge (max_{in} \geq 0) \wedge (min_{in} < max_{in}) \\
d_5 = d(s_2) \cap d(p_1) &= (min_{in} < 0) \wedge (max_{in} < 0) \wedge (-max_{in} < -min_{in}) \\
d_6 = d(s_2) \cap d(p_2) &= (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (-max_{in} < min_{in}) \\
d_7 = d(s_2) \cap d(p_3) &= (min_{in} < 0) \wedge (max_{in} \geq 0) \wedge (max_{in} < -min_{in}) \\
d_8 = d(s_2) \cap d(p_4) &= (min_{in} \geq 0) \wedge (max_{in} \geq 0) \wedge (max_{in} < min_{in}) \\
d_9 = d(s_3) \cap d(p_5) &= (min_{in} < 0) \wedge (max_{in} < 0) \wedge (-min_{in} = -max_{in}) \\
d_{10} = d(s_3) \cap d(p_6) &= (min_{in} \geq 0) \wedge (max_{in} < 0) \wedge (min_{in} = -max_{in}) \\
d_{11} = d(s_3) \cap d(p_7) &= (min_{in} < 0) \wedge (max_{in} \geq 0) \wedge (-min_{in} = max_{in}) \\
d_{12} = d(s_3) \cap d(p_8) &= (min_{in} \geq 0) \wedge (max_{in} \geq 0) \wedge (min_{in} = max_{in})
\end{aligned}$$

Die hier vorgestellten simulativen Verfahren zur expliziten Äquivalenzprüfung lassen sich auf speicherbehaftete Systeme erweitern, indem Sequenzen von Ein- und Ausgaben anstelle einzelner Ein- und Ausgabesymbole betrachtet werden, um den Einfluss des im Speicher abgebildeten Zustands erfassen zu können. Weiterführende Verfahren zur Generierung von funktions- und strukturorientierten Testfällen sind in Abschnitt 7.2 beschrieben. Formale Methoden zur expliziten Äquivalenzprüfung kombinatorischer Hardware sind in Abschnitt 6.1.2 zu finden.

4.3 Sequentielle Äquivalenzprüfung

Digitale Hardware/Software-Systeme arbeiten typischerweise über sehr lange Zeiträume ohne Neustart. Dabei verarbeiten sie nicht nur ein einzelnes Eingabesymbol zu einem Ausgabesymbol, sondern Sequenzen von Eingabesymbolen zu Sequenzen von Ausgabesymbolen. Dabei hängt die aktuelle Ausgabe eines Systems im Allgemeinen nicht nur von der momentanen, sondern auch von vorherigen Eingaben ab. Dies ist in Abb. 4.15 dargestellt, wobei die Ausgabe des speicherbehafteten Systems (Abb. 4.15b)) stets von dem aktuellen und den zwei vorherigen Eingangssymbolen abhängt.

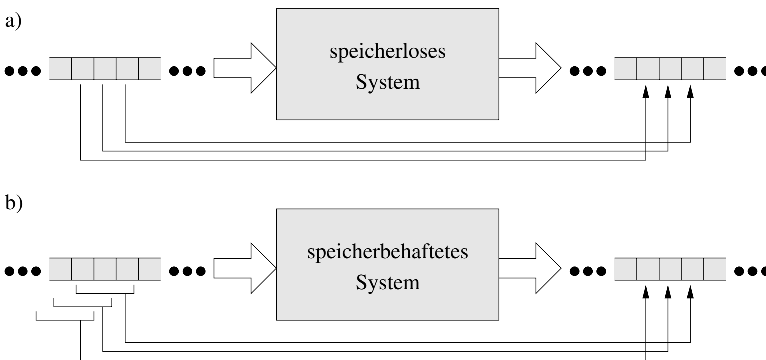


Abb. 4.15. Verarbeitung in Systemen a) ohne und b) mit Speicher [272]

Die Information zur Berechnung der aktuellen Ausgabe ist dabei in den Speicherelementen des Systems gespeichert. Viele wichtige speicherbehaftete Systeme lassen sich als *deterministische endliche Automaten* (engl. *Deterministic Finite Automaton, DFA*) (siehe Definition 2.2.13 auf Seite 47) modellieren. Aus diesem Grund wird die sequentielle Äquivalenzprüfung anhand von Automaten-Äquivalenz eingeführt. Verfahren zur sequentiellen Äquivalenzprüfung für Hardware sind in Abschnitt 6.1.3 beschrieben. Verfahren zur Prozessor- und Software-Äquivalenzprüfung sind in den Abschnitten 6.3 und 7.1 diskutiert.

Endliche Automaten können explizit als Zustandsdiagramme repräsentiert werden. Zustände $s \in S$ werden als Knoten gezeichnet. Zustandsübergänge werden entsprechend der Übergangsfunktion f als Kanten (s, s') modelliert, wobei s der aktuelle Zustand und $s' := f(s, i)$ der Folgezustand ist. Diese werden mit dem Eingabesymbol $i \in I$, das den Übergang auslöst, und dem erzeugten Ausgabesymbol $o \in O$ beschriftet, wobei $o := g(s, i)$ sich aus der Ausgabefunktion g ergibt.

Endliche Automaten arbeiten auf Sequenzen von Eingabesymbolen. Sei I^+ die Menge aller endlichen Sequenzen, die sich aus den Symbolen $i \in I$ bilden lassen. Entsprechend sei O^+ die Menge aller endlichen Sequenzen, die sich aus den Ausgabesymbolen $o \in O$ bilden lassen. Eingabesequenzen $w \in I^+$ der Länge n werden im

Folgenden als $w = \langle w_1, w_2, \dots w_n \rangle$ geschrieben. Analog werden Ausgabesequenzen $w \in O^+$ der Länge m im Folgenden als $w = \langle w_1, w_2, \dots w_m \rangle$ geschrieben. Mit diesen Definitionen lassen sich eine *erweiterte Übergangsfunktion* $f^+ : S \times I^+ \rightarrow S$ und eine *erweiterte Ausgabefunktion* $g^+ : S \times I^+ \rightarrow O^+$ definieren [329]:

$$f^+(s,w) := \begin{cases} f(s,w) & \text{falls } |w| = 1 \\ f(f^+(s, \langle w_1, \dots, w_{n-1} \rangle), w_n) & \text{falls } |w| \geq 2 \end{cases} \tag{4.3}$$

$$g^+(s,w) := \begin{cases} g(s,w) & \text{falls } |w| = 1 \\ \langle g^+(s, \langle w_1, \dots, w_{n-1} \rangle), g(f^+(s, \langle w_1, \dots, w_{n-1} \rangle), w_n) \rangle & \text{falls } |w| \geq 2 \end{cases} \tag{4.4}$$

Zu beachten ist, dass die erweiterte Übergangsfunktion f^+ einen einzelnen Zustand berechnet, während die erweiterte Ausgabefunktion g^+ eine Sequenz $w \in O^+$ an Ausgabesymbolen bestimmt.

4.3.1 Automaten-Äquivalenz

Mit Hilfe der Gleichungen (4.3) und (4.4) kann die Äquivalenz zweier endlicher Automaten gezeigt werden.

Definition 4.3.1 (Automaten-Äquivalenz). *Zwei deterministische endliche Automaten* $M := (I, O, S, s_0, f, g)$ *und* $M' := (I', O', S', s'_0, f', g')$ *mit Anfangszustand* s_0 *bzw.* s'_0 *sind* äquivalent *oder* verhaltensgleich*, wenn für eine beliebige Eingabesequenz* $w \in I^+$*, die an beide Automaten angelegt wird, die selbe Ausgabesequenz entsteht, d. h.*

$$\forall w \in I^+ : g^+(s_0, w) = g'^+(s'_0, w)\,.$$

Dieser Definition von Automaten-Äquivalenz liegt eine simulativen Methode zur expliziten Äquivalenzprüfung zu Grunde. Da simulative Verfahren in der Regel unvollständig sind, wird ein geeignetes Überdeckungsmaß zur Bestimmung der Verifikationsvollständigkeit benötigt. Hierin liegt aber ein Problem, da I^+ unendlich groß ist und alle enthaltenen Sequenzen überprüft werden müssen, um eine 100%-ige Vollständigkeit zu erzielen. Somit ist Definition 4.3.1 für einen Beweis der Äquivalenz zweier Automaten ungeeignet.

Um formal die Äquivalenz zweier Automaten zu zeigen, wird zunächst die *Zustandsäquivalenz* definiert.

Definition 4.3.2 (Zustandsäquivalenz). *Gegeben seien zwei deterministische endliche Automaten* $M := (I, O, S, s_0, f, g)$ *und* $M' := (I, O, S', s'_0, f', g')$ *mit Anfangszustand* s_0 *bzw.* s'_0 *und identischem Eingabe- und Ausgabealphabet. Die* Äquivalenzrelation für Zustände $\equiv\ \subseteq S \times S'$ *ist die größte Relation mit*

$$s \equiv s' \Leftrightarrow \forall i \in I : (g(s,i) = g'(s',i)) \wedge (f(s,i) \equiv f'(s',i))\,.$$

Aus Definition 4.3.2 folgt direkt:

$$\forall w \in I^+ : s \equiv s' \Leftrightarrow f^+(s,w) \equiv f'^+(s',w) \tag{4.5}$$

Somit gilt

Theorem 4.3.1. *Zwei deterministische endliche Automaten M und M' sind* äquivalent, *geschrieben als $M \equiv M'$, genau dann, wenn ihre Anfangszustände äquivalent sind, d. h. $s_0 \equiv s'_0$.*

Somit reduziert sich das Problem der Automaten-Äquivalenz auf eine rekursive Definition. Diese kann in eine iterative Überprüfung der Äquivalenz der Ausgaben $g(s,i) = g'(s',i)$ umgewandelt werden. Dabei müssen allerdings nicht alle möglichen Zustandspaare $(s,s') \in S \times S'$ überprüft werden, sondern nur solche, die von den Anfangszuständen s_0 und s'_0 aus unter identischen Eingaben $w \in I^+$ erreichbar sind. Mit anderen Worten: Es dürfen nur die Zustandspaare betrachtet werden, in denen beide Automaten gleichzeitig sein können.

Zur Vereinfachung der Paarbildung kann ein *Produktautomat* gebildet werden.

Definition 4.3.3 (Produktautomat). *Gegeben seien zwei deterministische endliche Automaten $M := (I,O,S,s_0,f,g)$ und $M' := (I,O,S',s'_0,f',g')$ mit Anfangszustand s_0 bzw. s'_0 und identischem Eingabe- und Ausgabealphabet. Der* Produktautomat $M_p := M \times M'$ *ist ein deterministischer endlicher Automat mit $M_p := (I,O,S_p,(s_0,s'_0,f_p,g_p)$ mit Anfangszustand (s_0,s'_0) und Zustandsmenge $S_p := S \times S'$ sowie*

$$f_p((s,s'),i) := (f(s,i), f'(s',i)) \quad \textit{und}$$

$$g_p((s,s'),i) := (g(s,i) = g'(s',i)) \; .$$

Mit anderen Worten: Die Zustände des Produktautomaten M_p sind assoziiert mit Paaren von Zuständen der Automaten M und M'. Der Anfangszustand des Produktautomaten wird mit dem Paar der Anfangszustände (s_0,s'_0) der Automaten M und M' assoziiert. Die Ausgabefunktion g_p liefert T, falls die Ausgabe der beiden Automaten M und M' identisch ist, andernfalls ist die Ausgabe F.

Beispiel 4.3.1. Abbildung 4.16a) zeigt zwei endliche Zustandsautomaten M und M'. Beide Automaten haben das gleiche Eingabe- und Ausgabealphabet mit $I = \{i_1, i_2\}$ und $O = \{o_1, o_2\}$. Während M drei Zustände hat ($S = \{s_1, s_2, s_3\}$), besitzt M lediglich zwei Zustände, d. h. $S' = \{s'_1, s'_2\}$. Der zugehörige Produktautomat M_p mit $|S_p| = 6$ Zuständen ist in Abb. 4.16b) zu sehen.

Mit der Definition von Produktautomaten kann die Definition 4.3.2 von Zustandsäquivalenz neu formuliert werden.

Definition 4.3.4 (Zustandsäquivalenz). *Gegeben seien zwei deterministische endliche Automaten $M := (I,O,S,s_0,f,g)$ und $M' := (I,O,S',s'_0,f',g')$ mit Anfangszustand s_0 bzw. s'_0 und der zugehörige Produktautomat M_p nach Definition 4.3.3. Die* Äquivalenzrelation für Zustände $\equiv \; \subseteq S \times S'$ *ist die größte Relation mit*

$$(s,s') \in \; \equiv \; \Leftrightarrow \forall i \in I : (g_p((s,s'),i) = \mathsf{T}) \wedge (f_p((s,s'),i) \in \; \equiv \;)$$

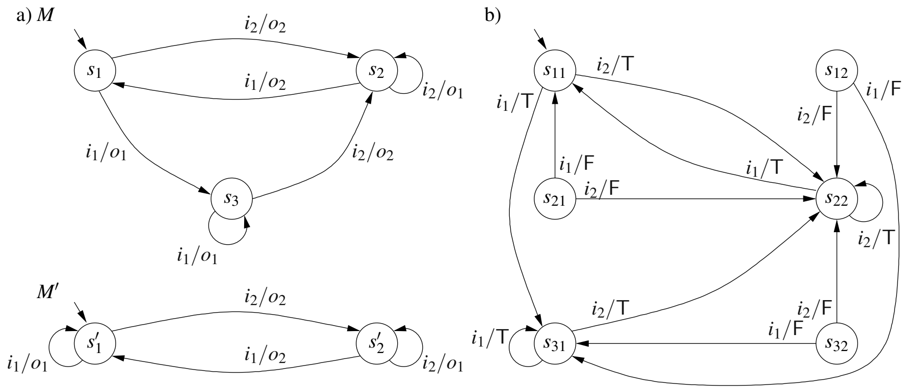


Abb. 4.16. a) Zwei endliche Automaten und b) zugehöriger Produktautomat

4.3.2 Zustandsraumtraversierung

Für den Beweis, dass zwei endliche Automaten äquivalent sind, ist es notwendig, den Zustandsraum des zugehörigen Produktautomaten $M_p = (I, O, S_p, (s_0, s_0'), f_p, g_p)$ zu traversieren, um Zustände zu identifizieren, in denen beide Automaten gleichzeitig sein können. Wird dabei ein Zustand (s, s') erreicht, der die Ausgabe $g_p((s, s'), i) = \mathsf{F}$ für ein beliebiges $i \in I$ erzeugt, so sind die beiden Automaten nicht äquivalent. Wird ein solcher Zustand nicht erreicht, so sind die Automaten äquivalent. Im Folgenden werden zwei Verfahren vorgestellt, die *Vorwärtstraversierung* und die *Rückwärtstraversierung* (siehe auch [329]).

Für die Zustandsraumtraversierung werden zunächst drei Hilfsfunktionen definiert. Gegeben sei eine Teilmenge $S' \subseteq S$ an Zuständen. Die Menge aller Folgezustände, die in einem Zustandsübergang aus Zuständen in S' erreichbar ist, lässt sich definieren als:

$$SUCC(S') := \{s \in S \mid \exists s' \in S', i \in I : f(s', i) = s\} \tag{4.6}$$

Ebenso kann die Menge aller Vorgängerzustände, die mindestens einen Zustand $s' \in S'$ in einem Zustandsübergang erreichen können, definiert werden als:

$$PRED(S') := \{s \in S \mid \exists s' \in S', i \in I : f(s, i) = s'\} \tag{4.7}$$

Mit Hilfe der erweiterten Übergangsfunktion lässt sich auch die Menge aller erreichbaren Zustände definieren:

$$REACH(S') := S' \cup \left\{s \in S \mid \exists s' \in S', w \in I^+ : f^+(s', w) = s\right\} \tag{4.8}$$

Mit diesen Hilfsfunktionen wird die Automaten-Äquivalenz wie folgt neu definiert:

Definition 4.3.5 (Automaten-Äquivalenz). *Zwei deterministische endliche Automaten $M := (I, O, S, s_0, f, g)$ und $M' := (I, O, S', s'_0, f', g')$ mit Anfangszustand s_0 bzw. s'_0 sind* äquivalent*, wenn für den zugehörigen Produktautomaten gilt:*

$$\forall (s, s') \in REACH(\{(s_0, s'_0)\}), i \in I : g_p((s, s'), i) = \mathsf{T}$$

Alle Zustände, die vom Anfangszustand des Produktautomaten erreichbar sind, müssen für alle ausgehenden Zustandsübergänge die Ausgabe T erzeugen. Dies kann auch so formuliert werden, dass kein Zustand existieren darf, der vom Anfangszustand des Produktautomaten aus erreichbar ist und der einen ausgehenden Zustandsübergang mit Ausgabe F besitzt. Das Automaten-Äquivalenzproblem wird somit zu einem Erreichbarkeitsproblem. Dies ist in Abb. 4.17 dargestellt. Dabei bezeichnet $S_= := \{s \in S_p \mid \forall i \in I : g_p(s, i) = \mathsf{T}\}$ die Menge der Zustände, bei denen alle ausgehenden Zustandsübergänge die Ausgabe T erzeugen, und $S_{\neq} := \{s \in S \mid \exists i \in I : g_p(s, i) = \mathsf{F}\}$, die Menge der Zustände, bei denen mindestens ein ausgehender Zustandsübergang die Ausgabe F erzeugt. Die Frage, die es zu beantworten gilt, lautet: Existiert einer der gestrichelten Zustandsübergänge?

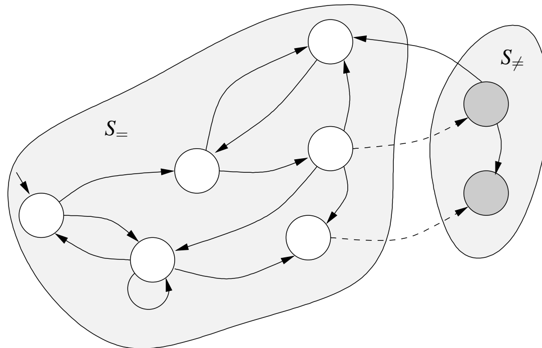


Abb. 4.17. Automaten-Äquivalenz als Erreichbarkeitsproblem [272]

Beispiel 4.3.2. Für den Produktautomaten in Abb. 4.16b) aus Beispiel 4.3.1 bestimmen sich die Mengen $S_=$ und $S_{\neq}$ zu:

$$S_= = \{s_{11}, s_{22}, s_{31}\}$$
$$S_{\neq} = \{s_{12}, s_{21}, s_{32}\}$$

Auf Basis von Definition 4.3.5 kann der folgende Algorithmus zur Erreichbarkeitsanalyse, der auf einer *Vorwärtstraversierung* des Zustandsraumes basiert, verwendet werden:

```
FORWARD_TRAVERSAL(M_p) {
    S_R := {s_0};
    S_N := S_R;
    REPEAT
        IF (∃s ∈ S_N, i ∈ I : g_p(s,i) = F)
            RETURN F;
        S_N := SUCC(S_N)\S_R;
        S_R := S_R ∪ S_N;
    UNTIL (S_N = ∅)
    RETURN T;
}
```

Eingabe ist der Produktautomat M_p mit Zustandsmenge S_p. Die beiden Mengen S_R und S_N enthalten diejenigen Zustände, die bereits traversiert wurden bzw. die in der letzten Iteration neu hinzu gekommen sind. Beide Mengen werden mit dem Anfangszustand des Produktautomaten initialisiert. Anschließend wird der Zustandsraum vorwärts traversiert, bis entweder ein Zustand erreicht wird, der einen ausgehenden Zustandsübergang besitzt, der F ausgibt, oder keine neuen erreichbaren Zustände gefunden werden ($S_N = \emptyset$). In jeder Iteration wird zunächst die Menge aller Zustände $SUCC(S_N)$, die von mindestens einem Zustand in S_N in exakt einem Übergang erreichbar sind, bestimmt. Aus dieser Menge werden die bereits traversierten Zustände gelöscht und anschließend die neu erreichten Zustände als bereits traversiert markiert ($S_R := S_R \cup S_N$). Ist die Menge der neu erreichten Zustände leer ($S_N = \emptyset$), so ist ein *Fixpunkt* erreicht, d. h. eine weitere Traversierung würde keine neuen zu erreichenden Zustände erreichen. Der Algorithmus ist in Abb. 4.18 visualisiert. Hierbei ist in eckigen Klammern der Iterationsschritt angegeben.

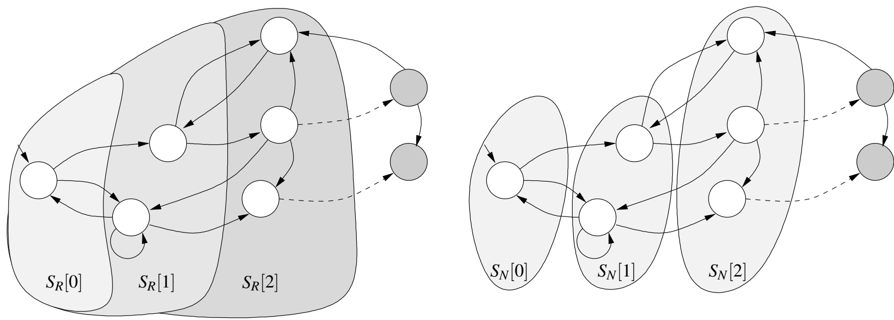


Abb. 4.18. Erreichbarkeitsanalyse durch Vorwärtstraversierung [272]

Sofern keine abweichenden Ausgaben erkannt werden, endet der Algorithmus, wenn keine neuen Zustände in einer Iteration erreicht werden und somit der Fixpunkt gefunden ist. Die Anzahl der Iterationen, nach denen dies eintritt, nennt man *sequentielle Tiefe* des Automaten. Für jeden erreichbaren Zustand $s \in S$ eines Auto-

maten lässt sich die minimale Anzahl an Zustandsübergängen von dem Anfangszustand s_0 zu s ermitteln. Diese wird als sequentielle Tiefe des Zustands s bezeichnet. Die sequentielle Tiefe eines Automaten ist die maximale sequentielle Tiefe seiner erreichbaren Zustände.

Beispiel 4.3.3. Für den Produktautomaten in Abb. 4.16b) aus Beispiel 4.3.1 bestimmen sich die Mengen S_R und S_N zu:

$$S_R[0] = \{s_{11}\},\ S_R[1] = \{s_{11}, s_{22}, s_{31}\},\ S_R[2] = \{s_{11}, s_{22}, s_{31}\} = S_R[1]$$
$$S_N[0] = \{s_{11}\},\ S_N[1] = \{s_{22}, s_{31}\},\qquad S_N[2] = \emptyset$$

Mit der in Beispiel 4.3.2 berechneten Menge $S_{\neq}$ gilt, dass $S_R \cap S_{\neq} = \emptyset$ und somit, dass M und M' aus Abb. 4.16a) äquivalent sind.

Neben vorwärts kann der Zustandsraum auch rückwärts traversiert werden. Um eine *Rückwärtstraversierung* durchführen zu können, muss das Verifikationsziel umformuliert werden, d. h. statt eines Beweises der Korrektheit wird nun eine Falsifikation das Ziel sein. Beginnend mit der Menge derjenigen Zustände, die einen ausgehenden Zustandsübergang besitzen, der die Ausgabe F erzeugt, wird der Zustandsraum rückwärts traversiert, bis entweder keine neuen Zustände besucht werden können, oder aber der Anfangszustand des Automaten erreicht wurde. Kann der Anfangszustand erreicht werden, bedeutet dies, dass auch ein Pfad vom Anfangszustand zu einem Zustand existiert, der einen ausgehenden Zustandsübergang besitzt, der F ausgibt. Hieraus kann geschlossen werden, dass die beiden zu untersuchenden Automaten nicht äquivalent sind. Obwohl hierbei das Verifikationsziel die Falsifikation ist, ist die Verifikationsmethodik vollständig, d. h. wird kein Gegenbeispiel gefunden, sind die Automaten äquivalent.

Die Erreichbarkeitsanalyse auf Basis einer Rückwärtstraversierung ist in folgendem Algorithmus zusammengefasst:

BACKWARD_TRAVERSAL(M_p) {
    $S_N := \{s \in S_p \mid \exists i \in I : g_p(s,i) = \mathsf{F}\}$;
    $S_E := S_P \backslash S_N$;
    REPEAT
        IF ($s_0 \in S_N$)
            RETURN F;
        $S_N := PRED(S_N) \cap S_E$;
        $S_E := S_E \backslash S_N$;
    UNTIL ($S_N = \emptyset$)
    RETURN T;
}

Dem Rückwärtstraversierungsalgorithmus wird als Argument der Produktautomat mit Zustandsmenge S_p übergeben. Die Mengen S_N und S_E bezeichnen die Menge der neu besuchten Zustände bzw. die Menge der noch nicht besuchten Zustände. Werden bei der Rückwärtstraversierung keine neuen Zustände erreicht ($S_N = \emptyset$), so ist ein *Fixpunkt* gefunden und eine weitere Traversierung würde keine zusätzliche Information hervorbringen.

Beispiel 4.3.4. Für den Produktautomaten in Abb. 4.16b) aus Beispiel 4.3.1 bestimmen sich die Mengen S_E und S_N zu:

$$S_E[0] = \{s_{13}, s_{21}, s_{32}\},\ S_E[1] = \{s_{13}, s_{21}, s_{32}\} = S_E[0]$$
$$S_N[0] = \{s_{13}, s_{21}, s_{32}\},\ S_N[1] = \emptyset$$

Da der Anfangszustand s_{11} weder in $S_N[0]$ noch in $S_N[1]$ enthalten ist, gilt wie bereits in Beispiel 4.3.3 gezeigt, dass M und M' aus Abb. 4.16a) äquivalent sind.

Welcher Traversierungsalgorithmus eingesetzt werden sollte, lässt sich nicht generell von vorn herein beantworten. Bei der Vorwärtstraversierung hängt die Anzahl der Iterationen von der sequentiellen Tiefe des Automaten ab. Ist die sequentielle Tiefe gering, ist es vorteilhaft, eine Vorwärtstraversierung durchzuführen. Ist die sequentielle Tiefe hingegen sehr groß, kann die Rückwärtstraversierung die besser Wahl sein, da eventuell alle Zustände des Produktautomaten mit möglicher Ausgabe F nicht erreichbar sind, und der Fixpunkt bei der Rückwärtstraversierung für diese Zustände schnell erreicht ist.

4.3.3 Symbolische Zustandsraumtraversierung

Der bisher beschriebene Ansatz zur sequentiellen Äquivalenzprüfung basiert auf der Traversierung des Produktautomaten zweier deterministischer endlicher Automaten. Dabei wurden die Zustandsmengen explizit aufgezählt, was limitierend für dieses Verfahren ist, da dies sehr speicherintensiv ist. Um dennoch größere Automaten auf Äquivalenz zu überprüfen, bietet es sich an, den Zustandsraum *symbolisch* zu traversieren. Hierzu werden die Zustandsmengen mit Hilfe von sog. *charakteristischen Funktionen implizit* repräsentiert.

Zunächst werden alle Zustände $s \in S_p$ und alle Ein- und Ausgabesymbole $i \in I$ und $o \in O$ binär codiert, d. h.

$$\sigma_{S_p} : S_p \to \mathbb{B}^k,$$
$$\sigma_I : I \to \mathbb{B}^n \text{ und}$$
$$\sigma_O : O \to \mathbb{B}^m.$$

Basierend auf diesen Codierungen kann eine Teilmenge von Zuständen $S' \subseteq S_p$ durch eine charakteristische Funktion repräsentiert werden.

Definition 4.3.6 (Implizite Repräsentation von Zustandsmengen). *Gegeben sei eine Teilmenge $S' \subseteq S_p$ an Zuständen sowie die Codierungsfunktion $\sigma_S : S_p \to \mathbb{B}^k$ der Zustände $s \in S_p$. Die* charakteristische Funktion $\psi_{S'} : \mathbb{B}^k \to \mathbb{B}$ *repräsentiert eindeutig die Teilmenge S' mit*

$$\forall s \in S_p : \psi_{S'}(\sigma_S(s)) = \mathsf{T} \Leftrightarrow s \in S'.$$

Beispiel 4.3.5. Betrachtet wird wiederum der Produktautomat aus Beispiel 4.3.1 der in Abb. 4.16b) zu sehen ist. Das Eingabealphabet I und das Ausgabealphabet O haben jeweils zwei Elemente. Somit können die Elemente beider Mengen jeweils mit

einer einzelnen binären Variablen codiert werden. Im Folgenden sind die Codierungen $\sigma_I(i_1) = \neg x_1$ und $\sigma_I(i_2) = x_1$, sowie $\sigma_O(\mathsf{F}) = \neg y_1$ und $\sigma_O(\mathsf{T}) = y_1$ angenommen. Weiterhin besteht der Produktautomat aus sechs Zuständen, die durch die drei binären Variablen z_1, z_2, z_3 repräsentiert werden. Die Codierung ist in der folgenden Tabelle gegeben:

s	$\sigma_{S_p}(s)$	s	$\sigma_{S_p}(s)$
s_{11}	$\neg z_3 \wedge \neg z_2 \wedge \neg z_1$	s_{12}	$\neg z_3 \wedge \neg z_2 \wedge z_1$
s_{21}	$\neg z_3 \wedge z_2 \wedge \neg z_1$	s_{22}	$\neg z_3 \wedge z_2 \wedge z_1$
s_{31}	$z_3 \wedge \neg z_2 \wedge \neg z_1$	s_{32}	$z_3 \wedge \neg z_2 \wedge z_1$

In Beispiel 4.3.2 wurden die Mengen $S_= = \{s_{11}, s_{22}, s_{31}\}$ und $S_{\neq} = \{s_{12}, s_{21}, s_{32}\}$ bestimmt. Mit der obigen Codierung σ_{S_p} lassen sich die charakteristischen Funktionen dieser beiden Funktionen erstellen:

$$\psi_{S_=}(z_3, z_2, z_1) = (\neg z_3 \wedge \neg z_2 \wedge \neg z_1) \vee (\neg z_3 \wedge z_2 \wedge z_1) \vee (z_3 \wedge \neg z_2 \wedge \neg z_1)$$

$$\psi_{S_{\neq}}(z_3, z_2, z_1) = (\neg z_3 \wedge \neg z_2 \wedge z_1) \vee (\neg z_3 \wedge z_2 \wedge \neg z_1) \vee (z_3 \wedge \neg z_2 \wedge z_1)$$

Neben der Darstellung von Mengen können charakteristische Funktionen auch zur Repräsentation von Funktionen verwendet werden. Für eine symbolische Traversierung des Zustandsraums ist vor allem die Repräsentation der Ausgabe- und Übergangsfunktion, g_p und f_p, von Interesse. Die Übergangsfunktion f_p kann als charakteristische Funktion ψ_{f_p} wie folgt definiert werden:

$$\psi_{f_p}(\sigma_{S_p}(s), \sigma_I(i), \sigma_{S_p}(s')) = \begin{cases} \mathsf{T} & \text{falls } f_p(s,i) = s' \\ \mathsf{F} & \text{sonst} \end{cases} \tag{4.9}$$

Analog lässt sich die Ausgabefunktion g_p als charakteristische Funktion ψ_{g_p} definieren:

$$\psi_{g_p}(\sigma_{S_p}(s), \sigma_I(i), \sigma_O(o)) = \begin{cases} \mathsf{T} & \text{falls } g_p(s,i) = o \\ \mathsf{F} & \text{sonst} \end{cases} \tag{4.10}$$

Beispiel 4.3.6. Betrachtet wird der Produktautomat aus Abb. 4.16b) sowie die in Beispiel 4.3.5 eingeführte Codierung. Die charakteristische Funktion ψ_{f_p} der Übergangsfunktion f_p ergibt sich zu:

$$\begin{aligned}
\psi_{f_p} = {} & (\neg z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (\neg z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge x_1 \wedge \neg z_3' \wedge z_2' \wedge z_1') \vee \\
& (\neg z_3 \wedge \neg z_2 \wedge z_1 \wedge \neg x_1 \wedge z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (\neg z_3 \wedge \neg z_2 \wedge z_1 \wedge x_1 \wedge \neg z_3' \wedge z_2' \wedge z_1') \vee \\
& (\neg z_3 \wedge z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge \neg z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (\neg z_3 \wedge z_2 \wedge \neg z_1 \wedge x_1 \wedge \neg z_3' \wedge z_2' \wedge z_1') \vee \\
& (\neg z_3 \wedge z_2 \wedge z_1 \wedge \neg x_1 \wedge \neg z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (\neg z_3 \wedge z_2 \wedge z_1 \wedge x_1 \wedge \neg z_3' \wedge z_2' \wedge z_1') \vee \\
& (z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge x_1 \wedge \neg z_3' \wedge z_2' \wedge z_1') \vee \\
& (z_3 \wedge \neg z_2 \wedge z_1 \wedge \neg x_1 \wedge z_3' \wedge \neg z_2' \wedge \neg z_1') \vee \\
& (z_3 \wedge \neg z_2 \wedge z_1 \wedge x_1 \wedge \neg z_3 \wedge z_2 \wedge z_1)
\end{aligned}$$

Die charakteristische Funktion ψ_{g_p} der Ausgabefunktion g_p ergibt sich zu:

$$\begin{aligned}\psi_{g_p} = {} & (\neg z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge y_1) \vee (\neg z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge x_1 \wedge y_1) \vee \\ & (\neg z_3 \wedge \neg z_2 \wedge z_1 \wedge \neg x_1 \wedge \neg y_1) \vee (\neg z_3 \wedge \neg z_2 \wedge z_1 \wedge x_1 \wedge \neg y_1) \vee \\ & (\neg z_3 \wedge z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge \neg y_1) \vee (\neg z_3 \wedge z_2 \wedge \neg z_1 \wedge x_1 \wedge \neg y_1) \vee \\ & (\neg z_3 \wedge z_2 \wedge z_1 \wedge \neg x_1 \wedge y_1) \vee (\neg z_3 \wedge z_2 \wedge z_1 \wedge x_1 \wedge y_1) \vee \\ & (z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge \neg x_1 \wedge y_1) \vee (z_3 \wedge \neg z_2 \wedge \neg z_1 \wedge x_1 \wedge y_1) \vee \\ & (z_3 \wedge \neg z_2 \wedge z_1 \wedge \neg x_1 \wedge \neg y_1) \vee (z_3 \wedge \neg z_2 \wedge z_1 \wedge x_1 \wedge \neg y_1)\end{aligned}$$

Charakteristische Funktionen lassen sich effizient als *reduzierte, geordnete binäre Entscheidungsdiagramme* (engl. *Reduced Ordered Binary Decision Diagrams, ROBDDs*) darstellen (siehe Anhang B.2). Neben der kompakten Darstellung erlauben ROBDDs auch eine effiziente Durchführung von Mengenoperationen. Die Schnittmenge $S_1 \cap S_2$ zweier Mengen, $S_1, S_2 \subseteq S$, lässt sich als Konjunktion der entsprechenden charakteristischen Funktionen ψ_{S_1} und ψ_{S_2} berechnen, d. h. $\psi_{S_1} \wedge \psi_{S_2}$. Die Vereinigungsmenge $S_1 \cup S_2$ zweier Mengen, $S_1, S_2 \subseteq S$, lässt sich als Disjunktion der charakteristischen Funktionen ψ_{S_1} und ψ_{S_2} berechnen, d. h. $\psi_{S_1} \vee \psi_{S_2}$.

Die Äquivalenzprüfung, basierend auf einer symbolischen Vorwärtstraversierung, kann nach folgendem Algorithmus durchgeführt werden:

SYMBOLIC_FORWARD_TRAVERSAL($M_p, \sigma_I, \sigma_O, \sigma_{S_p}$) {
  $\psi_R := \sigma_{S_p}(s_0)$;
  $\psi_N := \psi_R$;
  REPEAT
    IF ($\psi_N \wedge \psi_{g_p} \wedge \sigma_O(\mathsf{F}) \neq \mathsf{F}$)
      RETURN F;
    $\psi_{SUCC} := \left(\exists \sigma_{S_p}(s), \sigma_I(i) : \psi_N \wedge \psi_{f_p}\right)$;
    $\psi_N := \psi_{SUCC} \wedge \neg \psi_R$;
    $\psi_R := \psi_R \vee \psi_N$;
  UNTIL ($\psi_N = \mathsf{F}$)
  RETURN T;
}

Die Funktion SYMBOLIC_FORWARD_TRAVERSAL liefert T, falls die beiden Automaten M und M', aus denen der Produktautomat M_p gebildet wurde, äquivalent sind. Sind die beiden Automaten nicht äquivalent, so ist der Rückgabewert F. Der Algorithmus arbeitet analog zur nichtsymbolischen Vorwärtstraversierung und bestimmt dabei einen Fixpunkt ($\psi_N = \mathsf{F}$). Allerdings ist die Bestimmung, ob ein Zustand neu erreicht wurde, der einen ausgehenden Zustandsübergang mit Ausgabe F besitzt, die Überprüfung, ob das ROBDD der Funktion $\psi_N \wedge \psi_{g_p} \wedge \sigma_O(\mathsf{F})$, der Terminalknoten F ist. Ebenfalls lässt sich die Bestimmung der Menge aller in einem Schritt erreichbaren Zustände effizient durch eine Konjunktion und zwei Existenz-Quantifizierungen berechnen ($\exists \sigma_{S_p}(s), \sigma_I(i) : \psi_N \wedge \psi_{f_p}$).

Obwohl die symbolische Zustandsraumtraversierung deutlich größere Zustandsräume traversieren kann, als dies mit der nicht symbolischen Traversierung möglich wäre, hat auch dieser Ansatz Schwachpunkte. Ein wesentlicher Schwachpunkt liegt

in der relationalen Repräsentation der Übergangsfunktion f_p als charakteristische Funktion ψ_{f_p}. Erfolgt die Codierung $\sigma_{S_p} : S_p \to \mathbb{B}^k$ der Zustandsmenge S_p mit k Variablen und die Codierung $\sigma_I : I \to \mathbb{B}^n$ der Eingabesymbole I mit n Variablen, so besitzt die charakteristische Funktion ψ_{f_p} der Übergangsfunktion $2 \cdot k + n$ Variablen als Argument. Da ROBDDs exponentiell in der Anzahl der Variablen wachsen können, ist der vorhandene Speicher auch bei der symbolischen Äquivalenzprüfung häufig der limitierende Faktor.

4.3.4 Erzeugung von Gegenbeispielen

Sind zwei endliche Zustandsautomaten M und M' nicht äquivalent, so wäre es wünschenswert eine Sequenz $\langle i_1, \ldots, i_n \rangle$ zu erhalten, welche die Ausgabe F im Produktautomaten M_p, beziehungsweise die unterschiedliche Ausgabe in den beiden Automaten auslöst. Diese Sequenz kann beispielsweise später als Testfalleingabe in der Simulation verwendet werden. Hierzu muss die Traversierung des Zustandsraums leicht modifiziert werden:

```
FORWARD_TRAVERSAL(M_p) {
    k := −1;
    S_R := {s_0};
    S_N[0] := S_R;
    REPEAT
        k := k + 1;
        IF (∃s ∈ S_N[k], i ∈ I : g_p(s,i) = F)
            RETURN GENERATE_TRACE(M_p, k, s);
        S_N[k+1] := SUCC(S_N[k])\S_R;
        S_R := S_R ∪ S_N[k+1];
    UNTIL (S_N[k+1] = ∅)
    RETURN T;
}
```

Hierbei werden alle im Iterationsschritt k neu besuchten Zustände in der Menge $S_N[k]$ gespeichert. Man bezeichnet $S_N[k]$ auch als k-te Front der Zustandsraumtraversierung. Wird nun festgestellt, dass M und M' nicht äquivalent sind, so wird eine Eingabesequenz mit Hilfe der Funktion GENERATE_TRACE bestimmt und zurück gegeben. Dabei wird der Zustand *error*, der die Ausgabe F erzeugen kann, als Argument übergeben.

```
GENERATE_TRACE(M_p, k, error) {
    w := ⟨i | i ∈ I ∧ g_p(error,i) = F⟩;
    FOR (j := k; j ≥ 1; j := j − 1)
        pred :∈ PRED({error});
        w := ⟨i | i ∈ I ∧ f_p(pred,i) = error⟩ ∘ w;
        error := pred;
    RETURN w;
}
```

Zunächst wird ein Eingabesymbol i bestimmt, welches ausgehend vom Zustand *error* die Ausgabe F erzeugt. Anschließend wird für jeden Iterationsschritt ein Vorgängerzustand *pred* ausgewählt und eine Eingabe bestimmt, die den Übergang von *pred* nach *error* auslöst. Diese Eingabe wird per Konkatenation $\circ$ an den Anfang der Sequenz gestellt. Dies wird k-mal wiederholt. Dann ist der Anfangszustand erreicht und somit die Sequenz an Eingaben zur Erzeugung der Ausgabe von F von M_p abgeschlossen.

4.4 Strukturelle Äquivalenzprüfung

Während implizite Verfahren einen hohen Speicherbedarf haben, um Repräsentationen des Systems zu erstellen, leiden explizite Verfahren häufig an einer hohen Laufzeit. Orthogonal zu impliziten und expliziten Verfahren können sog. *strukturelle Verfahren* zur Äquivalenzprüfung verwendet werden. Strukturelle Verfahren basieren auf der Idee der Partitionierung. Durch Partitionierung können die betrachteten Systeme in überschaubare, kleinere Einheiten zerlegt werden. Dies ist in Abb. 4.19 für eine explizite Äquivalenzprüfung zu sehen.

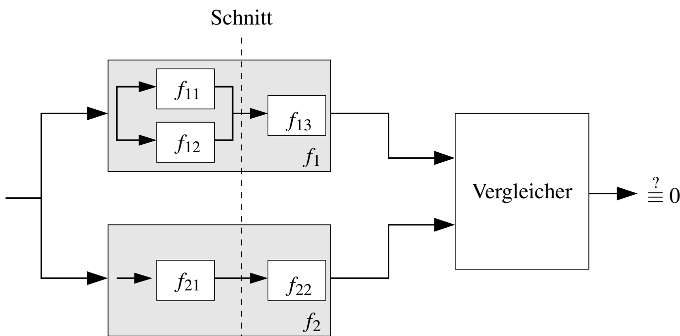


Abb. 4.19. Strukturelle Äquivalenzprüfung

Entlang des *Schnittes* (engl. *cut*) werden die beiden Systeme, welche die Funktionen f_1 und f_2 implementieren, in jeweils zwei Teilsysteme zerlegt. Eine mögliche Vorgehensweise, um die Äquivalenz von f_1 und f_2 zu zeigen, besteht nun darin, die Äquivalenz der beiden Teilsysteme von den primären Eingängen bis zu den *Schnittpunkten* (engl. *cut points*) zu zeigen und anschließend die Signale an den Schnittpunkten zu substituieren. Solch eine Substitution ist in Abb. 4.20 für das Beispiel aus Abb. 4.19 dargestellt. Nachdem gezeigt wurde, dass die beiden Teilsysteme an den Eingängen der beiden Systeme äquivalent sind, können die Signale des oberen Teilsystems im unteren Teilsystem wiederverwendet werden. Man sieht, dass dabei die Gesamtgröße beider Systeme reduziert wurde, was das eigentliche Verifikationsproblem verkleinert.

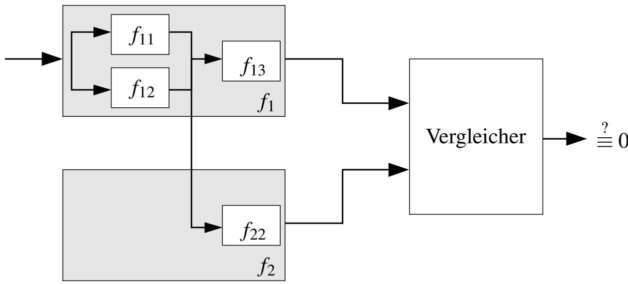


Abb. 4.20. Substitution nach Partitionierung zur strukturellen Äquivalenzprüfung

Ein alternativer Ansatz, der auf Partitionierung des Systems beruht, reduziert die Problemgröße noch drastischer: Nachdem die Äquivalenz der Teilsysteme (Partitionen) von den primären Eingängen der Systeme bis zu den Schnittpunkten gezeigt wurde, werden die Teilsysteme von den Schnittpunkten bis zu den primären Ausgängen unabhängig von den vorderen Teilsystemen auf Äquivalenz überprüft, d. h. die Schnittpunkte werden die neuen Eingänge für den Vergleich.

Auch wenn hierdurch der Verifikationsaufwand drastisch reduziert wird, so ist das Ergebnis dennoch kritisch zu betrachten. Durch den Schnitt sind die Teilsysteme entkoppelt, d. h. obwohl die Ausgänge der Teilsysteme zwischen primären Eingängen und Schnittpunkten die Eingaben für die Teilsysteme zwischen Schnitt und primären Ausgängen bilden, geht dieses Wissen durch den Schnitt verloren. Dies ist in Abb. 4.21 visualisiert. Abbildung 4.21a) zeigt ein System bestehend aus zwei Teilsystemen, welche die Funktionen f_1 und f_2 implementieren. Zusätzlich sind der Eingabe- und der Ausgabebereich des Systems zu sehen. Aufgrund der gewählten Abstraktion sind allerdings lediglich die dunkel hervorgehobenen Bereiche steuerbar bzw. beobachtbar.

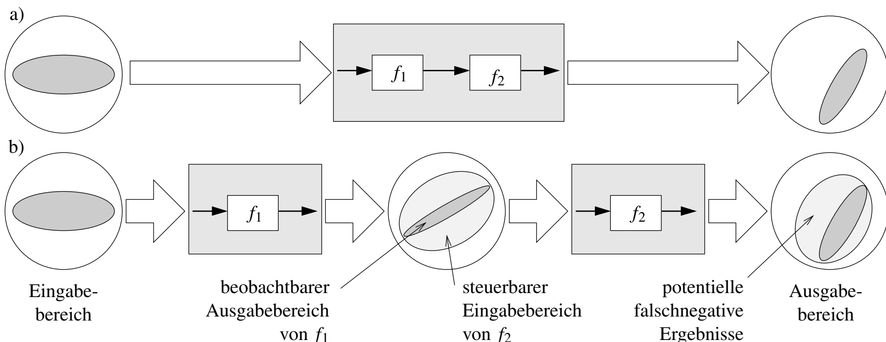


Abb. 4.21. Entstehung falschnegativer Ergebnissen bei der strukturellen Äquivalenzprüfung

In Abb. 4.21b) ist das partitionierte System zu sehen. Der steuerbare Eingabebereich ist identisch mit dem Gesamtsystem aus Abb. 4.21a). Am Schnittpunkt zwischen f_1 und f_2 ergibt sich für das linke Teilsystem ein beobachtbarer Ausgabebereich, der dunkel hervorgehoben ist. Durch die Entkopplung der Teilsysteme kann nun der Effekt beobachtet werden, dass der steuerbare Eingabebereich der Funktion f_2 nicht zwangsläufig deckungsgleich mit diesem Ausgabebereich sein muss. Dieser steuerbare Eingabebereich für f_2 führt zu einem vergrößerten beobachtbaren Ausgabebereich.

Werden bei einer expliziten Äquivalenzprüfung lediglich die Partitionen von zwei Systemen verglichen, kann es zu Nicht-Äquivalenzen kommen, die in dem vergrößerten beobachtbaren Ausgabebereich einer Partition liegen, d. h. die im Gesamtsystem nicht hätten auftreten können. Diese Fälle werden als *falschnegative Ergebnisse* (engl. *false negative*) bezeichnet, da sie zu einem negativen, aber falschen, Verifikationsergebnis führen. Aus diesem Grund ist es notwendig, bei einer strukturellen Äquivalenzprüfung nochmals zu überprüfen, ob das gefundene Gegenbeispiel für die Nichtäquivalenz tatsächlich von den primären Eingängen des Systems aus angesteuert werden kann. Allgemein kann man sagen, dass nicht aus der Äquivalenz von zwei Systemen und der Äquivalenz der Systempartitionen vor dem Schnitt auf die Äquivalenz der Systempartitionen nach dem Schnitt geschlossen werden kann. Der Grund hierfür liegt darin, dass nicht alle Belegungen an den Schnittpunkten angesteuert werden können. Entsprechend können die Systempartitionen hinter dem Schnitt auch nichtäquivalent bei nicht erreichbaren Belegungen sein.

Spezielle Verfahren zur strukturellen Äquivalenzprüfung von Hardware werden in Abschnitt 6.1.4 diskutiert. Strukturelle Verfahren zur Äquivalenzprüfung von Software werden in Abschnitt 7.1.2 beschrieben.

4.5 Literaturhinweise

Taylor-Expansion-Diagramme (TEDs) wurden von Ciesielski et al. eingeführt [94]. Ihre Anwendung für Verifikation auf Architekturebene und Logikebene sind in [251] bzw. [93] dargestellt. Fey et al. untersuchen in [163] Algorithmen für TEDs. Verfahren zum dynamischen Umordnen von Variablen in TEDs sind in [202] beschrieben. Eine Übersicht über TEDs und ihren Einsatz für die Verifikation findet man in [92].

Diversifizierende Methoden zur simulativen Äquivalenzprüfung sind ausführlich in [305] erläutert. Eine Übersicht zum Bereichstest findet man in [108]. Verfahren zur Prüfung der Äquivalenz von endlichen Automaten sind ausführlich in [272, 329] behandelt. An selber Stelle sind auch strukturelle Verfahren zur Äquivalenzprüfung beschrieben.

5

Eigenschaftsprüfung

Ziel der Eigenschaftsprüfung ist es, zu testen, ob die Implementierung funktionale und nichtfunktionale Anforderungen erfüllt (siehe Abb. 5.1). Bei der Prüfung funktionaler Eigenschaften wird das Strukturmodell der Implementierung dahin gehend überprüft, ob dieses alle funktionalen Anforderungen erfüllt. Dabei werden typischerweise Gefahrlosigkeits- und Lebendigkeitseigenschaften geprüft. Eine Gefahrlosigkeitseigenschaft besagt, dass nie etwas Schlimmes passieren wird, während eine Lebendigkeitseigenschaft besagt, dass immer irgendwann etwas Gutes passieren wird.

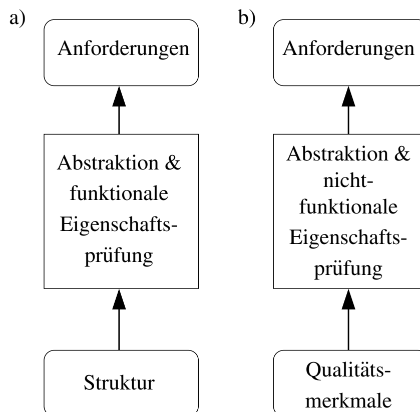


Abb. 5.1. a) funktionale Eigenschaftsprüfung und b) Prüfung nichtfunktionaler Eigenschaften

Die Prüfung nichtfunktionaler Eigenschaften zeigt, dass die abgeschätzten Qualitätsmerkmale der Implementierung allen nichtfunktionalen Anforderungen der Spezifikation genügen. Nichtfunktionale Anforderungen können dabei z. B. eine maximale Antwortzeit, ein minimaler Durchsatz, eine minimale MTTF (engl. *Mean Ti-*

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_5, © Springer-Verlag Berlin Heidelberg 2010

me To Failure) oder eine maximale Leistungs- und Energieaufnahme sein. Obwohl viele der genannten nichtfunktionalen Eigenschaften eingebetteter Systeme in der Praxis von großer Bedeutung sind, liegt der Fokus des vorliegenden Buchs auf der Betrachtung rein zeitlicher Eigenschaften, da eingebettete Systeme nahezu immer Echtzeitbeschränkungen unterliegen.

Im Folgenden werden die Grundlagen zur Prüfung funktionaler und nichtfunktionaler Eigenschaften präsentiert. Zunächst werden unterschiedliche Verfahren zur funktionalen Eigenschaftsprüfung für Petri-Netze vorgestellt, wobei sich zeigt, dass insbesondere Methoden, die Systemzustände aufzählen, eine besondere Bedeutung gewonnen haben. Aus diesem Grund wird einer der wichtigsten Vertreter aus diesem Bereich, die sog. *Modellprüfung* detaillierter betrachtet. Hierbei wird zunächst die *explizite* und anschließend die *symbolische* Modellprüfung vorgestellt. Dabei wird sowohl auf die *BDD-basierte* als auch die *SAT-basierte* Modellprüfung eingegangen. Letztere ist heutzutage der wichtigste Vertreter der Modellprüfungsverfahren. Zum Abschluss dieses Kapitels wird noch auf die Prüfung des Zeitverhaltens eingegangen, wobei sowohl kontrollfluss- als auch datenflussdominante Modelle betrachtet werden.

5.1 Prüfung funktionaler Eigenschaften

In Kapitel 2.2.1 wurden Definitionen für dynamische Eigenschaften von Petri-Netzen gegeben. Die wichtigsten Eigenschaften, die in diesem Abschnitt betrachtet werden, lauten zusammengefasst: *Beschränktheit* (Definition 2.2.5 auf Seite 44), *Lebendigkeit* (Definition 2.2.8 auf Seite 44) und *Reversibilität* (Definition 2.2.9 auf Seite 45). Bei den Definitionen wurden nicht darauf eingegangen, wie für ein gegebenes Petri-Netz gezeigt werden kann, ob es diese Eigenschaften besitzt. Dies ist Gegenstand der Eigenschaftsprüfung, die in diesem Kapitel diskutiert wird. Hierzu werden zunächst die wesentlichen Eigenschaften nochmals an Beispielen wiederholt.

Beispiel 5.1.1. Betrachtet wird das Petri-Netz G mit der Anfangsmarkierung M_0 in Abb. 5.2a). Durch Schalten der Transition t_2 wird die Folgemarkierung M_1 erreicht, also $M_0[t_2\rangle M_1$, wobei $M_1(p_1) = M_1(p_2) = 0$ und $M_1(p_3) = M_1(p_4) = 1$ ist. Durch Schalten von Transition t_4 wird die Folgemarkierung M_2 erreicht, wobei $M_2(p_2) = M_2(p_4) = 0$ und $M_2(p_1) = M_2(p_3) = 1$ ist.

Wiederholt man die Sequenz $\langle t_2, t_4\rangle$ insgesamt k mal so erreicht man die Folgemarkierung $M_{2k}(p_2) = M_{2k}(p_4) = 0$, $M_{2k}(p_1) = 1$ und $M_{2k}(p_3) = k$. Da man die Sequenz beliebig oft wiederholen kann, also k beliebig groß werden kann, ist die Stelle p_3 unbeschränkt. Als Modell eines eingebetteten Systems würde die Stelle p_3 beispielsweise einen Speicher darstellen. Da physikalische Speicher eine endliche Größe besitzen, würde es zwangsläufig bei wiederholter Ausführung der Sequenz $\langle t_2, t_4\rangle$ zu Speicherüberläufen kommen. Da G eine unbeschränkte Stelle enthält, ist G selbst *unbeschränkt*.

Andererseits kann beginnend mit der Anfangsmarkierung M_0, dargestellt in Abb. 5.2a), die Transition t_1 schalten. Die Folgemarkierung M_1 mit $M_0[t_1\rangle M_1$ ergibt sich zu $M_1(p_1) = M_1(p_3) = M_1(p_4) = 0$ und $M_1(p_2) = 1$. Allerdings ist unter

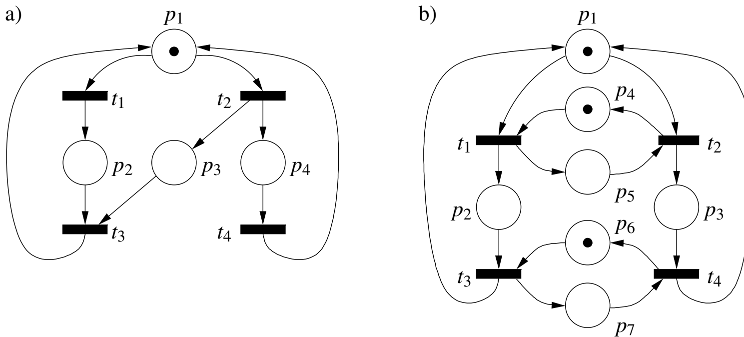


Abb. 5.2. a) unbeschränktes Petri-Netz und b) sicheres Petri-Netz [189]

der Markierung M_1 keine weitere Transition schaltbereit. Somit ist G zusätzlich auch nicht *verklemmungsfrei*.

Betrachtet man hingegen das Petri-Netz G aus Abb. 5.2b), so sieht man, dass G sowohl *1-beschränkt* als auch *stark lebendig* ist. Allerdings müssen diese Eigenschaften nicht monoton in der Anfangsmarkierung sein, d. h. wird die Menge der Marken in der Anfangsmarkierung vergrößert, können Eigenschaften wie die Verklemmungsfreiheit verloren gehen. In dem Beispiel in Abb. 5.2b) könnte zusätzlich zu der dargestellten Anfangsmarkierung eine Marke auf p_5 positioniert werden. In diesem Fall ist es denkbar, dass t_2 zuerst schaltet, was zu einer Verklemmung des Petri-Netzes führt.

In einem Petri-Netz wird eine Markierung M als *Grundzustand* bezeichnet, falls dieser Zustand aus allen aus M erreichbaren Zuständen wiederum erreichbar ist. Ist der Anfangszustand M_0 ein Grundzustand, so heißt das Petri-Netz *reversibel*.

Beispiel 5.1.2. Das Petri-Netz G, das in Abb. 5.3 dargestellt ist, ist nicht reversibel. Allerdings sind alle Folgezustände $M \in [M_0\rangle \setminus \{M_0\}$ Grundzustände.

Beschränktheit, Lebendigkeit und Reversibilität sind „gute“ Eigenschaften, die typischerweise von Systemen gefordert werden. Dabei sind diese drei Eigenschaften unabhängig voneinander, was leicht mit Hilfe von kleinen Beispielen gezeigt werden kann [189]. Die acht möglichen Kombinationen aus dem Vorhandensein bzw. Nichtvorhandensein der Eigenschaften sind in Abb. 5.4 zu sehen.

Prinzipiell können bei Methoden zur Eigenschaftsprüfung auf Petri-Netzen zwei Ansätze unterschieden werden. Zum einen werden *aufzählende* Verfahren eingesetzt, um erreichbare Zustände eines Petri-Netzes zu erfassen. Zum anderen werden *strukturelle* Verfahren eingesetzt, um das dynamische Verhalten von Petri-Netzen zu beschreiben. Beide Ansätze werden im Folgenden genauer beschrieben. Zum Abschluss wird noch ein zentrales Problem bei der Verifikation digitaler Systeme, die sich als Petri-Netz modellieren lassen, betrachtet: Durch die potentiell nebenläufige

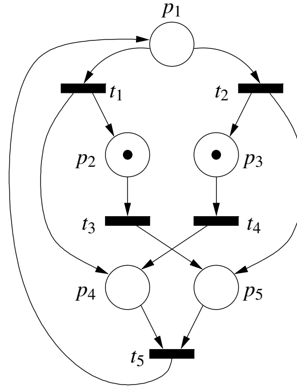


Abb. 5.3. Nichtreversibles Petri-Netz [189]

Ausführung von Transitionen, kann es unterschiedliche Schaltfolgen in einem Petri-Netz geben, die zu der selben Markierung führen. Die Anzahl dieser unterschiedlichen Sequenzen kann sehr groß werden, was eine effiziente Eigenschaftsprüfung verhindern kann. Eine sog. *Partialordnungsreduktion* hilft, lediglich repräsentative Schaltfolgen auszuwählen.

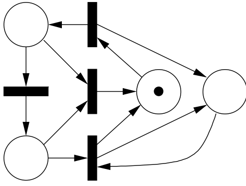
5.1.1 Eigenschaftsprüfung auf Erreichbarkeitsgraphen

Eine Möglichkeit Systeme, die als Petri-Netz modelliert sind, bezüglich der Eigenschaften Beschränktheit, Lebendigkeit und Reversibilität zu überprüfen, besteht darin, die Erreichbarkeitsmenge $[M_0\rangle$ als sog. *Erreichbarkeitsgraphen* zu repräsentieren. Eine Verifikation im Allgemeinen ist allerdings nur möglich, wenn das Petri-Netz beschränkt und somit die Erreichbarkeitsmenge endlich ist. Ist das Petri-Netz hingegen unbeschränkt, muss der resultierende unendliche Erreichbarkeitsgraph durch einen endlichen Erreichbarkeitsgraphen angenähert werden. In diesem Fall können nur noch eingeschränkte Anforderungen verifiziert werden.

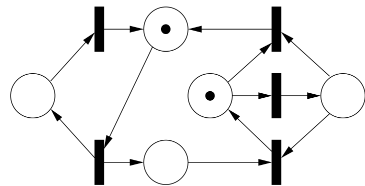
Definition 5.1.1 (Erreichbarkeitsgraph). *Gegeben sei ein Petri-Netz $G(P,T,F,K,W,M_0)$ mit Erreichbarkeitsmenge $[M_0\rangle$. Der* Erreichbarkeitsgraph $RG(G) = (V,E,L_v,L_e,v_0)$ *von G besteht aus Knoten $v \in V$, welche die erreichbaren Markierungen $M \in [M_0\rangle$ repräsentieren, Kanten $e \in E$, die Markierungsänderungen durch Schalten einer Transition $t \in T$ darstellen, einer* Knotenmarkierungsfunktion $L_v : V \rightarrow [M_0\rangle$, *die mit jedem Knoten $v \in V$ eine Markierung assoziiert, einer* Kantenmarkierungsfunktion $L_e : E \rightarrow T$, *die jeder Kante eine Transition zuordnet, und einem Quellknoten $v_0 \in V$.*

Um den Erreichbarkeitsgraphen für unbeschränkte Petri-Netze konstruieren zu können, bedarf es einer Abstraktion für unbeschränkte Stellen. Im Folgenden wird das Symbol ∞ verwendet, um eine unbeschränkte Stelle zu kennzeichnen. Zur Detektion unbeschränkter Stellen wird der Begriff der *Zustandsdominanz* eingeführt.

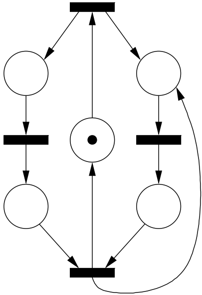
a)



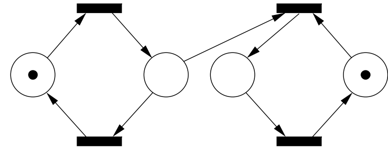
b) R



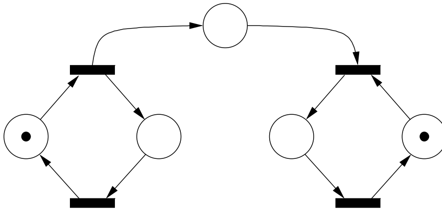
c) V



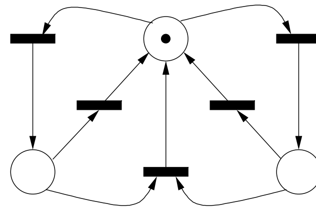
d) B



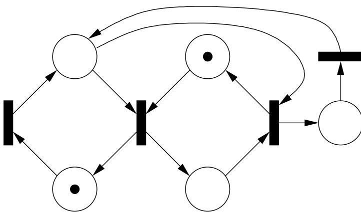
e) VR



f) BR



g) BV



h) BVR

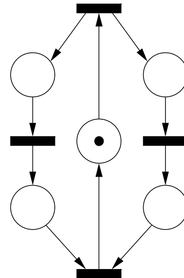


Abb. 5.4. Beschränktheit (B), Verklemmungsfreiheit (V) und Reversibilität (R) sind voneinander unabhängige Eigenschaften [189]

Definition 5.1.2 (Zustandsdominanz). *Gegeben sei ein Petri-Netz $G(P,T,F,K,W,M_0)$ sowie zwei Markierungen $M_i, M_j \in [M_0\rangle$. M_i* dominiert *M_j, geschrieben als $M_i \succ M_j$, wenn $M_i \succ M_j \Leftrightarrow (\forall p \in P : M_i(p) \geq M_j(p)) \wedge (\exists p \in P : M_i(p) > M_j(p))$ gilt.*

Die Konstruktion des Erreichbarkeitsgraphen kann dann wie folgt geschehen:

```
CONSTRUCT_REACHABILITY_GRAPH(G) {
    RG := ({v_0}, {}, (v_0, M_0), {}, v_0);
    V_N := V;
    FOREACH v ∈ V_N
        FOREACH t ∈ {t' ∈ T | L_v(v)[t'⟩}
            L(v)[t⟩M';
            IF (∃v' ∈ V : L_v(v') = M')
                E := E ∪ {(v, v')};
                L_e(v, v') := t;
            ELSE
                M' := UPDATE(RG, v, M');
                V := V ∪ {v''};
                V_N := V_N ∪ {v''};
                E := E ∪ {(v, v'')};
                L_v(v'') := M';
                L_e(v, v') := t;
        V_N := V_N \ {v};
    RETURN RG;
}
```

Zunächst wird der Erreichbarkeitsgraph RG mit einem Quellknoten v_0, mit dem die Anfangsmarkierung M_0 assoziiert ist, initialisiert. Dabei ist die Menge der Kanten sowie die Kantenmarkierungsfunktion leer. Anschließend werden solange neue Knoten und Kanten hinzugefügt, wie neue Markierungen durch Schalten von Transitionen ermittelt werden können. Für jeden neuen Knoten werden die schaltbereiten Transitionen und die resultierenden Folgemarkierungen M' ermittelt. Existiert bereits ein Knoten v' mit einer solchen Markierung in RG, so wird lediglich eine neue Kante $e = (v, v')$ hinzugefügt und mit der entsprechenden Transition beschriftet. Existiert hingegen kein Knoten v' mit der selben Markierung, wird ein neuer Knoten v'' und eine neue Kante $e = (v, v'')$ hinzugefügt. Die neue Kante wird ebenfalls wieder mit der entsprechenden Transition beschriftet. Der neue Knoten v'' erhält die Knotenmarkierung $L_v(v'') := M'$, wobei diese zunächst mit der Funktion UPDATE auf eine mögliche Zustandsdominanz untersucht wird. Die Funktion UPDATE sieht wie folgt aus:

```
UPDATE(RG, v_N, M) {
    FOREACH ṽ ∈ {v̂ = (v_0, ..., v_i, ... v_N) ∈ RG | ∀1 ≤ i ≤ N : (v_{i-1}, v_i) ∈ E}
        IF (∃v_i ∈ ṽ : M ≻ L_v(v_i))
            ∀p ∈ P, M(p) > L_v(v_i)(p) : M(p) := ∞;
    RETURN M;
}
```

In der Prozedur UPDATE wird überprüft, ob es im Erreichbarkeitsgraphen auf den Pfaden vom Quellknoten v_0 zum Knoten v_N einen Knoten v_i gibt, der von der Markierung M dominiert wird. Ist dies der Fall, wird jedes Element $M(p)$ der Markierung, was echt größer dem Element $L_v(v_i)(p)$ in der dominierten Markierung ist, durch das Symbol ∞ ersetzt. Dies zeigt an, dass die Stelle p unbeschränkt ist. Falls eine solche Markierung nicht existiert, bleibt M unverändert.

Beispiel 5.1.3. Gegeben ist das Petri-Netz in Abb. 5.5a). Die Anfangsmarkierung M_0 ist gegeben als $M_0(p_1) = M_0(p_2) = 1$ und $M_0(p_3) = 0$. In Vektorschreibweise wird dies als $M_0 = [1,1,0]$ dargestellt. Für die Konstruktion des Erreichbarkeitsgraphen wird zunächst ein Knoten v_0 erzeugt und mit der Anfangsmarkierung M_0 beschriftet. Anschließend werden alle schaltbereiten Transition $t \in \{t' \in T \mid M_0[t'\rangle\}$ bestimmt und die resultierenden Folgemarkierungen bestimmt. Unter der Markierung M_0 ist lediglich die Transition t_1 schaltbereit. Die resultierende Folgemarkierung M_1 mit $M_0[t_1\rangle M_1$ ist $[0,0,1]$. Da diese Markierung weder gleich der Anfangsmarkierung M_0 ist noch diese dominiert, wird ein neuer Knoten v_1 in den Erreichbarkeitsgraphen eingefügt, mit der Markierung M_1 beschriftet und mit einer neuen Kante $e = (v_0, v_1)$ verbunden, die mit t_1 beschriftet ist.

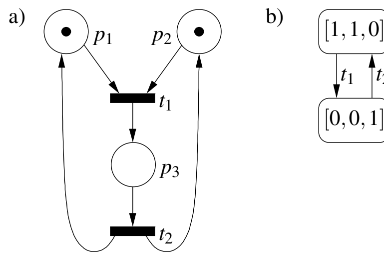


Abb. 5.5. a) Petri-Netz und b) zugehöriger Erreichbarkeitsgraph [87]

Für die Markierung M_1 werden wieder alle schaltbereiten Transitionen bestimmt. In diesem Fall die Transition t_2. Die neue Markierung, die sich durch Schalten von t_2 ergibt, lautet $[1,1,0]$. Diese Markierung ist identisch zu M_0, weshalb eine Kante von v_1 zu v_0 zu dem Erreichbarkeitsgraphen hinzugefügt wird. Da in diesem Schritt kein neuer Knoten hinzugefügt wurde, ist die Konstruktion des Erreichbarkeitsgraphen abgeschlossen. Das Ergebnis ist in Abb. 5.5b) zu sehen.

Beispiel 5.1.4. Ein Erreichbarkeitsgraph mit unbeschränkten Stellen ist in Abb. 5.6b) dargestellt. Das zugehörige Petri-Netz ist in Abb. 5.6a) zu sehen.

Die Konstruktion des Erreichbarkeitsgraphen erfolgt in sechs Schritten:

1. Zu Beginn wird der Knoten v_0 mit der Anfangsmarkierung $M_0 = [1,0,0,0]$ beschriftet und dem Erreichbarkeitsgraphen hinzugefügt.
2. Ausgehend von M_0 werden alle schaltbereiten Transitionen $\{t' \in T \mid M_0[t'\rangle\}$ bestimmt. Unter der Markierung M_0 ist lediglich die Transition t_1 schaltbereit.

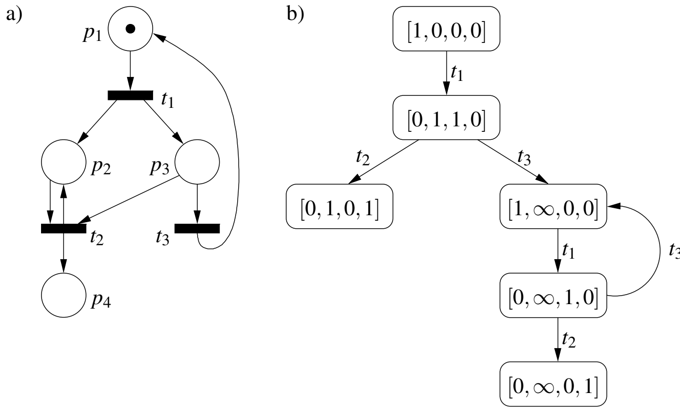


Abb. 5.6. a) Petri-Netz und b) zugehöriger Erreichbarkeitsgraph [87]

Die Folgemarkierung nach Schalten von t_1 lautet $M_1 = [0,1,1,0]$. Da $M_1 \not\succ M_0$ und $M_1 \neq M_0$ gilt, wird ein neuer Knoten v_1 erzeugt, dieser mit M_1 markiert, d. h. $L_v(v_1) := M_1$, und dem Erreichbarkeitsgraphen hinzugefügt. Schließlich wird eine Kanten (v_0, v_1) zu der Menge der Kanten E des Erreichbarkeitsgraphen hinzugefügt und mit t_1 markiert.

3. Basierend auf M_1 werden die schaltbereiten Transitionen bestimmt. Die Transitionen t_2 und t_3 sind unter M_1 schaltbereit.
   a) Zunächst wird die Transition t_2 betrachtet. Die Folgemarkierung M_2 mit $M_1[t_2\rangle M_2$ lautet $M_2 = [0,1,0,1]$. Es gilt sowohl $M_2 \not\succ M_1$ als auch $M_2 \not\succ M_0$ und ein Knoten v mit $L_v(v) = M_2$ existiert nicht. Deshalb wird ein neuer Knoten v_2 erzeugt, $L_v(v_2) := M_2$ gesetzt und v_2 zu dem Erreichbarkeitsgraphen hinzugefügt. Eine Kante (v_1, v_2) verbindet die Knoten v_1 und v_2 und wird mit $L_e(v_1, v_2) := t_2$ markiert.
   b) Die Folgemarkierung M_3, welche durch Schalten der Transition t_3 erreicht wird, lautet $M_3 = [1,1,0,0]$. Diese Markierung dominiert zwar nicht die Markierung M_1 des Knoten v_1, jedoch gilt $M_3 \succ M_0$. Aus diesem Grund wird in der Prozedur UPDATE die Markierung M_3 in $M_3 := [1,\infty,0,0]$ geändert. Anschließend wird ein Knoten v_3 erzeugt, dieser mit der geänderten Markierung M_3 beschriftet und über eine Kante (v_1, v_3) mit dem Knoten v_1 verbunden. Die Kante wird mit $L_e(v_1, v_3) := t_3$ markiert.
4. Für die im letzten Schritt neu hinzugefügten Knoten v_2 und v_3 und deren assoziierten Markierungen werden die schaltbereiten Transitionen bestimmt. Unter der Markierung M_2 ist keine Transition schaltbereit, weshalb der Knoten v_2 ein Blatt ist. Unter der Markierung $M_3 = [1,\infty,0,0]$ ist lediglich die Transition t_1 schaltbereit. Durch Schalten der Transition t_1 ergibt sich die Folgemarkierung M_4 mit $M_3[t_1\rangle M_4$ zu $M_4 = [0,\infty,1,0]$. M_4 dominiert die Markierung M_1. Allerdings führt die Prozedur UPDATE keine Änderung auf M_4 aus, da $M_4(p_2) > M_1(p_2)$ ist, $M_4(p_2)$ aber bereits ∞ ist. Es wird entsprechend ein neuer Knoten v_4 er-

zeugt, mit der Knotenmarkierung $L_v(v_4) := M_4$ beschriftet und über die Kante (v_3, v_4) mit Kantenmarkierung $L_e(v_3, v_4) := t_1$ verbunden.

5. Unter der Markierung M_4 sind wiederum die Transitionen t_2 und t_3 schaltbereit:
   a) Durch Schalten der Transition t_2 wird die Folgemarkierung $M_5 = [0, \infty, 0, 1]$ erreicht. Diese Markierung dominiert keine Markierung, die mit einem Knoten auf dem Pfad zum Quellknoten v_0 assoziiert ist und kein Knoten mit dieser Markierung bereits existiert. Somit wird ein Knoten v_5 erzeugt, mit der Markierung M_5 beschriftet und über die mit $L_e(v_4, v_5) := t_2$ beschriftete Kante (v_4, v_5) verbunden.
   b) Durch Schalten der Transition t_3 wird die Folgemarkierung $[1, \infty, 0, 0]$ erreicht. Diese Markierung ist bereits mit dem Knoten v_3 assoziiert, weshalb lediglich eine neue Kante (v_4, v_3) im Erreichbarkeitsgraphen aufgenommen wird. Diese Kante wird mit t_3 markiert.
6. Schließlich wird für die Markierung M_5 des neuen Knoten die Menge der schaltbereiten Transitionen bestimmt. Da diese Menge leer ist, sind keine weiteren Markierungen erreichbar und v_5 ist ein Blatt.

Erreichbarkeitsgraphen $RG(V, E, L_v, L_e, v_0)$ für verklemmungsfreie Petri-Netze können als *temporale Struktur* $M(S, R, L)$ interpretiert werden (siehe Definition 2.4.1 auf Seite 73). Die Zustandsmenge S der temporalen Struktur entspricht den Knoten V des Erreichbarkeitsgraphen. Die Kanten E des Erreichbarkeitsgraphen entsprechen der Übergangsrelation R der temporalen Struktur. Die Markierungsfunktion L der temporalen Struktur ist identisch mit der Knotenmarkierungsfunktion L_v des Erreichbarkeitsgraphen, wobei im Fall des Erreichbarkeitsgraphen die Knoten mit erreichbaren Markierungen des Petri-Netzes markiert werden. Der Anfangszustand der temporalen Struktur stimmt mit dem Quellknoten des Erreichbarkeitsgraphen überein.

Verifikationsmethode für Gefahrlosigkeitseigenschaften

Der Erreichbarkeitsgraph wird im Folgenden als Datenstruktur für Verifikationsmethoden verwendet. Ziel dabei ist es, zu zeigen, dass ein gegebenes Petri-Netz geforderte funktionale Eigenschaften erfüllt. Die vorgestellten Verfahren besitzen dabei im schlechtesten Fall eine exponentielle Laufzeit in der Anzahl der Stellen $|P|$ des Petri-Netzes, und eine polynomielle Laufzeit in der Anzahl der Knoten $|V|$ des Erreichbarkeitsgraphen.

Für die Eigenschaftsprüfung wird zunächst eine Verifikationsmethode für *Gefahrlosigkeitseigenschaften* vorgestellt. Das Verifikationsziel ist hierbei der Beweis, dass die Annahme, diese Gefahrlosigkeitseigenschaft gelte, erfüllt ist. Anschließend wird eine Verifikationsmethode für *Lebendigkeitseigenschaften* präsentiert. Hierbei ist das Verifikationsziel der Beweis, dass ein bestimmter Zustand immer irgendwann eintritt. Zur Formulierung des Beweisziels werden sog. *Markierungsprädikate* definiert.

Definition 5.1.3 (Markierungsprädikat). *Ein* Markierungsprädikat *ist eine aussagenlogische Formel* φ*, deren atomare Formeln aus Ungleichungen der Form*

$$\sum_{p \in \widetilde{P}} k_p \cdot M(p) \leq k$$

bestehen, wobei k_p und k Konstanten sind und $\widetilde{P}$ eine Teilmenge von Stellen ist.

Bei einer Gefahrlosigkeitseigenschaft muss die Formel φ in allen erreichbaren Zustände gelten, d. h. $\forall M \in [M_0\rangle : \varphi$. Beispiele für Gefahrlosigkeitseigenschaften sind [189]:

1. *k-Beschränktheit* einer Stelle $p \in P$ (mit $\varphi := M(p) \leq k$):

$$\forall M \in [M_0\rangle : M(p) \leq k$$

2. Gegenseitiger Ausschluss von p und p' (mit $\varphi := (M(p) = 0) \vee (M(p') = 0)$):

$$\forall M \in [M_0\rangle : (M(p) = 0) \vee (M(p') = 0)$$

3. *Verklemmungsfreiheit* (mit $\varphi := \exists t \in T : M[t\rangle$):

$$\forall M \in [M_0\rangle : \exists t \in T : M[t\rangle$$

Ein Beweisverfahren für Gefahrlosigkeitseigenschaften sieht wie folgt aus [189]:

```
DECIDE_SAFETY(RG, φ) {
        FOREACH v ∈ V
                IF (L_v(v) ⊭ φ)
                        RETURN F;
        RETURN T;
}
```

Es werden also lediglich alle Knoten des Erreichbarkeitsgraphen traversiert und die Formel φ überprüft. Ist diese nicht erfüllt ($L_v(v) \not\models \varphi$), so ist das Verifikationsziel verfehlt. Sind hingegen alle Knoten traversiert und alle assoziierten Markierungen erfüllen die Formel, so ist die Gefahrlosigkeitseigenschaft bewiesen und es wird T zurück gegeben.

Beispiel 5.1.5. Betrachtet wird das Petri-Netz aus Abb. 5.6a) und die Formeln:

$$\begin{aligned}
\varphi_1 &:= M(p_2) \leq 1 \\
\varphi_2 &:= M(p_3) + M(p_4) \leq 1 \\
\varphi_3 &:= \exists t \in T : M[t\rangle
\end{aligned}$$

Die Formel φ_1 ist zwar in den Knoten v_0, v_1 und v_2 erfüllt, da für Knoten v_3 aber $L_v(v_3)(p_2) = \infty$ gilt, ist φ_1 unter der Markierung $L_v(v_3)$ verletzt. Die Algorithmus DECIDE_SAFETY gibt F zurück. Die Formel φ_2 ist hingegen in allen Knoten $v \in V$ des Erreichbarkeitsgraphen erfüllt. Somit liefert der Algorithmus dafür T zurück.

Die Formel φ_3 wird entsprechend Definition 2.2.2 auf Seite 43 wie folgt umgewandelt:

$$\begin{aligned}\exists t \in T: &(\forall p \in \bullet t : M(p) \geq W(p,t)) \wedge \\ &(\forall p \in t\bullet \setminus \bullet t : M(p) \leq K(p) - W(t,p)) \wedge \\ &(\forall p \in t\bullet \cap \bullet t : M(p) \leq K(p) - W(t,p) + W(p,t))\end{aligned}$$

Die Existenzquantifizierung kann durch eine Prüfung über alle Transitionen realisiert werden. Für das Petri-Netz aus Abb. 5.6a) liefert der Algorithmus F zurück, da sowohl unter der Markierung $L_v(v_2)$ als auch $L_v(v_5)$ keine Transition schaltbereit ist.

Aus dem Beispiel kann man schließen, dass ein Petri-Netz genau dann sicher ist, wenn in den Knotenmarkierungen des Erreichbarkeitsgraphen das Symbol ∞ nicht auftritt. Weiterhin ist ein Petri-Netz verklemmungsfrei, wenn im Erreichbarkeitsgraphen kein Blatt (Knoten ohne Nachfolger) existiert.

Verifikationsmethode für Lebendigkeitseigenschaften

Die oben betrachteten Gefahrlosigkeitseigenschaften können durch eine Traversierung der Knoten des Erreichbarkeitsgraphen geprüft werden. Lebendigkeitseigenschaften besitzen typischerweise eine komplexere Struktur. Eine Klasse von Lebendigkeitseigenschaften besitzt beispielsweise die Form $\forall M \in [M_0\rangle : \exists M' \in [M\rangle : \varphi$, wobei φ die irgendwann zu erfüllende Formel beschreibt.

Beispiele für Lebendigkeitseigenschaften dieser Form sind:

1. *Aktivierbarkeit* der Transition t (mit $\varphi := M'[t\rangle$):

$$\forall M \in [M_0\rangle : \exists M' \in [M\rangle : M'[t\rangle$$

2. Die Markierung M'' ist ein *Grundzustand* (mit $\varphi := (M' = M'')$):

$$\forall M \in [M_0\rangle : \exists M' \in [M\rangle : (M' = M'')$$

3. *Reversibilität* (mit $\varphi := (M' = M_0)$):

$$\forall M \in [M_0\rangle : \exists M' \in [M\rangle : (M' = M_0)$$

Ein Algorithmus für den Beweis von Lebendigkeitseigenschaften als Verifikationsziel sieht wie folgt aus [189]:

```
DECIDE_LIVENESS(RG, φ) {
    C := DECOMPOSE(RG);
    RGC := REDUCE(RG,C);
    V_L := LEAF_NODES(RGC);
    WHILE (V_L ≠ ∅)
        FOREACH C_i ∈ V_L
            IF (∄M ∈ C_i : M |= φ)
                RETURN F;
            V_L := V_L\{C_i};
    RETURN T;
}
```

Die Funktion DECOMPOSE bestimmt die größten Teilgraphen im Erreichbarkeitsgraphen, die stark zusammenhängend sind (siehe Anhang A). Diese werden als starke Zusammenhangskomponenten (engl. *Strongly Connected Components, SCC*) bezeichnet. Die wesentliche Eigenschaft einer starken Zusammenhangskomponente ist, dass jeder enthaltene Knoten von jedem anderen enthaltenen Knoten aus über einen gerichteten Pfad erreichbar ist. Die Menge C der Zusammenhangskomponenten wird von der Funktion DECOMPOSE zurückgegeben.

Anschließend wird der Erreichbarkeitsgraph reduziert, indem jede SCC in RG durch einen einzelnen Knoten ersetzt wird, d. h. $RGC := (V_C, E_C)$ mit $V_C := C$ und $E_C := \{e = (C_i, C_j) \in C \times C \mid \exists e' = (v', v'') \in E : L_v(v') \in C_i \wedge L_v(v'') \in C_j \wedge i \neq j\}$. Es gibt also genau dann eine gerichtete Kante zwischen zwei SCCs C_i und C_j, wenn vorher mindestens eine Kante zwischen einem Knoten $v' \in C_i$ und einem Knoten $v'' \in C_j$ existierte. Dabei werden alle Knotenmarkierungen, mit denen Knoten in einer SCC beschriftet sind, mit der SCC selbst assoziiert. Die Blätter des reduzierten Erreichbarkeitsgraphen RGC, also diejenigen Knoten ohne ausgehende Kanten, werden mit der Funktion LEAF_NODES bestimmt und in der Menge V_L gespeichert. Anschließend wird in jedem Blatt des reduzierten Erreichbarkeitsgraphen geprüft, ob mit diesem Knoten eine Markierung assoziiert ist, durch die φ erfüllt ist. Ist dies nicht der Fall, so gibt der Algorithmus F zurück. Erfüllt hingegen jedes Blatt diese Bedingung, so ist der Beweis erbracht, dass die geforderte Eigenschaft gilt. Der Algorithmus liefert T zurück.

Die Idee bei dieser Verifikationsmethode ist, dass Lebendigkeitseigenschaften nicht durch eine sequentielle Traversierung der Knoten des Erreichbarkeitsgraphen geprüft werden können, da von jeder Markierung aus gelten muss, dass wieder eine Markierung erreichbar ist, welche die Formel φ erfüllt. Die Blätter des reduzierten Erreichbarkeitsgraphen RGC repräsentieren SCCs ohne wegführende Kanten, also Teilgraphen des Erreichbarkeitsgraphen, in denen jeder Knoten von jedem anderen Knoten aus erreichbar ist. Außerdem führt keine Kante aus einer solchen SCC heraus, so dass diese nicht mehr verlassen werden kann. Erfüllt eine Markierung, die mit diesem Blatt assoziiert ist, die Formel φ, so muss in der entsprechenden SCC ein Knoten existieren, der mit dieser Markierung assoziiert wird. Weiterhin ist dieser Knoten immer wieder erreichbar, sobald ein Knoten der selben SCC erreicht wurde (per definitionem). Besitzt nun jedes Blatt des reduzierten Erreichbarkeitsgraphen eine assoziierte Knotenmarkierung, welche die Formel φ erfüllt, ist mit der Tatsache, dass irgendwann eine Knotenmarkierung aus genau einem Blatt erreicht wird, bewiesen, dass die Lebendigkeitseigenschaft erfüllt ist.

Beispiel 5.1.6. Für das Petri-Netz in Abb. 5.3 auf Seite 158 sollen Lebendigkeitseigenschaften mit folgenden Formeln gezeigt werden:

$$\varphi_1 := M'[t\rangle$$
$$\varphi_2 := M' = [1,0,0,0,0]$$
$$\varphi_3 := M' = M_0$$

Zunächst wird der Erreichbarkeitsgraph konstruiert. Dieser ist in Abb. 5.7 zu sehen.

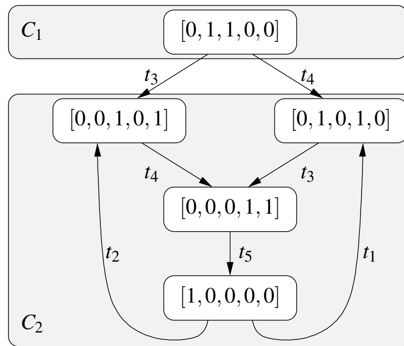


Abb. 5.7. Erreichbarkeitsgraph des Petri-Netzes aus Abb. 5.3

Der Erreichbarkeitsgraph enthält zwei starke Zusammenhangskomponenten C_1 und C_2. Um zu zeigen, dass das Petri-Netz aus Abb. 5.3 *stark lebendig* ist, muss die Lebendigkeitseigenschaft $\forall M \in [M_0\rangle : \exists M' \in [M\rangle : \varphi_1$ bewiesen werden. Hierzu reicht es aus, in der starken Zusammenhangskomponente C_2, dem einzigen Blatt, die Aktivierbarkeit $\exists M : M[t\rangle$ jeder Transition $t \in T$ zu zeigen. Dies ist erfüllt, da alle fünf möglichen Transitionen als Kantenmarkierung in L_e erscheinen.

Dass es sich bei der Markierung $M'' = [1,0,0,0,0]$ um einen *Grundzustand* handelt, kann man beweisen, indem man die Lebendigkeitseigenschaft $\forall M \in [M_0\rangle : \exists M' \in [M\rangle : \varphi_2$ beweist. Hierfür muss in der starken Zusammenhangskomponente C_2 die Existenz der Markierung M'' gezeigt werden, was durch eine Traversierung der Knoten in C_2 erreicht wird.

Schließlich soll gezeigt werden, dass das Petri-Netz *reversibel* ist, d. h. dass es von jeder erreichbaren Markierung in den Anfangszustand zurückversetzt werden kann. Hierfür wird die Lebendigkeitseigenschaft $\forall M \in [M_0\rangle : \exists M' \in [M\rangle : (M' = M_0)$ geprüft. Wiederum wird versucht, die Existenz eines Knotens mit der entsprechenden Markierung M_0 in C_2 zu finden. Dies schlägt allerdings fehl, da M_0 mit der starken Zusammenhangskomponente C_1 assoziiert ist. Somit ist, wie bereits vorher behauptet, das Petri-Netz aus Abb. 5.3 nicht reversibel.

5.1.2 Strukturelle Eigenschaftsprüfung von Petri-Netzen

Das dynamische Verhalten eines Petri-Netzes kann durch ein lineares Gleichungssystem beschrieben werden. Dieses Gleichungssystem bildet die Grundlage für eine andere Art von Verifikationsmethode für Petri-Netze, die sog. *strukturelle Eigenschaftsprüfung*. Gegeben sei ein Petri-Netz $G(P,T,F,K,W,M_0)$ (siehe Definition 2.2.1 auf Seite 41). Das dynamische Verhalten, welches sich durch das Schalten einer Transition ergibt, ist in Definition 2.2.3 auf Seite 43 beschrieben. Falls keine Kante zwischen p und t bzw. t und p existiert, werden die Kantengewichte $W(p,t) := 0$ bzw. $W(t,p) := 0$ angenommen. Damit kann die Folgemarkierung

$M'(p)$ einer Stelle $p \in P$ aus der momentanen Markierung $M(p)$ beim Schalten der Transition $t \in T$ durch

$$M'(p) := M(p) - W(p,t) + W(t,p) \tag{5.1}$$

beschrieben werden. Gleichung (5.1) beschreibt also, wie das Schalten einer Transition $t \in T$ die Markierung einer Stelle $p \in P$ ändert. Diese Gleichung kann direkt in eine Vektorgleichung erweitert werden, die beschreibt, wie sich die Markierungen aller Stellen des Petri-Netzes ändern, unter der Annahme, dass eine bestimmte Transition $t \in T$ schaltet.

$$\begin{pmatrix} M'(p_1) \\ \vdots \\ M'(p_{|P|}) \end{pmatrix} := \begin{pmatrix} M(p_1) \\ \vdots \\ M(p_{|P|}) \end{pmatrix} - \begin{pmatrix} W(p_1,t) + W(t,p_1) \\ \vdots \\ W(p_{|P|},t) + W(t,p_{|P|}) \end{pmatrix} \tag{5.2}$$

Im Folgenden gibt der sog. *Schaltvektor* u an, ob eine Transition $t \in T$ schaltet oder nicht, d. h.

$$u := \left(u_1, \ldots, u_i, \ldots, u_{|T|}\right)^\top$$

mit

$$u_i := \begin{cases} 0 \text{ falls } t_i \text{ nicht schaltet} \\ 1 \text{ falls } t_i \text{ schaltet} \end{cases}$$

Schaltet also lediglich die Transition $t_i \in T$, so lautet der zugehörige Schaltvektor $u = (0, \ldots, 0, 1, 0 \ldots, 0)$, also derjenige Schaltvektor, der lediglich eine 1 an der i-ten Position und 0 an allen anderen Positionen hat.

Weiterhin wird eine sog. *Inzidenzmatrix* $A \in \mathbb{Z}^{|P| \times |T|}$ definiert, die angibt wie sich die Markierung aller Stellen $p \in P$ beim Schalten von Transitionen $t \in T$ ändern.

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1j} & \cdots & a_{1|T|} \\ \vdots & \ddots & & & \vdots \\ a_{i1} & \cdots & a_{ij} & \cdots & a_{i|T|} \\ \vdots & & & \ddots & \vdots \\ a_{|P|1} & \cdots & a_{|P|j} & \cdots & a_{|P||T|} \end{pmatrix}$$

mit

$$a_{ij} := W(t_j, p_i) - W(p_i, t_j)$$

Mit anderen Worten: In der Inzidenzmatrix A zeigt jedes Element a_{ij} die Differenz zwischen den durch die Transition t_j auf der Stelle p_i produzierten und den von der Transition t_j aus der Stelle p_i konsumierten Marken bei Schalten von Transition t_j an. Die Inzidenzmatrix eines Petri-Netzes kann aus der Struktur des Petri-Netzes abgeleitet werden, weshalb Verifikationsmethoden auf Basis der Inzidenzmatrix *strukturelle Methoden* heißen.

Das dynamische Verhalten eines Petri-Netzes G mit Inzidenzmatrix A kann aus Gleichung (5.2) abgeleitet und durch die folgende sog. *Zustandsgleichung* beschrieben werden:

$$m' := m + A \cdot u \tag{5.3}$$

Dabei stellen $m = (M(p_1), \ldots, M(p_{|P|}))^\top$ die momentane Markierung von G und $m' = (M'(p_1), \ldots, M'(p_{|P|}))^\top$ die Folgemarkierung nach dem Schalten der im Schaltvektor u definierten Transitionen dar.

Beispiel 5.1.7. Betrachtet wird wiederum das Petri-Netz aus Abb. 5.3 auf Seite 158. Die Inzidenzmatrix ergibt sich zu:

$$A := \begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix}$$

Die Anfangsmarkierung M_0 ist gegeben durch $M_0(p_2) = M_0(p_3) = 1$ und $M_0(p_1) = M(p_4) = M(p_5) = 0$. Dies wird durch den Vektor $m_0 = (0,1,1,0,0)^\top$ repräsentiert. Das Schalten der Transition t_3 wird durch den Schaltvektor $u = (0,0,1,0,0)^\top$ repräsentiert. Die Folgemarkierung M_1 mit $M_0[t_3\rangle M_1$ kann mit Hilfe von Gleichung (5.3) berechnet werden:

$$\begin{pmatrix} M_1(p_1) \\ M_1(p_2) \\ M_1(p_3) \\ M_1(p_4) \\ M_1(p_5) \end{pmatrix} := \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Die Folgemarkierung ergibt sich somit zu $M_1(p_1) = M_1(p_2) = M_1(p_4) = 0$ und $M_1(p_3) = M_1(p_5) = 1$. Unter dieser Folgemarkierung ist lediglich t_4 schaltbereit. Das Schalten von t_4 wird durch den Schaltvektor $u = (0,0,0,1,0)^\top$ repräsentiert. Die resultierende Folgemarkierung M_2 ergibt sich zu:

$$\begin{pmatrix} M_2(p_1) \\ M_2(p_2) \\ M_2(p_3) \\ M_2(p_4) \\ M_2(p_5) \end{pmatrix} := \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Bei Gleichung (5.3) handelt es sich um ganzzahliges lineares Gleichungssystem. Aufgrund der Linearität können Schaltfolgen im Schaltvektor u zusammengefasst werden. Dazu wird der Schaltvektor u neu definiert, wobei $u_i \in \mathbb{Z}_{\geq 0}$ die Anzahl der Schaltungen von t_i beschreibt. Dabei ist allerdings zu beachten, dass die Lösbarkeit von Gleichung (5.3) lediglich eine notwendige Bedingung für die Erreichbarkeit der Folgemarkierung m' aus m darstellt. Eine hinreichende Bedingung ist die Existenz einer gültigen Schaltfolge, die durch den Schaltvektor repräsentiert wird.

Beispiel 5.1.8. Für das Beispiel 5.1.7 soll die Folgemarkierung M' aus der Markierung M_2 für die Schaltfolge $\langle t_5, t_1, t_3, t_5, t_2, t_4\rangle$ bestimmt werden. Diese Schaltfolge

wird durch den Schaltvektor $u = (1,1,1,1,2)$ repräsentiert. Man sieht, dass dabei auch das mehrfache Schalten der selben Transition berücksichtigt ist. Die Folgemarkierung M' ergibt sich wiederum aus Gleichung (5.3):

$$\begin{pmatrix} M_2(p_1) \\ M_2(p_2) \\ M_2(p_3) \\ M_2(p_4) \\ M_2(p_5) \end{pmatrix} := \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Man sieht, dass die Folgemarkierung M' identisch zur Ausgangsmarkierung M_2 ist. Durch die Lösung des linearen Gleichungssystems ist allerdings noch keine hinreichende Bedingung zur Erreichbarkeit von M' aus M_2 gezeigt. Durch Simulation der Schaltfolge $\langle t_5, t_1, t_3, t_5, t_2, t_4 \rangle$ kann dies aber gezeigt werden.

Auch wenn im vorhergehenden Beispiel die Lösung des Gleichungssystems und anschließende Simulation als unnötiger Mehraufwand scheint, ist im Allgemeinen das Lösen des Gleichungssystems schneller als eine Simulation. Insbesondere bei unlösbaren Gleichungssystemen kann man unnötige Simulationszeit einsparen.

Verifikationsmethoden

Strukturelle Eigenschaftsprüfung von Petri-Netzen erfolgt unter Verwendung der Inzidenzmatrix und der Zustandsgleichung (5.3). Im Folgenden wird gezeigt, wie die Konservativität und die Reversibilität eines Petri-Netzes mit Hilfe der strukturellen Beweismethode geprüft werden kann.

Konservativität

Ein *konservatives* Petri-Netz besitzt die Eigenschaft, dass es seine Marken konserviert, d. h. die gewichtete Summe der Marken im Netz konstant ist. Dies kann (folgend Definition 2.2.10 auf Seite 45) durch folgende Gleichung beschrieben werden:

$$\forall M \in [M_0\rangle : \sum_{p \in P} w(p) \cdot M(p) = \sum_{p \in P} w(p) \cdot M_0(p) \tag{5.4}$$

mit

$$\forall p \in P : w(p) > 0$$

Man beachte, dass die Konservativität eine stärkere Bedingung als die Beschränktheit ist, d. h. jedes konservative Petri-Netz nach Definition 2.2.10 ist auch beschränkt.

Um zu einer Verifikationsmethode mit dem Verifikationsziel des Beweises für die Konservativität von Petri-Netzen zu gelangen, muss Gleichung (5.4) durch Umformung aus Gleichung (5.3) erhalten werden. Hierfür wird Gleichung (5.3) zunächst von links mit dem Zeilenvektor $i_\mathrm{P} = (i_1, \ldots, i_{|P|})$ multipliziert:

$$i_\mathrm{P} \cdot m' = i_\mathrm{P} \cdot m + i_\mathrm{P} \cdot A \cdot u \tag{5.5}$$

Eine hinreichende Bedingung für die Konservativität eines Petri-Netzes ist das Verschwinden des letzten Terms $i_P \cdot A \cdot u$. Dies muss aber für alle gültigen Schaltfolgen, die durch u dargestellt werden können, gelten, weshalb

$$i_P \cdot A = 0 \tag{5.6}$$

gefordert wird. Die Existenz eines solchen strikt positiven Vektors i_P ist eine hinreichende Bedingung für die Konservativität eines Petri-Netzes. Da i_P die Stellen des Petri-Netzes gewichtet, wird i_P auch als *Stelleninvariante* bezeichnet.

Beispiel 5.1.9. Für das Petri-Netz aus Abb. 5.3 auf Seite 158 aus Beispiel 5.1.7 folgt aus Gleichung (5.6):

$$(i_1, i_2, i_3, i_4, i_5) \cdot \begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Die kleinste ganzzahlige, strikt positive Lösung lautet $i_P = (2, 1, 1, 1, 1)$. Also ist das Petri-Netz konservativ, da die Existenz dieser Lösung für den Beweis bereits hinreichend ist.

Reversibilität

Für ein *reversibles* Petri-Netz gilt laut Definition 2.2.9 auf Seite 45:

$$\forall M \in [M_0\rangle : M_0 \in [M\rangle \tag{5.7}$$

Um eine strukturelle Verifikationsmethode für den Beweis der Reversibilität zu entwickeln, muss die Gleichung (5.3) so umgeformt werden, dass sie Gleichung (5.7) entspricht. Hierfür wird Gleichung (5.3) umgeschrieben:

$$m_0 + A \cdot u = m_0 + A \cdot (u + i_T) \tag{5.8}$$

Die linke Seite von Gleichung (5.8) beschreibt alle von der Anfangsmarkierung M_0 erreichbaren Folgemarkierungen. Die rechte Seite beschreibt alle (beliebigen) von dieser Markierung aus erreichbaren Folgemarkierungen. Hierzu wird die zusätzlich Schaltfolge in dem Schaltvektor $i_T = (i_1, \ldots, i_{|T|})^\top$ repräsentiert. Beide Seiten müssen nach Gleichung (5.7) gleich sein. Vereinfacht man Gleichung (5.8), so erhält man eine notwendige Bedingung für Reversibilität:

$$A \cdot i_T = 0 \tag{5.9}$$

Da das Schalten der Transitionen entsprechend dem Schaltvektor i_T den Zustand des Petri-Netzes nicht ändert, wird i_T als *Stelleninvariante* bezeichnet. Eine hinreichende Bedingung erhält man, wenn man für die Schaltfolge i_T einen gültigen Ablaufplan beginnend mit der Anfangsmarkierung M_0 findet.

Beispiel 5.1.10. Für das Petri-Netz aus Abb. 5.3 auf Seite 158 aus Beispiel 5.1.7 folgt aus Gleichung (5.9):

$$\begin{pmatrix} -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Die kleinste ganzzahlige, strikt positive Lösung lautet $i_\mathrm{T} = (1,1,1,1,2)^\top$.

Die Existenz dieser Lösung ist eine notwendige Bedingung für die Reversibilität des Petri-Netzes. Zu zeigen ist noch, dass ein gültiger Ablaufplan beginnend mit der Anfangsmarkierung für den Schaltvektor existiert. Diese existiert allerdings nicht, weshalb, wie bereits zuvor mit den aufzählenden Verfahren gezeigt, das Petri-Netz irreversibel ist. Für alle anderen erreichbaren Markierungen kann allerdings mit der Stelleninvariante i_T gezeigt werden, dass es sich bei diesen Markierungen um Grundzustände des Petri-Netzes handelt.

5.1.3 Partialordnungsreduktion

Eine wesentliche Eigenschaft von Systemen mit nebenläufigen Prozessen ist, dass gültige Ausführungen dieser Prozesse in unterschiedlichen Reihenfolgen stattfinden können. Während diese Eigenschaft im Entwurf zur Optimierung ausgenutzt werden kann, stellt dies ein großes Problem bei der Verifikation solcher Systeme dar, da hierdurch der zu untersuchende Zustandsraum sehr groß wird. Man spricht von der sog. *Zustandsraumexplosion*. Im Folgenden werden Möglichkeiten untersucht, die es erlauben, den Zustandsraum für Systeme mit nebenläufigen Prozessen zu reduzieren.

Oftmals hängt das Ergebnis der Verifikation von Systemen mit nebenläufigen Prozessen gar nicht von der tatsächlichen Ausführungsreihenfolge ab. In einem solchen Fall ist es denkbar, dass alle möglichen Permutationen in der Ausführungsreihenfolge durch eine einzelne, ausgewählte Ausführungsreihenfolge repräsentiert werden. Sei beispielsweise unter einer Markierung M die Ausführung der Folge $\langle t_1, t_2 \rangle$ möglich. Sei weiterhin unter M die Ausführung der Folge $\langle t_2, t_1 \rangle$ ebenfalls möglich und die beiden Schaltfolgen erreichen die selbe Folgemarkierung M', d. h. $M[\langle t_1, t_2 \rangle\rangle\, M' = M[\langle t_2, t_1 \rangle\rangle M'$, so können die beiden Sequenzen in einer Äquivalenzklasse zusammengefasst werden.

Man kann also beobachten, dass das Erreichen eines Zustands von der Schaltfolge der Transitionen unabhängig sein kann. Man sagt auch, das Schalten der Transitionen und somit die Transitionen selbst sind *unabhängig* voneinander. Das Schalten zweier Transitionen ist genau dann voneinander unabhängig, wenn sie nicht im Konflikt stehen (keine gemeinsame Stelle im Vorbereich besitzen) und keine kausale Abhängigkeit zwischen ihnen besteht (das Schalten einer Transition keine notwendige Bedingung für die Aktivierung der anderen Transition ist). Unabhängige Transitionen können in einem Petri-Netz existieren, da ein Petri-Netz lediglich eine partielle Ordnung für die Schaltfolge der Transitionen definiert. Entsprechend

werden Verfahren, welche die Unabhängigkeit von Transitionen zur Reduktion des Zustandsraumes ausnutzen, als *Partialordnungsreduktionen* bezeichnet.

Beispiel 5.1.11. Gegeben ist das Petri-Netz in Abb. 5.8a). Der zugehörige Erreichbarkeitsgraph ist in Abb. 5.8b) dargestellt. Man erkennt ein typisches Rautenmuster, welches durch die partielle Ordnung von Transitionen in einem Petri-Netz entstehen kann.

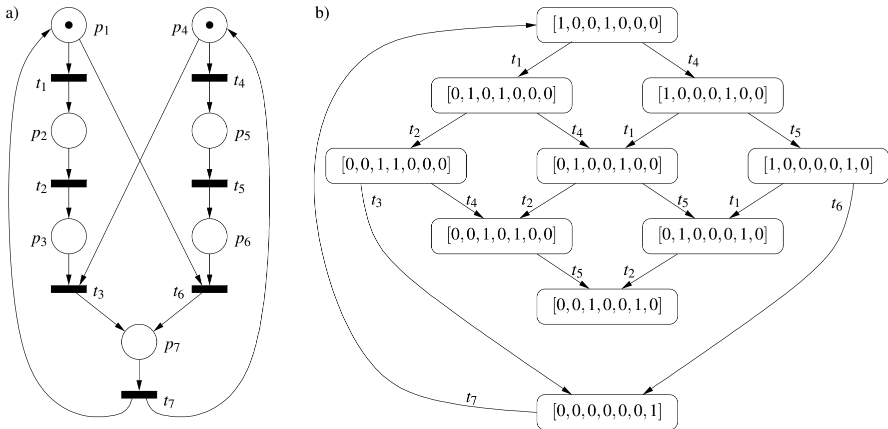


Abb. 5.8. a) Petri-Netz und b) zugehöriger Erreichbarkeitsgraph [189]

Man beachte, dass im Erreichbarkeitsgraphen in Abb. 5.8b) in der Markierung $[1,0,0,1,0,0,0]$ die Ausführung von t_1 und t_4 unabhängig voneinander ist, weshalb die beiden Schaltfolgen $\langle t_1, t_4 \rangle$ und $\langle t_4, t_1 \rangle$ in der selben Folgemarkierung resultieren. Weitere unabhängige Transitionen können im Erreichbarkeitsgraph identifiziert werden:

- Markierung $[0,1,0,1,0,0,0]$: Die Transitionen t_2 und t_4 sind unabhängig.
- Markierung $[1,0,0,0,1,0,0]$: Die Transitionen t_1 und t_5 sind unabhängig.
- Markierung $[0,1,0,0,1,0,0]$: Die Transitionen t_2 und t_5 sind unabhängig.

Somit sind ausgehende von der Markierung $[0,1,0,1,0,0,0]$ die Schaltfolgen $\langle t_2, t_4, t_5 \rangle$, $\langle t_4, t_2, t_5 \rangle$ und $\langle t_4, t_5, t_4 \rangle$ äquivalent und können beispielsweise durch $\langle t_2, t_4, t_5 \rangle$ repräsentiert werden.

In einem Erreichbarkeitsgraphen kann somit die Anzahl der möglichen Pfade beim Schalten der Transitionen reduziert werden, indem eine einzelne, repräsentative Schaltfolge gewählt wird. Das Ergebnis ist ein *reduzierter Erreichbarkeitsgraph*. Genauer gesagt wird eine Schaltfolge im Erreichbarkeitsgraphen beginnend bei einer Markierung M durch eine Referenzschaltfolge repräsentiert, wenn diese Folge einem Präfix einer beliebigen gültigen Permutation einer Schaltfolge im reduzierten Erreichbarkeitsgraphen entspricht. Im obigen Beispiel bedeutet dies, dass die

Schaltfolge $\langle t_4, t_2 \rangle$ ein Präfix der Schaltfolge $\langle t_4, t_2, t_5 \rangle$ darstellt, was eine gültige Permutation der Referenzschaltfolge $\langle t_2, t_4, t_5 \rangle$ ist, und deshalb durch diese repräsentiert werden kann.

Bei der Reduktion des Erreichbarkeitsgraphen ist allerdings die zu prüfende Eigenschaft zu berücksichtigen. Hierzu werden Transitionen in einem Petri-Netz zunächst in *sichtbare* und *nicht sichtbare* Transitionen klassifiziert. Sichtbare Transitionen werden entweder direkt in der zu prüfenden Eigenschaft verwendet oder es werden Stellen aus dem Vor- oder Nachbereich dieser Transition verwendet. Bei der Reduktion dürfen lediglich die nicht sichtbaren Transitionen verwendet werden. Dabei müssen drei grundlegende Anforderungen berücksichtigt werden:

1. Eine hinreichende Menge an Referenzschaltfolgen muss erhalten bleiben, um die Beobachtbarkeit aller Schaltreihenfolgen sichtbarer Transitionen sicherzustellen.
2. Als eine Konsequenz aus 1. dürfen lediglich nicht sichtbare Transitionen bei der Reduktion Verwendung finden. Es kann allerdings im Fall von Lebendigkeitseigenschaften notwendig sein, nicht sichtbare Transitionen zu erhalten. Aber nicht zwangsläufig alle.
3. Aus den ersten beiden Punkten folgt direkt die Frage, ob sich bei einer gegebenen zu prüfenden Eigenschaft eine Reduktion überhaupt lohnt. Dies ist offensichtlich nicht der Fall, wenn alle Transitionen sichtbar sind.

Die Konstruktion des reduzierten Erreichbarkeitsgraphen wird in der Regel nicht durch Reduktion des bereits konstruierten Erreichbarkeitsgraphen erfolgen, sondern die Reduktion erfolgt bereits bei der Konstruktion. Dies ist möglich, da die Unabhängigkeit von Transitionen aus der Struktur des Petri-Netzes erkennbar ist. Dabei werden im Wesentlichen Konflikte zwischen Teilmengen von schaltbereiten Transitionen analysiert. Existiert eine Teilmenge an schaltbereiten Transitionen, die nicht im Konflikt mit schaltbereiten Transitionen außerhalb dieser Menge stehen, können die Transitionen dieser Teilmenge schalten. Dabei wird keiner der nicht berücksichtigten Transitionen die Schaltbereitschaft entzogen. Diese bleiben somit in späteren Markierung weiterhin schaltbereit. Allerdings kann es bei diesem Vorgehen passieren, dass relevante Schaltfolgen übersehen werden. Hierfür gibt es zwei Gründe:

1. Das *Ignoranzproblem* entsteht, wenn eine schaltbereite Transition niemals schaltet. Dies kann insbesondere auftreten, wenn Transitionen unter sämtlichen Markierungen unabhängig bleiben. Ein einfaches Beispiel hierfür sind zwei nicht zusammenhängende Petri-Netze.
2. Das *Irritationsproblem* entsteht, wenn eine Markierung niemals betreten wird, obwohl diese eine Anfangsmarkierung für eine Referenzschaltfolge ist. Ein Beispiel hierfür ist in Abb. 5.8a) zu sehen. Die Transitionen t_1 und t_4 sind in der Anfangsmarkierung M_0 unabhängig voneinander. Wird eine der beiden Transitionen niemals ausgeführt (z. B. t_4), gehen hierdurch alle Schaltfolgen verloren, welche die Transition t_6 enthalten. Dies liegt daran, dass der Konflikt, der zwischen t_1 und t_6 besteht, unter der Anfangsmarkierung nicht gesehen wird. Allerdings wird dieser nach Ausführung der Schaltfolge $\langle t_4, t_5 \rangle$ sichtbar.

Im Folgenden werden zwei Verfahren zur Partialordnungsreduktion vorgestellt. Die *Stubborn-Set-Methode* und die *Sleep-Set-Methode* nutzen dabei die oben diskutierten Eigenschaften bei der Reduktion aus.

Die Stubborn-Set-Methode

Bei der Stubborn-Set-Methode wird in jedem Knoten des vollständigen Erreichbarkeitsgraphen eine Teilmenge schaltbereiter Transitionen ausgewählt, die unabhängig von anderen schaltbereiten Transitionen sind. Diese Menge wird als engl. *stubborn set* bezeichnet. Neben schaltbereiten Transitionen kann diese Menge auch zusätzlich durch nicht schaltbereite Transitionen erweitert werden. Hierdurch wird das Irritationsproblem behandelt, falls diese Transitionen in Folgemarkierungen einen Konflikt auslösen können. Ausgehende Kanten aus dem Knoten im Erreichbarkeitsgraphen, die mit Transitionen markiert sind, die nicht in dem *stubborn set* enthalten sind, werden gelöscht. Da nicht zwangsläufig alle schaltbereiten Transitionen enthalten sind, kann dies zu einer Reduktion des Erreichbarkeitsgraphen führen. Durch die Auswahl der Transitionen ist dabei sichergestellt, dass jede schaltbereite Transition, die nicht in dem *stubborn set* aufgenommen wird, unter jeder Folgemarkierung ebenfalls schaltbereit ist.

Jede ausgewählte Transition muss hierfür eine der beiden folgenden Bedingungen erfüllen:

1. Falls die ausgewählte Transition $t \in T$ schaltbereit ist, wähle ebenfalls alle Transitionen $t' \in T$ aus, die potentiell mit t in Konflikt stehen können, d. h. $\bullet t \cap \bullet t' \neq \emptyset$.
2. Falls die ausgewählte Transition $t \in T$ nicht schaltbereit ist, wähle eine Stelle $p \in \bullet t$ mit $M(p) < W(p,t)$, also eine Stelle, die aufgrund fehlender Marken die Schaltbereitschaft von t verhindert. Zu dieser Stelle werden alle Transitionen aus deren Vorbereich ebenfalls in das *stubborn set* aufgenommen, d. h. $\{t' \in T \mid t' \in \bullet p\}$.

Diese Bedingungen haben zur Folge, dass die Bestimmung des *stubborn set* eine Fixpunktberechnung ist. Es können in jedem Schritt, wenn neue Transitionen hinzugenommen werden, weitere Bedingungen verletzt sein, die wiederum die Aufnahme weiterer Transitionen in das *stubborn set* verlangen. Im schlimmsten Fall sind anschließend alle Transitionen des Petri-Netzes in dem *stubborn set* enthalten. Das andere Extrem ist, dass lediglich eine unabhängige Transition in der *stubborn set* enthalten ist.

Beispiel 5.1.12. Es wird das Petri-Netz aus Abb. 5.8a) betrachtet. Unter der Anfangsmarkierung M_0 sind die Transitionen t_1 und t_4 schaltbereit. Das *stubborn set* ergibt sich hierbei zu $\{t_1, t_6, t_5, t_4, t_3, t_2\}$. Dabei wurde zunächst die schaltbereite Transition t_1 ausgewählt. Diese besitzt allerdings einen strukturellen Konflikt mit Transition t_6, weshalb t_6 ebenfalls mit in das *stubborn set* aufgenommen werden muss. Da t_6 nicht aktiviert ist, muss die Eingangsstelle p_6 betrachtet werden und alle Transitionen aus dem Vorbereich von p_6 ebenfalls in das *stubborn set* aufgenommen werden. Die einzige Transition aus dem Vorbereich $\bullet p_6$ ist Transition t_5. Da auch t_5 nicht schaltbereit

ist, wird auch t_4 mit aufgenommen. Welche wiederum einen strukturellen Konflikt mit t_3 besitzt. Schließlich wird auch noch t_2 mit in das *stubborn set* aufgenommen, da t_3 nicht schaltbereit ist. An dieser Stelle ist ein Fixpunkt erreicht. Da die Transitionen t_1 und t_4 in dem *stubborn set* enthalten sind, kann in diesem ersten Schritt keine Vereinfachung des Erreichbarkeitsgraphen (siehe Abb. 5.8b)) erzielt werden.

In der Folgemarkierung $[0,1,0,1,0,0,0]$ kann die schaltbereite Transition t_2 ausgewählt werden. Da diese bereits alle Anforderungen erfüllt, schaltet lediglich diese. Man beachte, dass aber ebenfalls t_4 weiterhin schaltbereit war. Hätte man zuerst t_4 in das *stubborn set* aufgenommen, hätte t_4 aufgrund der geforderten Bedingungen auf $\{t_4, t_3, t_2\}$ erweitert werden müssen, was in diesem Schritt wiederum zu keiner Zustandsraumreduktion geführt hätte. Der reduzierte Erreichbarkeitsgraph nach Anwendung der Stubborn-Set-Methode ist in Abb. 5.9 zu sehen.

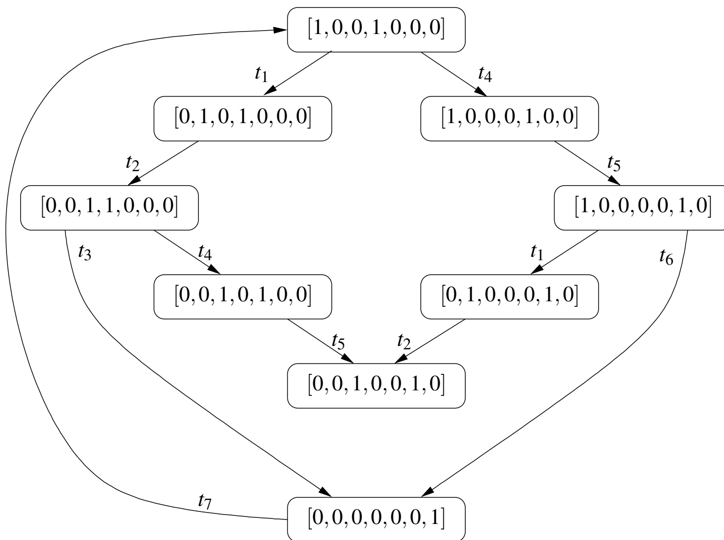


Abb. 5.9. Reduzierter Erreichbarkeitsgraph des Petri-Netzes aus Abb. 5.8a) nach der Stubborn-Set-Methode [189]

Bereits an diesem kleinen Beispiel erkennt man, dass die Effizienz der Stubborn-Set-Methode von vielen Parametern abhängt:

- Der Wahl einer schaltbereiten Transition.
- Der Wahl der Eingangsstelle einer nicht schaltbereiten Transition.
- Der Prozedur, um dem Ignoranzproblem zu begegnen.

Je nachdem wie die Stubborn-Set-Methode umgesetzt ist, können unterschiedliche *stubborn sets* ermittelt werden. Die Frage, welche dieser Mengen verwendet werden sollte, kann durch die Einschlussbedingung bestimmt werden. In obi-

gen Beispiel wäre unter der Markierung $[0,1,0,1,0,0,0]$ das *stubborn set* $\{t_2\}$ oder $\{t_4,t_3,t_2\}$ möglich. Da $\{t_2\}$ in $\{t_4,t_3,t_2\}$ enthalten ist, kann $\{t_2\}$ ausgewählt werden, da t_2 schaltbereit ist.

Die Sleep-Set-Methode

In der Sleep-Set-Methode wird mit jedem Knoten im vollständigen Erreichbarkeitsgraphen ein sog. engl. *sleep set* assoziiert. Im Gegensatz zur Stubborn-Set-Methode repräsentiert das *sleep set*, welche schaltbereiten Transitionen nicht schalten sollten. Die Entscheidung, dass eine schaltbereite Transition nicht schalten soll, basiert auf dem Wissen, dass die Folgemarkierung, die durch das Schalten dieser Transition erreicht wird, auch auf anderem Weg erreicht werden kann. Somit eliminiert die Sleep-Set-Methode Kanten aus dem Erreichbarkeitsgraphen.

Sleep sets werden während der Konstruktion des Erreichbarkeitsgraphen aufgebaut. Zu Beginn sind alle *sleep sets* leer. Gegeben sei eine Markierung M und das assoziierte *sleep set*. Für alle unter M schaltbereiten Transitionen t mit Ausnahme der Transitionen in dem *sleep set* werden die Folgemarkierungen M' bestimmt. Ist durch Schalten von t ein noch nicht besuchter Knoten mit Markierung M' des Erreichbarkeitsgraphen erreicht worden, so wird das zu M' assoziierte *sleep set* berechnet, indem das *sleep set* von M kopiert wird. Zu dem mit M' assoziierten *sleep set* werden nun alle Transitionen, die in M vor t geschaltet haben, hinzugefügt und alle Transitionen entfernt, die in einem Konflikt mit t unter der Markierung M' stehen.

Beispiel 5.1.13. Es wird wiederum das Petri-Netz aus Abb. 5.8a) betrachtet. Für die Anfangsmarkierung M_0 ist das *sleep set* leer. Es wird das Schalten der Transitionen t_1 und t_4 in dieser Reihenfolge betrachtet. Für die Folgemarkierung M_1 mit $M_0[t_1\rangle M_1$ ist das *sleep set* weiterhin leer. Für die Folgemarkierung M_2 mit $M_0[t_4\rangle M_2$, welche sich durch das Schalten von t_4 ergibt, besteht das *sleep set* aus dem Element t_1, da t_1 vor t_4 geschaltet hat. Aus diesem Grund schaltet in M_2 lediglich t_5, während in M_1 die beiden Transitionen t_2 und t_4 schalten usw. Der reduzierte Erreichbarkeitsgraph, der sich durch Anwendung der Sleep-Set-Methode ergibt, ist in Abb. 5.10 dargestellt.

Die Sleep-Set-Methode reduziert die Kanten in einem Erreichbarkeitsgraphen und somit die Anzahl der Vergleiche von neuen und bereits besuchten Markierungen. Eine Zustandsraumreduktion in dem Sinne, dass Knoten im Erreichbarkeitsgraphen gelöscht werden, wie dies in der Stubborn-Set-Methode der Fall ist, wird dabei nicht vorgenommen, da die Sleep-Set-Methode das Ziel hat, alle erreichbaren Markierungen zu erhalten. Dies sieht man auch, wenn man Abb. 5.9 mit Abb. 5.10 vergleicht.

Da alle Knoten erhalten bleiben, kann man aber weiterhin alle Erreichbarkeitseigenschaften, die auf dem nichtreduzierten Erreichbarkeitsgraphen geprüft werden können, auch auf dem reduzierten Erreichbarkeitsgraphen prüfen. Allerdings lassen sich Lebendigkeitseigenschaften nicht mehr allgemein verifizieren. In Abb. 5.10 würde die Markierung $[0,1,0,0,0,1,0]$ fälschlicherweise als Verklemmung identifiziert.

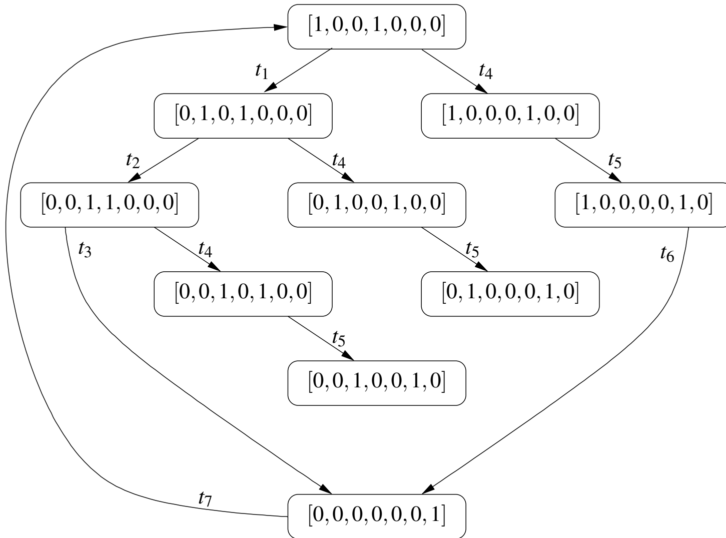


Abb. 5.10. Reduzierter Erreichbarkeitsgraph des Petri-Netzes aus Abb. 5.8a) nach der Sleep-Set-Methode [189]

Die Sleep-Set-Methode lässt sich auch mit der Stubborn-Set-Methode kombinieren. Dabei werden in jeder Markierung die Transitionen des *sleep set* aus dem *stubborn set* entfernt.

Beispiel 5.1.14. Das Ergebnis der Kombination von Sleep-Set- und Stubborn-Set-Methode ist in Abb. 5.11 zu sehen. Es wird wiederum das Petri-Netz aus Abb. 5.8a) betrachtet. Ausgangspunkt ist diesmal allerdings der reduzierte Erreichbarkeitsgraph, der sich durch Anwendung der Stubborn-Set-Methode ergeben hat (siehe Abb. 5.9). Entscheidend ist, dass in der Markierung $[1,0,0,0,0,1,0]$ das *stubborn set* $\{t_1,t_6\}$ sowie das *sleep set* $\{t_1\}$ gilt, weshalb lediglich t_6 weiter betrachtet wird. Der in Abb. 5.11 gezeigte reduzierte Erreichbarkeitsgraph erlaubt keine weitere Reduktion.

5.2 Explizite Modellprüfung

Für die im vorangegangenen Kapitel beschriebenen Verifikationsmethoden zur Eigenschaftsprüfung von Petri-Netzen stellt der Erreichbarkeitsgraph ein zentrales Element dar. Ist die Erreichbarkeitsmenge und somit der Zustandsraum endlich, können Petri-Netze als temporale Strukturen (siehe Definition 2.4.1 auf Seite 73) repräsentiert werden. Für Modelle mit endlichem Zustandsraum wurden in den vergangenen Jahren effiziente Verfahren zur Überprüfung von Erreichbarkeits-, Gefahrlosigkeits- und Lebendigkeitseigenschaften entwickelt. Da all diese Verfahren ein Modell des zu

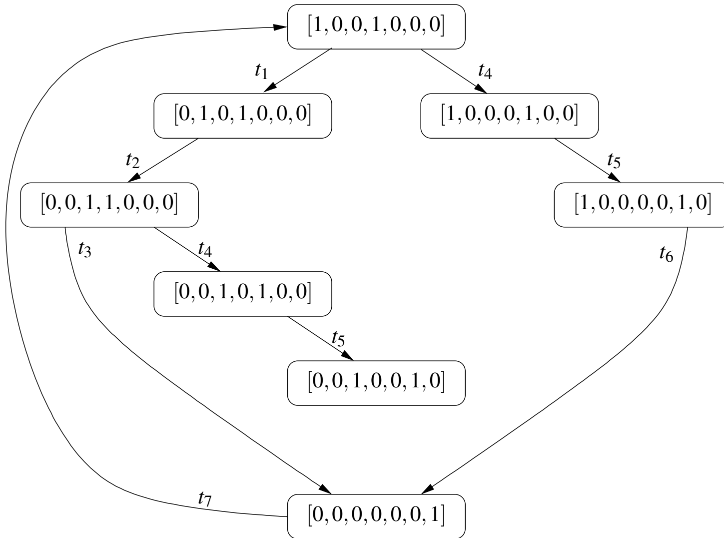


Abb. 5.11. Reduzierter Erreichbarkeitsgraph nach Kombination der Sleep-Set- und Stubborn-Set-Methode [189]

überprüfenden Systems verlangen, werden diese Verfahren auch als *Modellprüfung* bezeichnet.

Basierend auf der formalen Spezifikation funktionaler Eigenschaften, wie sie im Abschnitt 2.4 diskutiert wurde, werden im Folgenden formale Modellprüfungsmethoden für CTL- und LTL-Formeln präsentiert. Beide Verfahren basieren darauf, dass das zu verifizierende System als temporale Struktur modelliert wird. Anschließend wird eine simulative Verifikationsmethode zur Modellprüfung vorgestellt.

5.2.1 CTL-Modellprüfung

CTL-Formeln werden aus aussagenlogischen Formeln gebildet, indem temporale Operatoren, Pfadquantoren und logische Operatoren verwendet werden (siehe Definition 2.4.6 auf Seite 78). Da jedem temporalen Operator direkt ein Pfadquantor voran steht, gibt es in CTL acht Operatoren mit Verzweigungslogik, wobei die drei Operatoren EX, EG und EU mit den aussagenlogischen Operatoren $\vee$ und $\neg$ eine vollständige Basis bilden, in der sich alle acht Operatoren darstellen lassen. In CTL-lassen sich Erreichbarkeits- (EF), *Gefahrlosigkeits-* (AG) und *Lebendigkeitseigenschaften* (AG AF) formulieren. Im Folgenden wird beschrieben, wie CTL-Formeln auf einer gegebenen temporalen Struktur M geprüft werden können.

Gegeben sei eine temporale Struktur $M = (S,R,L)$, bestehend aus Zuständen S, einer Übergangsrelation R, einem Anfangszustand $s_0 \in S$ und einer Markierungsfunktion $L : S \rightarrow 2^V$, die jedem Zustand eine Teilmenge an aussagenlogischen Variablen (atomaren Formeln) aus V zuordnet. Weiterhin gegeben sei eine CTL-Formel φ.

Die Aufgabe der Modellprüfung besteht nun darin, alle Zustände in der temporalen Struktur M zu identifizieren, in denen φ gilt:

$$\mathrm{SAT}(M,\varphi) := \{s \in S \mid M,s \models \varphi\} \tag{5.10}$$

Oftmals ist die Verifikationsaufgabe in der Form $M,s_0 \models \varphi$ gegeben. In diesem Fall muss lediglich geprüft werden, ob $s_0 \in \mathrm{SAT}(M,\varphi)$ ist.

Ein Algorithmus zur CTL-Modellprüfung kann direkt aus der Semantik von CTL (siehe Definition 2.4.7 auf Seite 78) abgeleitet werden und basiert auf der Berechnung von kleinsten und größten Fixpunkten. CTL-Fixpunktberechnung ist in Anhang C.4 beschrieben. Gleichung (5.11) zeigt die Fixpunktdefinitionen für sechs Operatoren mit Verzweigungslogik, wobei μy den kleinsten und νy den größten Fixpunkt eines Funktionals y charakterisiert. Zur Definition der Operatoren EX φ und AX φ wird keine Fixpunktberechnung benötigt.

$$\begin{aligned}
\mathrm{AF}\ \varphi &:= \mu y : \varphi \vee \neg\mathrm{EX}\ \neg y \\
\mathrm{EF}\ \varphi &:= \mu y : \varphi \vee \mathrm{EX}\ y \\
\mathrm{AG}\ \varphi &:= \nu y : \varphi \wedge \neg\mathrm{EX}\ \neg y \\
\mathrm{EG}\ \varphi &:= \mu y : \varphi \wedge \mathrm{EX}\ y \\
\mathrm{A}\ \varphi\ \mathrm{U}\ \psi &:= \mu y : \psi \vee (\varphi \wedge \neg\mathrm{EX}\ \neg y) \\
\mathrm{E}\ \varphi\ \mathrm{U}\ \psi &:= \mu y : \psi \vee (\varphi \wedge \mathrm{EX}\ y)
\end{aligned} \tag{5.11}$$

Die grundlegende Idee in den Gleichungen ist es, jeden Operator durch zwei logisch verknüpfte Teile zu beschreiben:

1. Der erste Teil enthält lediglich Aussagen über die Markierung des momentan betrachteten Zustands.
2. Der zweite Teil enthält Aussagen über direkte Nachfolgezustände, wobei diese für mindestens einen (EX φ) oder alle (AX $\varphi = \neg$EX $\neg\varphi$) Nachfolgezustände gilt.

Die Berechnung der Zustände $\mathrm{SAT}(M,\varphi)$, in denen φ gilt, erfolgt analog zur Definition 2.4.7 der Relation $\models$, also auf der Struktur der Formel. Dies bedeutet, dass zunächst diejenigen Zustände bestimmt werden, in denen die verwendeten aussagenlogischen Variablen wahr sind. Anschließend werden, basierend auf der Semantik von CTL für komplexere Teilformeln, die assoziierten Zustände bestimmt. Dies wird sukzessive wiederholt, bis alle Operatoren der CTL-Formel abgearbeitet sind.

Beispiel 5.2.1. Gegeben ist eine temporale Struktur M und die CTL-Formel AG $(\varphi \Rightarrow$ AF $\psi)$. Diese Formel lässt sich als Syntaxbaum darstellen (siehe Abb. 5.12). Der Quellknoten repräsentiert die gesamte CTL-Formel. Blätter des Syntaxbaums sind aussagenlogische Variablen.

Zur Berechnung von $\mathrm{SAT}(M,\varphi)$ aus Gleichung (5.10) ist die Umsetzung der folgenden Teilberechnungen hinreichend:

- Berechnung der Markierungsfunktion $L(s)$ auf den Zuständen $s \in S$ der temporalen Struktur M

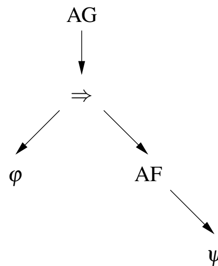


Abb. 5.12. Syntaxbaum der CTL-Formel AG $(\varphi \Rightarrow \text{AF } \psi)$ [272]

- Berechnung von logischen Operatoren auf Mengen
- Auswertung des Operators EX
- Berechnung der Fixpunkte für EG und EU

Im Folgenden wird die Umsetzung dieser Teilberechnungen diskutiert.

Die Verifikationsmethode

Zentraler Bestandteil der Fixpunktberechnungen und damit der Verifikationsmethode ist die Berechnung des Operators EX p. Ist für eine temporale Struktur $M = (S,R,L)$ bereits die Menge der Zustände $S_p \subseteq S$ bekannt, in denen p gilt, so müssen für diese Zustände die direkten Vorgänger bestimmt werden. Eine Funktion, die dies erfüllt, ist COMPUTE_EX:

```
COMPUTE_EX(M, S_p) {
        RETURN {s ∈ S | ∃s' ∈ S_p : (s,s') ∈ R};
}
```

Weiterhin wird eine Funktion benötigt, die den Operator EG p implementiert. Wiederum sei S_p die Menge an Zuständen, in denen p gilt. Dann muss die Implementierung von EG p diejenigen Zustände bestimmen, die einen unendlichen Pfad besitzen, der lediglich über Zustände in S_p verläuft. Die Funktion COMPUTE_EG implementiert dieses:

```
COMPUTE_EG(M, S_p) {
        S_N := S;
        DO
                S_R := S_N;
                S_N := S_p ∩ COMPUTE_EX(M, S_N);
        UNTIL (S_N = S_R)
        RETURN S_N;
}
```

Die Funktion erhält als Argumente die temporale Struktur M und die Menge S_p der Zustände, in denen p gilt. Beginnend mit der Menge aller Zustände $S_N := S$

wird die Fixpunktberechnung durchgeführt. Hierzu werden sukzessive Zustände aus S_N gelöscht. Dafür werden in jedem Schritt die Vorgängerzustände zu denjenigen Zuständen bestimmt, die noch in S_N enthalten sind und in denen p gilt. Dies wird solange wiederholt, bis keine Zustände mehr gelöscht werden, also ein Fixpunkt erreicht ist.

Des weiteren wird eine Funktion benötigt, die den Operator E p U q implementiert. Sei S_p die Menge aller Zustände, in denen p gilt, und S_q die Menge aller Zustände, in denen q gilt. Die Funktion muss diejenigen Zustände bestimmen, die mit q markiert sind oder Anfang eines Pfades mit Zuständen nur aus S_p sind, bis er mit mindestens einem Zustand aus S_q endet. Dies wird in der Funktion COMPUTE_EU berechnet:

```
COMPUTE_EU(M, S_p, S_q) {
    S_N := ∅;
    DO
        S_R := S_N;
        S_N := S_q ∪ (S_p ∩ COMPUTE_EX(M, S_N));
    UNTIL (S_N = S_R)
    RETURN S_N;
}
```

Die Funktion erhält als Argumente die temporale Struktur M und die Mengen S_p und S_q. Beginnend mit der leeren Menge $S_N := \emptyset$ wird die Fixpunktberechnung durchgeführt. Hierzu werden sukzessive Zustände zu S_N hinzugefügt. Dafür werden in jedem Schritt diejenigen Zustände bestimmt, in denen q gilt oder in denen p gilt und zusätzlich in einem darauf folgenden Schritt q. Dies wird solange wiederholt, bis keine Zustände mehr hinzugefügt werden.

Schließlich wird eine Methode benötigt, welche die Auswertung der Markierungsfunktion und die Berechnung von logischen Operatoren auf Mengen implementiert. Dies geschieht in der Methode COMPUTE. Als Argumente erhält die Methode die temporale Struktur M sowie die CTL-Formel φ.

```
COMPUTE(M, φ) {
    SWITCH
        CASE φ = F:
            S_φ := ∅;
        CASE φ = T:
            S_φ := S;
        CASE φ ∈ V:
            S_φ := {s ∈| L(s) = φ};
        CASE φ = ¬p:
            S_φ := S\COMPUTE(M, p);
        CASE φ = p ∨ q:
            S_φ := COMPUTE(M, p) ∪ COMPUTE(M, q);
        CASE φ = EX p:
            S_φ := COMPUTE_EX(M, COMPUTE(M, p));
```

```
            CASE φ = EG p:
                S_φ := COMPUTE_EG(M, COMPUTE(M, p));
            CASE φ = E p U q:
                S_φ := COMPUTE_EU(M, COMPUTE(M, p),
                                  COMPUTE(M, q));
    RETURN S_φ;
}
```

Die Berechnung erfolgt entsprechend Definition 2.4.7 auf Seite 78 durch Fallunterscheidung. Handelt es sich bei φ um eine atomare Formel ($\varphi \in V$), wird die Menge an Zuständen bestimmt, die mit φ markiert sind. Handelt es sich um eine negierte Formel ($\neg p$) wird die Differenzmenge zum gesamten Zustandsraum bestimmt. Eine ODER-Verknüpfung ($p \vee q$) wird als Vereinigung der beiden Zustandsmengen, in denen p bzw. q gilt, implementiert. Und die Operatoren EX, EG und EU werden durch rekursive Berechnung der Argumente und Aufruf der entsprechenden Funktionen realisiert.

Beispiel 5.2.2. Gegeben ist die temporale Struktur M in Abb. 5.13. Es sollen mit der oben beschriebenen Verifikationsmethode diejenigen Zustände bestimmt werden, welche die CTL-Formel φ erfüllen

$$\varphi = \mathrm{AF\,AG}\, p\,.$$

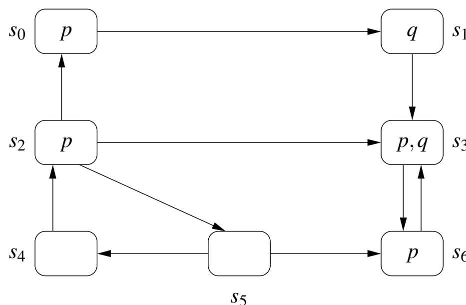


Abb. 5.13. Temporale Struktur

In einem ersten Schritt muss die CTL-Formel φ so umgeformt werden, dass sie von der Funktion COMPUTE bearbeitet werden kann, also nach Äquivalenzumformungen nur die berücksichtigte Basis an Operatoren verwendet wird:

$$\begin{aligned}\varphi &= \mathrm{AF\,AG}\, p \\ &= \mathrm{AF}\,\neg\mathrm{EF}\,\neg p \\ &= \neg\mathrm{EG\,EF}\,\neg p \\ &= \neg\mathrm{EG\,E}\,\top\,\mathrm{U}\,\neg p\end{aligned}$$

Nun kann mittels der Funktion COMPUTE die Menge der Zustände S_φ bestimmt werden, in denen φ gilt. Die Berechnung erfolgt in mehreren Schritten:

1. $S_p := \{s_0, s_2, s_3, s_6\}$
2. $S_{\neg p} := S \backslash S_p = \{s_1, s_4, s_5\}$
3. $S_{\mathrm{E}\top\mathrm{U}\neg p}$:=COMPUTE_EU(M, S, $S_{\neg p}$). Die entsprechende Fixpunktberechnung ist in Abb. 5.14 zu sehen. Zunächst werden dabei alle Zustände in $S_{\neg p}$ markiert (Abb. 5.14a)). Im zweiten Schritt (Abb. 5.14b)) werden diejenigen Zustände hinzugenommen, die einen Zustand aus $S_{\neg p}$ in einem Schritt erreichen können. Im dritten Schritten werden die Zustände hinzugenommen, die einen Zustand in $S_{\neg p}$ in zwei Zustandsübergängen erreichen können. In diesem dritten Schritt kommen keine weiteren Zustände hinzu, weshalb der kleinste Fixpunkt erreicht wurde. Die Menge $S_{\mathrm{E}\top\mathrm{U}\neg p}$ ergibt sich zu $\{s_0, s_1, s_2, s_4, s_5\}$.

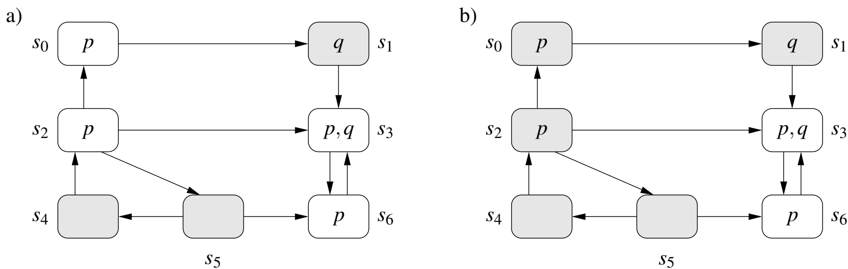


Abb. 5.14. Bestimmung von $\mathrm{E}\top\mathrm{U}\neg p$. In zwei Iterationsschritten ist der Fixpunkt erreicht

4. $S_{\mathrm{EG\,E}\top\mathrm{U}\neg p}$:= COMPUTE_EG(M, $S_{\mathrm{E}\top\mathrm{U}\neg p}$). Die Fixpunktberechnung ist in Abb. 5.15 zu sehen. Da es sich um die Berechnung des größten Fixpunktes handelt, sind zunächst alle Zustände Teil der Lösung (Abb. 5.15a)). Diese Menge wird reduziert, indem alle Zustände, die nicht Element in $S_{\mathrm{E}\top\mathrm{U}\neg p}$ sind, entfernt werden (Abb. 5.15b)). Von der resultierenden Menge an Zuständen werden diejenigen entfernt, die keinen Zustand $s \in S_{\mathrm{E}\top\mathrm{U}\neg p}$ in genau einem Zustandsübergang erreichen können (Abb. 5.15c)). Von dieser Menge an Zuständen werden wiederum diejenigen Zustände entfernt, die keinen Zustand $s \in S_{\mathrm{E}\top\mathrm{U}\neg p}$ in genau zwei Zustandsübergängen erreichen können (Abb. 5.15d)). Anschließend werden noch diejenigen Zustände entfernt, die keinen Zustand in $S_{\mathrm{E}\top\mathrm{U}\neg p}$ in genau drei Zustandsübergängen erreichen können. An dieser Stelle bleibt die Zustandsmenge konstant und der größte Fixpunkt ist gefunden. Die Menge $S_{\mathrm{EG\,E}\top\mathrm{U}\neg p}$ ergibt sich zu $\{s_2, s_4, s_5\}$.
5. In einem letzten Schritt muss noch die Differenzmenge zur Zustandsmenge der temporalen Struktur gebildet werden. $S_\varphi = S_{\neg\mathrm{EG\,E}\top\mathrm{U}\neg p} := S \backslash S_{\mathrm{EG\,E}\top\mathrm{U}\neg p} = \{s_0, s_1, s_3, s_6\}$.

Die Zustände in S_φ besitzen auf allen Berechnungspfaden, die in einem dieser Zustand beginnt, somit die Eigenschaft, dass irgendwann in der Zukunft alle weiteren

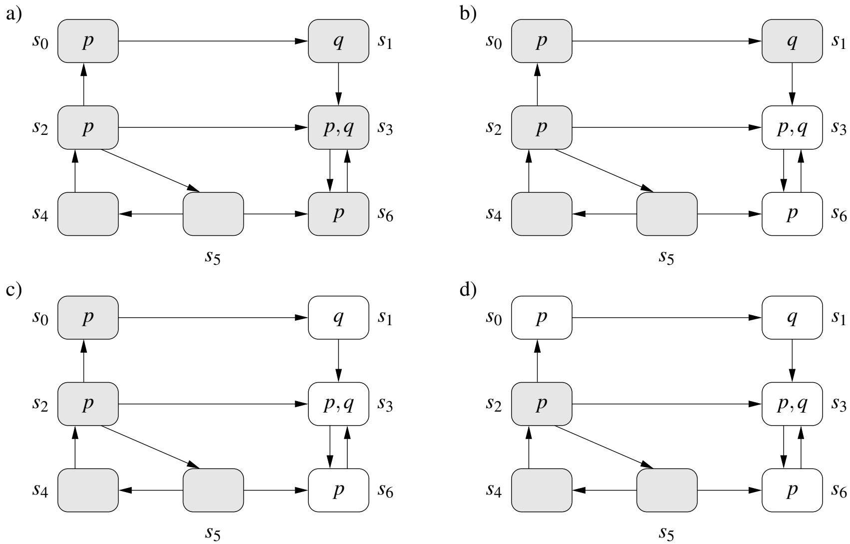


Abb. 5.15. Bestimmung von EG E ⊤ U $\neg p$

Berechnungen auf allen erdenklichen Pfaden die Markierung p besitzen. Dies resultiert aus der Schleife zwischen s_3 und s_6. Für die Zustände s_2, s_4 und s_5 existiert hingegen ein Pfad, auf dem nicht ständig p gilt. Dieser Pfad verläuft exakt über diese drei Zustände.

Die bis hier beschriebene Modellprüfung basiert auf der expliziten Aufzählung von Zuständen. Deshalb wird dieses Verfahren auch als *explizite Modellprüfung* bezeichnet. Eine *implizite Modellprüfung* basiert auf der symbolischen Repräsentation von Zustandsräumen mittels charakteristischer Funktionen. Diese Verfahren werden in Abschnitt 5.3 beschrieben.

5.2.2 LTL-Modellprüfung

Während CTL-Formeln sog. *Zustandsformeln* sind, handelt es sich bei LTL-Formeln um *Pfadformeln*, die funktionale Eigenschaften von unendlich langen Pfaden beschreiben (siehe Abschnitt 2.4.2). Modellprüfung für LTL-Formeln kann somit nicht mehr allein durch das bloße Markieren von Zuständen durchgeführt werden. Dies liegt darin begründet, dass ein endlicher Automat im Allgemeinen unendlich viele verschiedene Ausführungssequenzen mit unendlicher Länge besitzt.

Beispiel 5.2.3. Betrachtet man die LTL-Formel $\varphi :=$ GF p, so muss es in einer temporalen Struktur, die Modell für φ ist, eine unendliche Sequenz $\langle s_0, s_1, \dots \rangle$ mit unendlich vielen Positionen $s_{n_1}, s_{n_2}, \dots$ geben, in denen p gilt. Zwischen diesen Positionen kann es beliebig, aber endlich viele Zustände geben, in denen $\neg p$ gilt. Versucht

man dies in einer Formel auszudrücken, so erhält man $((\neg p)^* p)^\omega$, wobei $*$ die Bedeutung von „beliebig, aber endlich“ und ω von „unendlich“ vielen Wiederholungen besitzt. Ein Gegenbeispiel für φ hätte entsprechend die Form $(p \vee \neg p)^*(\neg p)^\omega$.

Die in diesem Beispiel eingeführte Schreibweise wird zur Formulierung von sog. *ω-regulären Ausdrücken* verwendet. Es handelt sich dabei um eine Erweiterung der bekannten *regulären Ausdrücke*. Während sich reguläre Ausdrücke noch mit endlichen Automaten modellieren lassen, ist dies für ω-reguläre Ausdrücke nicht mehr möglich.

LTL-Modellprüfung basiert auf der Erkenntnis, dass zu jeder LTL-Formel φ ein ω-regulärer Ausdruck $\mathcal{E}_\varphi$ gebildet werden kann, der alle gültigen Ausführungssequenzen für φ darstellt. Somit kann die Modellprüfung $M \models \varphi$ für eine gegebene temporale Struktur M in folgende Frage umformuliert werden: „Sind alle Ausführungssequenzen von M wie in $\mathcal{E}_\varphi$ beschrieben?“

Die Repräsentation eines ω-regulären Ausdrucks kann mittels eines sog. *Büchi-Automaten* erfolgen.

Definition 5.2.1 (Büchi-Automat). *Ein Büchi-Automat ist ein 4-Tupel $B = (S_B, R_B, L_B, A_B)$, wobei*

- *S_B die Menge der Zustände,*
- *$R_B \subseteq S_B \times S_B$ die Zustandsübergangsrelation,*
- *$L_B : S_B \to 2^{2^V}$ eine Markierungsfunktion mit V der Menge der atomaren Formeln und*
- *$A_B \subseteq S_B$ die Menge* akzeptierender Zustände *ist.*

Die Markierungsfunktion L_B ordnet jedem Zustand eine Menge aussagenlogischer Variablen $v \in V$ (atomaren Formeln) zu. Die Zustandsübergangsrelation muss nicht total sein. Weiterhin ist eine Menge an Anfangszuständen $S_{B,0} \subseteq S_B$ implizit gegeben.

Sei $\tilde{s} := \langle s_0, s_1, \ldots \rangle$ ein unendlicher Pfad aus Zuständen $s_i \in S_B$, mit $\forall i \geq 0 : (s_i, s_{i+1}) \in R_B$, durch den Büchi-Automaten B gegeben, dann sei $\inf(\tilde{s} \subseteq S_B)$ die Menge an Zuständen $s \in S_B$, die unendlich oft in $\tilde{s}$ auftreten. Die *Sprache* $\mathcal{L}(B)$, die von einem Büchi-Automaten akzeptiert wird, ist die Menge aller Pfade, in denen ein akzeptierender Zustand unendlich oft auftritt, d. h.

$$\mathcal{L}(B) = \{\tilde{s} = \langle s_0, s_1, \ldots \rangle \mid \inf(\tilde{s}) \cap A_B \neq \emptyset\}$$

Definition 5.2.2 (Temporale Struktur als Büchi-Automat). *Sei $M = (S, R, L)$ eine temporale Struktur nach Definition 2.4.1 auf Seite 73. Die Markierungsfunktion $L : S \to 2^V$ gibt an, für welchen Zustand welche atomaren Formeln $v \in V$ gelten. Dies ist gleichbedeutend mit: „Die Aussage $\hat{L}(s)$ gilt in s“, wobei*

$$\hat{L}(s) := \bigwedge_{v \in L(s)} v \wedge \bigwedge_{v \in (V \setminus L(s))} \neg v\,.$$

Somit kann jede temporale Struktur als Büchi-Automat *angesehen werden, in dem jeder Zustand akzeptierend ist, d. h. $A_B = S_B$.*

Da jede temporale Struktur als Büchi-Automat modellierbar ist, kann nun auch der *Produktautomat* einer temporalen Struktur M und eines Büchi-Automaten B gebildet werden. Sei $B = (S_B, R_B, L_B, A_B)$ ein Büchi-Automat mit Anfangszuständen $S_{B,0}$ und $M = (S, R, L)$ eine temporale Struktur mit Anfangszuständen S_0. Beide Automaten verwenden die selbe Menge V an atomaren Formeln. Der Produktautomat $M_p := M \times B = (S_p, R_p, L_p, A_p)$ ist ein Büchi-Automat und definiert durch:

- $S_p = \{(s, s_B) \in S \times S_B \mid \hat{L}(s) \Rightarrow L_B(s_B)\}$,
- $((s, s_B), (s', s'_B)) \in R_p$, genau dann, wenn $(s, s') \in R \wedge (s_B, s'_B) \in R_B$ und
- $(s, s_B) \in A_p$, genau dann, wenn $s_B \in A_B$.

Somit akzeptiert der Produktautomat $M \times B$ genau diejenigen Pfade von M, die mit B beschrieben sind. Die Anfangszustände ergeben sich direkt aus den Paaren von Anfangszustände der beiden Automaten. Ein anderes Ergebnis ist allerdings von größerer Bedeutung [183]:

Theorem 5.2.1. *Sei M eine temporale Struktur und φ eine LTL-Formel, so existiert ein Büchi-Automat $B_{\neg\varphi}$, so dass*

$$M \models \varphi \Leftrightarrow \mathcal{L}(M \times B_{\neg\varphi}) = \emptyset\,.$$

Somit wird für eine LTL-Formel φ ein Büchi-Automat generiert, der exakt die Sequenzen enthält, die φ nicht erfüllen. Durch die Bildung des Produktautomaten $M \times B_{\neg\varphi}$ entsteht ein Automat, der genau die Pfade von M enthält, die mit $B_{\neg\varphi}$ konsistent und somit mit φ inkonsistent sind. Nur wenn kein solcher Pfad existiert, erfüllt M die LTL-Formel φ.

Beispiel 5.2.4. Gegeben ist die temporale Struktur M in Abb. 5.16a) mit drei Zuständen. Der Zustand s_0 ist der Anfangszustand. Es soll gezeigt werden, dass

$$M, s_0 \models \mathrm{FG}\ p$$

gilt, d. h. irgendwann wird ein Zustand in M von s_0 aus erreicht, ab dem immer p erfüllt ist.

In einem ersten Schritt wird die temporale Struktur M in einen äquivalenten Büchi-Automaten transformiert. Dies ist in Abb. 5.16b) zu sehen. Anschließend wird der Büchi-Automat $B_{\neg\varphi}$ für die Negation von $\varphi = \mathrm{FG}\ p$ aufgebaut, d. h. $\neg\varphi = \mathrm{GF}\ \neg p$. Sequenzen, die $\neg\varphi$ erfüllen, enthalten unendlich oft einen Zustand, der mit $\neg p$ markiert ist. Im Allgemeinen ist die Konstruktion von Büchi-Automaten aus LTL-Formeln nicht trivial. In diesem Fall ist es aber hinreichend einfach zu sehen, dass der Büchi-Automat $B_{\neg\varphi}$ in Abb. 5.16b) die LTL-Formel $\neg\varphi$ beschreibt. Anfangszustände sind sowohl q_0 als auch q_1, d. h. $S_{B,0} = \{q_0, q_1\}$. Der einzige akzeptierende Zustand q_0 ist durch eine doppelte Umrandung gekennzeichnet.

Im folgenden Schritt wird der Produktautomat $M \times B_{\neg\varphi}$ aus M und $B_{\neg\varphi}$ gebildet. Dafür wird angenommen, dass jeder Zustand in M akzeptierend ist. Das Ergebnis ist in Abb. 5.17 zu sehen. Der Anfangszustand ist (s_0, q_1). Der einzige akzeptierende Zustand ist (s_1, q_0).

Um zu zeigen, dass $M \models \varphi$ gilt, muss gezeigt werden, dass

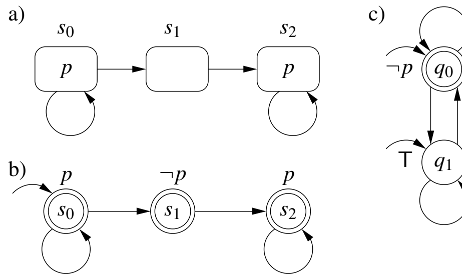


Abb. 5.16. a) Eine temporale Struktur M und b) äquivalenter Büchi-Automat sowie c) der Büchi-Automaten $B_{\neg\varphi}$ für $\varphi = \text{FG}\ p$ [457]

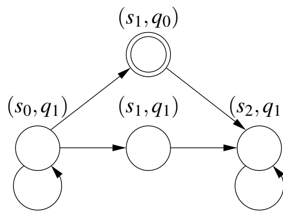


Abb. 5.17. Produktautomat von $M \times B_{\neg\varphi}$ [457]

$$\mathcal{L}(M \times B_{\neg\varphi}) = \emptyset$$

gilt, also die Sprache des Produktautomaten $M \times B_{\neg\varphi}$ leer sein soll. Oder mit anderen Worten, dass kein (unendlicher) Pfad durch den Produktautomaten existiert, der unendlich oft den Zustand (s_1, q_0) betritt. Dies ist für den Automaten in Abb. 5.17 offensichtlich.

5.2.3 Zusicherungsbasierte Eigenschaftsprüfung

In den letzten Jahren hat sich insbesondere im industriellen Umfeld eine neue Form der funktionalen Eigenschaftsprüfung etabliert, die als *zusicherungsbasierte Eigenschaftsprüfung* (engl. *assertion-based verification*) bezeichnet wird. Wie der Name bereits suggeriert, basiert die Methodik auf der Formulierung von Zusicherungen (engl. *assertions*). Dies erfolgt typischerweise in einer *Zusicherungssprache* wie PSL (siehe Abschnitt 2.4.3) und hat verschiedene Ziele:

1. Dokumentation der Verifikationsaufgaben
2. Formulierung funktionaler Eigenschaften für die formale Modellprüfung
3. Formulierung funktionaler Eigenschaften für die simulative Verifikation
4. Formulierung der Anforderungen an die Stimuli für die simulative Verifikation

5. Formulierung funktionaler Eigenschaften für die Überprüfung im späteren Betrieb des Systems

Da Zusicherungssprachen für die Formulierung von CTL- und LTL-Formeln geeignet sind, stellen Zusicherungssprachen eine Möglichkeit dar, die zu prüfenden Zusicherungen Werkzeugen zur formalen Modellprüfung zu übergeben. Darüber hinaus können Simulatoren, die Zusicherungssprachen unterstützen, sowohl diese Zusicherungen während der Simulation überprüfen, als auch die Stimuli-Erzeugung entsprechend der geforderten Überdeckungen und Restriktionen steuern. Hierfür werden die Zusicherungen zunächst in Büchi-Automaten (siehe Definition 5.2.1 auf Seite 186) übersetzt und anschließend in einen entsprechenden *Monitor* zur Überprüfung oder *Generator* zur Stimuli-Erzeugung übersetzt. Für die Überprüfung von Zusicherungen während des Betriebs werden die Monitore in Hardware oder Software synthetisiert und in das System integriert. Die formale Modellprüfung ist in den vorangegangenen Abschnitten bereits ausführlich dargestellt worden, weshalb im Folgenden ausschließlich der Einsatz von Zusicherungen in der Simulation diskutiert wird.

In einigen Programmiersprachen bzw. System- oder Hardware-Beschreibungssprachen sind Zusicherungen ein Bestandteil der Sprache. Wird während der Simulation oder des Betriebs die Verletzung einer Zusicherung registriert, wird dies unverzüglich angezeigt. Dies hat den Vorteil, dass die Verwendung von Zusicherungen die Beobachtbarkeit des Systems erhöht, da der Fehler nicht mehr zu den primären Ausgängen des Systems übertragen werden muss. Hierdurch kann auch die Zeit zwischen Fehlererzeugung und Fehlerdetektion in der Simulation verkürzt werden, was wichtig ist, da eine frühzeitige Fehlererkennung die Kosten zur Fehlerbeseitigung verringert. Hierzu muss allerdings in einer simulativen Verifikationsmethode der Fehler auch stimuliert werden, d. h. die Zusicherung während der Simulation verletzt werden. Hierzu ist es notwendig, aber nicht hinreichend, dass der Fehlerort simuliert wird, wie das folgende Beispiel zeigt.

Beispiel 5.2.5. Betrachtet werden die beiden Ausschnitte aus C-Programmen:

```
1      if(x > 1)
2        assert(x > 1);
```

und

```
1      if(x >= 1)
2        assert(x > 1);
```

Unter der Annahme, dass die Programme ansonsten identisch sind, wird die Simulation beider Programme stets zum selben Ergebnis führen, außer die Variable x besitzt bei erreichen der `if`-Anweisung exakt den Wert 1. Nimmt man weiterhin an, dass `x >= 1` tatsächlich einem Fehler entspricht, so sieht man, dass die Ausführung des Fehlerorts nicht hinreichend zur Stimulierung des Fehlers ist.

Während das obige Beispiel einer Zusicherung trivial ist und keinerlei zeitliche Relationen von Berechnungen enthält, ist für die Überprüfung temporaler Eigenschaften ein sog. *Monitor* notwendig – einfache `assert`-Anweisungen reichen dann nicht mehr aus. Im Folgenden wird die Generierung von Monitoren und Generatoren für die Simulation diskutiert (siehe auch [472]).

Monitore

Ein *Monitor* überwacht den Zustand eines Systems während dessen Simulation. Dabei bewertet der Monitor ständig den Status der Zusicherung, die es zu überwachen gilt. Technisch gesehen sind Monitore endliche Zustandsautomaten, welche bestimmte Simulationssequenzen akzeptieren oder ablehnen. Hierfür werden Zustände des Monitors als *akzeptierende* oder *ablehnende Zustände* markiert. Durch die Simulation wird der Monitor schließlich in einen akzeptierenden oder ablehnenden Zustand überführt, was auf die Erfüllung oder Nichterfüllung der zu prüfenden Eigenschaft (Zusicherung) schließen lässt.

Die genaue Arbeitsweise von Monitoren basiert auf Ergebnissen aus der Analyse formaler Sprachen. Historisch gesehen sind reguläre Ausdrücke und temporale Logiken als Spracherkenner entwickelt worden. Eine Sprache ist über eine endliche Menge von Symbolen definiert, dem sog. *Alphabet*. Folgen von Symbolen bilden Wörter einer Sprache. *Reguläre Ausdrücke* können Wörter endlicher Länge erkennen, während *ω-reguläre Ausdrücke* Wörter unendlicher Länge erkennen können. Die Zugehörigkeit eines Wortes zu einer Sprache kann bestimmt werden, in dem ein *nichtdeterministischer endlicher Automat* bzw. ein *Büchi-Automat* für den regulären Ausdruck bzw. ω-regulären Ausdruck konstruiert wird. Die Konstruktion von Automaten für reguläre Ausdrücke wurde bereits intensiv in den 1960er Jahren untersucht, während die Konstruktion von Büchi-Automaten für ω-reguläre Ausdrücke erst Mitte der 1980er Jahre entwickelt wurde. Die Automaten werden während der Simulation mit einem Wort stimuliert. Die Evaluierung erfolgt somit wie folgt:

1. *Aufbau des Automaten*: Einige Zustände des Automaten sind als Anfangszustand und manche als akzeptierender Zustand gekennzeichnet. Die Zustandsübergänge werden mit Elementen des Alphabets markiert.
2. *Evaluierung des Wortes*: Beginnend in den Anfangszuständen des Automaten werden die Zustandsübergänge entsprechend der Sequenz der Symbole im Wort und der Markierung der Zustandsübergänge verfolgt. Im Allgemeinen existieren mehrere Pfade zu einem einzelnen Wort. Dies liegt darin begründet, dass es 1) mehrere Anfangszustände geben kann und 2) jeder Zustand mehrere Zustandsübergänge mit gleicher Markierung zu Folgezuständen besitzen kann.
3. *Akzeptanz eines Wortes*: Im Falle regulärer Ausdrücke wird das Wort akzeptiert, sofern einer der berechneten Pfade in einem akzeptierenden Zustand endet. Im Falle ω-regulärer Ausdrücke wird ein Wort akzeptiert, sofern einer der berechneten Pfade einen akzeptierenden Zustand unendlich oft besucht, d. h. dieser Pfad nur Schleifen enthält, die nicht verlassen werden können und auf denen jeweils mindestens ein akzeptierender Zustand liegt.
4. *Ablehnung eines Wortes*: Wird ein Wort nicht akzeptiert, so gilt dieses als abgelehnt. Allerdings muss man bei simulativen Verifikationsmethoden zwei Fälle unterscheiden: 1) die Evaluierung des (endlichen) Wortes wurde beendet und das Wort nicht akzeptiert und 2) die Evaluierung des unendlichen Wortes wurde terminiert, da der letzte Zustand keinen ausgehenden Zustandsübergang enthält, der mit dem momentanen Symbol markiert ist.

Beispiel 5.2.6. Gegeben ist der reguläre Ausdruck $(ab)*$. Der zugehörige nichtdeterministische endliche Automat ist in Abb. 5.18a) zu sehen. Die Anfangszustände sind s_1 und s_3. Die akzeptierte Sprache ist $\mathcal{L} = \{\varepsilon, ab, abab, ababab, \dots\}$. $\mathcal{L}$ ist in Abb. 5.18b) als Berechnungsbaum dargestellt.

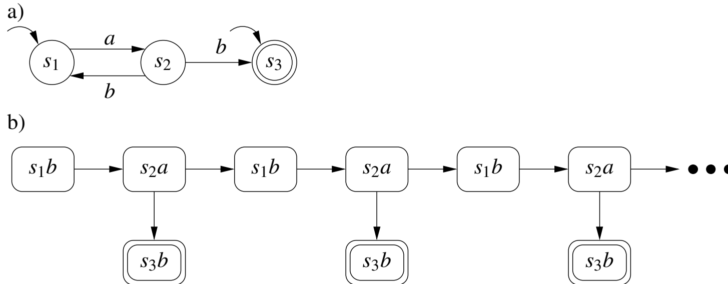


Abb. 5.18. a) nichtdeterministischer endlicher Automat für $(ab)*$ und b) zugehöriger Berechnungsbaum [472]

Während der Simulation ist es allerdings schwer, alle Pfade, die während der Evaluierung entstehen, zu speichern. Dies ist aber im Falle nichtdeterministischer endlicher Automaten notwendig, um tatsächlich alle möglichen Pfade zu simulieren. Aus diesem Grund werden die Automaten zunächst in deterministische Automaten übersetzt. Für nichtdeterministische endliche Automaten kann dies beispielsweise durch die sog. *Teilmengenkonstruktion* erfolgen (siehe z. B. [223]), die später beschrieben wird. Für Büchi-Automaten ist die Übersetzung deutlich aufwendiger [322, 385].

Generatoren

Die zuvor beschriebenen Automaten für die Monitore können wiederum als Generatoren verwendet werden. Hierbei ist die Interpretation des Automaten nicht die eines Spracherkenners, sondern die eines Spracherzeugers. In diesem Fall werden die Zustandsübergänge nicht durch die Simulation eines Wortes hervorgerufen. Vielmehr werden die Symbole mit denen die Zustandsübergänge markiert sind, beim Übergang erzeugt. Übergänge erfolgen zufällig, indem ein ausgehender Zustandsübergang aus dem momentanen Zustand gewählt wird. Dies kann man auch als Pfade im Berechnungsbaum, in Abb. 5.18b) für den regulären Ausdruck $(ab)*$, interpretieren.

Diese rein zufällige Generierung von Stimuli kann durch weitere Zusicherungen eingeschränkt werden. In diesem Fall muss zunächst evaluiert werden, ob überhaupt ein Pfad existiert, der die momentanen Beschränkungen erfüllt. Ist dies nicht der Fall, existiert kein gültiger Stimulus.

Monitor-Konstruktion für PSL-Zusicherungen

Im Folgenden wird die Konstruktion von Monitoren für PSL-Zusicherungen für eine Teilmenge der Foundation Language von PSL beschrieben (siehe Abschnitt 2.4.3). Die PSL-FL wird hierbei derart eingeschränkt, dass die Negation von sequentiell erweiterten regulären Ausdrücken (engl. *sequential extended regular expressions, SEREs*) nicht zulässig ist. Die resultierende Sprache wird mit PSL-FL$^-$ bezeichnet.

Definition 5.2.3 (PSL-FL$^-$). *Alle aussagenlogischen Formeln sind PSL-FL$^-$-Formeln. Sei r eine SERE und p_1 und p_2 zwei PSL-FL$^-$-Formeln, dann sind auch die folgenden Ausdrücke PSL-FL$^-$-Formeln:*

- $\{r\}$
- $p_1 \vee p_2$
- $p_1 \wedge p_2$
- $\mathrm{X}\, p_1$
- $p_1\, \mathrm{U}\, p_2$
- $p_1\, \mathrm{R}\, p_2$

Man muss beachten, dass der Ausschluss der Negation keinen Einfluss auf die Expressivität von LTL-Formeln hat, die keine SEREs verwenden. Die abkürzenden Schreibweisen F p_1 und G p_1 lassen sich aus obiger Definition ableiten und sind ebenfalls PSL-FL$^-$-Formeln.

Die Übersetzung einer PSL-FL$^-$-Formel in einen Monitor erfolgt, indem mittels der sog. *Tableau-Technik* zunächst ein nichtdeterministischer endlicher Automat konstruiert und dieser anschließend in einen deterministischen endlichen Automaten übersetzt wird. Mittels der Tableau-Technik wird die Erfüllbarkeit einer LTL-Formel in einen aussagenlogischen Teil und eine Verpflichtung (engl. *obligation*) für den nächsten Zustand des Modells zerlegt. Beide Teile werden anschließend disjunktiv verknüpft.

Beispiel 5.2.7. Gegeben ist die LTL-Formel p_1 U p_2. Um diese zu erfüllen, muss entweder p_2 im aktuellen Zustand gelten oder es gilt p_1 und im nächsten Zustand gilt verpflichtend p_1 U p_2. Ersteres hebt die Verpflichtung auf, da hierdurch bereits die LTL-Formel im momentanen Zustand erfüllt ist. Letzteres verlangt, dass nachdem p_1 im momentanen Zustand gilt, die ursprüngliche LTL-Formel im nächsten Zustand erfüllt ist.

Überdeckung von PSL-FL$^-$-Formeln

Die disjunktive Verknüpfung beider Teile einer zerlegten LTL-Formel wird als *Überdeckung* der LTL-Formel bezeichnet. Im Folgenden wird die Notation $\mathrm{N}(p)$ verwendet, um anzuzeigen, dass die Formel p als Verpflichtung für den nächsten Zustand gilt. Man beachte, dass $\mathrm{N}(p_1) \wedge \mathrm{N}(p_2) = \mathrm{N}(p_1 \wedge p_2)$ und $!\mathrm{N}(p) = \mathrm{N}(!p)$ ist.

Die Berechnung der Überdeckung einer PSL-FL$^-$-Formel wird durch die Bestimmung der Überdeckung von LTL-Formeln und SEREs bestimmt. Für LTL-Formeln kann die Überdeckung in zwei Schritten bestimmt werden:

1. Jede LTL-Formel wird entsprechend der folgenden Umformungen in einen aussagenlogischen Teil und eine Verpflichtung für den nächsten Zustand zerlegt:

$$\begin{aligned} p_1 \mathrm{U}\, p_2 &= p_2 \vee (p_1 \wedge \mathrm{N}(p_1 \mathrm{U}\, p_2)) \\ p_1 \mathrm{R}\, p_2 &= p_2 \wedge (p_1 \vee \mathrm{N}(p_1 \mathrm{R}\, p_2)) \\ \mathrm{F}\, p &= p \vee \mathrm{N}(\mathrm{F}\, p) \\ \mathrm{G}\, p &= p \wedge \mathrm{N}(\mathrm{G}\, p) \end{aligned}$$

2. Umwandlung des Ergebnisses in disjunktive Normalform.

Die Berechnung der Überdeckung von regulären Ausdrücken erfolgt indem reguläre Ausdrücke in Disjunktionen aus dem leeren Symbol ε, einer atomaren Aussage b oder eines regulären Ausdrucks zerlegt wird. Dies erfolgt in drei Schritten:

1. Für reguläre Ausdrücke r, s, t werden zunächst die folgenden Distributivregeln so oft wie möglich angewendet:

$$\begin{aligned} (r \mid s)t &= (rt \mid st) \\ t(r \mid s) &= (tr \mid ts) \end{aligned}$$

2. Enthält der reguläre Ausdruck den $*$-Operator, dann wird der entsprechend reguläre Teilausdruck wie folgt umgeschrieben und anschließend Schritt 1. wiederholt:

$$r* = rr* \mid \varepsilon$$

3. Ersetze alle $|$-Symbole durch Disjunktionen ($\vee$) und jeden Teilausdruck in der Form br wie folgt:

$$br = b \wedge \mathrm{N}(r)$$

   wobei b eine atomare Aussage und r ein regulärer Ausdruck ist

Beispiel 5.2.8. Die Berechnung der Überdeckung für den regulären Ausdruck $(a{*}b){*}$ sieht somit zunächst durch Anwendung der Schritte 1. und 2. wie folgt aus:

$$\begin{aligned} (a{*}b){*} &= (a{*}b)(a{*}b){*} \mid \varepsilon \\ &= ((aa{*} \mid \varepsilon)b)(a{*}b){*} \mid \varepsilon \\ &= (aa{*}b \mid b)(a{*}b){*} \mid \varepsilon \\ &= (aa{*}b)(a{*}b){*} \mid b(a{*}b){*} \mid \varepsilon \end{aligned}$$

Somit ergibt sich die Überdeckung zu (Schritt 3.):

$$a \wedge \mathrm{N}((a{*}b)(a{*}b){*}) \vee b \wedge \mathrm{N}((a{*}b){*}) \vee \varepsilon$$

Konstruktion des nichtdeterministischen endlichen Automaten

Durch die wiederholte Berechnung der Überdeckung kann für eine PSL-FL$^-$-Formel der nichtdeterministische endliche Automat konstruiert werden. Die Überdeckung einer PSL-FL$^-$-Formel besteht aus Bedingungen für den aktuellen Zustand und Verpflichtungen für den folgenden Zustand. Für die Verpflichtung wird wiederum die Überdeckung berechnet, sofern dies noch nicht geschehen ist. Dies kann zu weiteren Verpflichtungen (dann für den übernächsten Zustand) führen. Die wiederholte Berechnung der Überdeckung terminiert irgendwann, da die verwendeten Umformungen stets nur Teilformeln der Originalformel verwenden, welche eine endliche Länge haben.

Die Terme der Überdeckungen ergeben schließlich die Zustände des nichtdeterministischen endlichen Automaten. Somit kann der nichtdeterministische endliche Automat wie folgt erzeugt werden:

1. Die Anfangszustände entsprechen den Termen der Originalformel.
2. Füge einen Zustandsübergang von einem Zustand x_1 zu einem Zustand x_2 hinzu, sofern der durch x_2 repräsentierte Term eine Teilformel enthält, welche die Verpflichtung im nächsten Zustand für die durch x_1 repräsentierte Formel ist.
3. Füge einen zusätzlichen Zustand hinzu und markiere ihn als akzeptierenden Zustand. Für jeden Zustand ohne ausgehenden Zustandsübergang füge einen Zustandsübergang von diesem Zustand zu dem akzeptierenden Zustand hinzu. Dies bedeutet auch, dass der akzeptierende Zustand eine Selbstschleife besitzt.
4. Markiere jeden Zustandsübergang mit der Konjunktion aller atomaren Formeln in der mit dem Zustand assoziierten Formel.
5. Entferne alle Zustandsübergänge, die mit ε markiert sind, indem die entsprechenden beiden Zustände verschmolzen werden.

Beispiel 5.2.9. Es wird wiederum der reguläre Ausdruck $(a{*}b){*}$ aus Beispiel 5.2.8 betrachtet. Die Überdeckung des regulären Ausdrucks hat zu folgender Formel geführt:

$$a \wedge \mathrm{N}((a{*}b)(a{*}b){*}) \vee b \wedge \mathrm{N}((a{*}b){*}) \vee \varepsilon$$

Diese enthält zwei Verpflichtungen für den nächsten Zustand. Die Verpflichtung $\mathrm{N}((a{*}b){*})$ entspricht der Originalformel und wird nicht weiter betrachtet. Für $\mathrm{N}(a{*}b\,(a{*}b){*})$ jedoch muss eine weitere Überdeckung berechnet werden:

$$\begin{aligned}(a{*}b)(a{*}b){*} &= ((aa{*} \mid \varepsilon)b)(a{*}b){*}\\ &= (aa{*}b \mid b)(a{*}b){*}\\ &= (aa{*}b)(a{*}b){*} \mid b(a{*}b){*}\end{aligned}$$

Die Überdeckung ergibt sich zu:

$$a \wedge \mathrm{N}((a{*}b)(a{*}b){*}) \vee b \wedge \mathrm{N}((a{*}b){*})$$

Diese Überdeckung enthält wiederum zwei Verpflichtungen für den nächsten Zustand. Für beide ist jedoch bereits die Überdeckung berechnet, weshalb die Berechnung der Überdeckungen beendet wird.

Der zugehörige nichtdeterministische endliche Automat ist in Abb. 5.19 zu sehen. Die Zustände s_0, s_1 und s_2 sind die Anfangszustände und entsprechen den drei Termen der Überdeckung der Originalformel aus Beispiel 5.2.8. Die Zustände s_3 und s_4 entsprechen den Termen der Überdeckung der Verpflichtung $\mathrm{N}((a{*}b)(a{*}b){*})$. Der akzeptierende Zustand ist in diesem Beispiel mit s_2 verschmolzen worden, da s_2 lediglich einen ausgehenden Zustandsübergang mit der Markierung ε zu dem akzeptierenden Zustand besitzt.

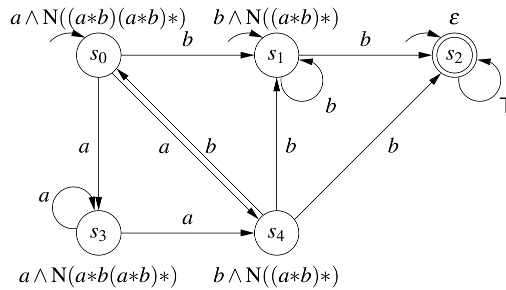


Abb. 5.19. Nichtdeterministischer endlicher Automat für den regulären Ausdruck $(a{*}b){*}$ [472]

Beispiel 5.2.10. Gegeben ist die PSL-FL$^-$-Formel $!p \rightarrow \mathrm{G}\,\{(a \mid b){*}\}$. Man beachte, dass die Negation (!) nicht auf eine SERE angewendet wurde. Die Überdeckung der Formel errechnet sich zu:

$$\begin{aligned}
!p \rightarrow \mathrm{G}\,\{(a \mid b){*}\} &= p \vee \{(a \mid b){*}\} \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\}) \\
&= p \vee \{((a \mid b)(a \mid b){*} \mid \varepsilon)\} \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\}) \\
&= p \vee \{(a(a \mid b){*} \mid b(a \mid b){*} \mid \varepsilon)\} \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\}) \\
&= p \vee \{a\} \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\}) \vee \{b\} \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\}) \mid \{\varepsilon)\} \\
&\qquad \wedge \mathrm{N}(\mathrm{G}\,\{(a \mid b){*}\})
\end{aligned}$$

Bei der letzten Umformung wurde ausgenutzt, dass $p \wedge \mathrm{G}\,p = \mathrm{G}\,p$ ist.

Man sieht, dass keine weiteren Verpflichtungen für den nächsten Zustand entstanden sind, für die noch keine Überdeckung bestimmt wurde. Der resultierende nichtdeterministische Automat ist in Abb. 5.20a) zu sehen. Anfangszustände sind die Zustände s_0, s_1, s_2 und s_3. Der akzeptierende Zustand ist s_4. Der nichtdeterministische endliche Automat ohne ε-Übergänge ist in Abb. 5.20b) zu sehen.

Determinierung

Ein einzelner Simulationslauf kann in einem nichtdeterministischen endlichen Automaten die Ausführung vieler unterschiedlicher Pfade anregen. All diese möglichen

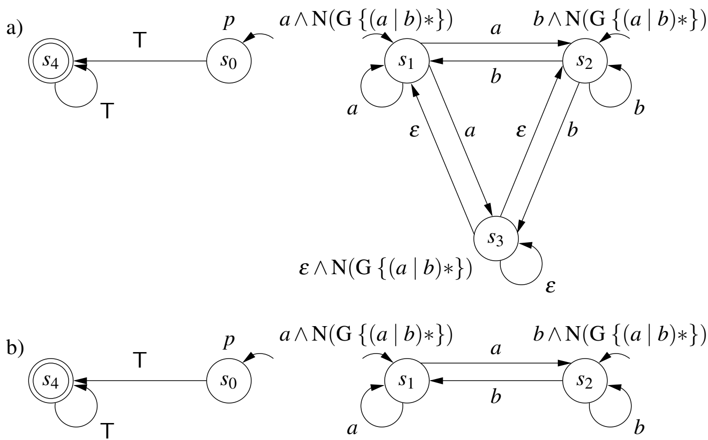


Abb. 5.20. a) nichtdeterministischer endlicher Automat für die PSL-FL$^-$-Formel $!p \rightarrow$ G $\{(a \mid b)*\}$ mit ε-Übergängen und b) ohne ε-Übergänge [472]

Pfade zu verfolgen kann sehr aufwendig sein. Andererseits kann bei der Determinierung eines nichtdeterministischen endlichen Automaten eine exponentielle Anzahl an Zuständen entstehen. Dennoch ist es wünschenswert einen nichtdeterministischen endlichen Automaten in einen endlichen Automaten zu transformieren, so dass dieser direkt als Monitor oder Generator in der Simulation dienen kann. Hierzu wird üblicherweise die sog. *Teilmengenkonstruktion* verwendet.

Die Teilmengenkonstruktion funktioniert wie folgt:

1. Erzeuge einen Anfangszustand für den deterministischen endlichen Automaten, der die Menge aller Anfangszustände des nichtdeterministischen endlichen Automaten repräsentiert.
2. Für jeden neuen Zustand s im deterministischen endlichen Automaten verfahre wie folgt:
   - Für jeden Buchstaben i im Alphabet bestimme den Folgezustand s'. Dieser repräsentiert alle Folgezustände im nichtdeterministischen endliche Automaten, die durch Eingabe von i in einem der Zustände, die in s repräsentiert sind, erreichbar sind.
   - Füge s' zu den Zuständen des deterministischen endlichen Automaten hinzu, sofern dieser Zustand noch nicht existiert, und füge einen mit i markierten Zustandsübergang von s nach s' hinzu.
   - Alle Zustände im deterministischen endlichen Automaten, die einen akzeptierenden Zustand des nichtdeterministischen endlichen Automaten repräsentieren, sind akzeptierende Zustände.

Beispiel 5.2.11. Betrachtet wird wiederum die PSL-FL$^-$-Formel $!p \rightarrow$ G $\{(a \mid b)*\}$ aus Beispiel 5.2.10. Der zugehörige deterministische Automat ist in Abb. 5.21 zu sehen.

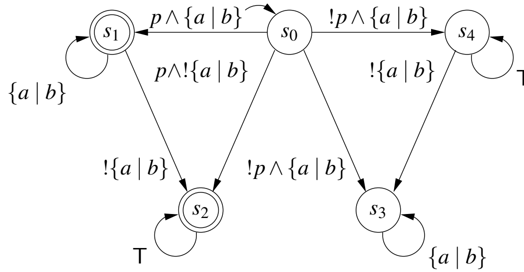


Abb. 5.21. Deterministischer endlicher Automat für die PSL-FL^{-}-Formel $!p \rightarrow \mathrm{G}\ \{(a \mid b)*\}$ [472]

Die Hardware-Synthese von Monitoren ist aus den generierten Monitoren unter der Annahme möglich, dass Zeitpunkte für Zustandsübergänge mittels eines Taktsignals definiert sind. Für Software- oder Systembeschreibungen müssen geeignete Ereignisse zunächst spezifiziert werden (siehe auch Abschnitt 8.1.4).

5.3 Symbolische Modellprüfung

Die in Abschnitt 5.2 vorgestellten Verifikationsmethoden zur CTL-Modellprüfung basieren auf einer expliziten Aufzählung der erreichbaren Zustände. Aus diesem Grund sind diese Verfahren nur auf relativ kleine Systeme anwendbar. Die Effizienz der Algorithmen kann jedoch erheblich gesteigert werden, wenn der Zustandsraum *symbolisch* repräsentiert wird. Dabei werden Zustandsmengen nicht mehr durch Aufzählung der Zustände, sondern implizit durch Formeln repräsentiert, wobei eine einzelne Formel eine Menge von vielen Zuständen beschreiben kann. Im Folgenden werden Verfahren zur symbolischen CTL- und LTL-Modellprüfung vorgestellt.

5.3.1 BDD-basierte CTL-Modellprüfung

Für eine symbolische CTL-Modellprüfung müssen analog zur symbolischen Erreichbarkeitsanalyse aus Abschnitt 4.3.3 die Zustände $s \in S$ der temporalen Struktur $M = (S, R, L)$ binär codiert werden. Hierzu dient die Funktion:

$$\sigma_S : S \rightarrow \mathbb{B}^k$$

Die Zahl $k \in \mathbb{N}$ ist dabei die Anzahl der für die Codierung verwendeten binären Variablen. Damit können Zustandsmengen $S' \subseteq S$ als charakteristische Funktionen $\psi_{S'}$ dargestellt werden, mit der Eigenschaft:

$$\forall s \in S : \psi_{S'}(\sigma_S(s)) = \mathsf{T} \Leftrightarrow s \in S'$$

Weiterhin ergibt sich die charakteristische Funktion $\psi_R : \mathbb{B}^k \times \mathbb{B}^k \rightarrow \mathbb{B}$ für die Zustandsübergangsrelation zu:

$$\psi_R(\sigma_S(s), \sigma_S(s')) = \begin{cases} \mathsf{T} & \text{falls } (s,s') \in R \\ \mathsf{F} & \text{sonst} \end{cases} \tag{5.12}$$

Die charakteristischen Funktionen können effizient mit reduzierten geordneten binären Entscheidungsdiagrammen repräsentiert werden (siehe Anhang B).

Mit den charakteristischen Funktionen können nun die Funktionen COMPUTE, COMPUTE_EX, COMPUTE_EG und COMPUTE_EU derart umgeschrieben werden, dass die CTL-Modellprüfung symbolisch durchgeführt werden kann. Jede symbolische Berechnung gibt dabei eine charakteristische Funktion $\psi : \mathbb{B}^k \rightarrow \mathbb{B}$ der berechneten Zustandsmenge zurück.

Zunächst wird die Funktion SYM_COMPUTE vorgestellt. Diese erhält als Argumente die temporale Struktur M und die CTL-Formel φ.

```
SYM_COMPUTE(M, φ) {
    SWITCH
        CASE φ = F:
            ψ := F;
        CASE φ = T:
            ψ := T;
        CASE φ ∈ V:
            ψ := ⋁_{s∈{s′∈S|φ∈L(s′)}} σ_S(s);
        CASE φ = ¬p:
            ψ := ¬SYM_COMPUTE(M, p);
        CASE φ = p ∨ q:
            ψ := SYM_COMPUTE(M, p) ∨ SYM_COMPUTE(M, q);
        CASE φ = EX p:
            ψ := SYM_COMPUTE_EX(M, SYM_COMPUTE(M, p));
        CASE φ = EG p:
            ψ := SYM_COMPUTE_EG(M, SYM_COMPUTE(M, p));
        CASE φ = E p U q:
            ψ := SYM_COMPUTE_EU(M, SYM_COMPUTE(M, p),
                            SYM_COMPUTE(M, q));
    RETURN ψ;
}
```

Im Falle, dass φ eine atomare Formel $v \in V$ darstellt, muss die Zustandsmenge derjenigen Zustände symbolisch repräsentiert werden, die mit der Variablen v markiert sind. Die symbolische Repräsentation dieser Menge ist die Disjunktion der Codierungen der entsprechenden Zustände. Die Methode SYM_COMPUTE_EX kann dann wie folgt implementiert werden:

```
SYM_COMPUTE_EX(M, ψ_p) {
    ψ := ∃σ_S(s′) : ψ_p|_{s:=s′} ∧ ψ_R;
    RETURN ψ;
}
```

Die Funktion SYM_COMPUTE_EX erhält als Argument die temporale Struktur und die charakteristische Funktion ψ_p, welche die Menge derjenigen Zustände repräsentiert, deren direkte Vorgänger bestimmt werden sollen. Die Funktion gibt die charakteristische Funktion der Menge der Zustände zurück, die direkte Vorgänger der Zustände codiert durch ψ_p sind.

Die Funktion SYM_COMPUTE_EG kann wie folgt realisiert werden:

```
SYM_COMPUTE_EG(M, ψp) {
    ψN := T;
    DO
        ψR := ψN;
        ψN := ψp ∧ SYM_COMPUTE_EX(M, ψN);
    UNTIL (ψN = ψR)
    RETURN ψN;
}
```

Die Funktion erhält die temporale Struktur M und die charakteristische Funktion ψ_p der Zustandsmenge S_p als Argument und liefert die charakteristische Funktion für die Menge derjenigen Zustände, in denen EGp gilt. Da es sich hierbei um die Berechnung des größten Fixpunktes handelt, wird ψ_N mit T initialisiert. Statt der Berechnung der Schnittmenge wie in der Funktion COMPUTE_EG, erfolgt nun die Konjunktion Boolescher Funktionen.

Schließlich lässt sich die Funktion SYM_COMPUTE_EU wie folgt umsetzen:

```
SYM_COMPUTE_EU(M, ψp, ψq) {
    ψN := F;
    DO
        ψR := ψN;
        ψN := ψq ∨ (ψp ∧ SYM_COMPUTE_EX(M, ψN));
    UNTIL (ψN = ψR)
    RETURN ψN;
}
```

Als Argumente erhält die Funktion die temporale Struktur M sowie die charakteristischen Funktionen ψ_p und ψ_q für die Zustandsmengen S_p und S_q. Die Funktion gibt eine charakteristische Funktion für die Menge der Zustände zurück, die E p U q erfüllen. Hierbei handelt es sich um die Bestimmung des kleinsten Fixpunktes, weshalb ψ_N vor der Iteration mit F (dem symbolischen Äquivalent der leeren Menge) initialisiert wird.

5.3.2 SAT-basierte Modellprüfung

Für die Modellprüfung wird eine Spezifikation der geforderten funktionalen Eigenschaften mittels einer Temporallogik vorgenommen, während das System als temporale Struktur modelliert wird. *Explizite Modellprüfung* (siehe Abschnitt 5.2) hat jedoch oft mit der Zustandsraumexplosion zu kämpfen, weshalb *symbolische Modellprüfung* weit häufiger zum Einsatz kommt. Wie oben beschrieben, haben sich

binäre Entscheidungsdiagramme (BDD) als geeignet herausgestellt, um große Zustandsräume zu codieren. Mit BDDs ist es möglich, Systeme mit mehreren hundert Zustandsvariablen effizient zu traversieren. Dabei hängt die Größe der BDDs jedoch stark von der gewählten Variablenordnung ab und eine optimale Variablenordnung zu finden ist wiederum ein schweres Problem. Vor diesem Hintergrund ist leicht zu erkennen, dass für reale Systeme mit mehreren tausend Zustandsvariablen eine BDD-basierte Modellprüfung nicht immer zielführend ist.

In [48, 49] stellen Biere et al. ein *SAT-basiertes Modellprüfungsverfahren* vor. Die grundlegende Idee besteht darin, ein Gegenbeispiel einer gegebenen Länge k zu suchen. Somit handelt es sich bei der SAT-basierten Modellprüfung um eine Methode, welche die Falsifikation zum Ziel hat. Hierfür wird eine aussagenlogische Formel konstruiert, die genau dann erfüllbar ist, wenn ein solches Gegenbeispiel existiert. Somit spricht man bei SAT-basierter Modellprüfung auch von einer *beschränkten Modellprüfung* (engl. *bounded model checking*), wobei die maximale Länge des gesuchten Gegenbeispiels die Schranke angibt. Die Vorteile dieses Ansatzes liegen auf der Hand: Erstens können Gegenbeispiele mittels SAT-basierter Modellprüfung oftmals sehr schnell gefunden werden. Dies liegt in der SAT-Solvern zugrundeliegenden Tiefensuche (siehe Anhang C.2). Dabei sollte beachtet werden, dass das Auffinden von Gegenbeispielen eine hohe industrielle Relevanz besitzt. Zweitens liefert SAT-basierte Modellprüfung, sofern die Schranke k sukzessive erhöht wird, kürzest mögliche Gegenbeispiele, was oftmals das Verständnis für das Gegenbeispiel erhöht. Drittens ist der Speicherbedarf von SAT-basierter Modellprüfung wesentlich geringer als bei BDD-basierter Modellprüfung. Dies erlaubt die Prüfung wesentlich größerer Systeme. Schließlich benötigt SAT-basierte Modellprüfung keine optimierte Variablenordnung. Standardverfahren wie in Anhang C.2 sind ausreichend.

Im Folgenden wird die SAT-basierte symbolische Modellprüfung für funktionale Eigenschaften, die in linearer temporaler Aussagenlogik (LTL, siehe Abschnitt 2.4.2) spezifiziert sind, vorgestellt. Die Semantik von LTL-Formeln ist in Definition 2.4.4 auf Seite 76 bezüglich einer gegebenen temporalen Struktur $M = (S, R, L)$ und einem Pfad $\tilde{s}$ angegeben. Die temporale Struktur besteht aus einer Zustandsmenge S, einer Übergangsrelation $R \subseteq S \times S$ und einer Markierungsfunktion $L : S \to 2^V$, wobei V die Menge der atomaren Formeln (aussagenlogische Variablen) ist. Ein Pfad $\tilde{s} = \langle s_0, s_1, \ldots, s_i, \ldots \rangle$ ist eine Sequenz an Zuständen. $\tilde{s}^i := \langle s_i, s_{i+1}, \ldots \rangle$ beschreibt einen Suffix von $\tilde{s}$. ${}^i\tilde{s} := \langle s_0, \ldots, s_i \rangle$ beschreibt einen Präfix von $\tilde{s}$.

Handelt es sich bei M nicht um eine lineare Struktur, kann das Modellprüfungsproblem für LTL-Formeln unterschiedlich betrachtet werden. Die wird als *universelle* bzw. *existentielle Modellprüfung* bezeichnet und basiert auf der Gültigkeit der Formel.

Definition 5.3.1 (Gültigkeit von LTL-Formeln). *Eine LTL-Formel φ heißt* universell gültig*, falls für eine temporale Struktur M gilt, dass $\forall \tilde{s} : M, \tilde{s} \models \varphi$ ist. Eine LTL-Formel φ heißt* existentiell gültig*, falls für eine temporale Struktur M gilt, dass $\exists \tilde{s} : M, \tilde{s} \models \varphi$ ist.*

Universelle und existentielle Gültigkeit ergeben sich aus der Darstellung von LTL-Formeln in CTL*. Aus dieser Darstellung ergibt sich ebenfalls, dass eine LTL-

Formel φ für eine gegebene temporale Struktur M universell gültig ist, genau dann, wenn $\neg\varphi$ nicht existentiell gültig ist. Um also das universelle Modellprüfungsproblem zu lösen, reicht es, die gegebene LTL-Formel zu negieren und anschließend zu zeigen, dass das korrespondierende existentielle Modellprüfungsproblem keine Lösung besitzt. Dies kann geschehen, indem man versucht, ein Gegenbeispiel zu finden. Gelingt dies nicht, ist die Originalformel universell gültig. Da typischerweise eine Darstellung von LTL-Formeln in CTL* über die universelle Gültigkeit erfolgt, ist für die Einführung der SAT-basierten Modellprüfung die Diskussion der existentiellen Modellprüfung hinreichend.

In SAT-basierter Modellprüfung wird eine beschränkte Modellprüfung durchgeführt, wobei lediglich ein endlicher Präfix ${}^{i}\tilde{s}$ eines Pfades $\tilde{s}$ betrachtet wird, der eine Lösung für das existentielle Modellprüfungsproblem sein könnte. Dabei wird die Länge des Präfixes mit einer Schranke k begrenzt. Diese Schranke wird typischerweise sukzessive inkrementiert, wenn es nicht möglich ist, ein Gegenbeispiel für die gewählte Schranke zu finden.

Eine wichtige Beobachtung ist, dass ein endlicher Präfix dennoch eine unendliche Sequenz von Zuständen repräsentieren kann, wenn dieser Schleifen enthält (siehe Abb. 5.22b)). Falls der Präfix des Pfades keine Schleife enthält (Abb. 5.22a)), sagt dieser nichts über das endlos laufende System aus. Betrachtet man beispielsweise die LTL-Formel G φ. Nur ein Präfix mit einer Schleife und Gültigkeit von φ in jedem Zustand kann ein Modell für diese LTL-Formel darstellen. Ein Präfix, bei dem in jedem Zustand s_0 bis s_k φ erfüllt ist, der aber keine Schleife von s_k zu einem früheren Zustand enthält, kann kein Modell für G φ sein.

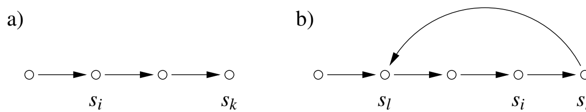


Abb. 5.22. a) Präfix ohne Schleife und b) (k,l)-Schleife [49]

Definition 5.3.2 ($(\mathbf{k},\mathbf{l})$-Schleife). *Ein Pfad $\tilde{s}$ wird für ein gegebenes l und k mit $l \leq k$ als (k,l)-*Schleife *bezeichnet, wenn gilt $s_{k+1} = s_l$ und $\tilde{s} = u \cdot v^{\infty}$ mit $u = \langle s_0,\ldots,s_{l-1}\rangle$ und $v = \langle s_l,\ldots,s_k\rangle$.*

Definition 5.3.3 (k-Schleife). *Existiert ein $l \in \mathbb{Z}_{\geq 0}$ mit $l \leq k$, so dass $\tilde{s}$ eine (k,l)-Schleife ist, so wird $\tilde{s}$ auch einfach als k-*Schleife *bezeichnet.*

Für eine beschränkte SAT-basierte Modellprüfung ist es notwendig, dass die Semantik von LTL-Formeln (siehe Definition 2.4.4 auf Seite 76) durch eine *beschränkte Semantik von LTL-Formeln* approximiert wird. Für die beschränkte Semantik wird lediglich ein Präfix von einem Pfad bestehend aus $k+1$ Zuständen betrachtet, d. h. ${}^{k}\tilde{s} = \langle s_0,\ldots,s_k\rangle$. Falls dieser Pfad eine k-Schleife ist, so bleibt die LTL-Semantik aus Definition 2.4.4 erhalten:

Definition 5.3.4 (Beschränkte Semantik von LTL-Formeln (mit Schleife)). *Sei M eine temporale Struktur mit Anfangszustand s_0. Sei ferner $k \in \mathbb{Z}_{\geq 0}$ und $\tilde{s}$ eine k-Schleife. Dann ist eine LTL-Formel φ gültig entlang des Pfades $\tilde{s} = \langle s_0, s_1, \dots \rangle$ mit Schranke k, geschrieben $M, \tilde{s} \models_k \varphi$, genau dann, wenn $M, \tilde{s} \models \varphi$.*

Handelt es sich bei $\tilde{s}$ hingegen nicht um eine k-Schleife, muss die Semantik approximiert werden. Betrachtet wird die LTL-Formel F φ. Diese Formel ist nur gültig entlang des Pfades $\tilde{s}$, wenn ein $i \in \mathbb{Z}_{\geq 0}$ existiert, so dass $\tilde{s}^i \models \varphi$. Mit anderen Worten: Es muss ein Suffix von $\tilde{s}$ existieren, so dass der Anfangszustand dieses Suffixes φ erfüllt. Wird der Präfix der Länge $k+1$ betrachtet, so besitzt der Zustand s_k allerdings keinen Nachfolgezustand, so dass sich die Semantik nicht rekursiv über Suffixe definieren lässt. Um dies zu umgehen, wird der Parameter i zur Definition der Semantik hinzugenommen und dafür $\models_k^i$ als Symbol verwendet. Dabei beschreibt i die momentane Position in der Sequenz und k die verwendete Schranke.

Definition 5.3.5 (Beschränkte Semantik von LTL-Formeln (ohne Schleife)). *Sei M eine temporale Struktur mit Anfangszustand s_0. Sei ferner $k \in \mathbb{Z}_{\geq 0}$ und $\tilde{s}$ ein Pfad, der keine k-Schleife ist. Dann ist eine LTL-Formel φ gültig entlang des Pfades $\tilde{s} = \langle s_0, s_1, \dots \rangle$ mit Schranke k, geschrieben $M, \tilde{s} \models_k \varphi$, genau dann, wenn $M, \tilde{s} \models_k^0 \varphi$, wobei*

$$
\begin{array}{lll}
M, \tilde{s} \models_k^i \varphi & \Leftrightarrow & \varphi \in L(s_i) \textit{ falls } \varphi \in V \\
M, \tilde{s} \models_k^i \neg\varphi & \Leftrightarrow & \varphi \notin L(s_i) \textit{ falls } \varphi \in V \\
M, \tilde{s} \models_k^i \varphi \vee \psi & \Leftrightarrow & M, \tilde{s} \models_k^i \varphi \textit{ oder } M, \tilde{s} \models_k^i \psi \\
M, \tilde{s} \models_k^i \varphi \wedge \psi & \Leftrightarrow & M, \tilde{s} \models_k^i \varphi \textit{ und } M, \tilde{s} \models_k^i \psi \\
M, \tilde{s} \models_k^i \mathrm{X}\, \varphi & \Leftrightarrow & i < k \textit{ und } M, \tilde{s} \models_k^{i+1} \varphi \\
M, \tilde{s} \models_k^i \mathrm{G}\, \varphi & \Leftrightarrow & \textit{gilt niemals} \\
M, \tilde{s} \models_k^i \varphi \,\mathrm{U}\, \psi & \Leftrightarrow & \exists i \leq j \leq k : M, \tilde{s} \models_k^j \psi \wedge \forall i \leq n < j : M, \tilde{s} \models_k^n \varphi \\
M, \tilde{s} \models_k^i \varphi \,\mathrm{R}\, \psi & \Leftrightarrow & \exists i \leq j \leq k : M, \tilde{s} \models_k^j \varphi \wedge \forall i \leq n \leq j : M, \tilde{s} \models_k^n \psi
\end{array}
$$

Falls $\tilde{s}$ keine k-Schleife ist, so ist G φ ungültig in der beschränkten Semantik, da nicht sichergestellt werden kann, dass φ in s_{k+1} weiterhin gilt. Aus dem selben Grund muss aus der Definition des R-Operators der Fall ausgeschlossen werden, bei dem ψ immer erfüllt ist, aber φ niemals. Diese Beschränkungen heben auch die Dualität der Operatoren G und F ($\neg \mathrm{F}\, \varphi = \mathrm{G}\, \neg\varphi$) sowie U und R ($\neg(\varphi \,\mathrm{U}\, \psi) = (\neg\varphi) \,\mathrm{R}\, (\neg\psi)$) auf. Der Operator F ergibt sich aus der Definition des U-Operators zu $M, \tilde{s} \models_k^i \mathrm{F}\, \varphi \Leftrightarrow \exists i \leq j \leq k : M, \tilde{s} \models_k^j \varphi$.

Eine zentrale Frage ist, wie mittels beschränkter Modellprüfung ($M \models_k \mathrm{E}\, \varphi$) eine unbeschränkte Modellprüfung ($M \models \mathrm{E}\, \varphi$) durchgeführt werden kann. In [49] beweisen Biere et al. hierzu das folgende Lemma:

Lemma 5.3.1. *Sei φ eine LTL-Formel und M eine temporale Struktur. Dann gilt $M \models \mathrm{E}\, \varphi$, genau dann, wenn $\exists k \in \mathbb{N} : M \models_k \mathrm{E}\, \varphi$.*

Hierdurch wird gezeigt, dass die SAT-basierte Modellprüfung zu einer vollständigen Verifikationsmethode wird, sofern die Schranke k groß genug gewählt wird. Somit lässt sich die SAT-basierte Modellprüfung auch für das Verifikationsziel eines Beweises einsetzen.

Reduktion auf das Boolesche Erfüllbarkeitsproblem

Nachdem im vorangegangenen Abschnitt gezeigt wurde, dass die beschränkte und die unbeschränkte Semantik äquivalent sind, wird nun gezeigt, dass beschränkte (SAT-basierte) Modellprüfung einen wesentlichen Vorteil liefert: Das Problem der beschränkten Modellprüfung lässt sich in Polynomialzeit auf das Boolesche Erfüllbarkeitsproblem reduzieren. Diese Reduktion erlaubt die Nutzung effizienter Entscheidungsstrategien für aussagenlogische Formeln bei der Modellprüfung.

Sei M eine temporale Struktur, φ eine LTL-Formel und k eine gegebene Schranke. Im Folgenden wird eine aussagenlogische Formel $[[M,\varphi]]_k$ konstruiert. Die Variablen $s_0,\ldots,s_k$ in $[[M,\varphi]]_k$ beschreiben einen Sequenz an Zuständen auf dem Pfad $\tilde{s}$. Die Zustände sind symbolisch codiert, d. h. jeder Zustand s_i ist als Vektor von Zustandsvariablen repräsentiert. Die konstruierte Formel $[[M,\varphi]]_k$ drückt Beschränkungen der Zustände $s_0,\ldots,s_k$ aus, so dass $[[M,\varphi]]_k$ genau dann erfüllbar ist, wenn φ auf dem Pfad $\tilde{s}$ gültig ist.

In einem ersten Schritt wird die Formel $[[M]]_k$ konstruiert. $[[M]]_k$ beschränkt alle Pfade $\tilde{s} := \langle s_0,\ldots,s_k\rangle$ derart, dass $\tilde{s}$ einen gültigen Pfad der Länge k in M darstellt:

Definition 5.3.6 (Entfaltung der Übergangsrelation). *Sei $M=(S,R,L)$ eine temporale Struktur mit Anfangszustand s_0 und k eine Schranke, dann kann über die k-fache Entfaltung der Übergangsrelation R die temporale Struktur als aussagenlogische Formel wie folgt dargestellt werden:*

$$[[M]]_k = \sigma_S(s_0) \wedge \bigwedge_{i=0}^{k-1} \Psi_R(\sigma_S(s_i), \sigma_S(s_{i+1}))$$

Dabei ist σ_S die verwendete symbolische Codierung der Zustände und Ψ_R die charakteristische Funktion der Übergangsrelation.

Das Ergebnis ist eine charakteristische Funktion $[[M]]_k$, die alle gültigen Pfade der Länge k in M beschreibt.

In einem zweiten Schritt wird die LTL-Formel φ ebenfalls in eine aussagenlogische Formel übersetzt. Hierbei muss der Fall unterschieden werden, ob ein Pfad eine k-Schleife ist oder nicht. Zunächst wird der Fall betrachtet, dass der Pfad keine k-Schleife ist. Hierbei übersetzt der Operator $[[\cdot]]_k^i$ eine LTL-Formel in eine aussagenlogische Formel. Dabei ist k die Länge des Präfixes des Pfades und i die aktuelle Position im Präfix. Werden die Formeln rekursiv codiert, ändert sich i, während k konstant bleibt.

Definition 5.3.7 (Übersetzung einer LTL-Formeln ohne Schleife). *Sei φ eine LTL-Formel. Sei ferner $k,i \in \mathbb{Z}_{\geq 0}$ mit $i \leq k$. Dann ist*

$$\begin{aligned}
[[\varphi]]_k^i &:= \varphi(s_i),\ \textit{falls}\ \varphi \in V \\
[[\neg\varphi]]_k^i &:= \neg\varphi(s_i),\ \textit{falls}\ \varphi \in V \\
[[\varphi \wedge \psi]]_k^i &:= [[\varphi]]_k^i \wedge [[\psi]]_k^i \\
[[\varphi \vee \psi]]_k^i &:= [[\varphi]]_k^i \vee [[\psi]]_k^i
\end{aligned}$$

$$
\begin{aligned}
[[\mathrm{X}\,\varphi]]_k^i &:= [[\varphi]]_k^{i+1}, \textit{ falls } i < k, \textit{ sonst } \mathsf{F} \\
[[\mathrm{G}\,\varphi]]_k^i &:= \mathsf{F} \\
[[\mathrm{F}\,\varphi]]_k^i &:= \bigvee_{j=i}^{k} [[\varphi]]_k^j \\
[[\varphi\,\mathrm{U}\,\psi]]_k^i &:= \bigvee_{j=i}^{k} \left([[\psi]]_k^j \wedge \bigwedge_{n=i}^{j-1} [[\varphi]]_k^n \right) \\
[[\varphi\,\mathrm{R}\,\psi]]_k^i &:= \bigvee_{j=i}^{k} \left([[\varphi]]_k^j \wedge \bigwedge_{n=i}^{j} [[\psi]]_k^n \right)
\end{aligned}
$$

Man beachte, dass die Codierung konsistent mit der beschränkten Semantik von LTL-Formeln aus Definition 5.3.5 ist.

Nun werden noch diejenigen Pfade betrachtet, die k-Schleifen sind. Die Codierung ${}_l[[\cdot]]_k^i$ hängt sowohl von der Länge des Präfixes k, der momentanen Position i als auch der Position l ab, an der eine Schleife endet. Im Folgenden wird die Funktion $\mathrm{succ}(i)$ verwendet, um den Nachfolger einer (k,l)-Schleife zu bestimmen. Die Funktion $\mathrm{succ}(i)$ ist wie folgt definiert (siehe Abb. 5.22b)):

$$
\mathrm{succ}(i) := \begin{cases} i+1 & \text{falls } i < k \\ l & \text{falls } i = k \end{cases}
$$

Definition 5.3.8 (Codierung einer LTL-Formeln mit Schleife). *Sei φ eine LTL-Formel. Sei ferner $k,l,i \in \mathbb{Z}_{\geq 0}$ mit $i,l \leq k$. Dann ist*

$$
\begin{aligned}
{}_l[[\varphi]]_k^i &:= \varphi(s_i), \textit{ falls } \varphi \in V \\
{}_l[[\neg\varphi]]_k^i &:= \neg\varphi(s_i), \textit{ falls } \varphi \in V \\
{}_l[[\varphi \vee \psi]]_k^i &:= {}_l[[\varphi]]_k^i \vee {}_l[[\psi]]_k^i \\
{}_l[[\varphi \wedge \psi]]_k^i &:= {}_l[[\varphi]]_k^i \wedge {}_l[[\psi]]_k^i \\
{}_l[[\mathrm{X}\,\varphi]]_k^i &:= {}_l[[\varphi]]_k^{\mathrm{succ}(i)} \\
{}_l[[\mathrm{G}\,\varphi]]_k^i &:= \bigwedge_{j=\min\{i,l\}}^{k} {}_l[[\varphi]]_k^j \\
{}_l[[\mathrm{F}\,\varphi]]_k^i &:= \bigvee_{j=\min\{i,l\}}^{k} {}_l[[\varphi]]_k^j \\
{}_l[[\varphi\,\mathrm{U}\,\psi]]_k^i &:= \bigvee_{j=i}^{k} \left({}_l[[\psi]]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[[\varphi]]_k^n \right) \vee \bigvee_{j=l}^{i-1} \left({}_l[[\psi]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[\varphi]]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[[\varphi]]_k^n \right) \\
{}_l[[\varphi\,\mathrm{R}\,\psi]]_k^i &:= \bigwedge_{j=\min\{i,l\}}^{k} {}_l[[\psi]]_k^j \vee \bigvee_{j=i}^{k} \left({}_l[[\varphi]]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[[\psi]]_k^n \right) \vee \\
&\quad \bigvee_{j=l}^{i-1} \left({}_l[[\varphi]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[\psi]]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[[\psi]]_k^n \right)
\end{aligned}
$$

Zur allgemeinen Codierung einer LTL-Formel als aussagenlogische Formel bedarf es jetzt noch der Fallunterscheidung, ob ein Pfad eine Schleife ist oder nicht.

Definition 5.3.9 (Schleifenbedingung). *Für* $k, l \in \mathbb{Z}_{\geq 0}$, *sei* ${}_lL_k := \Psi_R(\sigma_S(s_k), \sigma_S(s_l))$ *und* $L_k := \bigvee_{l=0}^{k} {}_lL_k$.

Damit kann nun die allgemeine Codierung definiert werden:

Definition 5.3.10 (Codierung einer LTL-Formel). *Sei* φ *eine LTL-Formel,* M *eine temporale Struktur und* $k \in \mathbb{Z}_{\geq 0}$. *Die* Codierung *als aussagenlogische Formel lautet:*

$$[[M, \varphi]]_k := [[M]]_k \wedge \left(\left(\neg L_k \wedge [[\varphi]]_k^0\right) \vee \bigvee_{l=0}^{k} \left({}_lL_k \wedge {}_l[[\varphi]]_k^0\right) \right)$$

Der erste Term der Disjunktion $(\neg L_k \wedge [[\varphi]]_k^0)$ beschreibt den Fall, dass keine Schleife vorhanden ist. Die restlichen Terme der Disjunktion probieren alle möglichen Endpunkte l der Schleife aus.

Mit der Codierung der temporalen Struktur und der LTL-Formel kann nun das beschränkte Modellprüfungsproblem auf das Boolesche Erfüllbarkeitsproblem reduziert werden [49]:

Theorem 5.3.1. $[[M, \varphi]]_k$ *ist erfüllbar, genau dann, wenn* $M \models_k \mathrm{E}\, \varphi$.

Hieraus leitet sich ab:

Theorem 5.3.2. $M \models \mathrm{A}\, \neg\varphi$ *gilt, genau dann, wenn* $[[M, \varphi]]_k$ *unerfüllbar ist für alle* $k \in \mathbb{N}$.

Dies wird an einem Beispiel aus [48] verdeutlicht.

Beispiel 5.3.1. Betrachtet wird ein 2-Bit-Zähler. Die temporale Struktur, die den Zähler beschreibt, ist in Abb. 5.23 dargestellt. Jeder der vier Zustände $s_0, \ldots, s_3$ ist durch zwei Zustandsvariablen z_1 und z_0 repräsentiert. Zu Beginn ist der Zähler im Zustand s_0, d. h. $\sigma_S(s_0) = \neg z_1 \wedge \neg z_0$. Die charakteristische Funktion der Übergangsrelation eines korrekten 2-Bit-Zählers ist gegeben durch $\Psi_{inc}(z_1, z_0, z_1', z_0') = (\neg z_1 \wedge \neg z_0 \wedge \neg z_1' \wedge z_0') \vee (\neg z_1 \wedge z_0 \wedge z_1' \wedge \neg z_0') \vee (z_1 \wedge \neg z_0 \wedge z_1' \wedge z_0') \vee (z_1 \wedge z_0 \wedge \neg z_1' \wedge \neg z_0')$ und beschreibt das Inkrementieren des Zählers. In Abb. 5.23 ist allerdings ein fehlerhafter Zustandsübergang enthalten, weshalb die charakteristische Funktion für diesen 2-Bit-Zähler gegeben ist als $\Psi_R(z_1, z_0, z_1', z_0') = \Psi_{inc}(z_1, z_0, z_1', z_0') \vee (z_1 \wedge \neg z_0 \wedge z_1' \wedge \neg z_0')$.

Es soll gezeigt werden, dass der 2-Bit-Zähler auf jedem seiner Berechnungspfade irgendwann den Zustand s_3 mit $\sigma_S(s_3) = z_1 \wedge z_0$ erreicht, d. h. AF $z_1 \wedge z_0$. Diese universelle Modellprüfung kann in die äquivalente existentielle Modellprüfung $\neg$EG $\neg(z_1 \wedge z_0)$ umgeformt werden.

In der SAT-basierten Modellprüfung wird, beginnend mit $k = 0$, ein Gegenbeispiel gesucht. Existiert dieses nicht, muss k inkrementiert oder der Prüfungsversuch aufgegeben werden. Für den Fall $k = 2$ wird zunächst die temporale Struktur in eine aussagenlogische Formel übersetzt:

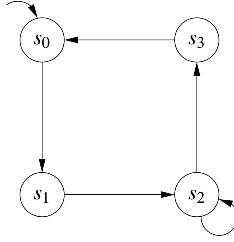


Abb. 5.23. 2-Bit-Zähler mit fehlerhaftem Zustandsübergang [48]

$$\begin{aligned}[[M]]_2 &= \sigma_S(s_3) \wedge \Psi_R(z_1, z_0, z_1', z_0') \wedge \Psi_R(z_1', z_0', z_1'', z_0'') \\ &= (\neg z_1 \wedge \neg z_0) \wedge (\neg z_1 \wedge \neg z_0 \wedge \neg z_1' \wedge z_0') \wedge (\neg z_1' \wedge z_0' \wedge z_1'' \wedge \neg z_0'')\end{aligned}$$

Als nächstes muss mittels der Schleifenbedingung aus Definition 5.3.9 überprüft werden, ob die Pfade eine Schleife enthalten. Es ergibt sich: ${}_0L_2 = \mathsf{F}$, ${}_1L_2 = \mathsf{F}$ und ${}_2L_2 = \mathsf{T}$. Somit ist $L_2 = \mathsf{T}$ und es müssen nach Definition 5.3.10 Codierungen mit Schleife berücksichtigt werden. Die übersetzte LTL-Formel $\mathrm{G}\,\neg(z_1 \wedge z_0)$ ergibt sich zu:

$$\begin{aligned}{}_2[[\mathrm{G}\,\neg(z_1 \wedge z_0)]]_k^0 &= {}_2[[\neg(z_1 \wedge z_0)]]_2^0 \wedge {}_2[[\neg(z_1 \wedge z_0)]]_2^1 \wedge {}_2[[\neg(z_1 \wedge z_0)]]_2^2 \\ &= \neg(z_1 \wedge z_0) \wedge \neg(z_1' \wedge z_0') \wedge \neg(z_1'' \wedge z_0'')\end{aligned}$$

Somit ergibt sich die gesamte aussagenlogische Formel zu:

$$\begin{aligned}[[M, \mathrm{G}\,\neg(z_1 \wedge z_0)]]_2 = ((&\neg z_1 \wedge \neg z_0) \wedge (\neg z_1 \wedge \neg z_0 \wedge \neg z_1' \wedge z_0') \\ &\wedge(\neg z_1' \wedge z_0' \wedge z_1'' \wedge \neg z_0'')) \wedge \mathsf{T} \wedge \neg(z_1 \wedge z_0) \\ &\wedge \neg(z_1' \wedge z_0') \wedge \neg(z_1'' \wedge z_0'')\end{aligned}$$

Diese Formel ist in der Tat erfüllbar: $z_1 = z_0 = z_1' = z_0'' = \mathsf{F}$ und $z_0' = z_1'' = \mathsf{T}$. Dies ist die Codierung für den Präfix $\tilde{s} = \langle s_0, s_1, s_2 \rangle$, der eine $(2,2)$-Schleife ist. Somit ist die gegebene erfüllende Belegung ein Modell für $[[M, \mathrm{G}\,\neg(z_1 \wedge z_0)]]_2$. Dies hat schließlich zur Folge, dass $\tilde{s}$ ein Gegenbeispiel für die ursprüngliche Formel $\mathrm{AF}(z_1 \wedge z_0)$ ist. Es handelt sich hierbei sogar um das kürzeste Gegenbeispiel.

Berechnung der größten Schranke

Oben wurde gezeigt, dass die beschränkte und die unbeschränkte Semantik für LTL-Formeln äquivalent sind, wenn alle denkbaren Schranken berücksichtigt werden. Dies kann direkt in eine Modellprüfungsmethode umgesetzt werden: Um zu zeigen, dass $M \models \mathrm{E}\,\varphi$ ist, kann alternativ auch gezeigt werden, dass $\exists k \in \mathbb{Z}_{\geq 0} : M \models_k \mathrm{E}\,\varphi$. Kann für ein gegebenes k gezeigt werden, dass $M \models_k \mathrm{E}\,\varphi$, so wurde ein Modell für $M \models \mathrm{E}\,\varphi$ gefunden. Ist das Ergebnis allerdings $M \not\models_k \mathrm{E}\,\varphi$, so muss k inkrementiert werden. Somit stellt sich allerdings die Frage, für welche größte Schranke k kann aus $M \not\models_k \mathrm{E}\,\varphi$ geschlossen werden, dass $M \not\models \mathrm{E}\,\varphi$?

Leider ist im allgemeinen Fall nicht zu erwarten, dass eine größte Schranke, die polynomiell von der Anzahl der Zustände der temporalen Struktur und der Größe der LTL-Formel abhängt, existiert. Allerdings besitzen viele temporale Strukturen für praktische Probleme eine bestimmte Form, wobei sich jeder Pfad $\tilde{s}$, der im Anfangszustand s_0 der temporalen Struktur beginnt, als Konkatenation aus rückkopplungsfreiem Pfad $u_{\tilde{s}}$ und Schleife $v_{\tilde{s}}$ beschreiben lässt, d. h. $\tilde{s} = u_{\tilde{s}} v_{\tilde{s}}^{\infty}$. Dabei besitze $u_{\tilde{s}}$ höchstens die Länge $|u_{\tilde{s}}|$ und $v_{\tilde{s}}$ höchstens die Länge $|v_{\tilde{s}}|$. Dann gilt $M \models \mathrm{E}\ \varphi$, genau dann, wenn $\exists k \leq |u_{\tilde{s}}| + |v_{\tilde{s}}| : M \models_k \mathrm{E}\ \varphi$ [49].

5.4 Prüfung nichtfunktionaler Eigenschaften

In der bisherigen Diskussion von Verifikationsmethoden, wurden lediglich Aufgaben der funktionalen Verifikation betrachtet. Daneben spielt insbesondere im Bereich eingebetteter Systeme auch die Überprüfung nichtfunktionaler Eigenschaften eine zentrale Rolle. Dabei sind zeitliche Eigenschaften nahezu immer Bestandteil der zu erfüllenden Anforderungen. Der Grund hierfür liegt darin, dass eingebettete Systeme zum überwiegenden Teil *Echtzeitanforderungen* unterliegen. Diese können entweder sog. *„weiche“* oder *„harte“ Echtzeitanforderungen* sein. Während die Verletzung weicher Echtzeitanforderungen lediglich zu einem Qualitätsverlust in der erbrachten Funktionalität führt, hat die Verletzung harter Echtzeitanforderungen Folgen für die Gefahrlosigkeits- oder Lebendigkeitseigenschaften des Systems. Dies bedeutet aber auch, dass im Fall harter Echtzeitanforderungen, funktionale und nichtfunktionale Eigenschaften nicht mehr getrennt betrachtet werden können.

Für die Verifikation von Echtzeitanforderungen spielt es keine Rolle, ob es sich um harte oder weiche Echtzeitanforderungen handelt. Mit anderen Worten: Die Verifikationsmethoden sind hierzu identisch. Einen Unterschied wird man allerdings bei dem Aufwand der Verifikation von Echtzeitanforderungen feststellen können. So werden bei sicherheitskritischen Systemen mit harten Echtzeitanforderungen im Wesentlichen vollständige Verifikationsmethoden zum Einsatz kommen, während beispielsweise bei Multi-Media-Geräten eine Zeitsimulation häufig ausreichend sein wird, um genügend Vertrauen in die zeitlichen Eigenschaften einer Implementierung zu erlangen.

Die Verifikation des Zeitverhaltens erfolgt, indem zeitliche Eigenschaften des Systems analysiert werden und anschließend mit den nichtfunktionalen Anforderungen aus der Spezifikation verglichen werden. Die Analyse des Zeitverhaltens stellt dabei den komplexeren Teil der Verifikation dar, weshalb im Folgenden wichtige Ansätze zur Zeitanalyse betrachtet werden. Hierbei werden zunächst zeitbehaftete Petri-Netze diskutiert. Anschließend werden Methoden zur Zeitanalyse zeitbehafteter Automaten und zeitbehafteter Datenflussmodelle vorgestellt.

5.4.1 Zeitbehaftete Petri-Netze

Wie in Abschnitt 5.1 beschrieben, kann die funktionale Eigenschaftsprüfung für ein Petri-Netz auf dem Erreichbarkeitsgraphen erfolgen. Dieser repräsentiert im Fall ei-

nes beschränkten Petri-Netzes alle möglichen Markierungen, in denen sich das Petri-Netz befinden kann, sowie die möglichen Übergänge zwischen diesen Markierungen. Im Fall eines unbeschränkten Petri-Netzes stellt der Erreichbarkeitsgraph eine Approximation der erreichbaren Markierungen dar. In beiden Fällen ist der Erreichbarkeitsgraph endlich und Gefahrlosigkeits- und Lebendigkeitseigenschaften lassen sich damit entscheiden. Anders betrachtet handelt es sich bei dem Erreichbarkeitsgraphen um eine temporale Struktur, bei der die erreichbaren Markierungen im Petri-Netz als atomare Aussagen interpretiert werden. Demzufolge können Modellprüfungsverfahren mittels Erreichbarkeitsgraphen durchgeführt werden.

Für die nichtfunktionale Eigenschaftsprüfung wird ein analoger Ansatz verfolgt. Hierbei wird für ein zeitbehaftetes Petri-Netz (siehe Abschnitt 2.2.1) der zugehöriger Erreichbarkeitsgraph konstruiert. Dieser kann wiederum als zeitbehaftete temporale Struktur interpretiert werden. Die Anwendung von Modellprüfungsverfahren auf diese zeitbehaftete Struktur liefert schließlich Aussagen, die zur Zeitanalyse herangezogen werden können.

Der Zustand eines zeitbehafteter Petri-Netzes $G(P,T,F,K,W,M_0,\mathcal{B})$ nach Definition 2.2.11 auf Seite 45, mit Stellen P, Transitionen T, Flussrelationen F, Stellenkapazitäten K, Kantengewichten W, Anfangsmarkierung M_0 und Zeitbeschränkungen $\mathcal{B}$, ist gegeben durch die Markierung M der Stellen in dem zeitbehafteten Petri-Netz sowie die Zeitstruktur $\mathcal{T} : T \to \mathbb{T} \times (\mathbb{T} \cup \{\infty\})$, die jeder aktivierten Transition einen frühesten und einen spätesten Schaltzeitpunkt zuweist. Dabei ist $\mathbb{T}$ die Menge aller möglichen Zeitpunkte, die gleich der Mengen $\mathbb{Z}_{\geq 0}$, $\mathbb{Q}_{\geq 0}$ oder $\mathbb{R}_{\geq 0}$ sein kann. Die so definierte Zustandsmenge eines zeitbehafteten Petri-Netzes ist somit nicht zwangsläufig diskret und eignet sich nur bedingt zur Definition des Erreichbarkeitsgraphen. Deshalb werden zunächst sog. *Zustandsklassen* der Form $(M,\mathcal{D})$ gebildet. Darin beschreibt M die Markierung des zeitbehafteten Petri-Netzes und $\mathcal{D}$ den sog. *Feuerbereich*, der alle möglichen Schaltzeitpunkte der schaltbereiten Transitionen unter M beschreibt.

Definition 5.4.1 (Feuerbereich). *Gegeben sei ein zeitbehaftetes Petri-Netz $G(P,T,F,K,W,M_0,\mathcal{B})$, wobei $\mathcal{B}$ für jede Transition $t \in T$ einen frühesten $\tau_l(t)$ und einen spätesten Schaltzeitpunkt $\tau_u(t)$ relativ zum Aktivierungszeitpunkt von t definiert. Sei M eine Markierung des zeitbehafteten Petri-Netzes und $\tau(t)$ der Schaltzeitpunkt der Transition t unter M, dann lässt sich der* Feuerbereich *wie folgt definieren.*

$$\mathcal{D} := \begin{cases} \tau_l(t_i) \leq \tau(t_i) \leq \tau_u(t_i) & \forall t_i \in T : M[t_i\rangle \\ (\tau_l(t_i) - \tau_u(t_j)) \leq (\tau(t_i) - \tau(t_j)) \leq (\tau_u(t_i) - \tau_l(t_j)) & \forall (i \neq j) : M[t_i\rangle \wedge M[t_j\rangle \end{cases}$$

Mit anderen Worten: Der Feuerbereich beschreibt, dass jede aktivierte Transition innerhalb ihrer Zeitschranken feuern muss und dass die Differenz der Feuerungszeitpunkte von zwei aktivierten Transitionen durch die maximalen Differenzen derer Schranken beschränkt ist. Das Paar $(M_0,\mathcal{D}_0)$ ist dann die Anfangszustandsklasse eines zeitbehafteten Petri-Netzes. Im Folgenden beschreibe $[M_0,\mathcal{D}_0\rangle$ die Menge der von Zustandsklasse $(M_0,\mathcal{D}_0)$ aus erreichbaren Zustandsklassen. Somit kann der *Erreichbarkeitsgraph* für zeitbehaftete Petri-Netze formal definiert werden.

Definition 5.4.2 (Erreichbarkeitsgraph). *Gegeben sei ein zeitbehaftetes Petri-Netz* $G(P,T,F,K,W,M_0,\mathcal{B})$ *mit Erreichbarkeitsmenge* $[M_0,\mathcal{D}_0\rangle$. *Der* Erreichbarkeitsgraph $RG(G) = (V,E,L_v, L_e,v_0)$ *von G besteht aus Knoten* $v \in V$, *welche die erreichbaren Zustandsklassen* $(M,\mathcal{D}) \in [M_0,\mathcal{D}_0\rangle$ *repräsentieren, Kanten* $e \in E$, *die Änderungen der Zustandsklassen durch Schalten einer Transition* $t \in T$ *darstellen, einer Knotenmarkierungsfunktion* $L_v : V \to [M_0,\mathcal{D}_0\rangle$, *die mit jedem Knoten* $v \in V$ *eine erreichbare Zustandsklasse assoziiert, einer Kantenmarkierungsfunktion* $L_e : E \to T$, *die jeder Kante eine Transition zuordnet, und einem Quellknoten* $v_0 \in V$.

Der Erreichbarkeitsgraph eines zeitbehafteten Petri-Netzes wird nach folgenden Regeln konstruiert:

1. Der Knoten v_0 des Erreichbarkeitsgraphen ist mit $L_v(v_0) := (M_0,\mathcal{D}_0)$ markiert. $\mathcal{D}_0$ wird dabei entsprechend Definition 5.4.1 konstruiert.
2. Jeder Knoten $v \in V$ mit Markierung $L_v(v) = (M,\mathcal{D})$ erhält ein wegführende Kante e_{t_0}, falls $t_0 \in T$ in dieser Zustandsklasse schaltbereit ist, d. h.

$$(M,\mathcal{D})[t_0\rangle$$

gilt, und es eine Lösung gibt zu dem Ungleichungssystem $\mathcal{D}$ erweitert um:

$$\forall t_i \in T, (M,\mathcal{D})[t_i\rangle : \tau(t_0) \leq \tau(t_i)$$

3. Für jede wegführende Kante e_{t_0} aus Knoten v mit Markierung $(M,\mathcal{D})$ gibt es einen Endknoten v' mit Markierung $L_n(v') := (M',\mathcal{D}')$, also der aus M folgenden Markierung unter Schalten von t_0 ($M[t_0\rangle M'$) und einem Ungleichungssystem $\mathcal{D}'$. Das Ungleichungssystem $\mathcal{D}'$ wird wie folgt berechnet:
   a) Erweiterung des Ungleichungssystems $\mathcal{D}$:

$$\widetilde{\mathcal{D}} := \begin{cases} \mathcal{D} \\ \tau(t_0) \leq \tau(t_i) \ \forall t_i \in T : (M,\mathcal{D})[t_i\rangle \end{cases}$$

   b) Substitution der Variablen $\tau(t_i)$ durch $\tau(t_i) - \tau_0$ in $\widetilde{\mathcal{D}}$. Darin bezeichnet τ_0 den relativen zeitlichen Nullpunkt.
   c) Elimination von τ_0 aus $\widetilde{\mathcal{D}}$, z. B. durch Projektion des Polytops in Richtung der zu eliminierenden Variablen (Fourier-Motzkin-Elimination), und anschließendes Setzen von $\tau(t_0) := 0$.
   d) Elimination der Variablen $\tau(t_i)$ aus $\widetilde{\mathcal{D}}$ für jede neu aktivierte Transition t_i, d. h. $(M',\widetilde{\mathcal{D}})[t_i\rangle \wedge \neg(M,\mathcal{D})[t_i\rangle$.
   e) Erweiterung des Ungleichungssystems $\widetilde{\mathcal{D}}$:

$$\mathcal{D}' := \begin{cases} \widetilde{\mathcal{D}} \\ \tau_l(t_i) \leq \tau(t_i) \leq \tau_u(t_i) \ \forall t_i \in T : (M',\widetilde{\mathcal{D}})[t_i\rangle \wedge \neg(M,\mathcal{D})[t_i\rangle \end{cases}$$

Beispiel 5.4.1. Abbildung 5.24a) zeigt ein zeitbehaftetes Petri-Netz bestehend aus drei Stellen und zwei Transitionen. Die Zeitschranken sind an den Transitionen annotiert. Der Erreichbarkeitsgraph unter Vernachlässigung des Zeitverhaltens ist in Abb. 5.24b) dargestellt.

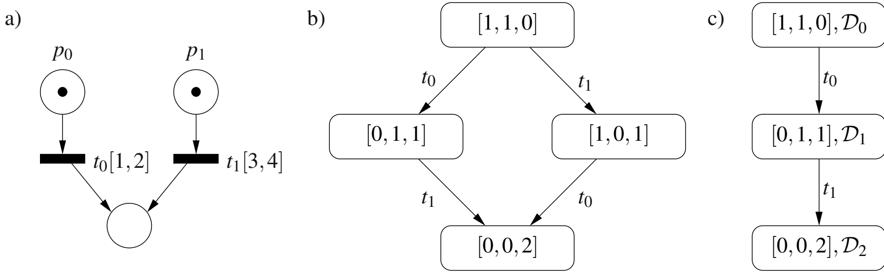


Abb. 5.24. a) zeitbehaftetes Petri-Netz mit b) Erreichbarkeitsgraph ohne und c) mit Zeitbeschränkungen

Die Konstruktion des Erreichbarkeitsgraphen unter Berücksichtigung der Zeitschranken erfolgt in drei Schritten.

1. Der erste Knoten v_0 des Erreichbarkeitsgraphen wird mit der Anfangsmarkierung $M_0 := [1,1,0]$ und dem Feuerbereich $\mathcal{D}_0$ markiert, wobei $\mathcal{D}_0$ wie folgt gegeben ist:
$$\begin{aligned}\mathcal{D}_0 : 1 \leq & \quad \tau(t_0) && \leq 2\\ 3 \leq & \quad \tau(t_1) && \leq 4\\ 1 \leq & \ \tau(t_1) - \tau(t_0) && \leq 3\end{aligned}$$
2. Beide Transitionen t_0 und t_1 sind unter Vernachlässigung der Zeitbeschränkungen schaltbereit. Berücksichtigt man hingegen die Zeitbeschränkungen, ist lediglich t_0 schaltbereit (siehe auch Definition 2.2.12 auf Seite 46), d. h. für das Ungleichungssystem $\mathcal{D}_0$ gibt es mit der Zusatzbedingung $\tau(t_1) \leq \tau(t_0)$ keine Lösung. Der einzige Nachfolgerknoten von v_0 im Erreichbarkeitsgraphen ist somit v_1 mit Markierung $L_v(v_1) = (M_1, \mathcal{D}_1)$, wobei $M_1 = [0,1,1]$ und
$$\begin{aligned}\widetilde{\mathcal{D}}_0 : \quad 1 \leq & \quad \tau(t_0) && \leq 2\\ 3 \leq & \quad \tau(t_1) && \leq 4\\ 1 \leq & \ \tau(t_1) - \tau(t_0) && \leq 3\\ \tau(t_0) \leq & \quad \tau(t_1) &&\end{aligned}$$
ist. Durch Substitution von $\tau(t_i)$ durch $\tau(t_i) - \tau_0$ erhält man:
$$\begin{aligned}\widetilde{\mathcal{D}}_0 : \quad 1 \leq & \ \tau(t_0) - \tau_0 && \leq 2\\ 3 \leq & \ \tau(t_1) - \tau_0 && \leq 4\\ 1 \leq & \ \tau(t_1) - \tau(t_0) && \leq 3\\ \tau(t_0) \leq & \quad \tau(t_1) &&\end{aligned}$$
Zur Elimination von τ_0 können die ersten beiden Ungleichungen wie folgt umgeschrieben werden:
$$\begin{aligned}\tau_0 &\leq \tau(t_0) - 1\\ \tau_0 &\geq \tau(t_0) - 2\\ \tau_0 &\leq \tau(t_1) - 3\\ \tau_0 &\geq \tau(t_1) - 4\end{aligned}$$

Somit muss jede Lösung des Ungleichungssystems $\widetilde{\mathcal{D}}_0$ die Bedingung

$$\max\{\tau(t_0)-2,\tau(t_1)-4\} \leq \min\{\tau(t_0)-1,\tau(t_1)-3\}$$

erfüllen. Diese Bedingung lässt sich wieder als Ungleichungssystem darstellen, in dem gefordert wird, dass jedes Argument der max-Funktion kleiner gleich jedem Argument der min-Funktion ist. Hiermit ergibt sich $\widetilde{\mathcal{D}}_0$ zu:

$$\begin{aligned}\widetilde{\mathcal{D}}_0 : \tau(t_0)-2 \leq& \quad \tau(t_0)-1\\ \tau(t_0)-2 \leq& \quad \tau(t_1)-3\\ \tau(t_1)-4 \leq& \quad \tau(t_0)-1\\ \tau(t_1)-4 \leq& \quad \tau(t_1)-3\\ 1 \leq& \ \tau(t_1)-\tau(t_0) \leq 3\\ \tau(t_0) \leq& \quad \tau(t_1)\end{aligned}$$

Man beachte, dass $\widetilde{\mathcal{D}}_0$ unabhängig von τ_0 ist. Nun wird $\tau(t_0) := 0$ gesetzt. Das somit vereinfachte Ungleichungssystem ist gleichzeitig der Feuerbereich $\mathcal{D}_1$, da es keine neu aktivierten Transitionen gibt, und sieht wie folgt aus:

$$\mathcal{D}_1 : 1 \leq \tau(t_1) \leq 3$$

3. Für die Zustandsklasse $(M_1,\mathcal{D}_1)$ ist lediglich die Transition t_1 aktiviert. Die Folgezustandsklasse ergibt sich zu $(M_2,\mathcal{D}_2)$ mit $M_2 = [0,0,2]$ und $\mathcal{D}_2 = \emptyset$.

Der Erreichbarkeitsgraph ist in Abb. 5.24c) dargestellt. Man sieht, dass der Erreichbarkeitsgraph mit Zeitbeschränkungen ein Teilgraph des Erreichbarkeitsgraphen ohne Zeitbeschränkungen ist, was immer der Fall ist.

TCTL-Modellprüfung

Der hier erzeugte Erreichbarkeitsgraph kann als zeitbehaftete temporale Struktur nach Definition 2.5.2 auf Seite 89 interpretiert werden, wenn die Übergänge im Erreichbarkeitsgraphen zusätzlich mit den Zeitschranken der Transitionen versehen werden. Auf einer zeitbehafteten temporalen Struktur können nun mittels Modellprüfungsverfahren Zeitanalysen vorgenommen werden. Daher wird das Zeitanalyseproblem als Erreichbarkeitsproblem formuliert. Hierfür werden die zu überprüfenden Zeitanforderungen als TCTL-Formeln (siehe Definitionen 2.5.1 auf Seite 89) formuliert. Im Folgenden werden die wichtigsten Funktionen bei der Modellprüfung von TCTL-Formeln vorgestellt. Dabei werden zunächst zwei spezielle Funktionen diskutiert, die nur Zustandsübergänge mit Verzögerungszeit $\delta = 0$ betrachten. Basierend auf diesen speziellen Funktionen können anschließend allgemeinere Zeitaspekte berücksichtigt werden. Schließlich wird gezeigt, wie die minimale Latenz zwischen zwei Zuständen in einer zeitbehafteten temporalen Struktur bestimmt werden kann.

Es werden zeitbehaftete temporale Strukturen $M = (S,R,L)$ mit exakten, diskreten Verzögerungszeiten betrachtet ($\mathbb{T} := \mathbb{Z}_{\geq 0}$). Weiterhin seien S_p und S_q diejenigen

Mengen an Zuständen, in denen p bzw. q gilt. Die Funktion COMPUTE_EU$_{=0}$ berechnet die Zustände, welche die TCTL-Formel E p U$_{=0}$ q erfüllen. Das sind diejenigen Zustände, von denen aus ein Pfad $\tilde{s}$ existiert, der einen Zustand in S_q erreicht und der auf dem Pfad nur Zustände aus S_p enthält. Zusätzlich muss die Latenz Λ dieses Pfades $\Lambda(\tilde{s}) = 0$ sein. Somit kann die Funktion COMPUTE_EU$_{=0}$ wie folgt definiert werden:

COMPUTE_EU$_{=0}$(M, S_p, S_q) {
    $S_N := \emptyset$;
    DO
        $S_R := S_N$;
        $S_A := \{s \in S \mid \exists s' \in S_N : s \xrightarrow{\delta=0} s' \in R\}$;
        $S_N := S_q \cup (S_p \cap S_A)$;
    UNTIL ($S_N = S_R$)
    RETURN S_N;
}

Die Funktion COMPUTE_EG$_{=0}$ berechnet die Zustände, welche die TCTL-Formel EG$_{=0}$ p erfüllen. Das sind diejenigen Zustände, die einen Pfad $\tilde{s}$ besitzen, der lediglich Zustände aus S_p beinhaltet und bei dem alle Zustandsübergänge die Zeit $\delta = 0$ haben. Mit anderen Worten: Es wird die Verfolgung eines Pfades abgebrochen, sobald ein Zustandsübergang mit $\delta > 0$ auftritt.

COMPUTE_EG$_{=0}$(M, S_p) {
    $S_N := S$;
    DO
        $S_R := S_N$;
        $S_A := \{s \in S \mid \exists s' \in S_N : s \xrightarrow{\delta=0} s' \in R\}$;
        $S_N := S_p \cap S_A$;
    UNTIL ($S_N = S_R$)
    RETURN S_N;
}

Mit Hilfe der Funktionen COMPUTE_EU$_{=0}$ und COMPUTE_EG$_{=0}$ lassen sich nun allgemeinere Zeitaspekte überprüfen. Diese lassen sich durch die Formeln E p U$_{[l,u]}$ q und EG$_{[l,u]}$ p formulieren. Dabei beschreiben $[l,u]$ Zeitintervalle innerhalb derer die Latenz liegen muss. Im Folgenden wird lediglich die Funktion COMPUTE_EU$_{[l,u]}$ näher betrachtet, welche diejenigen Zustände $s \in S$ ermittelt, die Präfix eines Pfades $\tilde{s}$ sind, der lediglich mit p markierte Zustände enthält, und dessen letzter Zustand mit q markiert ist. Für die Latenz dieses Pfades $\tilde{s}$ gilt $l \leq \Lambda(\tilde{s}) \leq u$. Die maximale Verzögerungszeit eines Zustandsübergangs sei dabei $\delta_{\max}$.

COMPUTE_EU$_{[l,u]}$(M, S_p, S_q, l, u) {
    $w := u - l + 1$;
    $i := 0$;
    IF (($w \leq 0$) $\vee$ ($u \leq 0$))
        RETURN F;

```
        FOR (j := 1; j ≤ min{w, δ_max}; j := j + 1)
                S_A := COMPUTE_EU_=0(M, S, {s ∈ S | ∃s' ∈ S : s --δ≥j--> s'});
                S_I[j] := COMPUTE_EU_=0(M, S_p, S_q ∩ S_A);
        FOR (j := w + 1; j ≤ δ_max; j := j + 1)
                S_I[j] := ∅;
        S_I[δ_max + 1] := ∅;
        S_N := COMPUTE_EU_=0(M, S_p, S_q);
        S_R := S_N;
        FOR (i := 1; i ≤ u; i := i + 1)
                FOR (j := 1; j ≤ δ_max; j := j + 1)
                        S_A := {s ∈ S | ∃s' ∈ S_R : s --δ=j--> s'};
                        S_I[j] := S_I[j + 1] ∪ COMPUTE_EU_=0(S_p, S_q ∩ S_A);
                S_R := S_I[1];
                IF (i ≤ w)
                        S_R := S_R ∪ S_N;
        RETURN S_R;
}
```

Zunächst wird die Größe w des betrachteten Intervall bestimmt. Ist diese negativ oder ist die obere Schranke negativ, wird die Funktionsberechnung mit einer Fehleranzeige beendet. Anschließend erfolgt die Initialisierung der Mengen $S_I[k]$ mit $1 \leq k \leq \delta_{\max} + 1$. Dabei werden alle Mengen $S_I[k]$ mit $k > w$ als leere Menge initialisiert. Nach jedem Schleifendurchlauf der äußeren FOREACH-Schleife gelten die folgenden Bedingungen:

$$s \in S_I[j] \Leftrightarrow \exists \tilde{s} = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{\delta} s'' \wedge \delta \geq j \wedge M, \tilde{s} \models_\tau \mathrm{E}\, p\, \mathrm{U}_{[i+j-w, i+j-1]}\, q$$

und

$$s \in S_R \Leftrightarrow M, s \models_\tau \mathrm{E}\, p\, \mathrm{U}_{[i+1-w, i]}\, q$$

Die zweite Bedingung garantiert, dass nach allen Schleifendurchläufen ($i = u$) das korrekte Ergebnis ermittelt wurde. Der Beweis hierzu findet sich in [312].

Bestimmung minimaler Latenzen

Für zeitbehaftete temporale Strukturen (engl. *Timed Temporal Structure, TTS*) mit exakten, diskreten Verzögerungszeiten ($\mathbb{T} := \mathbb{Z}_{\geq 0}$) lässt sich die minimale Latenz zwischen Zuständen bestimmen. Dies bildet die Grundlage zur Bestimmung von Antwortzeiten und Durchsatz. Im Folgenden sei $M := (S, R, L)$ eine TTS nach Definition 2.5.2 auf Seite 89 mit exakten Verzögerungszeiten. Die maximale Verzögerungszeit eines Zustandsübergangs sei $\delta_{\max}$. Die *minimale Latenz* zwischen einer Zustandsmenge $S_S \subseteq S$ und einer Zustandsmenge $S_E \subseteq S$ ist diejenige minimale Latenz $\Lambda_{\min} := \Lambda(\tilde{s})$, für die es einen Pfad $\tilde{s} := s_0 \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} s_n$ mit $(s_0 \in S_S) \wedge (s_n \in S_E)$ gibt. Dabei ist $\Lambda(\tilde{s})$ als $\sum_{i:=0}^{n-1} \delta_i$ definiert. Die minimale Latenz $\Lambda_{\min}$ ist ∞, falls kein solcher Pfad existiert. Sie ist null, falls $S_S \cap S_E \neq \emptyset$ gilt.

Die Berechnung der minimalen Latenz kann dann wie folgt durchgeführt werden:

```
TTS_MINIMAL_LATENCY(M, S_S, S_E) {
    n := 0;
    S_I[n] := COMPUTE_EF_=0(M, S_E);
    S_D[n] := S_I[n];
    REPEAT
        IF (S_S ∩ S_D[n] ≠ ∅)
            RETURN n;
        n := n + 1;
        S_A := ⋃_{δ:=0}^{n} {s ∈ S | ∃s' ∈ S_D[n − δ] : s --δ--> s' ∈ R};
        S_D[n] := COMPUTE_EU_=0(M, ¬S_I[n − 1], ¬S_I[n − 1] ∩ S_A);
        S_I[n] := S_I[n − 1] ∪ S_D[n];
    UNTIL (n ≥ δ_max ∧ S_I[n] = S_I[n − δ_max])
    RETURN ∞;
}
```

Im Folgenden seien die Zustände in S_E mit p markiert, d. h. $\forall s \in S_E : L(s) = p$. Bei jedem Eintritt in die REPEAT-Schleife gelten die folgenden Eigenschaften:

$$0 \leq i \leq n \Rightarrow (s \in S_I[i] \Leftrightarrow M, s \models_\tau \text{EF}_{\leq i}\, p) \tag{5.13}$$

und

$$0 \leq i \leq n \Rightarrow (s \in S_D[i] \Leftrightarrow s \in (S_I[i] \setminus S_I[i-1])) \tag{5.14}$$

Formel (5.13) besagt, dass man aus einem Zustand in $S_I[i]$ in höchstens i Zeiteinheiten einen Zustand in S_E erreichen kann. Formel (5.14) besagt, dass man aus einem Zustand in $S_D[i]$ in genau i Zeiteinheiten einen Zustand in S_E erreichen kann.

Die Funktion TTS_MINIMAL_LATENCY gibt die Latenz n zurück, sobald $S_S \cap S_D[n] \neq \emptyset$ gilt. Mit anderen Worten: Sobald festgestellt wird, dass ein Zustand aus der Startmenge einen Zustand in der Zielmenge in n Zeitschritten erreichen kann, wird dieses als minimale Latenz ausgegeben. Mit Formel (5.14) ist sichergestellt, dass es auch keinen Pfad mit kürzerer Latenz gibt. Wenn allerdings $S_I[n] = S_I[n - \delta_{\max}]$ für ein $n \geq \delta_{\max}$ ist, so bedeutet dies nach Formel (5.13), dass kein Pfad aus S_S zu S_E existiert. Andernfalls müsste $|S_I[n]| > |S_I[n - \delta_{\max}]|$ sein.

5.4.2 Zeitbehaftete Automaten

Auch bei zeitbehafteten Automaten lassen sich viele Zeitanalyseprobleme als Erreichbarkeitsproblem auf einer zeitbehafteten temporalen Struktur (TTS) definieren. Wird ein zeitbehafteter Automat $(I, S, s_0, C, \text{inv}, f)$ (siehe Definition 2.2.15 auf Seite 49), mit Zustandsmenge S, als Verhaltensmodell verwendet, so muss für die Erreichbarkeitsanalyse noch die Teilmenge $S_R \subseteq S$ zu erreichender Zustände gegeben sein. Das Erreichbarkeitsproblem besteht dann darin, zu bestimmen, ob mindestens einer der Zustände in S_R innerhalb vorgegebener Zeitschranken erreichbar ist.

Um dieses Problem zu lösen muss zunächst der Zustandsraum eines zeitbehafteten Automaten in Form einer TTS definiert werden: Gegeben sei ein *zeitbehafteter Automat* $(I, S, s_0, C, \text{inv}, f)$, mit dem Eingabealphabet I, der Zustandsmenge S, dem

Anfangszustand s_0, der Menge der Zeitvariablen C, der Funktion $\text{inv} : S \rightarrow \mathcal{B}(C)$, die jedem Zustand $s \in S$ eine Invariante $\beta \in \mathcal{B}(C)$ zuweist, sowie der Übergangsrelation $f \subseteq S \times \mathcal{B}(C) \times I \times 2^C \times S$. Die Beschränkungen $\beta \in \mathcal{B}(C)$ sind konjunktive Formeln über atomare Beschränkungen, also $b_0 \wedge b_1 \cdots \wedge b_n$. Dabei haben die b_i die Form $c \sim \gamma$, mit $c \in C$, $\gamma \in \mathbb{Q}$ und $\sim \in \{\leq, <, =, >, \geq\}$. Der momentane Wert einer Zeitvariablen $c \in C$ ergibt sich zu $\nu(c) \in \mathbb{T}$ mit $\mathbb{T} = \mathbb{R}_{\geq 0}$. Entsprechend beschreibt die Funktion ν die momentanen Werte aller Zeitvariablen.

Der Zustandsraum ist durch eine markierte TTS M mit Zuständen Q_M beschrieben, die über die Ausführungssemantik zeitbehafteter Automaten (siehe Abschnitt 2.2.2) gegeben ist. Die Zustandsänderung aufgrund eines Zeitfortschritts $\delta \in \mathbb{T}$ ist definiert mit:

$$(s, \nu) \xrightarrow{\delta \in \mathbb{T}} (s, \nu') \text{ falls } \forall c \in C, 0 \leq \delta' \leq \delta : \nu(c) + \delta' \models \text{inv}(s)$$

Die neue Zeitstruktur ν' ergibt sich zu $\forall c \in C : \nu'(c) := \nu(c) + \delta$. Eine Zustandsänderung aufgrund einer Eingabe $i \in I$ ist definiert mit:

$$(s, \nu) \xrightarrow{i \in I} (s', \nu') \text{ falls } \nu \models \beta \wedge \nu' \models \text{inv}(s')$$

Der Zustandsübergang ist $s \xrightarrow{\beta, i, \mathcal{R}} s'$, wobei $\mathcal{R}$ die Menge der zurück zu setzenden Zeitvariablen darstellt. Die aktualisierte Zeitstruktur ergibt sich zu:

$$\nu'(c) := \begin{cases} 0 & \text{für } c \in \mathcal{R} \\ \nu(c) & \text{sonst} \end{cases}$$

Der Anfangszustand ist gegeben durch $q_0 := (s_0, \nu_0)$ mit $\forall c \in C : \nu_0(c) := 0$. Die Markierungen der Zustandsübergänge sind dabei der Zeitfortschritt $\delta \in \mathbb{T}$ bzw. die Eingabe $i \in I$.

Regionen

Die oben beschriebene zeitbehaftete temporale Struktur M mit Anfangszustand q_0 besteht für $\mathbb{T} = \mathbb{R}_{\geq 0}$ aus unendlich vielen Zuständen Q_M und unendlich vielen Übergangsbedingungen. Somit ist M für die Anwendung von TCTL-Modellprüfungsverfahren nicht geeignet. Um zu einer endlichen Struktur zu gelangen, wird zunächst eine Abstraktion von der Zeit in den Zustandsübergängen vorgenommen, d. h. es werden lediglich die Eingaben aus dem Eingabealphabet I betrachtet. Eine solche Abstraktion erhält man, wenn man Übergangsfolgen der Form $q \xrightarrow{\delta \in \mathbb{T}} q' \xrightarrow{i \in I} q''$ durch einen einzelnen Zustandsübergang $q \overset{i}{\Longrightarrow} q''$ ersetzt. Hierdurch entsteht eine TTS $\widetilde{M}$ mit einer zu M identischen Zustandsmenge Q_M, d. h. $\widetilde{M}$ besitzt unendlich viele Zustände, allerdings nur noch endlich viele ($|I|$) Zustandsübergangsbedingungen.

Um die Anzahl der Zustände ebenfalls zu beschränken, wird eine stabile Äquivalenzrelation $\sim$ über die Zustandsmenge Q_M definiert. Eine Äquivalenzrelation ist stabil, wenn $\forall q \sim p : (q \overset{i}{\Longrightarrow} q') \Leftrightarrow \exists p' : (p \overset{i}{\Longrightarrow} p') \wedge (q' \sim p')$ gilt.

Sei (I,S,s_0,C,inv,f) ein zeitbehafteter Automat und $\sim$ eine stabile Äquivalenzrelation der Zustandsmenge Q_M. Seien weiterhin $Q_{M,1}$ und $Q_{M,2}$ zwei Äquivalenzklassen aus der Relation $\sim$, so gibt es genau dann einen Zustandsübergang $q \stackrel{i}{\Longrightarrow} q'$ mit $q \in Q_{M,1}$ und $q' \in Q_{M,2}$, wenn $\forall q \in Q_{M,1} : \exists q' \in Q_{M,2} : q \stackrel{i}{\Longrightarrow} q'$. Dies bedeutet, dass bei der Erreichbarkeitsanalyse lediglich die Äquivalenzklassen zu betrachten sind. Dies kann durch die markierte TTS $\widetilde{M}$ beschrieben werden, deren Zustände die Äquivalenzklassen aus der Relation $\sim$ sind. Eine Äquivalenzklasse $Q_{M,k}$ ist Anfangszustand von der TTS $\widetilde{M}$, wenn $q_0 \in Q_{M,k}$ ist. Die Markierung der Zustandsübergänge sind die Symbole $i \in I$ und $\widetilde{M}$ enthält einen Zustandsübergang von Äquivalenzklasse $Q_{M,1}$ zu Äquivalenzklasse $Q_{M,2}$, wenn ein Zustand $q \in Q_{M,1}$ und ein Zustand $q' \in Q_{M,2}$ existiert, für den M einen Übergang $q \stackrel{i}{\Longrightarrow} q'$ besitzt.

Die obige Transformation von M zu $\widetilde{M}$ muss neben der Stabilität der Äquivalenzrelation $\sim$ auch garantieren, dass keine Zustände aus S_R mit Zuständen aus $S \backslash S_R$ zusammengefasst werden, da ansonsten das Erreichbarkeitsproblem für die Zustände in S_R nicht auf der markierten TTS $\widetilde{M}$ gelöst werden kann. Eine Äquivalenzrelation $\sim$ auf Q_M heißt auch *S_R-sensitiv*, wenn für $(s,\nu) \sim (s',\nu')$ entweder $s,s' \in S_R$ oder $s,s' \in S \backslash S_R$ gilt. Dies ist in dem folgenden Lemma beschrieben [7].

Lemma 5.4.1. *Sei (I,S,s_0,C,inv,f) ein zeitbehafteter Automat und $\sim$ eine Äquivalenzrelation der Zustandsmenge Q_M. Sei weiterhin $S_R \subseteq S$, so dass die Relation $\sim$ sowohl stabil als auch S_R-sensitiv ist, dann ist ein Zustand in S_R genau dann erreichbar, wenn eine Äquivalenzklasse $Q_{M,k}$ in der Äquivalenzrelation $\sim$ existiert, so dass $Q_{M,k}$ in der TTS $\widetilde{M}$ erreichbar ist und $Q_{M,k}$ einen Zustand aus S_R enthält.*

Was somit noch benötigt wird, um für eine gegebene Zeitanforderung eine endliche TTS zu erzeugen, ist eine stabile und S_R-sensitive Äquivalenzrelation, welche die Zustandsmenge Q_M beschränkt. Da die zu erreichenden Zustände in S_R während der Äquivalenzklassenbildung einfach zu berücksichtigen sind, wird hier lediglich die Konstruktion einer stabilen Äquivalenzrelation für zeitbehaftete Automaten vorgestellt. Dabei ist zu beachten, dass die Zeitbeschränkungen $\mathcal{B}(C)$ eines zeitbehafteten Automaten diskret sind ($\gamma \in \mathbb{Q}$, durch Multiplikation aller Zeitbeschränkungen mit dem kleinsten gemeinsamen Vielfachen aller Divisoren in den Zeitbeschränkungen, erhält man ganzzahlige Beschränkungen).

Als nächstes werden die Zeitstrukturen $\nu(c) \in \mathbb{R}_{\geq 0}$ für alle Zeitvariablen $c \in C$ wie folgt aufgeteilt: $\lceil \nu(c) \rceil$ ist eine ganze Zahl, die sich durch Aufrunden aus $\nu(c)$ ergibt, während $\{\nu(c)\}$ den nicht ganzzahligen Rest beschreibt. Weiterhin sei $\gamma_{\max}(c)$ die größte Schranke für die Zeitvariable $c \in C$ in einer Zeitbeschränkung $\beta \in \mathcal{B}(C)$ bzw. einer Invariante $\text{inv}(s)$ eines Zustandes s. Wie oben bereits erläutert ist $\gamma_{\max}(c)$ ganzzahlig darstellbar. Mit diesen Festlegungen lässt sich eine Äquivalenzrelation über *Zeitregionen* definieren:

Definition 5.4.3 (Regionen-Äquivalenz). *Seien ν und ν' zwei Bewertungen der Zeitvariablen $c \in C$. Die* Regionen-Äquivalenz *$\nu \cong \nu'$ gilt, genau dann, wenn alle der folgenden Bedingungen erfüllt sind:*

1. $\forall c \in C : (\lceil \nu(c) \rceil = \lceil \nu' \rceil) \vee ((\nu(c) > \gamma_{\max}(c)) \wedge (\nu'(c) > \gamma_{\max}(c)))$

2. $\forall c_1, c_2 \in C : ((\nu(c_1) \leq \gamma_{\max}(c_1)) \wedge (\nu(c_2) \leq \gamma_{\max}(c_2))) \Rightarrow$
   $((\{\nu(c_1)\} \leq \{\nu(c_2)\}) \Leftrightarrow (\{\nu'(c_1)\} \leq \{\nu'(c_2)\}))$
3. $\forall c \in C : (\nu(c) \leq \gamma_{\max}(c)) \Rightarrow ((\{\nu(c)\} = 0) \Leftrightarrow (\{\nu'(c)\} = 0))$

Eine Zeitregion für einen zeitbehafteten Automaten $(I, S, s_0, C, \text{inv}, f)$ ist eine Äquivalenzklasse von Zeitbewertungen aus der Regionen-Äquivalenz $\cong$. Die Zeitregion einer Zeitbewertung ν wird im Folgenden mit $[\nu]$ bezeichnet. Jede Zeitregion kann dabei durch eine endliche Menge an Zeitbeschränkungen charakterisiert werden.

Beispiel 5.4.2. Betrachtet wird eine markierte zeitbehaftete temporale Struktur mit zwei Zeitvariablen c_1 und c_2. Es gilt $\gamma_{\max}(c_1) = 2$ und $\gamma_{\max}(c_2) = 1$. Die sich hieraus ergebenden Zeitregionen sind in Abb. 5.25 dargestellt. Es gibt insgesamt 28 Zeitregionen: Davon sind sechs Eckpunkte (z. B. $[(0,1)]$), 14 Liniensegment (z. B. $[0 < c_1 = c_2 < 1]$) und acht offene Regionen (z. B. $[0 < c_1 < c_2 < 1]$).

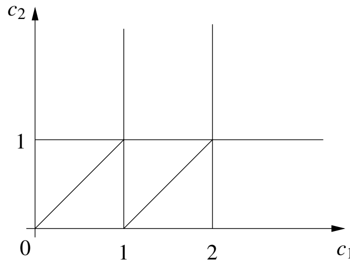


Abb. 5.25. Zeitregionen für zwei Zeitvariablen c_1 und c_2 [7]

An diesem Beispiel kann man bereits erahnen, dass die Anzahl der Zeitregionen für endlich viele Zeitvariablen auch endlich ist. Die maximale Anzahl möglicher Zeitregionen ergibt sich zu $|C|! \cdot 2^{|C|} \cdot \prod_{c \in C}(2 \cdot \gamma_{\max}(c) + 2)$ (siehe z. B. [7]). Weiterhin gilt: Wenn β eine Zeitbeschränkung in $\mathcal{B}(C)$ des zeitbehafteten Automaten und $\nu \cong \nu'$ ist, so gilt $\nu \models \beta$, genau dann, wenn $\nu' \models \beta$. Damit gilt:

$$[\nu] \models \beta \Leftrightarrow \forall \tilde{\nu} \in [\nu] : \tilde{\nu} \models \beta$$

Regionen-Äquivalenz kann auf die Zustandsmenge Q_M erweitert werden, indem für $(q, \nu), (q', \nu') \in Q_M$ gilt: $(q, \nu) \cong (q', \nu') \Leftrightarrow ((q = q') \wedge (\nu \cong \nu'))$. Ein zentrales Ergebnis hierbei ist, dass die Regionen-Äquivalenz $\cong$ eine stabile Äquivalenzrelation auf Q_M ist. Somit kann die Regionen-Äquivalenz verwendet werden, um eine TTS $\widetilde{M}$ mit endlicher Zustandsmenge $Q_{\widetilde{M}}$ für einen zeitbehafteten Automaten zu konstruieren. Die TTS $\widetilde{M}$, die mittels der Regionen-Äquivalenz konstruiert wurde, nennt man *Regionenautomat*.

Beispiel 5.4.3. Gegeben sei der zeitbehaftete Automat in Abb. 5.26. Das Eingabealphabet ist $I := \{i_0, i_1, i_2, i_3\}$ und es gibt zwei Zeitvariablen c_0 und c_1 mit $\gamma_{\max}(c_0) = \gamma_{\max}(c_1) = 1$. Die Zeitbeschränkungen in dem Automaten sorgen dafür, dass der Zustandsübergang von s_2 nach s_3 niemals auftreten kann. Der resultierende Regionenautomat ist in Abb. 5.27 dargestellt. Dabei sind nur die von $(s_0, [0,0])$ erreichbaren Regionen dargestellt. Die einzige erreichbare Zeitregion mit Zustand s_2 erfüllt die Zeitbedingung $[c_1 = 1, c_0 > 1]$. Diese Zeitregion hat keine wegführenden Übergänge.

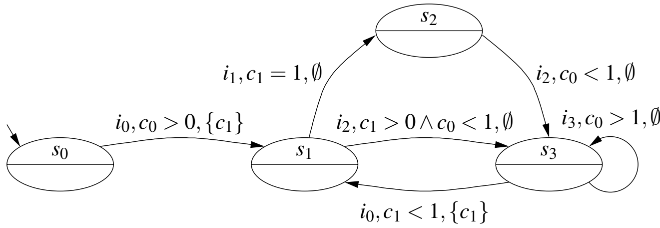


Abb. 5.26. Zeitbehafteter Automat [7]

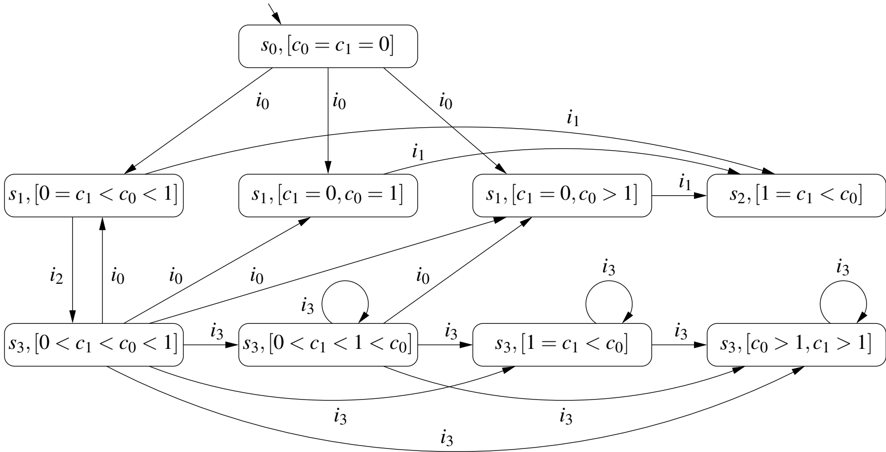


Abb. 5.27. Regionenautomat zu dem zeitbehafteten Automaten aus Abb. 5.26 [7]

Zonen

Der Regionenautomat in Abb. 5.27 startet im Zustand $(s_0, [c_1 = c_1 = 0])$. Von diesem Anfangszustand aus sind drei Folgezustände erreichbar: $(s_1, [c_1 = 0 < c_0 < 1])$,

$(s_1, [c_1 = 0, c_0 = 1])$ und $(s_1, [c_1 = 0, c_0 > 1])$. Um die Zustandsmenge des Regionenautomaten weiter zu reduzieren, könnten diese drei Zustände zu einem Zustand zusammengefasst werden. Eine solche konvexe Vereinigung von Zeitregionen wird als *Zeitzone* bezeichnet. Zeitzonen besitzen die folgenden Eigenschaften:

- Jede Zeitregion $[v]$ ist eine Zeitzone.
- Seien $[v_1]$ eine Zeitregion und $[v_2]$ eine Zeitzone, dann ist entweder $[v_1]$ vollständig in $[v_2]$ enthalten oder deren Schnittmenge ist leer.
- Die Schnittmenge zweier Zeitzonen ist ebenfalls eine Zeitzone.
- Jede Zeitbeschränkung, die als Zeitschranke eines Zustandsübergangs oder in einer Invariante $\text{inv}(s)$ eines Zustands s verwendet wird, ist eine Zeitzone.
- Seien $[v_1]$ und $[v_2]$ zwei Zeitzonen. Ist deren Vereinigungsmenge $[v_1] \cup [v_2]$ konvex, so ist diese ebenfalls eine Zeitzone.

Aus der letzten Eigenschaft folgt, dass sich die Zeitzonen aus der konvexen Vereinigung der Zeitregionen ergeben. Für die Erreichbarkeitsanalyse basierend auf Zeitzonen werden die folgenden Operatoren definiert:

1. Der Schnitt zweier Zeitzonen $[v_1]$ und $[v_2]$ wird als $[v_1] \wedge [v_2]$ geschrieben.
2. Sei $[v]$ eine Zeitzone, dann bezeichnet $[v]^{\uparrow}$ die Zeitbewertungen $v + \delta$ für $v \in [v]$ und $\delta \in \mathbb{R}_{\geq 0}$. Somit beschreibt $[v]^{\uparrow}$ diejenigen Zeitbewertungen, die sich aus den Zeitbewertungen in $[v]$ durch einen Zeitfortschritt δ ergeben.
3. Für eine Teilmenge $\mathcal{R} \subseteq C$ der Zeitvariablen und eine Zeitzone $[v]$ beschreibt $[v]|_{\mathcal{R}:=0}$ diejenigen Zeitbewertungen v', die sich aus $v \in [v]$ wie folgt berechnen lassen:
$$v'(c) := \begin{cases} 0 & \text{falls } c \in \mathcal{R} \\ v(c) & \text{sonst} \end{cases}$$

Eine *Zone* ist ein Paar $(s, [v])$ mit Zustand $s \in S$ und Zeitzone $[v]$. Basierend auf Zonen lässt sich ein markiertes TTS konstruieren: Gegeben sei eine Zone $(s, [v])$ und ein Zustandsübergang $s \xrightarrow{\beta,i,\mathcal{R}} s'$. Sei $\text{succ}([v], \beta, i, \mathcal{R})$ die Menge an Zeitbewertungen v', so dass für ein $v \in [v]$ der Zustand (s', v') von (s, v) durch Zeitfortschritt und Durchführung von $s \xrightarrow{\beta,i,\mathcal{R}} s'$ erreicht werden kann. Somit beschreibt $(s', \text{succ}([v], \beta, i, \mathcal{R}))$ die Nachfolger der Zone $(s, [v])$ unter diesem Zustandsübergang. Die Funktionen succ lässt sich wie folgt berechnen:

$$\text{succ}([v], \beta, i, \mathcal{R}) := \big(([v] \wedge \text{inv}(s))^{\uparrow} \wedge \text{inv}(s) \wedge \beta\big)\big|_{\mathcal{R}:=0}$$

Damit kann die markierte TTS konstruiert werden: Der Anfangszustand ist (s_0, v_0) mit $\forall c \in C : v_0(c) := 0$. Für jeden Zustand $(s, [v])$ der markierten TTS werden für alle Übergänge $s \xrightarrow{\beta,i,\mathcal{R}} s'$ die Folgezustände $(s', \text{succ}([v], \beta, i, \mathcal{R}))$ bestimmt und mit Übergängen $(s, [v]) \xrightarrow{i} (s', \text{succ}([v], \beta, i, \mathcal{R}))$ verbunden. Die resultierende markierte TTS wird als *Zonenautomat* bezeichnet.

Beispiel 5.4.4. Betrachtet wird der Regionenautomat in Abb. 5.27. Der entsprechende Zonenautomat ist in Abb. 5.28 dargestellt. Man sieht, dass die Anzahl der Zustände geringer ist als im Fall des Regionenautomaten.

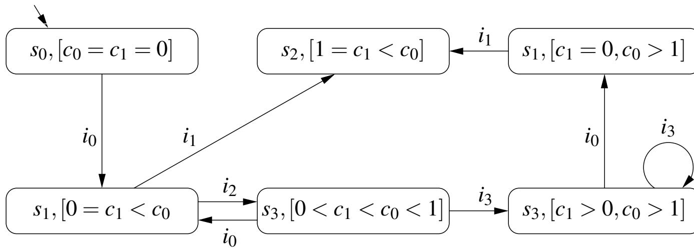


Abb. 5.28. Zonenautomat zu dem zeitbehafteten Automaten aus Abb. 5.26

Simulation zeitbehafteter Automaten

Zeitbehaftete Automaten, bei denen die Zeitbeschränkungen zu exakten Verzögerungszeiten führen, können auch in folgernder Form definiert werden:

Definition 5.4.4 (Zeitbehafteter Automat). *Ein* zeitbehafteter Automat *ist ein* 6-*Tupel* $(I, S, s_0, \Gamma, f, \mathcal{V})$*:*

- I *ist das* Eingabealphabet*,*
- S *ist eine endliche nichtleere Menge von* Zuständen*,*
- $s_0 \in S$ *ist der* Anfangszustand*,*
- $\Gamma(s) \subseteq I$ *ist die Menge zulässiger Eingaben im Zustand s,*
- $f : S \times I \to S$ *ist die* Übergangsfunktion *und*
- $\mathcal{V}$ *ist eine* Zeitstruktur.

Die Zeitstruktur $\mathcal{V} := \{v_i \mid i \in I\}$ besteht aus Zeitsequenzen $v_i := \langle v_{i,1}, v_{i,2}, \dots \rangle$ mit $v_{i,k} \in \mathbb{T}$.

Der Zustand eines zeitbehafteten Automaten mit exakten Verzögerungszeiten ist gegeben durch $(s, \mathcal{N}, \mathcal{T})$ mit Zustand $s \in S$, einer Funktion $\mathcal{N} : I \to \mathbb{Z}_{\geq 0}$ und einer Funktion $\mathcal{T} : I \times \mathbb{N} \to \mathbb{T}$. Die Funktion $\mathcal{N}$ repräsentiert die sog. *Ereigniszähler*, die für jedes Ereignis $i \in I$ angeben, zum wievielten Mal dieses bei der nächsten Aktivierung auftritt. Die Funktion $\mathcal{T}(i,k)$ gibt die Verzögerungszeit des Ereignisses i beim k-ten Auftreten an. Im Folgenden gilt $\mathbb{T} = \mathbb{R}_{\geq 0}$.

Die Semantik zeitbehafteter Automaten wird am einfachsten anhand der Simulation dieser Modelle erläutert: Ereignisse, die in der Zukunft auftreten, werden in einer geordneten Liste L verwaltet. Jedem Ereignis i ist eine Zeitvariable $\tau(i)$ mit Wert größer oder gleich 0 zugeordnet. Die Zeitvariablen werden während der Simulation dekrementiert. Erreicht eine der Zeitvariablen den Wert 0, so tritt das entsprechende Ereignis auf. Das Setzen der Zeitvariablen wird durch die Funktion $\Gamma(s)$ (Menge der in Zustand s zulässigen Ereignisse) gesteuert. Diese Zusammenhänge sind in Abb. 5.29 zusammengefasst.

Die Simulation erfolgt im Wesentlichen in sechs Schritten:

1. *Initialisierung*: Während der Initialisierung wird die Ereignisliste initialisiert ($L := \{\}$), die Anfangszeit $\tau := 0$ und der Anfangszustand $s := s_0$ gesetzt, sowie alle Ereigniszähler initialisiert ($\forall i \in I : \mathcal{N}(i) := 1$).

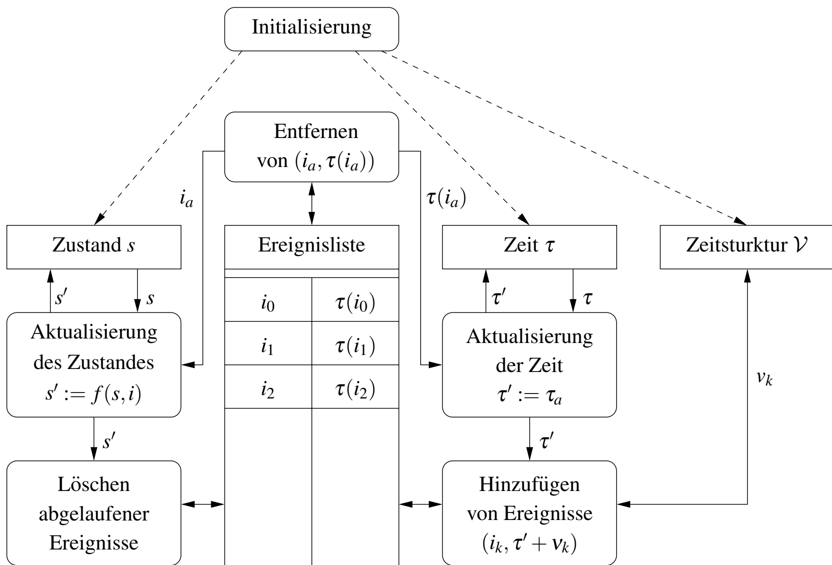


Abb. 5.29. Die Ereignisplanung bei der Simulation zeitbehafteter Automaten [87]

2. *Hinzufügen von Ereignissen*: Alle Ereignisse, die noch nicht in der Liste L enthalten sind, werden dieser hinzugefügt ($L := L \cup \{(i_k, \tau + v_{k,\mathcal{N}(i_k)}) \mid i_k \notin L\}$) und die entsprechenden Ereigniszähler inkrementiert ($\mathcal{N}(i_k) := \mathcal{N}(i_k) + 1$).
3. *Beendigung der Simulation*: Wenn $L = \{\}$, dann ist die Simulation beendet.
4. *Bestimmung des aktuellen Ereignisses*: Zunächst wird der nächste zu simulierende Zeitpunkt $\tau := \min\{\tau(i_k) \mid (i_k, \tau(i_k)) \in L\}$ und ein zugehöriges Ereignis $i_a := \tilde{i}$, wobei $\tilde{i} \in \{i_k \mid \tau(i_k) = \tau\}$ gilt, bestimmt. Die Auswahl des aktuellen Ereignisses erfolgt nichtdeterministisch. Nun muss allerdings überprüft werden, ob das Ereignis i_a zulässig im augenblicklichen Zustand s ist:
   - Falls $i_a \in \Gamma(s)$ ist, dann wird zu 5. gesprungen.
   - Falls $i_a \notin \Gamma(s)$ ist, dann wird der Eintrag $(i_a, \tau(i_a))$ aus L gelöscht ($L := L \backslash \{(i_a, \tau(i_a))\}$), ein neuer Eintrag für dieses Ereignis erzeugt ($L := L \cup \{(i_a, \tau + v_{a,\mathcal{N}(i_a)})\}$) und der zugehörige Ereigniszähler erhöht ($\mathcal{N}(i_a) := \mathcal{N}(i_a) + 1$). Anschließend erfolgt der Rücksprung zu 4.
5. *Durchführung des Zustandsübergangs*: Der Folgezustand ergibt sich aus $s := f(s, i_a)$.
6. *Entfernen der abgelaufenen Ereignisse*: Die neue Ereignisliste ergibt sich zu $L := L \backslash \{(i_k, \tau(i_k)) \mid \tau(i_k) = \tau\}$.
7. *Fortführung der Simulation*: Rücksprung zu 2.

Beispiel 5.4.5. Gegeben ist der zeitbehaftete Automat in Abb. 5.30a) mit den dargestellten Zeitstrukturen, dessen Simulation in Abb. 5.30b) zu sehen ist. Die simulierte Sequenz lautet: $\langle (s_0; 0,0) \xrightarrow{1,0} (s_0; 1,0) \xrightarrow{i_0} (s_1; 1,0) \xrightarrow{1,5} (s_1; 2,5) \xrightarrow{i_0} (s_2; 2,5) \xrightarrow{0,5}$

$(s_2;3,0) \xrightarrow{i_1} (s_0;3,0) \xrightarrow{1,0} (s_0;4,0) \xrightarrow{i_0} (s_1;4,0)\ldots\rangle$. Die simulierten Ereignisse sind als vertikale Pfeile auf der Zeitachse aufgetragen.

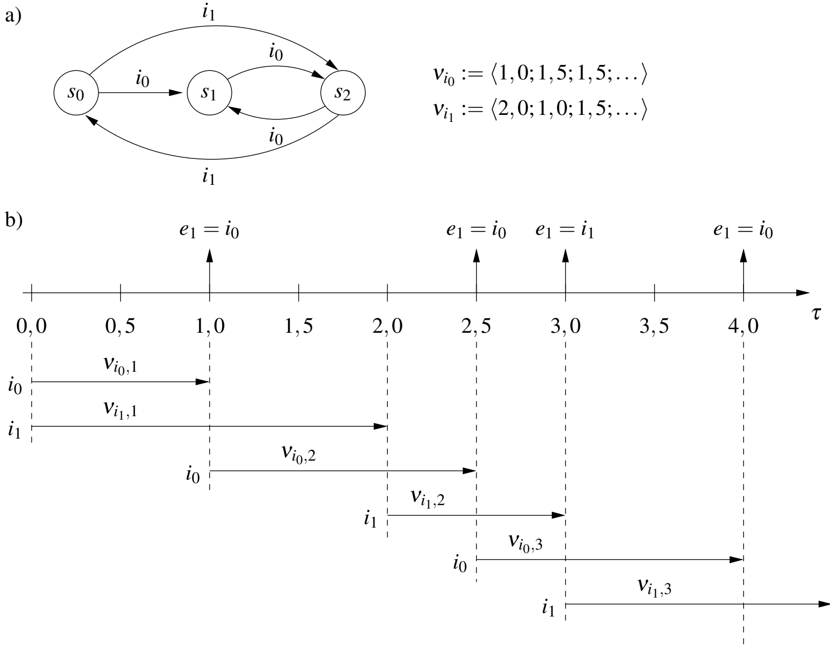


Abb. 5.30. a) zeitbehafteter Automat und b) Simulation des Automaten

5.4.3 Zeitbehaftete SDF-Graphen

Eine Interpretation eingeschränkter Petri-Netze sind *SDF-Graphen* (siehe Abschnitt 2.2.3). Dabei ist das Petri-Netz derart beschränkt, dass jede Stelle genau eine Transition im Vorbereich und genau eine Transition im Nachbereich besitzt, d. h. $\forall p \in P : |\bullet p| = |p \bullet| = 1$. Die Transitionen des Petri-Netzes stellen in diesem Fall die Aktoren des SDF-Graphen dar.

In einem *zeitbehafteten SDF-Graphen* (engl. *Timed SDF, TSDF*) werden konstante exakte Verzögerungszeiten mit den Aktoren assoziiert. Ein zeitbehafteter SDF-Graph kann wie folgt definiert werden:

Definition 5.4.5 (TSDF-Graph). *Ein* zeitbehafteter SDF-Graph *G ist ein 6-Tupel* $(A, C, \mathrm{prod}, \mathrm{cons}, M_0, \delta_{\max})$, *wobei*

- *A die Menge der* Aktoren,
- $C \subseteq A \times A$ *die Menge der* Kanäle *mit FIFO-Semantik,*

- prod : $C \to \mathbb{N}$ *eine Funktion, die jedem Kanal eine* Produktionsrate *zuweist,*
- cons : $C \to \mathbb{N}$ *eine Funktion, die jedem Kanal eine* Konsumptionsrate *zuweist,*
- $M_0 : C \to \mathbb{Z}_{\geq 0}$ *die* Anfangsmarkierung *der Kanäle und*
- $\delta_{\max} : A \to \mathbb{T}$ *eine Funktion, die jedem Aktor* $a \in A$ *eine maximale* Verzögerungszeit $\delta_{\max}(a) \in \mathbb{T}$ *zuweist, ist.*

Als Verzögerungszeit verwendet man die maximale Verzögerungszeit (engl. *Worst Case Execution Times, WCETs*) der durch den Aktor repräsentierten Funktionalität (siehe auch Abschnitt 7.4.1). Dies ist eine Abstraktion, die aufgrund der Monotonie des SDF-Modells garantiert, dass obere Schranken der tatsächlichen Zeiteigenschaften des Systems berechnet werden. Dies gilt sowohl für die Berechnungszeitpunkte (Aktorfeuerungen) als auch Datenproduktionszeitpunkte (Marken).

Beispiel 5.4.6. Gegeben ist der TSDF-Graph $(A, C, \text{prod}, \text{cons}, M_0, \delta_{\max})$ in Abb. 5.31. Der TSDF-Graph besteht aus vier Aktoren $a_0, \ldots, a_3$. Die Kanten stellen die Kanäle $c \in C$ dar. Die Kantenenden sind mit den Produktions- bzw. Konsumptionsraten beschriftet. Die Anfangsmarkierung ist durch die Marken auf den Kanten dargestellt. Die Aktoren im TSDF-Graph haben die Verzögerungszeiten $\delta_{\max}(a_0) := 2$, $\delta_{\max}(a_1) = \delta_{\max}(a_3) := 1$ und $\delta_{\max}(a_2) := 3$.

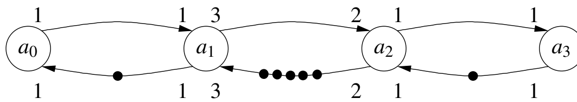


Abb. 5.31. Zeitbehafteter SDF-Graph [185]

Ein TSDF-Aktor $a \in A$ konsumiert von allen seinen eingehenden Kanten $\{c \in C \mid \text{succ}(c) = a\}$ die Anzahl $\text{cons}(c)$ an Marken und produziert auf jeder ausgehenden Kante $\{c \in C \mid \text{pred}(c) = a\}$ die Anzahl $\text{prod}(c)$ an Marken, jeweils am Ende seiner Feuerung. Die Kapazität der Kanäle ist bei SDF-Graphen unendlich. Wie im Fall der zeitbehafteten Petri-Netze, müssen für die Zeitanalyse von TSDF-Graphen die Markenverteilung und die aktuellen Verzögerungszeiten betrachtet werden.

Definition 5.4.6 (Zustand eines TSDF-Graphen). *Sei* $G = (A, C, \text{prod}, \text{cons}, M_0, \delta_{\max})$ *ein TSDF-Graph. Der* Zustand *ist ein Paar* $(M, \mathcal{T})$*, wobei* $M : C \to \mathbb{Z}_{\geq 0}$ *die Markierungsfunktion und* $\mathcal{T} : A \to \mathbb{T}$ *eine Zeitstruktur ist, die jedem Aktor die verbleibende Verzögerungszeit zuordnet.*

Die Markierungsfunktion ordnet jedem Kanal die momentane Markenzahl zu. Eine Markierung M_i dominiert eine andere Markierung M_j, geschrieben als $M_i \succ M_j$ (siehe auch Definition 5.1.2 auf Seite 160), wenn

$$M_i \succ M_j \Leftrightarrow (\forall c \in C : M_i(c) \geq M_j(c)) \wedge (\exists c \in C : M_i(c) > M_j(c)) \; .$$

Die Verzögerungszeiten sind exakt und können entweder diskret ($\mathbb{T} = \mathbb{Z}_{\geq 0}$) oder kontinuierlich sein ($\mathbb{T} = \mathbb{R}_{\geq 0}$). Typischerweise wird bei SDF-Graphen von diskreten Zeitpunkten ausgegangen.

Zur Definition der Zustandsübergänge werden die Anzahl der konsumierten ($M_{\text{cons}}(a)$) und produzierten Marken ($M_{\text{prod}}(a)$) eines Aktors $a \in A$ ebenfalls als Markierungsfunktion beschrieben:

$$M_{\text{cons}}(a) := \{(c,n) \mid \exists c \in C : (\text{succ}(c) = a) \wedge (\text{cons}(c) = n)\}$$
$$M_{\text{prod}}(a) := \{(c,n) \mid \exists c \in C : (\text{pred}(c) = a) \wedge (\text{prod}(c) = n)\}$$

Es ergeben sich drei Arten von Zustandsübergängen:

1. Zustandsübergang durch den Start einer Aktorfeuerung (angezeigt durch e_s) des Aktors $a_0 \in A$.

$$(M,\mathcal{T}) \xrightarrow{(a_0,e_s)} (M,\mathcal{T}'), \text{ falls } (M \succ M_{\text{cons}}(a_0)) \wedge (\mathcal{T}(a_0) = \infty)$$

Die aktualisierte Zeitstruktur $\mathcal{T}'$ ergibt sich zu:

$$\mathcal{T}'(a) := \begin{cases} \mathcal{T}(a) & \text{für } a \neq a_0 \\ \delta_{\max}(a_0) & \text{sonst} \end{cases}$$

2. Zustandsübergang durch Beendigung einer Aktorfeuerung (angezeigt durch e_e) nach Ablauf der Verzögerungszeit des Aktors $a_0 \in A$. Der Endzeitpunkt der Feuerung wird mit $\tau_e(a_0)$ bezeichnet.

$$(M,\mathcal{T}) \xrightarrow{(a_0,e_e)} (M',\mathcal{T}'), \text{ falls } \mathcal{T}(a_0) = 0$$

Die Folgemarkierung M' ergibt sich zu $M' := M - M_{\text{cons}}(a_0) + M_{\text{prod}}(a_0)$. Die aktualisierte Zeitstruktur $\mathcal{T}'$ ergibt sich zu:

$$\mathcal{T}'(a) := \begin{cases} \mathcal{T}(a) & \text{für } a \neq a_0 \\ \infty & \text{sonst} \end{cases}$$

3. Zustandsübergang durch einen Zeitfortschritt δ_{clk}. Um keine Beendigung einer Aktorfeuerung zu verpassen, wird dabei angenommen, dass $\delta_{clk} = 1$ ist, oder δ_{clk} anderweitig geeignet gewählt wurde.

$$(M,\mathcal{T}) \xrightarrow{\delta_{clk}} (M,\mathcal{T}'), \text{ falls } \forall a \in A : \mathcal{T}(a) \neq 0$$

Die aktualisierte Zeitstruktur $\mathcal{T}'$ ergibt sich zu:

$$\mathcal{T}'(a) := \mathcal{T}(a) - \delta_{clk}$$

Die Definition der Zustände und Zustandsübergänge in einem zeitbehafteten SDF-Graphen resultiert in einer zeitbehafteten temporalen Struktur (TTS, siehe Definition 2.5.2 auf Seite 89). Der Anfangszustand s_0 ergibt sich für einen gegeben

TSDF-Graphen $G = (A, C, \text{prod}, \text{cons}, M_0, \delta_{\max})$ zu $(M_0, \mathcal{T}_0)$ mit $\forall a \in A : \mathcal{T}_0(a) := \infty$. Die *Ausführung* eines zeitbehafteten SDF-Graphen wird durch einen Pfad $\tilde{s} = s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \cdots \xrightarrow{i_{n-1}} s_n$ beschrieben. Darin stellen die Eingaben $i \in (A \times \{e_s, e_e\}) \cup \{\delta_{clk}\}$ den Start einer Aktorfeuerung eines Aktors $a \in A$ mit (a, e_s), die Beendigung der Aktorfeuerung mit (a, e_e) oder einen Zeitfortschritt δ_{clk} dar. Die Ausführung startet mit der Anfangsmarkierung des zeitbehafteten SDF-Graphen.

Eine *selbstplanende Ausführung* (engl. *self-timed execution*) ist eine Ausführung, bei der Zustandsübergänge aufgrund eines Zeitfortschritts δ_{clk} nur auftreten, wenn kein Zustandsübergang aufgrund des Starts einer Aktorfeuerung möglich ist. Somit wird bei der selbstplanenden Ausführung jeder Aktor so früh wie möglich gefeuert. Damit ist sicher gestellt, dass bei der Zeitanalyse unter Annahme der selbstplanenden Ausführung die bestmögliche Latenz und der bestmögliche Durchsatz berechnet wird.

Beispiel 5.4.7. Abbildung 5.32 zeigt die selbstplanende Ausführung des TSDF-Graphen aus Abb. 5.31. In dieser Darstellung sind die Zustandsübergänge durch Zeitfortschritt explizit dargestellt (mit δ_{clk} beschriftete Kanten). Zustandsübergänge aufgrund des Starts oder der Beendigung einer Aktorfeuerung sind in einem Zustandsübergang zusammengefasst, wenn der Start und die Beendigung zum gleichen Zeitpunkt stattfinden. Ausgewählte Zustände sind in Abb. 5.32 mit gestrichelten Kanten erläutert.

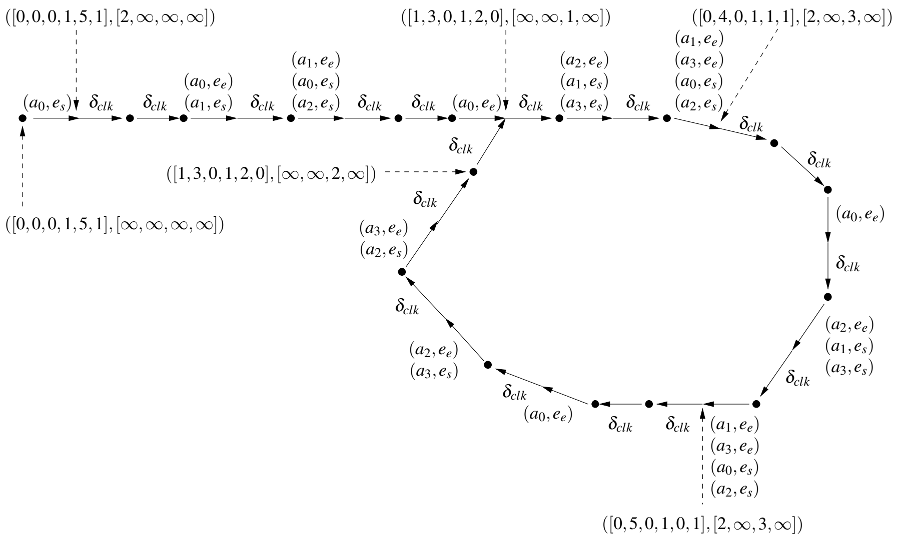


Abb. 5.32. Selbstplanende Ausführung des TSDF-Graphen aus Abb. 5.31 [185]

In Abb. 5.32 erkennt man, dass die Ausführung aus einer transienten Phase zu Beginn und einer periodischen Phase besteht. In [185] zeigen Ghamarian et al., dass dies bei der selbstplanenden Ausführung für jeden konsistenten und stark zusammenhängenden SDF-Graphen gilt. Für konsistente SDF-Graphen existiert ein sog. *Repetitionsvektor* $\gamma : A \to \mathbb{Z}_{\geq 0}$, der jedem Aktor $a \in A$ eine nichtnegative Anzahl an Feuerungen zuordnet, so dass für jeden Kanal $c \in C$ gilt:

$$\text{prod}(c) \cdot \gamma(\text{pred}(c)) = \text{cons}(c) \cdot \gamma(\text{succ}(c)) \tag{5.15}$$

Mit diesen $|C|$ Gleichungen lässt sich der Repetitionsvektor eines SDF-Graphen berechnen.

Definition 5.4.7 (Iteration). *Sei* $G = (A, C, \text{prod}, \text{cons}, M_0, \delta_{\text{max}})$ *ein TSDF-Graph mit Repetitionsvektor* γ*. Eine* Iteration *ist eine Menge an Aktorfeuerung, so dass jeder Aktor* $a \in A$ *exakt* $\gamma(a)$*-mal gefeuert wird.*

Beispiel 5.4.8. Der Repetitionsvektor γ für den zeitbehafteten SDF-Graphen aus Abb. 5.31 berechnet sich zu $\gamma = (2, 2, 3, 3)^\top$. Die periodische Phase in Abb. 5.32 mit Latenz $12 \cdot \delta_{clk}$ stellt eine Iteration dar.

In [185] zeigen Ghamarian et al. weiterhin, dass für jeden konsistenten und stark zusammenhängenden SDF-Graphen die periodische Phase in der selbstplanenden Ausführung ein ganzzahliges Vielfaches einer Iteration ist.

Bestimmung des maximalen Durchsatzes

Eine wichtige Eigenschaft datenflussdominanter Systeme, wie beispielsweise Systeme der digitalen Signalverarbeitung, Netzwerk-Controller und Multi-Media-Systeme, ist der Durchsatz, also diejenige Menge an Daten, die pro Zeiteinheit verarbeitet wird. Somit unterliegt die Implementierung solcher Systeme oftmals einer nichtfunktionalen Anforderung in Form eines minimalen Durchsatzes. Basierend auf der oben beschriebenen zeitbehafteten temporalen Struktur für TSDF-Graphen kann der maximal erzielbare Durchsatz eines TSDF-Graphen bestimmt werden. Dabei handelt es sich um eine Abschätzung des besten Falls. Erfüllt nicht einmal dieser die Anforderung an den minimalen Durchsatz des Systems, muss das System optimiert werden. Zur Zeitanalyse von TSDF-Graphen muss der Durchsatz des Graphen formal definiert werden. Zunächst wird allerdings der Durchsatz eines einzelnen Aktors betrachtet:

Definition 5.4.8 (Aktordurchsatz). *Der* Durchsatz $\Theta(a_0, \tilde{s})$ *eines Aktors* $a_0 \in A$ *in einem TSDF-Graphen mit Ausführung* $\tilde{s}$ *ist die durchschnittliche Anzahl an Feuerungen pro Zeiteinheit. Da die Ausführungen in einem konsistenten und stark zusammenhängenden SDF-Graphen unendlich lange dauert, ist dies als folgender Grenzwert beschrieben:*

$$\Theta(a_0, \tilde{s}) := \lim_{\tau \to \infty} \frac{|\tilde{s}|_{a_0}^{\tau}}{t} \tag{5.16}$$

Darin beschreibt $|\tilde{s}|_{a_0}^{\tau}$ die Anzahl an Zustandsübergängen $s \xrightarrow{(a_0,e_s)} s'$ bis zur Zeit τ. Zur Bestimmung von $|\tilde{s}|_{a_0}^{\tau}$ wird zunächst dasjenige größte k bestimmt, so dass $\Lambda({}^k\tilde{s}) \leq \tau$ gilt. Die Funktion Λ ist für einen Präfix ${}^k\tilde{s} = \langle s_0 \xrightarrow{i_0} s_1 \dots s_{k-1} \xrightarrow{i_{k-1}} s_k \rangle$ der Länge k einer Ausführung $\tilde{s} = \langle s_0 \xrightarrow{i_0} s_1 \dots \rangle$ definiert ist zu:

$$\Lambda({}^k\tilde{s}) := \sum_{j:=0:i_j \in \mathbb{T}}^{k-1} \delta_{clk} \tag{5.17}$$

Es werden also die Zustandsübergänge gezählt, die aufgrund eines Zeitfortschritts stattfinden.

Der maximale Durchsatz eines zeitbehafteten SDF-Graphen ergibt sich bei einer selbstplanenden Ausführung [204]. Somit wird im Folgenden mit $\Theta(a_0)$ der Durchsatz eines Aktors bei selbstplanender Ausführung bezeichnet. Sei weiterhin $\tau_k(a_0)$ der Startzeitpunkt der k-ten Feuerung des Aktors a_0, so kann Gleichung (5.16) wie folgt umgeschrieben werden:

$$\Theta(a_0) = \lim_{k \to \infty} \frac{k}{\tau_k(a_0)} \tag{5.18}$$

Dieser Grenzwert ist gleich der Anzahl an Feuerungen $\gamma_p(a_0)$ des Aktors a_0 pro Zeiteinheit in der periodischen Phase der selbstplanenden Ausführung. Sei $\tilde{s} = \langle s_0 \xrightarrow{i_0} \dots s_p \xrightarrow{i_p} \dots s_q \xrightarrow{i_q} \dots \rangle$ eine selbstplanende Ausführung, wobei s_p der erste Zustand bei Beginn der periodischen Phase und s_q der erste Zustand der 1. Iteration in der periodischen Phase ist. Dann beschreibt ${}^p\tilde{s}^q := \langle s_p \xrightarrow{i_p} \cdots \xrightarrow{i_{q-1}} s_q \rangle$ genau eine Iteration in der selbstplanenden Ausführung. Die Latenz $\Lambda({}^p\tilde{s}^q)$ beschreibt dann die benötigte Zeit für eine Iteration. Der Durchsatz $\Theta(a_0)$ eines Aktors a_0 ergibt sich dann zu:

$$\Theta(a_0) = \frac{\gamma_p(a_0)}{\Lambda({}^p\tilde{s}^q)} \tag{5.19}$$

In [185] zeigen Ghamarian et al., dass folgendes Lemma gilt:

Lemma 5.4.2. *Sei $G = (A, C, \text{prod}, \text{cons}, M_0, \delta_{\max})$ ein konsistenter und stark zusammenhängender TSDF-Graph mit Repetitionsvektor γ und Aktoren $a_1, a_2 \in A$, dann gilt:*

$$\frac{\Theta(a_1)}{\Theta(a_2)} = \frac{\gamma_p(a_1)}{\gamma_p(a_2)} = \frac{\gamma(a_1)}{\gamma(a_2)}$$

Mit anderen Worten: Das Verhältnis des Durchsatzes zweier Aktoren ist gleich dem Verhältnis der Anzahl an deren Feuerungen, bestimmt durch den Repetitionsvektor γ. Mit Gleichung (5.19) und Lemma 5.4.2 kann nun der maximale Durchsatz eines SDF-Graphen definiert werden:

Definition 5.4.9 (TSDF-Durchsatz). *Sei $G = (A, C, \text{prod}, \text{cons}, M_0, \delta_{\max})$ ein konsistenter und stark zusammenhängender TSDF-Graph mit Repetitionsvektor γ und Aktor $a \in A$. Der* maximale Durchsatz *$\Theta(G)$ von G ist definiert zu:*

$$\Theta(G) := \frac{\Theta(a)}{\gamma(a)}$$

Beispiel 5.4.9. Für den TSDF-Graphen in Abb. 5.31 auf Seite 223 ist die selbstplanende Ausführung in Abb. 5.32 dargestellt. Die periodische Phase dauert 12 Zeiteinheiten und besteht aus einer Iteration. Betrachtet man beispielsweise den Aktor a_3, so sieht man, dass dieser insgesamt $\gamma_p(a_3) = \gamma(a_3) = 3$ Mal feuert. Somit ergibt sich der Durchsatz des Aktors zu $\Theta(a_3) = \frac{3}{12} = \frac{1}{4}$. Der Durchsatz des SDF-Graphen G ergibt sich nach Definition 5.4.9 zu: $\Theta(G) = \frac{1}{4} \cdot \frac{1}{3} = \frac{1}{12}$.

Neben dem Durchsatz ist oftmals auch die Latenz eines zeitbehafteten SDF-Graphen von Interesse. Beispielsweise wird es in einem Videokonferenz-System Anforderungen an die Latenz geben, d. h. es ist eine maximale Latenz vorgegeben, um die Qualität der Videokonferenz sicher zu stellen. Die oben beschriebene zeitbehaftete temporale Struktur kann direkt verwendet werden, um Latenzen zwischen Aktorfeuerungen zu bestimmen. Dies wird hier nicht näher betrachtet.

Zeitanalyse basierend auf Max-Plus-Algebra

Der oben beschriebene Ansatz zur Verifikation des Zeitverhaltens von TSDF-Graphen basiert auf der Konstruktion einer zeitbehafteten temporalen Struktur. Damit kann die eigentliche Zeitanalyse als TCTL-Modellprüfung formuliert werden. Ein alternativer Ansatz ergibt sich, wenn anstelle einer zeitbehafteten temporalen Struktur, die den Zustandsraum beschreibt, die Feuerungszeitpunkte der Aktoren erfasst werden. Dies wird als symbolische Simulation des zeitbehafteten SDF-Graphen bezeichnet, die einem ähnlichen Schema wie bei der Simulation zeitbehafteter Automaten folgt. Um die Analyse nachvollziehbarer zu beschreiben, wird im Folgenden davon ausgegangen, dass die Konsumptions- und Produktionsraten der Aktoren konstant 1 sind, d. h. die Analyse wird auf einem markierten Graphen (siehe Abschnitt 2.2.3) durchgeführt.

Der Zeitpunkt an dem der Aktor $a \in A$ zum k-ten Mal feuert sei $\tau_k(a)$. Da eine selbstplanende Ausführung betrachtet wird, kann dieser Zeitpunkt aus den Zeitpunkten $\tau_k(c)$ bestimmt werden, die angeben, wann die k-te Marke auf dem Kanal $c \in C$ erzeugt wurde. Für Marken aus der Anfangsmarkierung ergibt sich dieser Zeitpunkt für alle Kanäle zu $\forall 0 \leq k < |M_0(c)| : \tau_k(c) := 0$, d. h. diese Marken sind von Beginn an verfügbar. Der Aktor a kann gefeuert werden, sobald die letzte benötigte Marke produziert wurde, d. h.:

$$\tau_k(a) := \max_{c \in \{\tilde{c} \in C \mid \mathrm{succ}(\tilde{c}) = a\}} \{\tau_k(c)\} \tag{5.20}$$

Bei Beendigung der Feuerung des Aktors a ($(M, \tau) \xrightarrow{(a,e_e)} (M', \tau)$) produziert dieser die $(k + M_0(c))$-te Marke auf den Kanälen $c \in \{\tilde{c} \in C \mid \mathrm{pred}(\tilde{c}) = a\}$. Somit wird die k-te Marke auf jedem dieser Kanäle für $k \geq M_0(c)$ zu dem Zeitpunkt $\tau_k(c)$ produziert:

$$\tau_k(c) := \tau_{k-M_0(c)}(a) + \delta_{\max}(a) \tag{5.21}$$

Dabei beschreibt $\delta_{\max}(a)$ die maximale Verzögerungszeit des Aktors a.

Die Gleichungen (5.20) und (5.21) können zu einem Gleichungssystem der folgenden Form umgewandelt:

$$\tau_{i,n} = \max_j \{\tau_{j,n-1} + \tau_{i,j}\} \tag{5.22}$$

Dabei beinhalten die $\tau_{i,n}$ u. a. die Feuerungszeitpunkte $\tau_n(a_i)$ aller Aktoren.

Beispiel 5.4.10. Gegeben ist der markierte Graph in Abb. 5.33. Die Verzögerungszeiten der einzelnen Aktoren sind $\delta_{\max}(a_0) := 2$, $\delta_{\max}(a_1) = \delta_{\max}(a_2) := 1$. Hieraus ergeben sich die folgenden Gleichungen:

$$\begin{aligned}
\tau_k(a_0) &= \max\{\tau_k(a_2) + 1\} \\
\tau_k(a_1) &= \max\{\tau_{k-1}(a_0) + 2\} \\
\tau_k(a_2) &= \max\{\tau_{k-1}(a_1) + 1\}
\end{aligned}$$

Der erste Term bedarf einer Umformung, um das Gleichungssystem in die Form von

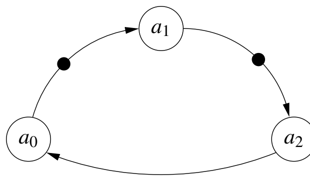


Abb. 5.33. Markierter Graph

Gleichung (5.22) zu bringen. Durch sukzessives Einsetzen des letzten und vorletzten Terms ergibt sich:

$$\begin{aligned}
\tau_k(a_0) &= \max\{\tau_{k-2}(a_2) + 5\} \\
\tau_k(a_1) &= \max\{\tau_{k-1}(a_0) + 2\} \\
\tau_k(a_2) &= \max\{\tau_{k-1}(a_1) + 1\}
\end{aligned}$$

Dies kann mit $\tau_k(a_2') := \tau_{k-1}(a_2) + 4$ wie folgt umgeschrieben werden:

$$\begin{aligned}
\tau_k(a_0) &= \max\{\tau_{k-1}(a_2') + 1\} \\
\tau_k(a_1) &= \max\{\tau_{k-1}(a_0) + 2\} \\
\tau_k(a_2) &= \max\{\tau_{k-1}(a_1) + 1\} \\
\tau_k(a_2') &= \max\{\tau_{k-1}(a_2) + 4\}
\end{aligned}$$

Somit wäre das System in einem Gleichungssystem entsprechend Gleichung (5.22) repräsentiert.

Das Gleichungssystem entsprechend Gleichung (5.22) kann in der sog. *Max-Plus-Algebra* beschrieben und analysiert werden. Die Operationen der Max-Plus-Algebra werden als *Addition* und *Multiplikation* bezeichnet und sind wie folgt definiert:

$$\begin{aligned}\text{Addition: } & a \oplus b := \max\{a,b\} \\ \text{Multiplkation: } & a \otimes b := a+b\end{aligned}$$

Die wichtigsten Gesetze der Max-Plus-Algebra sind:

- *Kommutativität*:

$$\begin{aligned} a \oplus b = \max\{a,b\} = \max\{b,a\} &= b \oplus a \\ a \otimes b = a+b = b+a &= b \otimes a \end{aligned}$$

- *Assoziativität*:

$$\begin{aligned} (a \oplus b) \oplus c = \max\{\max\{a,b\},c\} = \max\{a,\max\{b,c\}\} &= a \oplus (b \oplus c) \\ (a \otimes b) \otimes c = (a+b)+c = a+(b+c) &= a \otimes (b \otimes c) \end{aligned}$$

- *Distributivität*:

$$(a \oplus b) \otimes c = \max\{a,b\}+c = \max\{a+c,b+c\} = (a \otimes c) \oplus (b \otimes c)$$

Das neutrale Element für die Addition in Max-Plus-Algebra ist $-\infty$, da $a \oplus -\infty = \max\{a,-\infty\} = a$ ist. Für die Multiplikation in Max-Plus-Algebra gilt $a \otimes -\infty = a + -\infty = -\infty$. Traditionell verweist der Begriff Max-Plus-Algebra auf den Halbring $(\mathbb{R} \cup \{-\infty\}, \max, +)$. Die obigen Gesetze gelten ebenfalls für den Halbring $(\mathbb{R} \cup \{\infty\}, \min, +)$. Dies bedeutet, dass die Addition $\oplus$ in Max-Plus-Algebra durch die min-Operation und das neutrale Element der Addition durch ∞ definiert ist. Die Multiplikation $\otimes$ ist weiterhin als Addition $+$ definiert. Obwohl es sich hierbei augenscheinlich um eine „Min-Plus-Algebra“ handelt, wird diese trotzdem auch als Max-Plus-Algebra bezeichnet.

Die Max-Plus-Algebra lässt sich ebenfalls auf Matrizen anwenden, wie das folgende Beispiel aus [87] zeigt.

Beispiel 5.4.11.

$$\begin{pmatrix} 1 & 0 \\ 2 & -2 \end{pmatrix} \otimes \begin{pmatrix} 2 & -1 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} \max\{1+2,0+3\} & \max\{1-1,0+1\} \\ \max\{2+2,-2+3\} & \max\{2-1,-2+1\} \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix}$$

Schließlich beschreibe $A^{\otimes^k}$ die k-te Potenzmenge der Matrix A, wobei $A^{\otimes^1} := A$ und $A^{\otimes^k} := A^{\otimes^{k-1}} \otimes A$ ist.

Das Gleichungssystem in Gleichung (5.22) lässt sich in Max-Plus-Algebra als

$$\tau_{i,n} = \bigoplus_j (\tau_{j,n-1} \otimes \tau_{i,j}) \tag{5.23}$$

schreiben. Mit Matrixmultiplikation kann dies wie folgt formuliert werden:

$$\tau_n = X \oplus \tau_{n-1}$$

Der Vektor τ_n besteht aus den Komponenten $\tau_{i,n}$. Die Matrix X enthält die Koeffizienten $\tau_{i,j}$.

Der maximale Durchsatz $\Theta(G)$ für einen stark zusammenhängenden markierten Graphen G ergibt sich dann zu [185]:

$$\Theta(G) := \frac{1}{\lambda} = \frac{1}{P_{\min}}$$

wobei λ der Eigenwert der Matrix X in Max-Plus-Algebra darstellt. Dieser kann als $X \otimes \tau = \lambda \otimes \tau$ berechnet werden (siehe [110]). Dies bedeutet, dass die Feuerungszeitpunkte der nächsten Iteration gleich den Feuerungszeitpunkten der aktuellen Iteration verschoben um eine Konstante λ sind, was gleich dem minimalen Iterationsintervall $P_{\min}$ ist.

Beispiel 5.4.12. Für den markierten Graphen aus Beispiel 5.4.10 ergibt sich die Matrix X zu:

$$\begin{pmatrix} -\infty & -\infty & -\infty & 1 \\ 2 & -\infty & -\infty & -\infty \\ -\infty & 1 & -\infty & -\infty \\ -\infty & -\infty & 4 & -\infty \end{pmatrix}$$

Die Sequenz $\langle X^{\otimes^1}, X^{\otimes^2}, X^{\otimes^3}, \ldots \rangle$ ergibt sich zu:

$$\left\langle \begin{pmatrix} -\infty & -\infty & -\infty & 1 \\ 2 & -\infty & -\infty & -\infty \\ -\infty & 1 & -\infty & -\infty \\ -\infty & -\infty & 4 & -\infty \end{pmatrix}, \begin{pmatrix} -\infty & -\infty & 5 & -\infty \\ -\infty & -\infty & -\infty & 3 \\ 3 & -\infty & -\infty & -\infty \\ -\infty & 4 & -\infty & -\infty \end{pmatrix}, \begin{pmatrix} -\infty & 6 & -\infty & -\infty \\ -\infty & -\infty & 7 & -\infty \\ -\infty & -\infty & -\infty & 4 \\ 7 & -\infty & -\infty & -\infty \end{pmatrix}, \right.$$

$$\left. \begin{pmatrix} 8 & -\infty & -\infty & -\infty \\ -\infty & 8 & -\infty & -\infty \\ -\infty & -\infty & 8 & -\infty \\ -\infty & -\infty & -\infty & 8 \end{pmatrix}, \begin{pmatrix} -\infty & -\infty & -\infty & 9 \\ 10 & -\infty & -\infty & -\infty \\ -\infty & 9 & -\infty & -\infty \\ -\infty & -\infty & 12 & -\infty \end{pmatrix}, \ldots \right\rangle$$

Man beachte, dass $X^{\otimes^5} = 8 \otimes X^{\otimes^1}$ ist. Für $k > 1$ ergibt sich $X^{\otimes^{k+4}} = X^{\otimes^5} \otimes X^{\otimes^{k-1}} = 8 \otimes X^{\otimes^1} \otimes X^{\otimes^{k-1}} = 8 \otimes X^{\otimes^k}$. Hieraus ergibt sich $\lambda^{\otimes^4} = 8$ und somit $P_{\min} = \lambda = 2$.

Das Ergebnis lässt sich auch in folgendem Theorem [162] zusammenfassen:

Theorem 5.4.1. *Gegeben sei ein markierter Graph $G = (A, C, M_0)$ mit Knotenmenge A, Kantenmenge C, Markierungsfunktion $M_0 : C \to \mathbb{Z}_{\geq 0}$ und einer Funktion $\delta_{\max} : A \to \mathbb{Z}_{\geq 0}$, die jedem Aktor $a \in A$ eine Verzögerungszeit $\delta_{\max}(a)$ zuweist. Die* Iterationsintervallschranke $P_{\min}$ *ist gegeben durch:*

$$P_{\min} := \max \left\{ \frac{\sum_{c \in Z} \delta_{\max}(\text{pred}(c))}{\sum_{c \in Z} M(c)} \mid \forall \textit{ gerichteten Zyklen } Z \textit{ von } G \right\}$$

Beispiel 5.4.13. Gegeben sei der markierte Graph in Abb. 5.34. Die Verzögerungszeit der Aktoren sei $\delta_{max}(a_0) = \cdots = \delta_{max}(a_4) := 2$. Der Graph besitzt zwei Zyklen (a_0, a_1, a_2, a_0) und (a_1, a_2, a_3, a_1). Die Iterationsintervallschranke P_{min} ergibt sich zu:

$$P_{min} := \max\left\{\frac{2+2+2}{2}, \frac{2+2+2}{3}\right\} = 3$$

Ein periodischer Ablaufplan mit $P = P_{min}$ ist in Abb. 5.34 zu sehen.

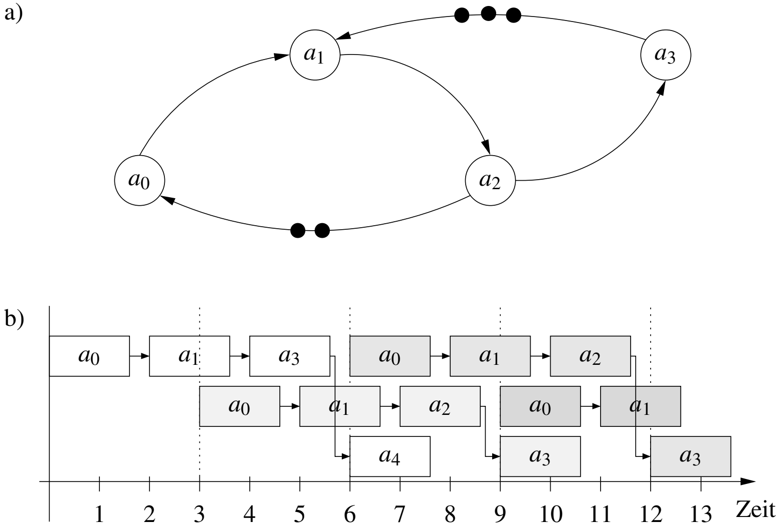


Abb. 5.34. a) markierter Graph und b) periodischer Ablaufplan

5.5 Literaturhinweise

Aufzählende und strukturelle Verfahren zur Verifikation von Petri-Netzen werden z. B. in [87] und [189] vorgestellt. In [189] sind auch Partialordnungsreduktionen und eine Erweiterung auf Modellprüfung für Petri-Netze ausführlich diskutiert. Dabei werden Verfahren unterschieden, die auf der Auswahl von Referenzschaltfolgen basieren sowie auf entfaltungsbasierten Verfahren. Verfahren zur Auswahl von Referenzschaltfolgen sind u. a. die Stubborn-Set- und Sleep-Set-Methoden. Die Stubborn-Set-Methode wurde von Valmari vorgeschlagen [446, 447, 448] Die Sleep-Set-Methode wird ausführlich in [194] diskutiert. Weitere Arbeiten zur Partialordnungsreduktion durch Auswahl von Referenzschaltfolgen sind in [196, 252, 197, 357, 358] zu finden. Andere Verfahren zur Partialordnungsreduktion basierend auf der Entfaltung des Petri-Netzes sind in [155, 317, 318, 265] beschrieben.

Modellprüfung für temporale Logiken sind in [107, 39] ausführlich beschrieben. CTL-Modellprüfung wurde unabhängig von Clarke und Emerson [97] und Queille und Sifakis [369] entwickelt. Der von Clarke und Emerson vorgeschlagene Algorithmus besitzt eine polynomielle Zeitkomplexität in Formellänge und Modellgröße. Einen verbesserten Algorithmus mit linearer Zeitkomplexität stellten Clarke, Emerson und Sistla in [98] vor. Einen ersten Algorithmus zur LTL-Modellprüfung stellten Lichtenstein und Pnueli 1985 in [304] vor. Ein alternativer Ansatz, der auf Büchi-Automaten und Sprachinklusion basiert, wurde von Vardi und Wolper vorgestellt [449, 183].

Zusicherungsbasierte Eigenschaftsprüfung ist ausführlich in [168, 111] beschrieben. Für die simulative Verifikation müssen die Zusicherungen zunächst in Monitore oder Generatoren übersetzt werden. Die frühesten Arbeiten zur Übersetzung von regulären Ausdrücken in Automaten sind [323, 437]. Die Übersetzung von LTL-Formel mittels *Tableau-Technik* wurde erstmals in [467] präsentiert. Aufgrund der Wichtigkeit von LTL-Formeln für die formale und simulative Verifikation wurde diese Methode ständig verbessert [127, 405, 181, 188, 436]. Nach der Standardisierung von PSL wurden auch spezielle Verfahren entwickelt, die PSL-Formeln in Automaten übersetzen [203, 95, 187, 77].

Symbolische CTL-Modellprüfung ist ausführlich in [107, 272] diskutiert. Die grundlegenden Arbeiten für die BDD-basierte Repräsentation von Zustandsmengen in der Modellprüfung finden sich in [74, 72, 316, 73]. Die SAT-basierte Modellprüfung wurde 1999 von Biere et al. [49] vorgestellt. Ihre technische Anwendung ist in [48] beschrieben.

Eine Vielzahl an Verifikationsaufgaben für zeitliche Anforderungen können als Modellprüfungsproblem für zeitbehaftete Modelle formuliert werden. Viele Methoden hierzu wurden im Umfeld von zeitbehafteten Automaten entwickelt [8]. Die Repräsentation und Manipulation der Zeitzonen für die Erreichbarkeitsanalyse bei zeitbehafteten Automaten kann effizient auf sog. engl. *difference bound matrices* durchgeführt werden [132]. Eine einfachere Klasse an Modelle kann direkt durch zeitbehaftete temporale Strukturen definiert werden. Für diese Modelle ist die TCTL-Modellprüfung in polynomieller Zeitkomplexität möglich. Ein erster Algorithmus zur TCTL-Modellprüfung ist in [151] gegeben. Erweiterungen sind in [152] und [288] diskutiert. Die Zeitanalyse von TSDF-Graphen ist ausführlich in [184] beschrieben. Die hier präsentierten Ergebnisse wurden erstmals in [185] und [186] vorgestellt. Dort verwenden Ghamarian et al. bei ihrer Analyse eine Ausführungssemantik für SDF-Graphen, die ein mehrfaches, gleichzeitiges Feuern von Aktoren zulässt. Dies ist eine offensichtliche Semantik, sofern man jegliche Ressourcenbeschränkungen vernachlässigt. Um die in diesem Buch vorgestellte Analyse konsistent zu dem eingeführten zeitbehafteten Petri-Netzen zu halten, wurde diese Semantik hier nicht übernommen. Die Bestimmung des maximalen Durchsatzes basierend auf der Spektralanalyse ist ebenfalls in [185] vorgestellt. Erste Ansätze dieser Methode gehen allerdings auf [375] zurück und wurden später mit dem Eigenwert-Problem der Max-Plus-Algebra [22] verbunden.

6

Hardware-Verifikation

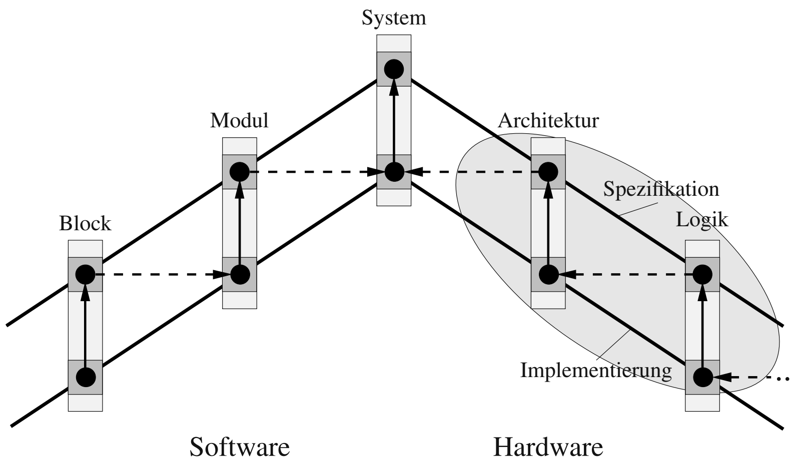


Abb. 6.1. Hardware-Verifikation

In diesem Kapitel werden wichtige Methoden zur Verifikation von Hardware vorgestellt. Diese Methoden sind notwendige Hilfsmittel zur Verifikation auf verschiedenen Abstraktionsebenen. In Abb. 6.1 sind die hier betrachteten Abstraktionsebenen hervorgehoben. Zunächst werden Methoden zur Äquivalenzprüfung von kombinatorischen und sequentiellen Schaltungen beschrieben, die überwiegend auf der Logikebene eingesetzt werden. Anschließend werden Methoden zur Äquivalenzprüfung von arithmetischen Schaltungen präsentiert, mit denen sich Schaltungen auf Architekturebene vergleichen lassen. Ein Spezialfall stellt dabei die Prozessorverifikation dar. Im Anschluss werden Methoden zur Modellprüfung von Hardware auf Logik- und Architekturebene beschrieben. Schließlich folgen Verfahren zur Verifikation des Zeitverhaltens auf Logik- und Architekturebene.

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_6, © Springer-Verlag Berlin Heidelberg 2010

6.1 Äquivalenzprüfung kombinatorischer und sequentieller Schaltungen

Äquivalenzprüfung von Hardware wird sowohl auf der Logik- als auch auf der Architekturebene durchgeführt. Hierbei unterscheidet man verschiedene Arten der Prüfung. Zum einen können zwei Implementierungen auf der selben Abstraktionsebene verglichen werden, zum anderen ist es oft notwendig, die Äquivalenz des Verhaltensmodells einer Spezifikation und des Strukturmodells der zugehörigen Implementierung zu zeigen. Darüber hinaus ist es aber oft wünschenswert, ebenfalls die Äquivalenz zweier Verhaltensmodelle bzw. zweier Strukturmodelle von unterschiedlichen Abstraktionsebenen zu zeigen.

Methoden zur Äquivalenzprüfung werden unterschieden in *implizite*, *explizite*, und *strukturelle Verfahren* (siehe auch Kapitel 4), die im Folgenden zunächst an Hand der Logikebene diskutiert werden. Anschließend werden Methoden für die Architekturebene präsentiert. Diese eignen sich dann sogar zur Äquivalenzprüfung zwischen Logik- und Architekturebene.

6.1.1 Implizite Äquivalenzprüfung auf der Logikebene

Auf der Logikebene (siehe Abb. 6.1) ist das Verhaltensmodell der Spezifikation häufig ein endlicher Zustandsautomat. Die Zustandsübergangsfunktion bzw. Ausgabefunktion kann dann mit Hilfe von Booleschen Funktionen beschrieben werden. Das Strukturmodell der Implementierung ist entweder eine *kombinatorische Schaltung*, auch *Schaltnetz* genannt, bzw. ein *Schaltwerk*. Im Gegensatz zu Schaltwerken enthalten Schaltnetze keine Speicherelemente. Sowohl Schaltnetze als auch Schaltwerke lassen sich in Form von Booleschen Funktionen repräsentieren. Somit lässt sich das grundlegende Problem bei der Äquivalenzprüfung auf Logikebene immer auf die Äquivalenzprüfung von Booleschen Funktionen reduzieren.

Äquivalenzprüfung zweier Implementierungen

Kombinatorische Schaltungen bezeichnen das strukturelle Modell einer Hardware-Implementierung auf der Logikebene ohne Speicher (Flip-Flops, Register etc.). Hierbei handelt es sich um eine Netzliste bestehend aus Logikgattern, wobei Rückkopplung nicht zulässig sind. Die Aufgabe der Äquivalenzprüfung auf Logikebene ist es, zu zeigen, dass zwei gegebene Schaltungen für die selben Eingaben die selben Ausgaben erzeugen. Dann implementieren sie die selbe Funktion. In diesem Fall bezeichnet man beide Schaltungen als *äquivalent*, andernfalls als *nicht äquivalent*.

Eine Möglichkeit, diesen Vergleich zweier kombinatorischer Schaltungen formal durchzuführen, besteht darin, beide Schaltungen durch eine kanonische Repräsentation darzustellen und die Repräsentanten zu vergleichen. Dies wird als *implizite Äquivalenzprüfung* bezeichnet. Da es sich um eine formale Verifikationsmethode handelt, ist sie auch für einen Beweis der Äquivalenz einsetzbar.

Das Verhalten kombinatorischer Schaltungen lässt sich mit Hilfe von Booleschen Funktionen beschreiben. Hierbei werden die Eingänge der Schaltung als Argumente

der Funktion interpretiert. Die Ausgänge der Schaltung stellen die Funktionswerte dar, wobei jeder Ausgang einzeln als eigene Boolesche Funktion dargestellt wird. Das Zeitverhalten, etwa die Verzögerungszeiten der Gatter oder Leitungen, wird bei dieser Abstraktion vernachlässigt.

Die Konstruktion der durch eine kombinatorische Schaltung implementierten Booleschen Funktionen wird auch als *symbolische Simulation* bezeichnet. Prinzipiell existieren hierzu zwei Verfahren, die *Vorwärtskonstruktion* und die *Rückwärtskonstruktion*. Bei der Vorwärtskonstruktion bestimmt man Gatterausgänge als Funktionen der Eingänge, beginnend mit den Schaltungseingängen. Bei der Rückwärtskonstruktion stellt man jeden Schaltungsausgang als Funktion des letzten Gatters dar und bestimmt für dieses die Eingänge aus den direkt davor liegenden Gattern usw. Im Folgenden werden Schaltungen mit nur einem Ausgang betrachtet. Das folgende Beispiel stammt aus [272].

Beispiel 6.1.1. Gegeben ist die kombinatorische Schaltung in Abb. 6.2. Durch Vorwärtskonstruktion ergibt sich die Funktion f_1 zu: $f_1(x_1,x_2) := x_1 \wedge x_2$. Das zweite Gatter ist ein OR-Gatter. Somit ist $f_2(x_1,x_2,x_3) := f_1(x_1,x_2) \vee x_2 \vee x_3 = (x_1 \wedge x_2) \vee x_2 \vee x_3$. Das dritter Gatter ist ein XOR-Gatter und somit $f(x_1,x_2,x_3,x_4) := f_2(x_1,x_2,x_3) \oplus x_4 = ((x_1 \wedge x_2) \vee x_2 \vee x_3) \oplus x_4$.

Die Rückwärtskonstruktion liefert hier das selbe Ergebnis, baut die Boolesche Funktion allerdings von dem Ausgang her auf, d. h. $f(f_2,x_4) := f_2 \oplus x_4$. Durch Substitution der Variablen f_2 mit $f_2 := f_1 \vee x_2 \vee x_3$ erhält man $f(f_1,x_2,x_3,x_4) = (f_1 \vee x_2 \vee x_3) \oplus x_4$. Zuletzt wird noch f_1 mit $f_1 := x_1 \wedge x_2$ substituiert, was zu dem Ergebnis $f(x_1,x_2,x_3,x_4) = ((x_1 \wedge x_2) \vee x_2 \vee x_3) \oplus x_4$ führt.

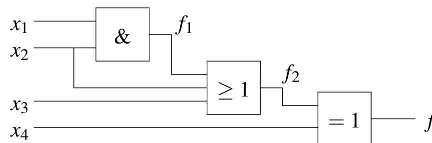


Abb. 6.2. Kombinatorische Schaltung [272]

Für eine implizite Äquivalenzprüfung werden die beiden Booleschen Funktionen, welche die zwei zu vergleichenden kombinatorischen Schaltungen implementieren, mit Hilfe einer kanonischen Repräsentation dargestellt. Kanonische Repräsentationen besitzen die Eigenschaft, dass diese *vollständig* und *eindeutig* sind, d. h. es existiert für jede Boolesche Funktion genau ein Repräsentant. Sind die zwei resultierenden Repräsentanten identisch, so implementieren die beiden kombinatorischen Schaltungen die selbe Boolesche Funktion. *Reduzierte geordnete binäre Entscheidungsdiagramme* (engl. *Reduced Ordered Binary Decision Diagrams, ROBDDs*) und *reduzierte geordnete Kronecker funktionale Entscheidungsdiagramme* (engl. *Reduced Ordered Kronecker Functional Decision Diagrams, ROKFDDs*) sind

solche kanonischen Repräsentationen Boolescher Funktionen (siehe Anhang B.2 und B.3).

Beispiel 6.1.2. Abbildung 6.3 zeigt zwei kombinatorische Schaltungen. Beide Schaltungen unterscheiden sich lediglich in einem Gatter: Während die erste Schaltung in Abb. 6.3a) die Eingänge x_5 und x_6 mit Hilfe eines NAND-Gatters verknüpft, wird in der zweiten Schaltung (Abb. 6.3b)) x_6 negiert. Die Frage, ob beide Schaltungen äquivalent sind, lässt sich formal durch die Konstruktion je eines ROBDD mit der selben Variablenordnung und deren Vergleich beantworten.

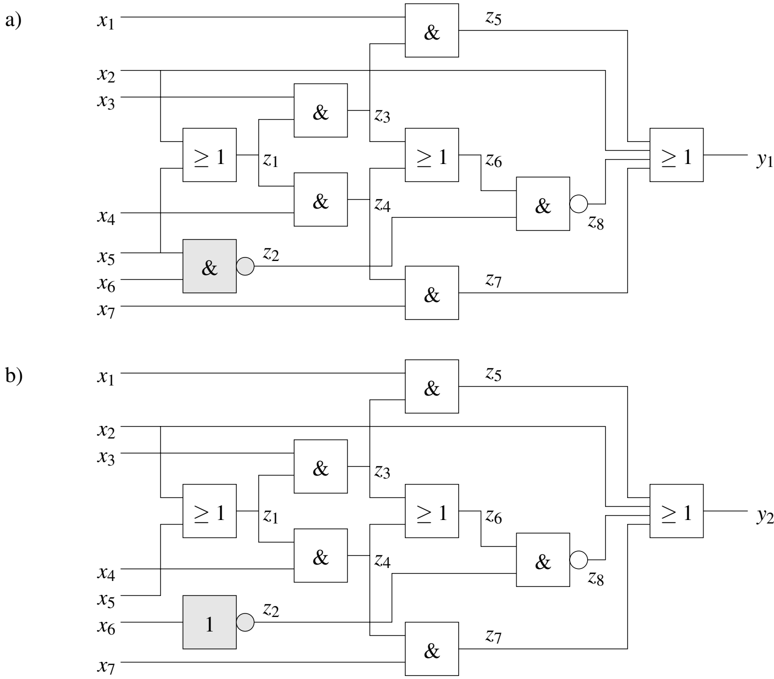


Abb. 6.3. Zwei kombinatorische Schaltungen [275]

Für die kombinatorische Schaltung in Abb. 6.3a) wird mit Hilfe der Vorwärtskonstruktion und des ITE-Operators (siehe Anhang B.2) ein ROBDD konstruiert. Die Variablenordnung ist $x_1 < x_4 < x_3 < x_5 < x_7 < x_6 < x_2$.

1. Im ersten Schritt werden die ROBDDs für die Funktionen der Variablen z_1 und z_2 bestimmt. Zur Konstruktion des ROBDD der Funktion des Ausgangs z_1 (das OR-Gatter mit den Eingängen x_2 und x_5) wird die Operation $\text{ITE}(x_2, \top, x_5)$ verwendet. Wegen der Variablenordnung (x_5 steht vor x_2) muss diese Funktion zunächst nach x_5 entwickelt werden.

$$z_1 = \text{ITE}(x_2, \top, x_5) = \text{ITE}(x_5, \top, x_2)$$

Zur Konstruktion des ROBDD der Funktion des Ausgangs z_2 (das NAND-Gatter mit den Eingängen x_5 und x_6) wird die Operation ITE$(x_5, \neg x_6, \mathsf{T})$ verwendet. Dies ist bereits die richtige Variablenordnung.

$$z_2 = \text{ITE}(x_5, \neg x_6, \mathsf{T})$$

Abbildung 6.4a) und b) zeigen die resultierenden ROBDDs.

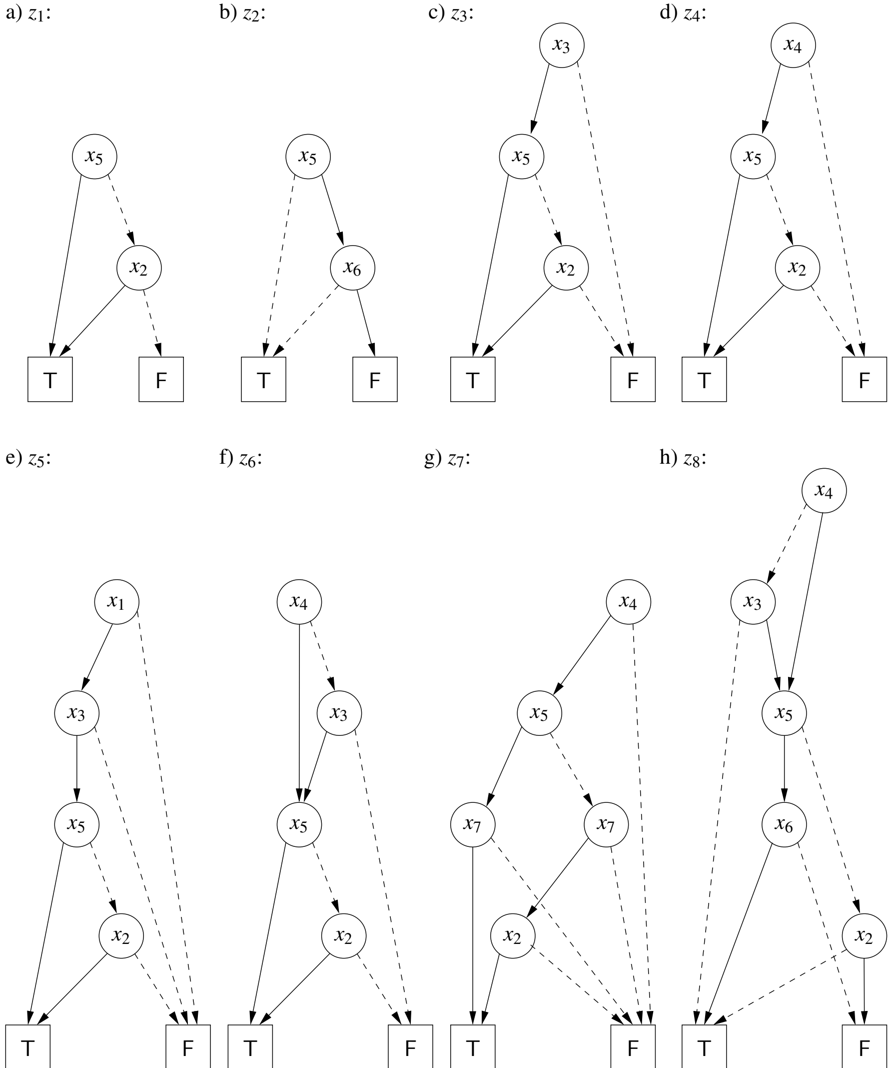


Abb. 6.4. ROBDD-Vorwärtskonstruktion für die Schaltung aus Abb. 6.3a)

2. Im zweiten Schritt werden die Funktionen der Variablen z_3 und z_4 bestimmt. Der Ausdruck für z_3 ergibt sich aus der Definition des ITE-Operators für AND-Gatter.

$$z_3 = \text{ITE}(x_3, z_1, \mathsf{F})$$

Da x_3 in der Variablenordnung vor x_5 und x_2 steht, ist keine weitere Umformung nötig. Das resultierende ROBDD ist in Abb. 6.4c) zu sehen. Die Funktion für z_4 ergibt sich zu $\text{ITE}(z_1, x_4, \mathsf{F})$. Aufgrund der gegebenen Variablenordnung muss zunächst nach x_4 entwickelt werden. Da z_1 unabhängig von der Variablen x_4 ist, bleibt dieser Term unverändert erhalten.

$$\begin{aligned} z_4 &= \text{ITE}(z_1, x_4, \mathsf{F}) \\ &= \text{ITE}(x_4, \text{ITE}(z_1, \mathsf{T}, \mathsf{F}), \text{ITE}(z_1, \mathsf{F}, \mathsf{F})) \\ &= \text{ITE}(x_4, z_1, \mathsf{F}) \end{aligned}$$

Das resultierende ROBDD für z_4 ist in Abb. 6.4d) dargestellt.

3. Im dritten Schritt erfolgt die Repräsentation der Variablen z_5, z_6 und z_7 als ROBDD. Die Konstruktion des ROBDD für die z_5 ergibt:

$$z_5 = \text{ITE}(x_1, z_3, \mathsf{F})$$

Ausgangspunkt ist wiederum die Definition des ITE-Operators für AND-Gatter. Da das ROBDD für die Funktion der Variablen z_3 lediglich die Variablen x_3, x_5 und x_2 enthält (siehe Abb. 6.4c)) und x_1 in der Variablenordnung an erster Stelle steht, ist keine weitere Umformung nötig. Das ROBDD für z_5 ist in Abb. 6.4e) dargestellt. Die Konstruktion des ROBDD für z_6 ergibt:

$$\begin{aligned} z_6 &= \text{ITE}(z_3, \mathsf{T}, z_4) \\ &= \text{ITE}(x_4, \text{ITE}(z_3, \mathsf{T}, z_1), \text{ITE}(z_3, \mathsf{T}, \mathsf{F})) \\ &= \text{ITE}(x_4, \text{ITE}(x_3, \text{ITE}(z_1, \mathsf{T}, z_1), \text{ITE}(\mathsf{F}, \mathsf{T}, z_1)), z_3) \\ &= \text{ITE}(x_4, z_1, z_3) \end{aligned}$$

Ausgangspunkt ist die Definition des ITE-Operators für OR-Gatter. Die höchste Variable in der Variablenordnung, die in den beiden ROBDDs z_3 und z_4 enthalten ist, ist x_4. Deshalb wird zunächst nach x_4 entwickelt. Der für den Fall $x_4 = \mathsf{T}$ verbleibende Term enthält x_3 als höchste Variable in der Variablenordnung. Durch Umformungen können die Ausdrücke auf z_1 und z_3 reduziert werden. Das ROBDD für z_6 ist in Abb. 6.4f) zu sehen. Die Konstruktion des ROBDD für z_7 ergibt:

$$\begin{aligned} z_7 &= \text{ITE}(z_4, x_7, \mathsf{F}) \\ &= \text{ITE}(x_4, \text{ITE}(z_1, x_7, \mathsf{F}), \text{ITE}(\mathsf{F}, x_7, \mathsf{F})) \\ &= \text{ITE}(x_4, \text{ITE}(x_5, \text{ITE}(\mathsf{T}, x_7, \mathsf{F}), \text{ITE}(x_2, x_7, \mathsf{F})), \mathsf{F}) \\ &= \text{ITE}(x_4, \text{ITE}(x_5, x_7, \text{ITE}(x_7, x_2, \mathsf{F})), \mathsf{F}) \end{aligned}$$

Ausgehend von der Definition des ITE-Operators für AND-Gatter werden die ROBDDs für z_4 und x_7 verknüpft. Dieser Ausdruck wird für die Variable x_4 und der für $x_4 = \mathsf{T}$ entstehende Ausdruck nach x_5 (entsprechend der Variablenordnung) entwickelt. Zur endgültigen Konstruktion des ROBDD für z_7 wird dann die Kommutativität der UND-Verknüpfung ausgenutzt ($\text{ITE}(x_2, x_7, \mathsf{F}) = \text{ITE}(x_7, x_2, \mathsf{F})$). Das resultierende ROBDD für z_7 ist in Abb. 6.4g) dargestellt.

4. Im vierten Schritt wird das ROBDD für die Variable z_8 konstruiert.

$$
\begin{aligned}
z_8 &= \text{ITE}(z_6, \neg z_3, \mathsf{T}) \\
&= \text{ITE}(x_4, \text{ITE}(z_1, \neg z_2, \mathsf{T}), \text{ITE}(z_3, \neg z_2, \mathsf{T})) \\
&= \text{ITE}(x_4, \text{ITE}(x_5, \text{ITE}(\mathsf{T}, x_6, \mathsf{T}), \text{ITE}(x_2, \mathsf{F}, \mathsf{T})), \\
&\quad \text{ITE}(x_3, \text{ITE}(z_1, \neg z_2, \mathsf{T}), \text{ITE}(\mathsf{F}, \neg z_2, \mathsf{T}))) \\
&= \text{ITE}(x_4, \text{ITE}(x_5, x_6, \neg x_2), \text{ITE}(x_3, \text{ITE}(x_5, \text{ITE}(\mathsf{T}, x_6, \mathsf{T}), \text{ITE}(x_2, \mathsf{F}, \mathsf{T})), \mathsf{T})) \\
&= \text{ITE}(x_4, \text{ITE}(x_5, x_6, \neg x_2), \text{ITE}(x_3, \text{ITE}(x_5, x_6, \neg x_2), \mathsf{T}))
\end{aligned}
$$

Ausgehend von der Definition des ITE-Operators für NAND-Gatter erfolgt die Entwicklung entsprechend der Variablenordnung. Das Ergebnis ist in Abb. 6.4h) zu sehen.

5. Im letzte Schritt wird das ROBDD für den Ausgang y_1 durch eine ODER-Verknüpfung konstruiert. Der resultierende ROBDD für y_1 ist in Abb. 6.5 dargestellt.

Neben der Vorwärtskonstruktion ist es ebenfalls möglich, das ROBDD vom Ausgang her zu konstruieren. Aufgrund der Konstruktionsregeln sind die Ergebnisse der Vorwärts- und der Rückwärtskonstruktion identisch.

Beispiel 6.1.3. Für die Schaltung aus Abb. 6.3b) soll das ROBDD durch Rückwärtskonstruktion für die Variablenordnung $x_1 < x_4 < x_3 < x_5 < x_7 < x_6 < x_2$ erstellt werden. Es handelt sich hierbei um die selbe Variablenordnung wie in Beispiel 6.1.2.

1. Im ersten Schritt wird das ROBDD für das OR-Gatter am Ausgang y_2 als Funktion $y_2 = z_5 \vee x_2 \vee z_8 \vee z_7$ erstellt.
2. Im zweiten Schritt werden die Variablen z_5, z_7 und z_8 durch die ROBDDs der entsprechenden Gatter (zweimal AND-Gatter, einmal NAND-Gatter) substituiert. Der entsprechende ITE-Operator ist definiert zu $f(x, g(x)) = \text{ITE}(g(x), f(x, \mathsf{T}), f(x, \mathsf{F}))$. Das resultierende ROBDD ist in Abb. 6.6a) zu sehen.
3. Im dritten Schritt werden die Variablen z_2 und z_6 durch die ROBDDs für die Booleschen Funktionen $z_2 = \neg x_6$ und $z_6 = z_3 \vee z_4 = \neg(z_3 \wedge z_4)$ substituiert. Das resultierende ROBDD ist in Abb. 6.6b) dargestellt.
4. Im vierten Schritt werden die Variablen z_3 und z_4 durch die ROBDDs der Funktionen $z_3 = x_3 \wedge z_1$ und $z_4 = z_1 \wedge x_4$ substituiert. Das resultierende ROBDD ist in Abb. 6.6c) dargestellt.
5. Im letzten Schritt wird z_1 durch $x_2 \vee x_5$ substituiert. Hiermit ist die Rückwärtskonstruktion vollständig. Das Ergebnis ist identisch mit der Vorwärtskonstruktion für die Schaltung aus Abb. 6.3a) wie in Abb. 6.5 dargestellt. Da beide

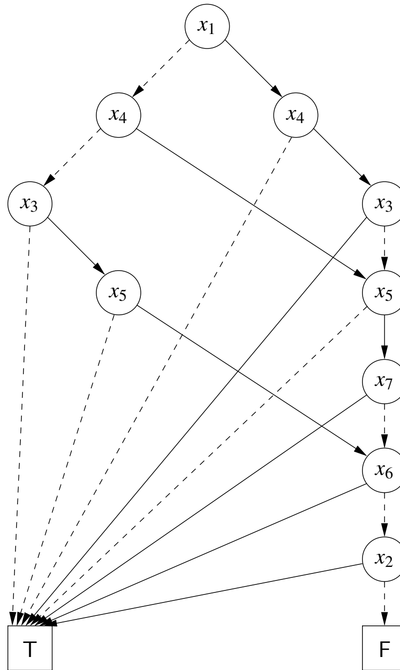


Abb. 6.5. ROBDD für die Ausgänge y_1 und y_2 für die Schaltungen aus Abb. 6.3

Schaltungen bei gleicher Variablenordnung durch das selbe ROBDD repräsentiert werden, sind beide Schaltungen äquivalent.

An den vorangegangenen beiden Beispielen kann man sehen, dass die ROBDDs in den Zwischenschritten bei der Vorwärts- und Rückwärtskonstruktion unterschiedlich groß sein können. Deshalb kann es nützlich sein, beide Verfahren zu kombinieren, um Spitzen im Speicherbedarf während der Konstruktion von ROBDDs zu verhindern [245].

Äquivalenzprüfung zwischen Spezifikation und Implementierung

Neben der Äquivalenzprüfung zweier Implementierungen wird oftmals auch die Äquivalenz zwischen dem Verhaltensmodell der Spezifikation und dem Strukturmodell der Implementierung geprüft. Während Strukturmodelle auf Logikebene durch *kombinatorische Schaltungen* bzw. *Schaltwerke* dargestellt werden, ist das Verhaltensmodell der Spezifikation durch Boolesche Funktionen beschrieben. Da kombinatorische Schaltungen Boolesche Funktionen implementieren, kann man die Schaltungen auch direkt in eine Boolesche Funktion übersetzen und auf Äquivalenz mit der Spezifikation prüfen. Dies kann wiederum *implizit* durch kanonische Repräsen-

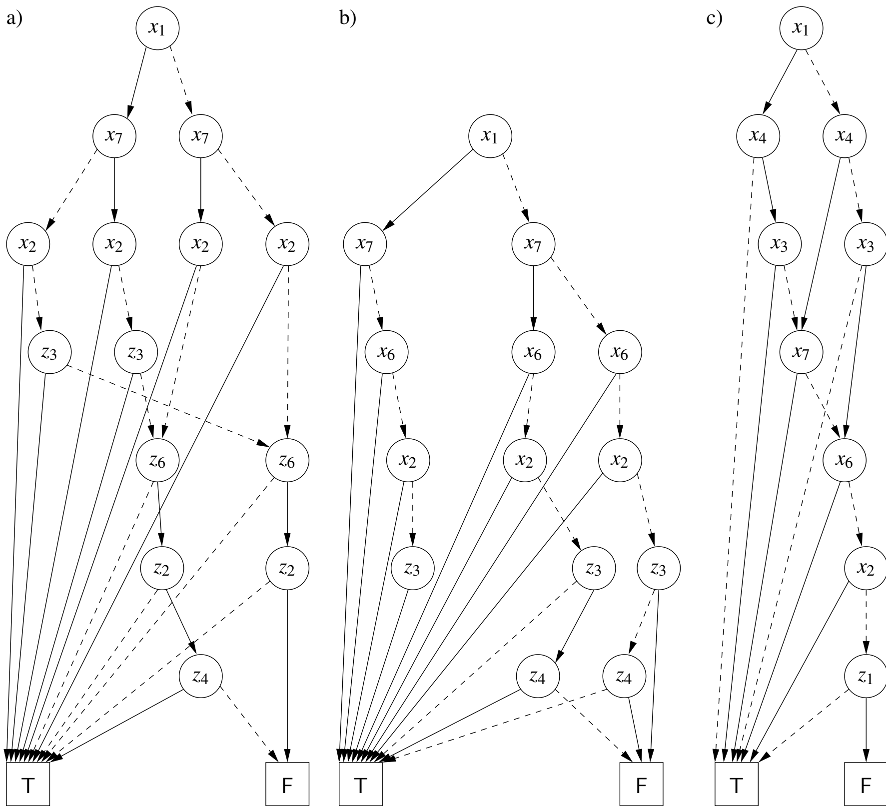


Abb. 6.6. Rückwärtskonstruktion des ROBDD für die Schaltungen aus Abb. 6.3b)

tationen erfolgen. Es handelt sich somit ebenfalls um eine formale Verifikationsmethode.

Beispiel 6.1.4. Gegeben ist die Boolesche Funktion $f = x_2 \vee \neg x_5 \vee x_6 \vee x_5 \wedge ((x_1 \wedge x_3) \vee (x_4 \wedge x_7) \vee (\neg x_3 \wedge \neg x_4))$. Diese dient als Verhaltensmodell der Spezifikation für die kombinatorischen Schaltungen in Abb. 6.3. Um die Äquivalenz der Spezifikation und der Implementierungen zu zeigen, wird ein *reduziertes geordnetes Kronecker funktionales Entscheidungsdiagramm* (engl. *Reduced Ordered Kronecker Functional Decision Diagram, ROKFDD*, siehe Anhang B.3) mit der Variablenordnung $x_2 < x_6 < x_5 < x_3 < x_4 < x_1 < x_7$ erstellt. Für die Variable x_3 wird dabei die *positive*, für die Variable x_4 die *negative Davio-Zerlegung* angewendet. Für alle anderen Variablen wird die *Shannon-Zerlegung* verwendet.

1. Im ersten Schritt wird die Boolesche Funktion f mit Hilfe der Shannon-Zerlegung nach der Variable x_2 entwickelt. Dabei ergibt sich $f|_{x_2:=\mathsf{T}} = \mathsf{T}$ und $f|_{x_2:=\mathsf{F}} = \neg x_5 \vee x_6 \vee x_5 \wedge ((x_1 \wedge x_3) \vee (x_4 \wedge x_7) \vee (\neg x_3 \wedge \neg x_4))$.

2. Im zweiten Schritt erfolgt die Entwicklung nach x_6 mit Hilfe der Shannon-Zerlegung. Da der positive Kofaktor von x_2 aus der vorherigen Zerlegung konstant T ist, muss dieser nicht weiter entwickelt werden. Es ergibt sich $f|_{x_2:=\mathsf{F},x_6:=\mathsf{T}} = \mathsf{T}$ und $f|_{x_2,x_6:=\mathsf{F}} = \neg x_5 \vee x_5 \wedge ((x_1 \wedge x_3) \vee (x_4 \wedge x_7) \vee (\neg x_3 \wedge \neg x_4))$.
3. Im dritten Schritt erfolgt die Entwicklung des negativen Kofaktors aus Schritt 2. bezüglich der Variablen x_5 mit Hilfe der Shannon-Zerlegung. Der positive Kofaktor von x_6 ist eine konstante Funktion und muss deshalb nicht zerlegt werden. Die Zerlegung der Funktion $f|_{x_2,x_6:=\mathsf{F}}$ ergibt $f|_{x_2,x_5,x_6:=\mathsf{F}} = \mathsf{T}$ und $f|_{x_2,x_6:=\mathsf{F},x_5:=\mathsf{T}} = (x_1 \wedge x_3) \vee (x_4 \wedge x_7) \vee (\neg x_3 \wedge \neg x_4)$.
4. Wieder bleibt aus dem vorherigen Schritt nur eine nicht konstante Funktion übrig, der positive Kofaktor $f|_{x_2,x_6:=\mathsf{F},x_5:=\mathsf{T}}$. Dieser wird nun bzgl. x_3 mit der positiven Davio-Zerlegung entwickelt. Dabei ergibt sich:
$$f|_{x_2,x_3,x_6:=\mathsf{F},x_5:=\mathsf{T}} = \neg x_4 \vee x_7$$
sowie
$$f|_{x_2,x_6:=\mathsf{F},x_3,x_5:=\mathsf{T}} \oplus f|_{x_2,x_3,x_6:=\mathsf{F},x_5:=\mathsf{T}} = (x_1 \vee x_4 \wedge x_7) \oplus (\neg x_4 \vee x_7)$$
5. Im nächsten Schritt müssen alle Kofaktoren aus Schritt 4. bezüglich x_4 mit der negativen Davio-Zerlegung entwickelt werden. Zunächst der negative Kofaktor:
$$f|_{x_2,x_3,x_6:=\mathsf{F},x_4,x_5:=\mathsf{T}} = x_7$$
und
$$f|_{x_2,x_3,x_6:=\mathsf{F},x_4,x_5:=\mathsf{T}} \oplus f|_{x_2,x_3,x_4,x_6:=\mathsf{F},x_5:=\mathsf{T}} = \neg x_7$$
Für den positiven Kofaktor ergibt sich:
$$\left(f|_{x_2,x_6:=\mathsf{F},x_3,x_5:=\mathsf{T}} \oplus f|_{x_2,x_3,x_6:=\mathsf{F},x_5:=\mathsf{T}}\right)|_{x_4:=\mathsf{T}} = x_7 \oplus (x_1 \vee x_7) = x_1 \wedge \neg x_7$$
und
$$\begin{aligned} & \left(f|_{x_2,x_6:=\mathsf{F},x_3,x_5:=\mathsf{T}} \oplus f|_{x_2,x_3,x_6:=\mathsf{F},x_5:=\mathsf{T}}\right)|_{x_4:=\mathsf{T}} \\ \oplus\ & \left(f|_{x_2,x_6:=\mathsf{F},x_3,x_5:=\mathsf{T}} \oplus f|_{x_2,x_3,x_6:=\mathsf{F},x_5:=\mathsf{T}}\right)|_{x_4:=\mathsf{F}} \\ =\ & \neg x_1 \oplus (x_1 \wedge \neg x_7) \\ =\ & \neg x_1 \vee \neg x_7 \end{aligned}$$
6. Die verbleibenden Schritte zur Entwicklung nach x_1 und x_7 sind trivial. Das resultierende OKFDD ist in Abb. 6.7 zu sehen. Da sich dieses nicht weiter reduzieren lässt (siehe Anhang B.3), handelt es sich um ein reduziertes OKFDD.

Um die Äquivalenz der Funktion f aus Beispiel 6.1.4 und den kombinatorischen Schaltungen aus Abb. 6.3 zu zeigen, muss ein ROKFDD mit der selben Variablenordnung wie in Beispiel 6.1.4 für eine der Schaltungen aufgestellt werden, und dieses mit dem ROKFDD aus Abb. 6.7 isomorph sein. Es ist ausreichend, das ROKFDD für eine der beiden Schaltungen aufzustellen, da die Äquivalenz der beiden Schaltungen

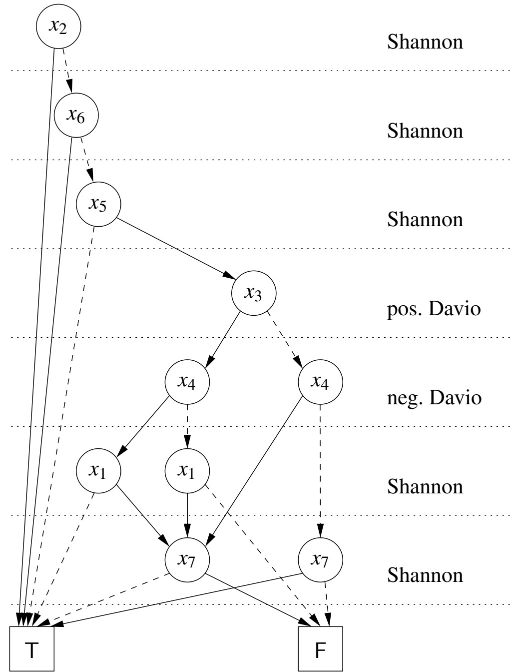


Abb. 6.7. ROKFDD der Booleschen Funktion f für die Ausgänge y_1 und y_2 für die Schaltungen aus Abb. 6.3

bereits zuvor gezeigt wurde. Zur Konstruktion des ROKFDD wird zunächst für die Schaltung in Abb. 6.3b) die Boolesche Funktion durch symbolische Simulation bestimmt. Die Boolesche Funktion, die y_2 implementiert, lautet:

$$\begin{aligned} y_2 = & \ x_1 \wedge x_3 \wedge (x_2 \vee x_5) \vee x_2 \\ & \vee \neg(((x_3 \wedge (x_2 \vee x_5)) \vee ((x_2 \vee x_5) \wedge x_4)) \wedge \neg x_6) \\ & \vee((x_2 \vee x_5) \wedge x_4 \wedge x_7) \end{aligned}$$

1. Im ersten Schritt wird die Funktion y_2 nach x_2 mit Hilfe der Shannon-Zerlegung entwickelt. Der positive Kofaktor ergibt sich zu T. Der negative Kofaktor ergibt sich zu:

$$f|_{x_2:=\mathsf{F}} = x_1 \wedge x_3 \wedge x_5 \vee \neg(((x_3 \wedge x_5) \vee (x_4 \wedge x_5)) \wedge \neg x_6) \vee x_4 \wedge x_5 \wedge x_7$$

2. Der positive Kofaktor $f|_{x_2:=\mathsf{T}}$ ist konstant T, weshalb keine weitere Entwicklung für diesen notwendig ist. Für den negativen Kofaktor $f|_{x_2:=\mathsf{F}}$ erfolgt die Entwicklung entsprechend der Variablenordnung per Shannon-Zerlegung nach Variable x_6. Der resultierende positive Kofaktor ist wieder konstant T. Der negative Kofaktor ergibt sich zu:

$$f|_{x_2,x_6:=\mathsf{F}} = x_1 \wedge x_3 \wedge x_5 \vee \neg((x_3 \wedge x_5) \vee (x_4 \wedge x_5)) \vee x_4 \wedge x_5 \wedge x_7$$

3. Im folgenden Schritt wird der negative Kofaktor $f|_{x_2,x_6:=\mathsf{F}}$ nach Variable x_5 mit Hilfe der Shannon-Zerlegung entwickelt. Der negative Kofaktor $f|_{x_2,x_5,x_6:=\mathsf{F}}$ ergibt konstant T. Der positive Kofaktor ist:

$$\begin{aligned} f|_{x_2,x_6:=\mathsf{F},x_5:=\mathsf{T}} &= (x_1 \wedge x_3) \vee \neg(x_3 \vee x_4) \vee (x_4 \wedge x_7) \\ &= (x_1 \wedge x_3) \vee (x_4 \wedge x_7) \vee (\neg x_3 \wedge \neg x_4) \end{aligned}$$

Dies ist die selbe aussagenlogische Formel wie in Schritt 3. der obigen ROKFDD-Entwicklung aus Beispiel 6.1.4. Da die Variablenordnung und die Zerlegungen hier die selben sind, verläuft auch die weitere Konstruktion des ROKFDD identisch. Als Ergebnis entsteht bei der Entwicklung des ROKFDD für die kombinatorische Schaltung in Abb. 6.3b) auch das in Abb. 6.7 gezeigte ROKFDD. Dies bedeutet, dass die Implementierung (kombinatorische Schaltung) äquivalent zur Spezifikation (Boolesche Funktion f) ist.

6.1.2 Explizite Äquivalenzprüfung auf der Logikebene

Neben der impliziten Äquivalenzprüfung von kombinatorischen Schaltungen mittels kanonischer Funktionsrepräsentationen, gibt es effiziente Verfahren zur *expliziten Äquivalenzprüfung* auf der Logikebene. Hierzu können entweder Verfahren zur formalen Erfüllbarkeitsprüfung mittels SAT-Solver oder Verfahren zur automatischen *Testfallgenerierung* (auch *Testmustergenerierung* engl. *Automatic Test Pattern Generation*, *ATPG*) zum Einsatz kommen. Beiden Verfahren liegt eine gemeinsame Datenstruktur zugrunde, die als engl. *Miter* bezeichnet wird [58].

Die Idee bei der Konstruktion einer Miter-Schaltung besteht darin, dass die beiden Ausgänge von zwei kombinatorischen Schaltungenf_1 und f_2 mit einem XOR-Gatter verglichen werden. Gleichzeitig werden die Eingänge beider Schaltungen miteinander verbunden, so dass diese stets die selben Eingaben verarbeiten. Der Beweis der Äquivalenz von f_1 und f_2 erfolgt, indem gezeigt wird, dass der Ausgang des XOR-Gatters niemals den Wert T annehmen kann. Bei der Falsifikation ist die Aufgabe, eine Testfalleingabe zu finden, so dass der Ausgang des XOR-Gatters T wird. Besitzen die beiden kombinatorischen Schaltungen mehr als einen Ausgang, so werden die Ausgänge paarweise mit XOR-Gattern verglichen und der Ausgang der Miter-Schaltung mit einem OR-Gatter berechnet, welches als Eingänge die Ausgänge der XOR-Gatter erhält. Dies ist in Abb. 6.8 dargestellt.

Vor diesem Hintergrund sind die folgenden drei Aussagen äquivalent [329]:

1. f_1 und f_2 sind funktional nicht äquivalent.
2. Der Ausgang der Miter-Schaltung aus f_1 und f_2 ist erfüllbar, d. h. es existiert eine Eingabe, so dass der Ausgang der Miter-Schaltung zu T evaluiert.
3. Der Ausgang der Miter-Schaltung ist auf sog. *Haftfehler* (engl. *stuck-at faults*) prüfbar (siehe auch [1]).

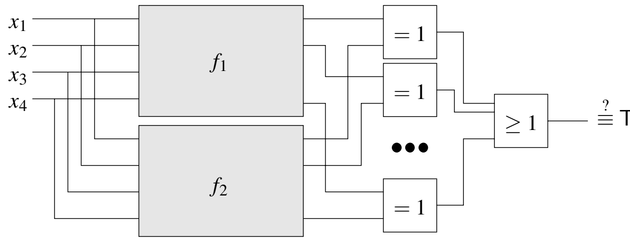


Abb. 6.8. Miter-Schaltung

SAT-basierte Äquivalenzprüfung

Um eine explizite Äquivalenzprüfung zweier kombinatorischer Schaltungen mit SAT-Solvern durchzuführen, wird die entsprechende Miter-Schaltung zunächst in eine aussagenlogische Formel in *konjunktiver Normalform* (*KNF*) übersetzt [289]. Eine aussagenlogische Formel in KNF besteht aus der Konjunktion von sog. *Klauseln*. Jede Klausel besteht wiederum aus der Disjunktion von *Literalen*, wobei ein Literal eine Variable oder deren Negation ist. Um eine aussagenlogische Formel in KNF zu erfüllen, muss jede Klausel und somit mindestens ein Literal in jeder Klausel den Wert T zugewiesen bekommen.

Klauseln können als Menge von Literalen interpretiert werden. Die Anzahl der Literale in einer Klausel wird als $|c|$ geschrieben. Die *leere Klausel* repräsentiert den konstanten Wert F. Formeln in KNF können als Menge Φ von Klauseln repräsentiert werden. In diesem Fall beschreibt die leere Menge $\emptyset$ die konstante Boolesche Funktion T.

Die Konstruktion der Formel in KNF aus einer kombinatorischen Schaltung erfolgt anhand des Booleschen Netzwerkes $\mathcal{N}$ (siehe auch Definition A.2.1 auf Seite 525), welches die Miter-Schaltung repräsentiert. Jeder Knoten $v \in V$ im Booleschen Netzwerk $\mathcal{N}$ wird mit einer Formel in KNF annotiert, welche die implementierte Funktion des assoziierten Gatters beschreibt. Im Folgenden werden lediglich Gatter (Knoten) mit zwei Eingängen x_1 und x_2 und einem Ausgang z betrachtet. Der Typ des Knoten v sei gegeben durch die Funktion $\text{type}(v)$. Die Evaluierung eines Knotens wird als konsistent bezeichnet, wenn gilt:

$$z = \text{type}(v)(x_1, x_2)$$

Dies kann auch wie folgt geschrieben werden [329]:

$$(\text{type}(v)(x_1, x_2) \Rightarrow z) \wedge (z \Rightarrow \text{type}(v)(x_1, x_2))$$

Mit anderen Worten: Es muss die mit den beiden Eingängen berechnete Funktion am Ausgang zu sehen sein und weiterhin das am Ausgang ausgegebene Ergebnis mit der berechneten Funktion übereinstimmen. In KNF kann dieser Zusammenhang auch als

$$(\neg\text{type}(v)(x_1, x_2) \vee z) \wedge (\neg z \vee \text{type}(v)(x_1, x_2))$$

geschrieben werden. Ersetzt man schließlich noch die type-Funktion durch die entsprechenden Berechnungsvorschriften, erhält man so für jedes Gatter in der Miter-Schaltung die berechnete Boolesche Funktion, welche sich einfach in die konjunktive Normalform umschreiben lässt. Betrachtet man beispielsweise ein AND-Gatter, d. h. $\text{type}(v) = AND$ so ergibt sich:

$$(\neg(x_1 \wedge x_2) \vee z) \wedge (\neg z \vee (x_1 \wedge x_2))$$

Durch Umformung in KNF erhält man schließlich

$$(\neg z \vee x_1) \wedge (\neg z \vee x_2) \wedge (z \vee \neg x_1 \vee \neg x_2).$$

Diese Formel kann so interpretiert werden: Jedes Mal, wenn der Ausgang z den Wert T hat, müssen auch die Eingänge x_1 und x_2 den Wert T besitzen. Dies wird durch die ersten beiden Klauseln sichergestellt. Die dritte Klausel beschreibt den Fall, dass der Ausgang $z = \mathsf{F}$ ist. In diesem Fall muss gelten, dass mindestens einer der Eingänge x_1 oder x_2 ebenfalls den Wert F hat. Tabelle 6.1 gibt die aussagenlogischen Formeln der wichtigsten Logikgatter in KNF an. Eine Erweiterung auf Gatter auf mehr als zwei Eingängen ist einfach möglich.

Tabelle 6.1. Funktionen wichtiger Gatter in KNF [289]

$\text{type}(v)$	KNF
AND	$(\neg z \vee x_1) \wedge (\neg z \vee x_2) \wedge (z \vee \neg x_1 \vee \neg x_2)$
OR	$(z \vee \neg x_1) \wedge (z \vee \neg x_2) \wedge (\neg z \vee x_1 \vee x_2)$
$NAND$	$(z \vee x_1) \wedge (z \vee x_2) \wedge (\neg z \vee \neg x_1 \vee \neg x_2)$
NOR	$(\neg z \vee \neg x_1) \wedge (\neg z \vee \neg x_2) \wedge (z \vee x_1 \vee x_2)$
XOR	$(\neg z \vee x_1 \vee x_2) \wedge (\neg z \vee \neg x_1 \vee \neg x_2) \wedge (z \vee \neg x_1 \vee x_2) \wedge (z \vee x_1 \vee \neg x_2)$
NOT	$(\neg z \vee \neg x_1) \wedge (z \vee x_1)$

Die Konstruktion der Booleschen Funktion für ein gegebenes Booleschen Netzwerk $\mathcal{N}$ wird anhand eines Beispiels erläutert.

Beispiel 6.1.5. Gegeben ist die Miter-Schaltung für die Funktionen $f_1(x_1,x_2) := \neg x_1 \wedge x_2$ und $f_2(x_1,x_2) := \neg(x_1 \vee \neg x_2)$ in Abb. 6.9. Da die Formeln aller Gatter (inklusive dem XOR-Gatter) gleichzeitig erfüllt sein müssen, kann die Formel $\phi(\mathcal{N})$ in KNF für $\mathcal{N}$ durch Konjunktion aller Formeln der Gatter gewonnen werden, d. h.

$$\begin{aligned}
\phi(\mathcal{N}) = {} & (\neg z_1 \vee \neg x_1) \wedge (z_1 \vee x_1) \wedge \\
& (\neg z_2 \vee \neg x_2) \wedge (z_2 \vee x_2) \wedge \\
& (\neg z_3 \vee z_1) \wedge (\neg z_3 \vee x_2) \wedge (z_3 \vee \neg z_1 \vee \neg x_2) \wedge \\
& (z_4 \vee \neg x_1) \wedge (z_4 \vee \neg z_2) \wedge (\neg z_4 \vee x_1 \vee z_2) \wedge \\
& (\neg z_5 \vee \neg z_4) \wedge (z_5 \vee z_4) \wedge \\
& (\neg z \vee z_3 \vee z_5) \wedge (\neg z \vee \neg z_3 \vee \neg z_5) \wedge (z \vee \neg z_3 \vee z_5) \wedge (z \vee z_3 \vee \neg z_5)
\end{aligned}$$

Die Formel $\phi(\mathcal{N})$ ist nur erfüllbar, wenn f_1 und f_2 nicht äquivalent sind, d. h. der Ausgang der Schaltung den Wert T erhalten kann.

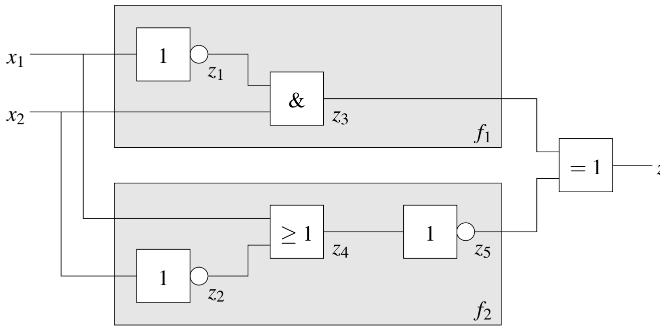


Abb. 6.9. Beispiel einer Miter-Schaltung

Um die Erfüllbarkeit von Formeln, wie im obigen Beispiel notwendig, formal zu überprüfen, können sog. *SAT-Solver* verwendet werden (siehe Anhang C.2). Die meisten SAT-Solver basieren auf dem DPLL-Algorithmus [128], der nach seinen Erfindern Davis, Putnam, Longman und Loveland benannt ist. Der DPLL-Algorithmus arbeitet auf der Menge Φ der Klauseln der zu erfüllenden Formel in KNF. Prinzipiell funktioniert der DPLL-Algorithmus so, dass einer Variablen, die bisher unspezifiziert ist, der Wert T oder der Wert F zugewiesen wird. Dies wird als Verzweigung bezeichnet. Anschließend wird überprüft, ob durch diese Variablenbelegung sog. *Einerklauseln* entstanden sind. Dies sind Klauseln, die lediglich aus einem einzelnen Literal bestehen. Einerklauseln implizieren Wertzuweisungen an Variablen, da dieses Literal erfüllt sein muss. Die implizierte bedingte Zuweisung und anschließende Suche nach Einerklauseln wird solange wiederholt, bis entweder keine Einerklauseln mehr entstehen oder ein Konflikt auftritt, d. h. einer Variablen sollen gegensätzliche Werte zugewiesen werden. Das wiederholte Suchen nach Einerklauseln und propagieren dadurch implizierter Variablenbelegungen wird als engl. *Boolean Constraint Propagation* (*BCP*) bezeichnet.

Entsteht ein Konflikt, so muss die letzte Verzweigung, bei der noch nicht beide Wertzuweisungen an die dort belegte Variable ausprobiert worden sind, rückgängig gemacht werden. Dies wird als *Zurückverfolgung* bezeichnet. Der Variablen wird dann der inverse Wert zugewiesen. Ist eine Zurückverfolgung nicht mehr möglich, da bereits alle Variablen mit beiden möglichen Wertzuweisungen betrachtet wurden, ist die gegebene Formel nicht erfüllbar. Werden keine weiteren Einerklauseln durch BCP gefunden, gibt es zwei Möglichkeiten:

1. Entweder wurde allen Variablen ein Wert zugewiesen und es wurde somit eine Variablenbelegung gefunden, welche die Formel erfüllt, oder
2. es gibt noch unspezifizierte Variablen. In diesem Fall wird mit einer weiteren Verzweigung fortgefahren.

Eine ausführliche Erläuterung zu SAT-Solvern und mögliche Verbesserungen sind in Anhang C.2 zu finden.

Beispiel 6.1.6. Für die Miter-Schaltung aus Abb. 6.9 und damit die in Beispiel 6.1.5 konstruierte Formel $\phi(\mathcal{N})$ in KNF wird die Erfüllbarkeit mit einem SAT-Solver geprüft. Dieser ergibt, dass $\phi(\mathcal{N})$ unerfüllbar ist. Das bedeutet wiederum, dass die Funktionen f_1 und f_2 äquivalent sind, da keine Variablenbelegung existiert, die $\phi(\mathcal{N})$ erfüllt. Dies kann durch einfaches Ausprobieren der vier möglichen Variablenbelegungen für x_1 und x_2 nachvollzogen werden.

Kombinierte BDD- und SAT-Solver-Ansätze

Die Erfüllbarkeit der Miter-Schaltung aus Abb. 6.9 kann auch implizit, basierend auf ROBDDs, geprüft werden. Die in Beispiel 6.1.5 entwickelte Boolesche Funktion $\phi(\mathcal{N})$ wird hierzu als ROBDD mit beliebiger Variablenordnung repräsentiert. Die Miter-Schaltung ist genau dann erfüllbar, wenn das ROBDD nicht nur aus dem Terminalknoten mit dem Wert F besteht. Auch wenn dieser abschließende Test in konstanter Zeit durchführbar ist, kann während der Konstruktion das BDD sehr groß werden, was bei dem Computersystem, auf dem die Verifikation durchgeführt wird, dazu führen kann, dass der Speicher nicht ausreicht. Auf der anderen Seite kann ein häufiges Zurückverfolgen im SAT-Solver zu schlechten Laufzeiten des Algorithmus führen. Aus diesem Grund werden in der Praxis auch Verfahren eingesetzt, die BDD-basierte Verfahren und SAT-Solver kombinieren.

Bei einem kombinierten Verfahren zur Äquivalenzprüfung kombinatorischer Schaltungen, wird die Miter-Schaltung zunächst in zwei Teile partitioniert. Der sog. *Eingangspartitionsblock* enthält alle Eingänge der Schaltungen, während der sog. *Ausgangspartitionsblock* den Ausgang beinhaltet. Der Ausgangspartitionsblock wird dann durch ein ROBDD repräsentiert [208].

Beispiel 6.1.7. Als Beispiel für die Partitionierung wird noch einmal die Miter-Schaltung aus Abb. 6.9 betrachtet. Eine mögliche Bipartitionierung ist in Abb. 6.10a) dargestellt. Der Eingangspartitionsblock besteht aus zwei NOT- und einem AND-Gatter. Die Eingänge des Ausgangspartitionsblock sind x_1, z_2 und z_3. Der Ausgangspartitionsblock besteht aus einem OR, einem NOT- und einem XOR-Gatter. Das ROBDD, das den Ausgangspartitionsblock mit der Variablenordnung $x_1 < z_2 < z_3$ repräsentiert, ist in Abb. 6.10b) dargestellt.

Die Formel in KNF, die den Eingangspartitionsblock beschreibt, lautet:

$$\begin{aligned}\phi(\mathcal{N}_{in}) = &(\neg z_1 \vee \neg x_1) \wedge (z_1 \vee x_1) \wedge \\ &(\neg z_2 \vee \neg x_2) \wedge (z_2 \vee x_2) \wedge \\ &(\neg z_3 \vee z_1) \wedge (\neg z_3 \vee x_2) \wedge (z_3 \vee \neg z_1 \vee \neg x_2)\end{aligned}$$

Man sieht, dass die Formel des Eingangspartitionsblocks und das ROBDD für den Ausgangspartitionsblock gemeinsame Variablen besitzen. Im Beispiel 6.1.7 sind dies die Variablen x_1, z_2 und z_3.

Um zu zeigen, dass die Funktionen f_1 und f_2 nicht äquivalent sind, muss eine Variablenbelegung gefunden werden, die den Wert T auf den Ausgang der Miter-Schaltung erzeugt. Dies bedeutet für den kombinierten Ansatz, dass eine Variablenbelegung gefunden werden muss, deren Pfad in dem ROBDD zu dem Terminalkno-

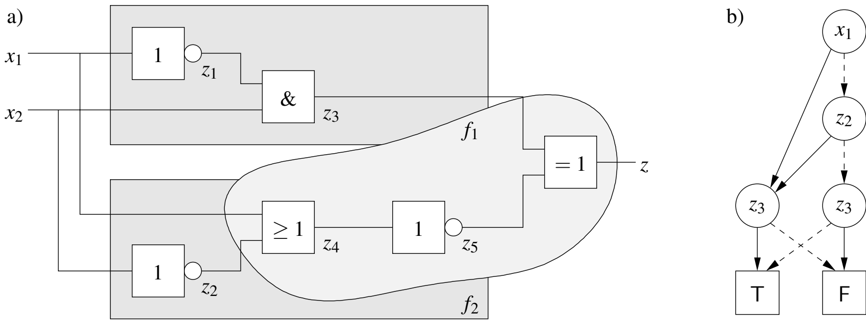


Abb. 6.10. a) Bipartition der Miter-Schaltung und b) ROBDD des Ausgangspartitionsblocks

ten mit dem Wert T führt. In dem obigen Beispiel sind dies drei mögliche Belegungen: $\beta_1 : x_1 = z_3 := \mathsf{T}$, $\beta_2 : x_1 := \mathsf{F} \wedge z_2 = z_3 := \mathsf{T}$ und $\beta_3 : x_1 = z_2 = z_3 := \mathsf{F}$. Diese Belegungen können nacheinander in die Formel für den Eingangspartitionsblock propagiert werden. Anschließend wird mit Hilfe eines SAT-Solvers eine konsistente Belegung der übrigen Eingangsvariablen gesucht. Dies ist allerdings für die drei Belegungen β_1, β_2 und β_3 nicht möglich, also sind die beiden Schaltungen äquivalent.

Die in dem Beispiel verwendete Aufzählung der Belegungen, die das ROBDD erfüllen, kann bei komplexeren Schaltungen schnell zu aufwendig werden. Aus diesem Grund ist es sinnvoller, den SAT-Solver ohne Beschränkungen durch vorher gesuchte Variablenbelegungen zu starten und jedes Mal, wenn der SAT-Solver eine gemeinsame Variable belegt, zu prüfen, ob noch eine erfüllende Vervollständigung der Variablenbelegung für die Funktion des ROBDD existiert. Mit anderen Worten: Es wird geprüft, ob die Belegung einer gemeinsamen Variablen dazu führt, dass der resultierende Pfad in dem ROBDD zu einen Terminalknoten mit Wert F führt. Dieser Test kann sehr effizient durchgeführt werden und grenzt schnell große Teile des Suchraums aus.

Neben der Möglichkeit, den Ausgangspartitionsblock mit einem ROBDD zu repräsentieren, kann alternativ auch der Eingangspartitionsblock durch mehrere ROBDDs dargestellt werden [372]. Der Ausgangspartitionsblock wird als Formel in KNF repräsentiert. In diesem Fall muss für jede gemeinsame Variable ein ROBDD aufgebaut werden.

Beispiel 6.1.8. In Abb. 6.11a) ist eine Bipartition für die Miter-Schaltung aus Abb. 6.9 zu sehen. Die gemeinsamen Variablen für den Eingangs- und Ausgangspartitionsblock sind wiederum x_1, z_2 und z_3. Für jede gemeinsame Variable wird ein ROBDD mit der Variablenordnung $x_1 < x_2$ aufgebaut. Da die ROBDDs für x_1 und z_2 trivial sind, zeigt Abb. 6.11b) lediglich das ROBDD für die gemeinsame Variable z_3.

Der Ausgangspartitionsblock wird als KNF repräsentiert:

$$\begin{aligned}\phi(\mathcal{N}_{out}) = {} & (z_4 \vee \neg x_1) \wedge (z_4 \vee \neg z_2) \wedge (\neg z_4 \vee x_1 \vee z_2) \wedge \\ & (\neg z_5 \vee \neg z_4) \wedge (z_5 \vee z_4) \wedge \\ & (\neg z \vee z_3 \vee z_5) \wedge (\neg z \vee \neg z_3 \vee \neg z_5) \wedge (z \vee \neg z_3 \vee z_5) \wedge (z \vee z_3 \vee \neg z_5)\end{aligned}$$

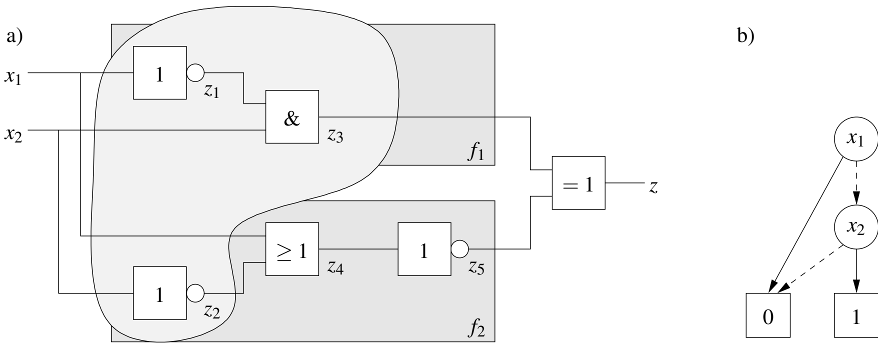


Abb. 6.11. a) Bipartition der Miter-Schaltung und b) ROBDD die gemeinsame Variable z_3

Zur Koordination zwischen ROBDDs und SAT-Solver wird die charakteristische Funktion der Menge der Variablenbelegungen, die durch den SAT-Solver vorgenommen wurden, in einem zusätzlichen ROBDD gespeichert. Dieses ROBDD entspricht zu Beginn der konstanten Funktion T. Jedes Mal, wenn der SAT-Solver eine gemeinsame Variable x belegt, wird dieses ROBDD entsprechend aktualisiert. Hierzu wird, entsprechend der Belegung, das ROBDD der charakteristischen Funktion mit dem ROBDD der Formel x bzw. $\neg x$ konjugiert. Repräsentiert das ROBDD für die charakteristische Funktion die konstante Funktion F, so muss der SAT-Solver eine Zurückverfolgung durchführen.

ATPG-basierte Äquivalenzprüfung

Parallel zur Entwicklung SAT-basierter Methoden für die Äquivalenzprüfung entstanden Verfahren zur Äquivalenzprüfung auf Basis von Methoden zur automatischen Testfallgenerierung (engl. *Automatic Test Pattern Generation, ATPG*). Diese basieren wie die SAT-basierte Äquivalenzprüfung auf der Miter-Schaltung von zu vergleichenden Schaltungen. ATPG wurde vor dem Hintergrund von Herstellungsfehlern entwickelt: Für den Test einer gefertigten Schaltung werden aus einer Miter-Schaltung automatisch Testfälle und dafür notwendige Testfalleingaben (Stimuli) abgeleitet. Die Miter-Schaltung besteht dabei aus einem Modell der entwickelten Schaltung und einer Schaltung, die sich aus der entwickelten Schaltung durch einen Herstellungsfehler ergeben kann. Somit wird zur Erstellung einer Testfalleingabe zunächst die gegebene Schaltung entsprechend eines möglichen Herstellungsfehlers mutiert und anschließend aus der gegebenen und der mutierten Schaltung eine Miter-Schaltung aufgebaut. Für diese Miter-Schaltung wird dann eine Testfalleingabe generiert, so dass der Ausgang der Miter-Schaltung den Wert T zugewiesen bekommt, d. h. der eingebaute Fehler erkannt wird. Eine zu testende Schaltung mit dem betrachteten Herstellungsfehler erzeugt also garantiert eine andere Ausgabe, als eine Schaltung ohne diesen Fehler, sobald diese mit der generierten Testfalleingabe sti-

muliert wird. Die folgende Einführung zur Äquivalenzprüfung basierend auf einer automatischen Testfallgenerierung basiert auf der Darstellung in [329].

Gegeben sei ein Boolesches Netzwerk $\mathcal{N}$, welches eine Boolesche Funktion $\phi(\mathcal{N})$ repräsentiert. Ein Herstellungsfehler führt dazu, dass $\mathcal{N}$ in ein anderes kombinatorisches Schaltwerk $\mathcal{N}'$ mutiert. Dieses implementiert die Boolesche Funktion $\phi(\mathcal{N}')$. Um eine kombinatorische Schaltung auf genau diesen aufgetretenen Herstellungsfehler testen zu können, wird eine Testfalleingabe (auch *Testvektor*) $x = (x_1, \ldots, x_n)$ benötigt, für den gilt:

$$\phi(\mathcal{N})(x_1, \ldots, x_n) \neq \phi(\mathcal{N}')(x_1, \ldots, x_n)$$

Besitzt $\mathcal{N}$ lediglich einen Ausgang, so kann die Testfalleingabe x gefunden werden, indem eine erfüllende Belegung für

$$\phi(\mathcal{N})(x_1, \ldots, x_n) \oplus \phi(\mathcal{N}')(x_1, \ldots, x_n)$$

gefunden wird.

An dieser Beschreibung erkennt man auch, dass man Testfalleingaben nur für bestimmte Fehler generiert. Der zu überprüfende Fehler wird dabei als Variation $\mathcal{N}'$ des Booleschen Netzwerkes $\mathcal{N}$ codiert. Die Annahme hierbei ist, dass physikalische Fehler, die z. B. durch Defekte in der Herstellung entstehen, als logische Fehler dargestellt werden können. Dabei können unterschiedliche physikalische Fehler zu dem selben logischen Fehler führen.

Ein häufig verwendetes Fehlermodell ist das sog. *Haftfehlermodell* (engl. *stuck-at fault model*). Die zugrundeliegende Annahme lautet, dass in vielen Technologien ein Kurzschluss zwischen Versorgungsspannung oder Masse und einem Signal, oder ein Leerlauf auf einer Signalleitung zu einem konstanten Potential der Signalleitung führt. Man spricht von einem Haftfehler 0 bzw. Haftfehler 1, wenn ein Signal stets den logischen Wert F bzw. T trägt. Bei der automatischen Testfallgenerierung ist das Ziel, eine Belegung der primären Eingänge der Schaltung zu finden, so dass einem (internen) Signal s der Wert T bzw. F zugewiesen wird, falls bei diesem Signal ein Haftfehler 0 bzw. Haftfehler 1 vermutet wird. Dies wird als *Aktivierung* des Fehlers bezeichnet. Weiterhin muss die Belegung der primären Eingänge so gewählt werden, dass der Fehler an einem primären Ausgang beobachtbar wird. Dies wird als *Übertragung* des Fehlers bezeichnet. Lässt sich ein Fehler nicht aktivieren oder übertragen, so ist dieser Fehler nicht steuerbar bzw. nicht beobachtbar, und somit auch nicht überprüfbar.

Neben dem Haftfehlermodell gibt es weitere Fehlermodelle. Das Haftfehlermodell ist ein statisches Fehlermodell, da dauerhaft eine falsche (interne) Funktion berechnet wird. Daneben existieren dynamische Fehlermodelle, bei denen das Zeitverhalten von Signalen fehlerhaft ist. So ist die Berechnung einer internen Funktion zwar korrekt, allerdings gibt es beim Schalten von F nach T oder umgekehrt ungewöhnlich lange Umschaltzeiten in einzelnen Gattern. Dies kann dazu führen, dass nachfolgende Schaltungsteile mit alten Ergebnissen rechnen. Im Folgenden wird lediglich vom Haftfehlermodell ausgegangen.

Der am häufigsten verwendete Algorithmus zur automatischen Testfallgenerierung ist der sog. *D-Algorithmus* [381]. Der D-Algorithmus basiert auf einer fünfwertigen Logik mit der Wertemenge $\{0,1,X,D,\overline{D}\}$. Der Wert 0 (1) zeigt an, dass jedes Mal, wenn ein Signal den logischen Wert F (T) tragen soll, dieses auch den Wert F (T) trägt. Aktivierte Haftfehler 0 (Haftfehler 1) werden durch den Wert D ($\overline{D}$) modelliert. Ist der Wert eines Signals unbekannt, so wird dies durch den Wert X dargestellt. Die Funktionstabellen für drei wichtige Logikgatter in fünfwertiger Logik sind in Tabelle 6.2 zu sehen.

Tabelle 6.2. OR-, AND- und NOT-Gatter in fünfwertiger Logik [1]

OR	0	1	D	$\overline{D}$	X
0	0	1	D	$\overline{D}$	X
1	1	1	1	1	1
D	D	1	D	1	X
$\overline{D}$	$\overline{D}$	1	1	$\overline{D}$	X
X	X	1	X	X	X

AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

NOT	
0	1
1	0
D	$\overline{D}$
$\overline{D}$	D
X	X

Nachdem ein Haftfehler 0 (Haftfehler 1) für ein Signal in der Schaltung gesetzt ist, muss dieser Fehler aktiviert werden, indem eine Belegung der primären Eingänge gesucht wird, die dem Signal den logischen Wert T (F) zuweist. Daneben muss der Fehler an die primären Ausgänge übertragen werden.

Die explizite Äquivalenzprüfung mit Hilfe einer Miter-Schaltung basiert auf der Idee, durch geeignete Belegung der primären Eingänge der Schaltung dem Ausgang der Schaltung den Wert T zuzuweisen. Somit kann diese explizite Äquivalenzprüfung auch als automatische Testfallgenerierung betrachtet werden. Lediglich die Aktivierungsphase wird dabei durchlaufen. Eine Übertragung an einen primären Ausgang ist nicht mehr notwendig. Aus diesem Grund wird im Folgenden lediglich die Aktivierungsphase betrachtet. Dies wird zunächst an einem Beispiel verdeutlicht.

Beispiel 6.1.9. In Abb. 6.12 ist eine Miter-Schaltung dargestellt. Um die Nichtäquivalenz der beiden Funktionen f_1 und f_2 zu zeigen, muss der Ausgang z der Miter-Schaltung den Wert T erhalten. Dies ist durch die Funktion activate(T) dargestellt.

Durch Rückwärts-Implikation erhält man, dass eine Möglichkeit, dies zu erfüllen, darin besteht, dem Signal z_3 den Wert T und dem Signal z_4 den Wert F zuzuweisen. Dies ist wiederum durch die entsprechende activate-Funktion symbolisiert.

Um eine Belegung der primären Eingänge zu erhalten, müssen weitere Rückwärts-Implikationen durchgeführt werden. Die Aktivierung des Wertes $z_3 = \mathsf{T}$ verlangt, dass sowohl z_1 als auch x_2 den Wert T tragen. Dies wiederum bedeutet, dass $x_1 = \mathsf{F}$ sein muss. Hiermit ist eine Belegung der primären Eingänge gefunden: $x_1 := \mathsf{F}$ und $x_2 := \mathsf{T}$. Allerdings muss diese Belegung auch konsistent für die Aktivierung des Signals z_4 sein. Aus diesem Grund werden die Rückwärts-Implikationen für das OR- und NOT-Gatter der Funktion f_2 auch noch ausgewertet.

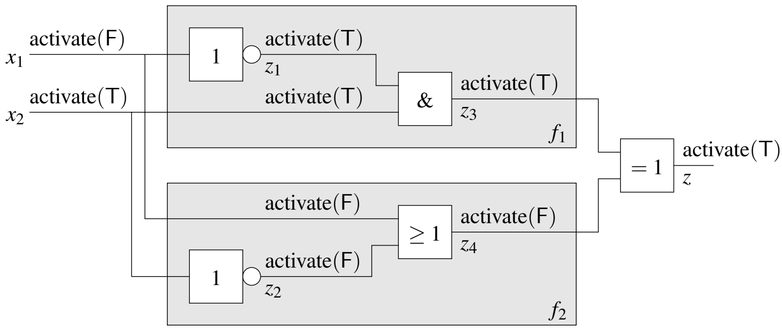


Abb. 6.12. Aktivierung eines Haftfehlers 1 in einer Miter-Schaltung

Die Aktivierung activate(F) für das Signal z_4 impliziert, dass sowohl das Signal x_1 als auch das Signal z_2 den Wert F zugewiesen bekommen, was wiederum bedeutet, dass x_2 den Wert T erhält. Dies ist konsistent mit der bereits gefundenen Belegung β. Somit sind die Funktionen f_1 und f_2 nicht äquivalent, was z. B. durch die Testfalleingabe $x_1 :=$ F und $x_2 :=$ T per Simulation gezeigt werden kann.

Die Aktivierungsfunktion

Vor der Aktivierungsphase wird zunächst allen Signalen, außer dem Ausgang der Miter-Schaltung, der Wert X zugewiesen. Der Ausgang der Miter-Schaltung erhält den Wert T. Während der Aktivierungsphase müssen nun diejenigen Gatter-Ausgänge, die durch die activate-Funktion einen Wert zugewiesen bekommen haben, dieser aber noch nicht durch deren Gatter-Eingänge impliziert wird, verfolgt werden. Diese Menge an Gatterausgängen wird als *Aktivierungsfront* bezeichnet. Die Aktivierungsfront enthält zu Beginn lediglich das XOR-Gatter am Ausgang der Miter-Schaltung.

Nach der Initialisierung wird durch wiederholte Anwendung der Rückwärtsimplikation versucht, die Aktivierungsfront zu den primären Eingängen der Schaltung zu verschieben. Der Algorithmus ist durch den folgenden Pseudo-Code dargestellt, wobei Φ die Aktivierungsfront darstellt:

```
ACTIVATE(Φ) {
    IF (IMPLICATION(Φ) = F)
        RETURN F;
    IF (Φ = ∅)
        RETURN T;
    WHILE (ungetestete Variablenbelegungen existieren)
        Φ' := Φ;
        Wähle ein Signal z aus Φ;
        Wähle eine noch nicht versuchte Belegung der Eingänge des
        zugehörigen Gatters, so dass die Aktivierung von z erfüllt ist;
        Aktualisiere Φ;
```

```
        IF (ACTIVATE(Φ) = T)
            RETURN T;
        Φ := Φ';
    RETURN F;
}
```

Die Funktion IMPLICATION berücksichtigt lokale und globale Implikationen. Lokale Implikationen ergeben sich, wenn ein Gatter-Ausgang mit der activate-Funktion einen Wert zugewiesen bekommen hat und dieser eindeutig auf die Belegung der Eingangsvariablen des zugehörigen Gatters führt. Für das XOR-Gatter am Ausgang der Miter-Schaltung in Abb. 6.12 existieren die in Tabelle 6.3 gezeigten lokalen Implikationen.

Tabelle 6.3. Lokale Implikationen für ein XOR-Gatter [329]

momentane Belegung			Lokal implizierte Belegung		
z	z_3	z_4	z	z_3	z_4
T	F	X	T	F	T
T	X	T	T	F	T
T	T	X	T	T	F
T	X	F	T	T	F
F	X	T	F	T	T
F	T	X	F	T	T
F	X	F	F	F	F
F	F	X	F	F	F
X	F	F	F	F	F
X	T	T	F	T	T
X	F	T	T	F	T
X	T	F	T	T	F

Wie man in Tabelle 6.3 sieht, gibt es für den Fall $z := \mathsf{T}$ und $z_3 = z_4 := X$ keine lokale Implikation. Dies liegt darin begründet, dass es für diesen Fall zwei mögliche Variablenbelegungen an den Eingängen des XOR-Gatters gibt: $z_3 := \mathsf{T}$ und $z_4 := \mathsf{F}$ oder $z_4 := \mathsf{T}$ und $z_3 := \mathsf{F}$. Welche dieser Belegungen gewählt werden soll wird entweder durch eine globale Implikation aufgelöst oder muss durch eine Verzweigung in der ACTIVATE()-Funktion ausprobiert werden. D. h. aber auch, dass es im Allgemeinen mehr als eine Testfalleingabe gibt, welche die Nichtäquivalenz von zwei Funktionen in einer Miter-Schaltung zeigen kann.

Beispiel 6.1.10. Für das Beispiel 6.1.9 ist die Konstruktion aller möglichen Testfalleingaben in Abb. 6.13 zu sehen. Für das AND- und das OR-Gatter gibt es für den zweiten Fall weitere Alternativen, die an die Eingänge übertragen werden. Die beiden möglichen Belegungen der Eingangsvariablen des XOR-Gatters sind als alternative Aktivierungen (durch $|$) dargestellt. Diejenigen Kombinationen, die eine konsistente Belegung der primären Eingänge ergeben, sind in Abb. 6.13 zu sehen.

D. h. in diesem Fall können die vier möglichen Belegungen ($x_1 := \mathsf{F}, x_2 := \mathsf{T}$), ($x_1 := \mathsf{T}, x_2 := \mathsf{T}$), ($x_1 := \mathsf{T}, x_2 := \mathsf{F}$) und ($x_1 := \mathsf{F}, x_2 := \mathsf{F}$) der primären Eingänge, verwendet werden, um die Nichtäquivalenz von f_1 und f_2 simulativ zu zeigen. Die Belegung ist also beliebig. Im Allgemeinen ist dies nicht der Fall.

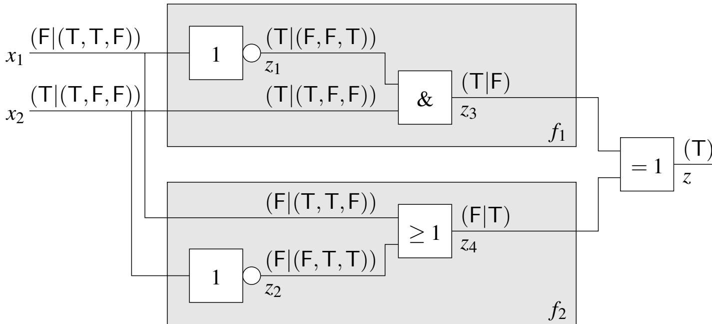


Abb. 6.13. Mögliche Testfalleingaben für die Miter-Schaltung aus Abb. 6.12

Globale Implikationen ergeben sich daraus, dass nicht nur einzelne Gatter getrennt voneinander betrachtet werden, sondern ganze Teilnetze. Dies wird an dem folgenden Beispiel aus [276] illustriert.

Beispiel 6.1.11. Gegeben ist das Boolesche Netzwerk aus Abb. 6.14. Um den Ausgang des OR-Gatters zu erfüllen, gibt es mehrere mögliche Belegungen der Signale z_1 und z_2. Somit kann die Aktivierung nicht allein mit Hilfe einer lokalen Implikation erfolgen.

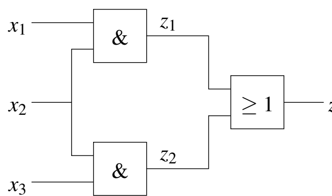


Abb. 6.14. Globale Implikation [276]

Betrachtet man das Boolesche Netzwerk genauer, so erkennt man, dass der primäre Eingang x_2 in allen Fällen, den Wert T zugewiesen bekommen muss. Dies kann erkannt werden, indem allen Signalen außer x_2 der Wert X zugewiesen wird. Belegt man schließlich x_2 mit F, erhält man durch Implikation, dass $z = \mathsf{F}$ und somit nicht erfüllbar ist, d. h. $x_2 = \mathsf{F} \Rightarrow z = \mathsf{F}$. Im Umkehrschluss bedeutet dies:

$$z = \mathsf{T} \Rightarrow x_2 = \mathsf{T}$$

Berücksichtigt man diese globale Implikation, kann der Suchraum für Testfalleingaben deutlich reduziert werden.

6.1.3 Formale explizite Äquivalenzprüfung von Schaltwerken

Hardware-Implementierungen speicherbehafteter Systeme werden als Schaltwerke bezeichnet. Ein Schaltwerk lässt sich als *deterministischer endlicher Automat* (engl. *Deterministic Finite Automaton, DFA*) modellieren (siehe Abschnitt 2.2.2). DFAs verarbeiten Sequenzen von Eingabesymbolen $i \in I$ zu Sequenzen von Ausgabesymbolen $o \in O$. Das aktuelle Ausgabesymbol hängt im Allgemeinen vom aktuellen Eingabesymbol und dem aktuellen Zustand $s \in S$ des DFA ab. Der Folgezustand wird aus dem momentanen Zustand und dem aktuellen Eingabesymbol berechnet. Zu Beginn befindet sich der DFA im Anfangszustand $s_0 \in S$.

Jedes Schaltwerk lässt sich also als DFA repräsentieren. Ein DFA wiederum kann als Schaltwerk modelliert werden. Dazu werden Symbole und Zustände eines DFA als Bitvektoren codiert, da diese eine Eins-zu-Eins-Repräsentation in Hardware besitzen. Solche Schaltwerke lassen sich graphisch ähnlich dem in Abb. 6.15 gezeigtem darstellen. In dem Beispiel wird die Eingabe mit fünf und die Ausgabe mit drei Bits codiert. Die Übergangsfunktion f und die Ausgabefunktion g sind dabei auf der Logikebene als kombinatorische Schaltungen implementiert.

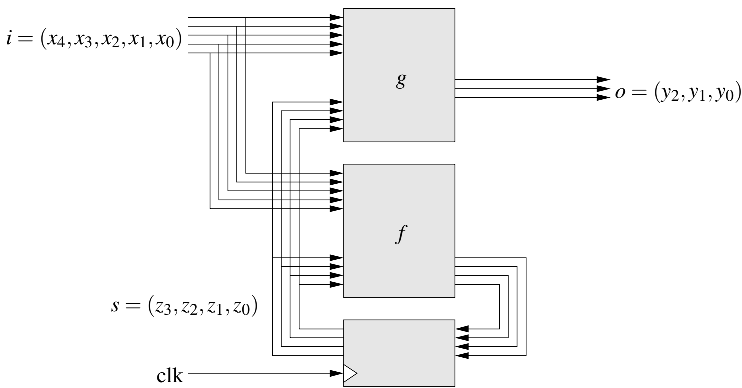


Abb. 6.15. Schaltwerk

Um die Äquivalenz zweier Schaltwerke zu zeigen, kann prinzipiell die sequentielle Äquivalenzprüfung, wie sie in Kapitel 4.3 beschrieben ist, zum Einsatz kommen. Die beiden Schaltwerke werden als DFA modelliert, der Produktautomat gebildet und durch eine Vorwärts- oder Rückwärtstraversierung die Menge der erreichbaren Zustände bestimmt. Enthält diese Menge einen Zustand mit ausgehendem Zu-

standsübergang, der die Ausgabe F erzeugt bzw. enthält diese Menge den Anfangszustand, so sind die beiden Schaltwerke nicht äquivalent.

Da die Übergangsfunktion f und die Ausgabefunktion g als kombinatorische Schaltungen implementiert sind, lassen sich diese durch Boolesche Funktionen und somit durch Boolesche Netzwerke oder ROBDDs repräsentieren. Dies bedeutet, dass die Traversierung auch symbolisch geschehen kann. Ein Nachteil ist allerdings, dass zur Speicherung der Zustände von DFAs Register verwendet werden. Bei n Registern gibt es 2^n mögliche Belegungen, die bei der Konstruktion des Produktautomaten berücksichtigt werden müssen. Dies ist selbst dann der Fall, wenn die Codierung der Zustände nicht alle Belegungen verwendet. Aus diesem Grund wird im Folgenden ein Verfahren basierend auf *symbolischer Simulation* zur formalen expliziten Äquivalenzprüfung von Schaltwerken vorgestellt.

Die sog. *Zeitschrittsimulation* (engl. *frame by frame simulation*) ist das Standardvorgehen bei der symbolischen Simulation von Schaltwerken. In der Zeitschrittsimulation wird der Zustandsraum iterativ exploriert. In jedem Simulationsschritt wird dabei den primären Eingängen und den Zustandsregistern der Schaltung je ein Boolescher Ausdruck zugewiesen. Dies können komplexe Ausdrücke oder konstante Werte sein. Entsprechend der topologischen Ordnung der Gatter der kombinatorischen Schaltungen, welche die Übergangsfunktion und die Ausgabefunktion implementieren, wird den internen Signalen entsprechend der Gattereingänge und der Gatterfunktionalität ein Boolescher Ausdruck zugewiesen, bis die Ausdrücke für die primären Ausgänge und die Registereingänge des Schaltwerks bestimmt sind. Die berechneten Ausdrücke an den Registereingängen stellen die in einem Zeitschritt erreichbaren Zustände dar, ausgehend von den, den Registern zugewiesenen Zuständen. Die Booleschen Ausdrücke für die erreichbaren Zustände werden als „Anfangszustand“ für den folgenden Zeitschritt (engl. *time frame*) verwendet. Den primären Eingängen werden in dem neuen Zeitschritt wieder geeignete Boolesche Ausdrücke zugewiesen. Durch symbolische Simulation der Übergangs- und Ausgabefunktion können die Ausgänge und die möglichen Folgezustände des Schaltwerks im nächsten Zeitschritt ermittelt werden. Dies kann graphisch mit Hilfe des sog. *iterativen Schaltungsmodells* visualisiert werden (siehe Abb. 6.16). Dabei ist in eckigen Klammern der jeweilige Zeitschritt angegeben.

Für eine formale explizite Äquivalenzprüfung auf Basis symbolischer Simulation müssen in jedem Zeitschritt die beiden zu vergleichenden Schaltwerke betrachtet werden und für jeden Zeitschritt eine geeignete Miter-Schaltung aus beiden Ausgabefunktionen gebildet werden. Durch symbolische Simulation, wobei den primären Eingängen die entsprechenden Variablen und den Registern die Anfangszustände zugewiesen werden, können dann die in jedem Zeitschritt erreichten Zustände und die Ausgaben der Miter-Schaltungen bestimmt werden. Liefert die Miter-Schaltung den Wert T zurück, so sind die beiden Schaltwerke nicht äquivalent. Zu prüfen ist außerdem nach jedem Zeitschritt, ob bereits ein Fixpunkt erreicht wurde.

Beispiel 6.1.12. Gegeben sind die beiden Schaltwerke in Abb. 6.17a) und b) (aus [411]). Der momentane Zustand des Schaltwerks M ist in zwei Registern z_1 und z_2 gespeichert, während M' lediglich über ein Zustandsregister z_1' verfügt. Die Aus-

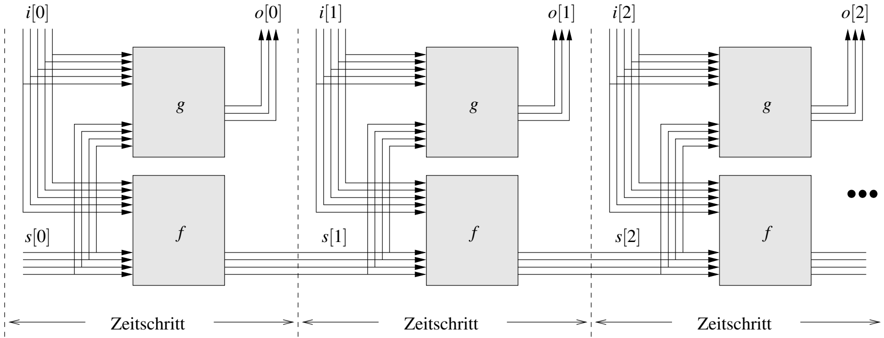


Abb. 6.16. Iteratives Schaltungsmodell des Schaltwerks aus Abb. 6.15

gabefunktionen der beiden Schaltwerke lauten: $f(z_1, z_2) = z_1 \vee z_2$ und $f'(z'_1) = z'_1$. Die Übergangsfunktionen sind wie folgt definiert: $g(x_1, x_2, z_1, z_2) = (x_1 \wedge (z_1 \vee z_2)) \wedge (\neg x_1 \wedge (z_1 \vee z_2))$ und $g'(x'_1, x'_2, z'_1) = (x'_1 \vee z'_1) \wedge (\neg x'_1 \vee x'_2)$.

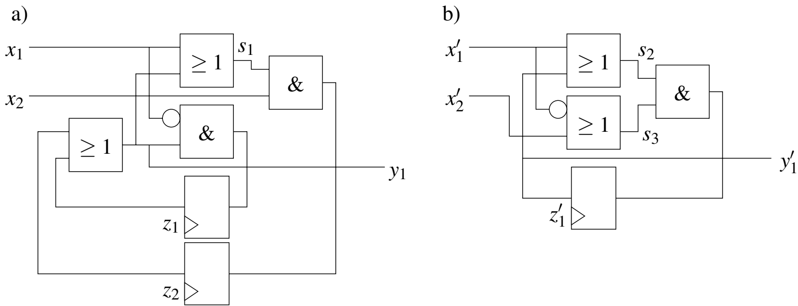


Abb. 6.17. Zwei Schaltwerke a) M und b) M' [411]

Zur Durchführung einer formalen expliziten Äquivalenzprüfung mit symbolischer Simulation muss aus der Miter-Schaltung von M und M' das iterative Schaltungsmodell entwickelt werden. Dies ist in Abb. 6.18 für die ersten drei Zeitschritte zu sehen.

Der Anfangszustand des Schaltwerks M ist durch $z_1[0] = z_2[0] := \mathsf{F}$ gegeben. Der Anfangszustand des Schaltwerks M' ist durch $z'_1[0] := \mathsf{F}$ gegeben. Für die Bestimmung des Fixpunktes durch Vorwärtstraversierung (siehe Kapitel 4.3) ergibt sich die Menge der erreichbaren Zustände somit zu $S_R[0] = \{(\mathsf{F}, \mathsf{F}, \mathsf{F})\}$, wobei der Zustandsvektor (z_1, z_2, z'_1) zugrunde gelegt wurde. Die symbolische Simulation erfolgt als Zeitschrittsimulation:

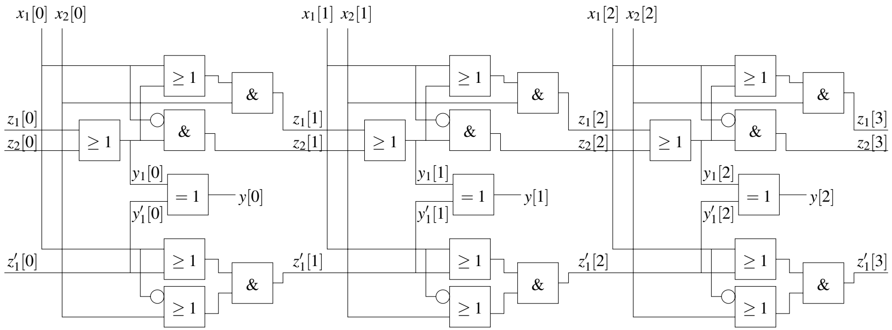


Abb. 6.18. Iteratives Schaltungsmodell für drei Zeitschritte der Miter-Schaltung von M und M' aus Abb. 6.17

1. Durch Zuweisung dieser Initialwerte an die Zustände und Zuweisung der symbolischen Werte $x_1[0]$ und $x_2[0]$ an die primären Eingänge der Schaltwerke erhält man nach der symbolischen Simulation des ersten Zeitschritts die Variablenbelegungen:

$$\begin{aligned} y_1[0] &= \mathsf{F} \\ y_1'[0] &= \mathsf{F} \\ z_1[1] &= x_1[0] \wedge x_2[0] \\ z_2[1] &= \mathsf{F} \\ z_1'[1] &= x_1[0] \wedge x_2[0] \end{aligned}$$

   Hieraus ergibt sich als Ausgabe der Miter-Schaltung nach symbolischer Simulation des ersten Zeitschritts $y[0] = \mathsf{F}$. Die Menge der erreichbaren Zustände $S_R[1]$ nach einem Zeitschritt ergibt sich zu $S_R[1] = \{(\mathsf{F},\mathsf{F},\mathsf{F}),(\mathsf{T},\mathsf{F},\mathsf{T})\}$.
2. Mit den symbolischen Werten $z_1[1]$, F und $z_1'[1]$ sowie den symbolischen Eingabewerten $x_1[1]$ und $x_2[1]$ kann der zweite Zeitschritt symbolisch simuliert werden. Die Variablenbelegungen ergeben sich zu:

$$\begin{aligned} y_1[1] &= z_1[1] \\ y_1'[1] &= z_1'[1] \\ z_1[2] &= (x_1[1] \vee z_1[1]) \wedge x_2[1] \\ z_2[2] &= \neg x_1[1] \wedge z_1[1] \\ z_1'[2] &= (x_1[1] \vee z_1'[1]) \wedge (\neg x_1[1] \vee x_2[1]) \end{aligned}$$

   Nach Substitution der im ersten Zeitschritt ermittelten Ausdrücke für $z_1[1]$ und $z_1'[1]$ ergibt sich somit $y[1] = \mathsf{F}$. Die Menge der erreichbaren Zustände nach zwei Zeitschritten ergibt sich zu $S_R[2] = \{(\mathsf{F},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{T},\mathsf{T}),(\mathsf{T},\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{T},\mathsf{T})\}$. Dabei muss beachtet werden, dass die Belegung der symbolischen Variablen $z_1[1]$

und $z'[1]$ nur entsprechend der im ersten Zeitschritt ermittelten Ausdrücke erfolgen darf, d. h. nur die in $S_R[1]$ enthaltenen Zuständen werden als Variablenbelegung berücksichtigt.

3. Mit den symbolischen Werten $z_1[2], z_2[2], z'_1[2]$ sowie den symbolischen Eingabewerten $x_1[2]$ und $x_2[2]$ kann der dritte Zeitschritt simuliert werden. Die Variablenbelegungen ergeben sich zu:

$$\begin{aligned} y_1[2] &= z_1[2] \vee z_2[2] \\ &= (x_1[1] \wedge x_2[1]) \vee (x_2[1] \wedge z_1[1]) \vee (\neg x_1[1] \wedge z_1[1]) \\ y'_1[2] &= z'_1[2] \\ &= (x_1[1] \wedge x_2[1]) \vee (x_2[1] \wedge z'_1[1]) \vee (\neg x_1[1] \wedge z'_1[1]) \\ z_1[3] &= (x_1[2] \vee z_1[2] \vee z_2[2]) \wedge x_2[2] \\ z_2[3] &= \neg x_1[2] \wedge (z_1[2] \vee z_2[2]) \\ z'_1[3] &= (x_1[2] \vee z'_1[2]) \wedge (\neg x_1[2] \vee x_2[2]) \end{aligned}$$

Für die Ausgabe der Miter-Schaltung nach symbolischer Simulation des dritten Zeitschritts ergibt sich $y[2] = \mathsf{F}$. Die Menge der erreichbaren Zustände $S_R[3]$ nach drei Zeitschritten ergibt sich zu $S_R[3] = \{(\mathsf{F},\mathsf{F},\mathsf{F}),(\mathsf{F},\mathsf{T},\mathsf{T}),(\mathsf{T},\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{T},\mathsf{T})\}$.

Bei der Bestimmung von $S_R[3]$ muss wiederum die Erreichbarkeitsmenge $S_R[2]$ berücksichtigt werden. An dieser Stelle sieht man, dass $S_R[3] = S_R[2]$ gilt. Somit ist ein Fixpunkt erreicht und die Menge der erreichbaren Zustände bestimmt. Da weiterhin gilt, dass in in jedem Zeitschritt i der Ausgang der Miter-Schaltung $y[i] = \mathsf{F}$ ist, sind die beiden Schaltwerke M und M' aus Abb. 6.17 äquivalent.

In dem Beispiel wurde in jedem Zeitschritt i mit den neuen symbolischen Werten $z_1[i], z_2[i]$ und $z'_1[i]$ weitergerechnet, um sehr große Boolesche Formeln zu vermeiden. Für die Auswertung der Erreichbarkeitsmenge müssen allerdings diese symbolischen Werte durch die berechneten Booleschen Formeln substituiert werden, um die bisher erreichten Zustände zu berücksichtigen. Dies wurde im Beispiel umgangen, indem die Mengen $S_R[i]$ immer explizit aus den bereits erreichten Zuständen bestimmt wurden. Diese explizite Darstellung der Erreichbarkeitsmenge ist für große Systeme mit vielen Zuständen ungeeignet, weshalb in der Regel eine implizite Darstellung z. B. durch ROBDDs vorgenommen wird.

Das hier vorgestellte Verfahren passt die Verfahren zur Prüfung von Automatenäquivalenz aus Abschnitt 4.3 auf das iterative Schaltungsmodell an. Allerdings stimmt die Fixpunktberechnung nicht mehr exakt mit der aus Abschnitt 4.3 überein: Zur Prüfung zweier Automaten auf Äquivalenz wird dort der Zustandsraum des Produktautomaten traversiert, bis entweder ein Zustand erreicht wird, in dem die beiden Automaten unterschiedliches Verhalten zeigen, oder bis ein Fixpunkt erreicht ist, d. h. keine zusätzlichen Zustände besucht werden können. Im letzteren Fall gilt $S_R[t-1] = S_R[t]$. Wird dieser Fixpunkt erreicht, ohne dass ein unterschiedliches Verhalten der beiden Automaten aufgetreten ist, so sind diese äquivalent.

Erfolgt der Vergleich auf Basis des iterativen Schaltungsmodells der Miter-Schaltung zweier Schaltwerke durch eine symbolische Zeitschrittsimulation, wird

während der symbolischen Simulation die Erreichbarkeitsmenge $S_R[t]$ für die *in Zeitschritt t erreichbaren Zustände* berechnet. Man beachte den Unterschied der beiden Definitionen von Erreichbarkeitsmengen: Während bei den Verfahren zur Prüfung von Automatenäquivalenz aus Kapitel 4.3 alle *während t Zeitschritten erreichbaren Zustände* berechnet werden ($S_R[t] := S_R[t-1] \cup SUCC(S_R[t-1])$), wird bei der symbolischen Simulation der Miter-Schaltung lediglich die Menge an Zuständen bestimmt, in der die Schaltwerke nach *genau t* Zeitschritten sein können ($S_R[t] := SUCC(S_R[t-1])$). $SUCC(S)$ ist dabei jeweils die Menge der aus der Menge S von Zuständen in genau einem Schritt erreichbaren Zuständen. Um den Fixpunkt bei der symbolischen Simulation erkennen zu können, muss somit vor der Simulation eines neuen Zeitschritts die Menge aller bisher erreichten Zustände bestimmt und mit der vorherigen verglichen werden.

6.1.4 Strukturelle Äquivalenzprüfung auf der Logikebene

Die implizite kombinatorische Äquivalenzprüfung auf der Logikebene mit ROBDDs oder ROKFDDs kann aufgrund des hohen Speicherbedarfs oftmals nicht für große kombinatorische Schaltungen durchgeführt werden. Auf der anderen Seite ist die formale explizite kombinatorische Äquivalenzprüfung für große Systeme aufgrund der großen Laufzeit auch nicht anwendbar.

Zusammenfassend kann man sagen, dass sich die formalen Methoden lediglich auf kleine bis mittlere Probleminstanzen anwenden lassen. Um diese Verfahren auch für größere Systeme zu verwenden, besteht die Möglichkeit, diese mit *strukturellen Verfahren* zu kombinieren. Im Folgenden werden zunächst wieder nur Schaltnetze, also Systeme ohne Speicher betrachtet. Anschließend wird ein strukturelles Verfahren für Schaltwerke vorgestellt.

Strukturelle Äquivalenzprüfung auf Basis von Schaltungspartitionierung

Für eine explizite strukturelle Äquivalenzprüfung ist der Ausgangspunkt die Miter-Schaltung zweier kombinatorischer Schaltkreise. Beginnend von den Eingängen, werden in beiden Schaltungen äquivalente (interne) Signale gesucht. Dies kann durch explizite oder implizite Verfahren erfolgen. Sind äquivalente Signale gefunden worden, kann das Signal der einen Schaltung durch das Signal der anderen Schaltung substituiert werden (vgl. auch Kapitel 4.4). Anschließend werden weitere äquivalente Signale gesucht und substituiert, bis die Erfüllbarkeit des Ausgangs der Miter-Schaltung gezeigt oder widerlegt ist.

Alternativ kann die Schaltung entlang eines Schnittes partitioniert werden. Zunächst muss dann die Äquivalenz der Signale an den Schnittpunkten gezeigt werden. Anschließend werden die Schnittpunkte als weitere primäre Schaltungseingänge betrachtet und die Erfüllbarkeit des Miter-Ausgang überprüft.

Beispiel 6.1.13. Für die beiden kombinatorischen Schaltungen aus Abb. 6.3 auf Seite 238 wurde bereits in Beispiel 6.1.2 mittels ROBDDs deren Äquivalenz bewiesen. Die Verwendung der strukturellen Äquivalenzprüfung auf Basis von Partitionierung

wird nun an diesem Beispiel gezeigt. Abbildung 6.19 zeigt die Miter-Schaltung der beiden kombinatorischen Schaltungen, sowie einen Schnitt durch beide Systeme. Der Schnitt basiert auf der Annahme, dass $z_5 \equiv z_5'$, $z_6 \equiv z_6'$ und $z_7 \equiv z_7'$.

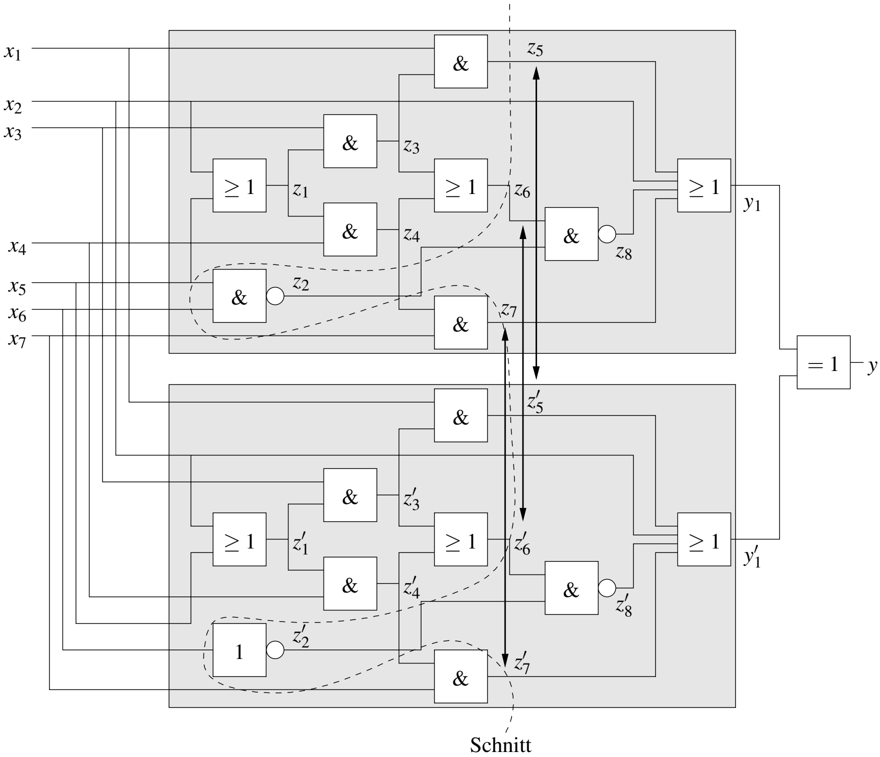


Abb. 6.19. Strukturelle Äquivalenzprüfung der beiden Schaltungen aus Abb. 6.3

Durch Betrachtung der Teilschaltungen vor den Schnittpunkten ergibt sich, dass in der Tat die oben genannten Äquivalenzen gelten. Entsprechend dem Partitionierungsansatz muss deshalb im folgenden Schritt gezeigt werden, dass die beiden Teilschaltungen hinter den Schnittpunkten äquivalent sind bzw. der Ausgang y der Miter-Schaltung nicht erfüllbar ist. Gelingt dies, so ist bewiesen, dass die beiden Schaltungen äquivalent sind.

Hierfür werden zunächst für beide Teilschaltungen ROBDDs mit der selben Variablenordnung $x_5 < x_6 < z_1 < z_2 < x_2 < z_3$ entwickelt. Die ROBDDs sind in Abb. 6.20b) und c) zu sehen. Man erkennt, dass die ROBDDs für y_1 und y_2 nicht isomorph sind und somit die beiden Teilschaltungen (dargestellt in Abb. 6.20a)) nicht äquivalent sind. Dies kann zusätzlich am ROBDD für den Miter-Ausgang y, dargestellt in Abb. 6.20d), gesehen werden. Dieses müsste bei Äquivalenz den konstanten Wert F repräsentieren, damit der Ausgang nie erfüllbar ist.

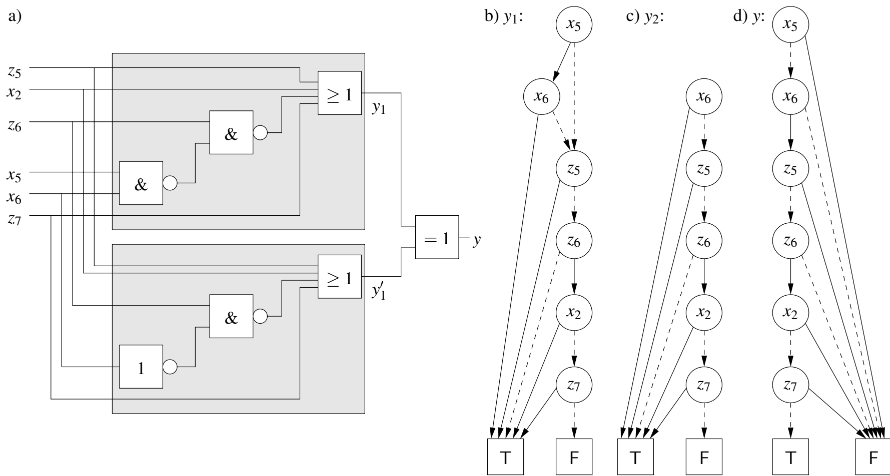


Abb. 6.20. ROBDDs für die Signale b) y_1 c) y_2 und d) y der Miter-Schaltung aus a)

Obwohl also die beiden kombinatorischen Schaltungen aus Abb. 6.3 äquivalent sind, kann dies nicht direkt durch eine strukturelle Äquivalenzprüfung auf Basis von Schaltungspartitionierung gezeigt werden. Das ROBDD für den Miter-Ausgang y der Teilschaltungen aus Abb. 6.20a) liefert eine Belegung, die den Ausgang erfüllt. Diese Belegung kann aus dem ROBDD als Pfad zum Terminalknoten T ausgelesen werden, d. h. $x_5 = z_5 = x_2 = z_7 := \mathsf{F}$ und $x_6 = z_6 := \mathsf{T}$. Allerdings muss, wie in Kapitel 4.4, noch überprüft werden, ob diese Belegung im Gesamtsystem überhaupt auftreten kann. Da bereits bekannt ist, dass die beiden Schaltungen äquivalent sind, muss das Ergebnis dieser Überprüfung sein, dass diese Belegung nicht erreicht werden kann.

Beispiel 6.1.14. Entsprechend dem einzigen Pfad zu dem Terminalknoten mit dem Wert T in Abb. 6.20d) werden die Eingänge x_2 und x_5 mit dem Wert F und der Eingang x_6 mit dem Wert T belegt. Die internen Signale z_5, z_6 und z_7 werden ebenfalls entsprechend dem Pfad belegt.

Die Werte für die internen Signale, die sich aus den Eingabewerten ergeben, sind in Abb. 6.21 zu sehen. Durch diese Belegung ergibt sich für die Signale z_1 und z_3 der Wert F. Hierdurch ergibt sich auch der Wert F für das Signal z_4. Mit $z_3 = z_4 = \mathsf{F}$ ergibt sich $z_6 = z_3 \vee z_4 = \mathsf{F}$. Dies steht aber in Konflikt zu dem zugewiesenen Wert $z_6 := \mathsf{T}$. Aus diesem Konflikt kann man schließen, dass die Variablenbelegung, welche die Miter-Schaltung in Abb. 6.20a) erfüllt, im unpartitionierten System nicht erreichbar ist. Dies bedeutet, dass es sich bei dem Ergebnis der strukturellen Äquivalenzprüfung auf Basis der Schaltungspartitionierung um ein *falschnegatives Ergebnis* handelt, also ein Gegenbeispiel für die Äquivalenz der beiden Systeme aus Abb. 6.19, welches aber falsch ist.

Das Auftreten von falschnegativen Ergebnissen kann zumindest teilweise vermieden werden, wenn die Schaltung in überlappende Partitionen zerlegt wird. Hier-

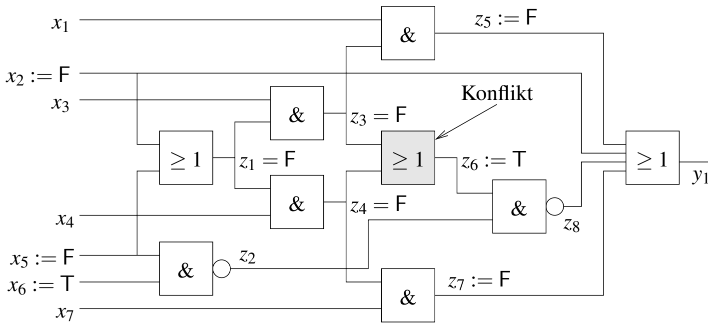


Abb. 6.21. Konflikt durch falschnegatives Ergebnis im unpartitionierten System

durch kann der steuerbare Wertebereich der Variablen an den Schnittpunkten eingeschränkt werden.

Strukturelle sequentielle Äquivalenzprüfung

In Abschnitt 6.1.3 wurde eine formale Methode zur Äquivalenzprüfung von zwei Schaltwerken auf Basis des iterativen Schaltungsmodells vorgestellt. Für eine strukturelle Äquivalenzprüfung von zwei Schaltwerken bietet sich die Verwendung des iterativen Schaltungsmodells ebenfalls an, da bei einem solchen Vorgehen die Ergebnisse für die strukturelle Äquivalenzprüfung von kombinatorischen Schaltungen wiederverwendet werden können.

Bei der formalen Äquivalenzprüfung von Schaltwerken wurden die Verfahren aus Abschnitt 4.3 zur Prüfung von Automatenäquivalenz auf das iterative Schaltungsmodell einer Miter-Schaltung angepasst. Ein Problem war allerdings die unterschiedliche Bestimmung der Erreichbarkeitsmengen. Wie in Abschnitt 6.1.3 dargestellt, wird bei der symbolischen Simulation die Erreichbarkeitsmenge für jeden Zeitschritt explizit bestimmt. Ein solches Vorgehen schränkt jedoch das Einsatzgebiet struktureller Prüfungsverfahren auf einen einzelnen Zeitschritt ein. Alternativ kann das iterative Schaltungsmodell der Miter-Schaltung bis zur sequentiellen Tiefe des zugehörigen Automaten aufgebaut werden. In diesem Fall ist sichergestellt, dass der Fixpunkt erreicht wird. Dies wird allerdings in der Regel zu sehr großen und somit nicht handhabbaren Probleminstanzen führen.

Sollen mehrere Zeitschritte bei der strukturellen Äquivalenzprüfung gleichzeitig Berücksichtigung finden, ohne das iterative Schaltungsmodell für die gesamte sequentielle Tiefe aufzubauen, muss der Fixpunkt aus der Menge der in genau t Zeitschritten erreichbaren Zustände ermittelbar sein. Dieser Fixpunkt kann im Allgemeinen allerdings nicht nur aus zwei aufeinander folgenden Erreichbarkeitsmengen bestimmt werden. Die Erreichbarkeitsmengen werden allerdings immer nach einer Zeit $t \geq t_{\text{fix}}$ anfangen, mit einer Periode T zu oszillieren, d. h.:

$$\forall t \geq t_{\text{fix}} : S_R[t] = S_R[t-T]$$

Hieraus folgt, dass der Fixpunkt detektiert werden kann, wenn gilt:

$$\exists T : S_R[t] = S_R[t-T]$$

Die Frage bleibt, wie das Auftreten der Periode T im iterativen Schaltungsmodell detektiert werden kann. Während bei der Automaten-Äquivalenzprüfung lediglich die ROBDDs der Mengen $S_R[t]$ und $S_R[t-1]$ auf Isomorphie überprüft werden, muss beim iterativen Schaltungsmodell überprüft werden, ob für zwei Zeitpunkte t und $t-T$ die Wertebereiche der durch die kombinatorischen Schaltungen implementierten Booleschen Funktionen äquivalent sind. Dabei müssen eventuell Schaltungen mit mehreren Ausgängen berücksichtigt werden. Hierfür schlagen Stoffel et al. [412] eine strukturelle Form der Existenz-Quantifizierung vor.

Die strukturelle Existenz-Quantifizierung beruht auf der funktionalen Dekomposition des iterativen Schaltungsmodells, dargestellt in Abb. 6.22. Abbildung 6.22a) zeigt das iterative Schaltungsmodell eines Schaltwerks für t Zeitschritte. Dabei wurde die Notation f^+ für die erweiterte Übergangsfunktion verwendet (siehe Kapitel 4.3). Diese bestimmt aus einem Anfangszustand $s[0]$ und einer Sequenz w der Länge t von Eingabesymbolen den Folgezustand $s[t]$ nach t Zeitschritten. Durch symbolische Simulation des iterativen Schaltungsmodells erhält man die Menge $S_R[t]$ von erreichbaren Zuständen in genau t Zeitschritten. $f^+[t]$ ist dabei die Boolesche Funktion, die durch das iterative Schaltungsmodell implementiert ist, und die für beliebige Sequenzen w der Länge t den Folgezustand berechnet.

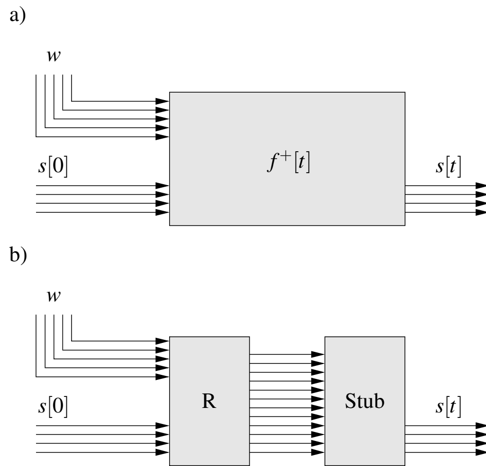


Abb. 6.22. Strukturelle Form der Existenz-Quantifizierung [412]

Die interessante Frage lautet nun, welche Zustände in $S_R[t]$ sind. Wie diese berechnet werden ist dabei nicht von Interesse. Mit anderen Worten: Wir sind lediglich an dem Wertebereich von $f^+[t]$ interessiert. Aus diesem Grund kann das iterative

Schaltungsmodell in zwei Teile zerlegt werden [411]: Eine sog. *Stub-Schaltung* Stub und eine sog. *Restschaltung* R. Dies ist in Abb. 6.22b) dargestellt.

Die von der Stub-Schaltung implementierte Funktion ist dabei *wertebereichäquivalent* zur Funktion des iterativen Schaltungsmodells, also der Funktion $f^+[t]$. Zwei Funktionen heißen wertebereichäquivalent, wenn diese die selbe Wertemenge besitzen. Dabei können die beiden Funktionen durchaus unterschiedliche Definitionsbereiche haben. Dies bedeutet, dass die Stub-Schaltung exakt die selben Werte produzieren kann, wie das iterative Schaltungsmodell. Wird nun die Restschaltung R aus der Schaltung herausgenommen, so ist der Zusammenhang zwischen Anfangszustand und Eingabe zu den erreichbaren Zuständen entfernt. Allerdings bleibt die Erreichbarkeitsmenge weiterhin durch die Stub-Schaltung repräsentiert. Somit wird eine strukturelle Existenz-Quantifizierung der Funktion $f^+[t]$ bezüglich der Eingabe w und des Anfangszustand $s[0]$ durchgeführt.

Während einer Zeitschrittsimulation des iterativen Schaltungsmodells kann die Dekomposition und das Heraustrennen der Restschaltung in jedem Zeitschritt erfolgen. Dies entspricht exakt der Quantifizierung bei der Automatentraversierung. Aber im Gegensatz zur symbolischen Simulation in Abschnitt 6.1.3 wird die Schaltung für einen neuen Zeitschritt nicht an das komplette bis dahin erstellte iterative Schaltungsmodell angehängt, sondern lediglich an die ermittelte Stub-Schaltung. Dies ist in Abb. 6.23a) zu sehen. Nach Anhängen der Schaltung für den neuen Zeitschritt (im Beispiel wird die kombinatorische Schaltung für die Funktion f angehängt), kann wiederum die funktionale Dekomposition der Schaltung in eine Restschaltung R und eine Stub-Schaltung vorgenommen werden.

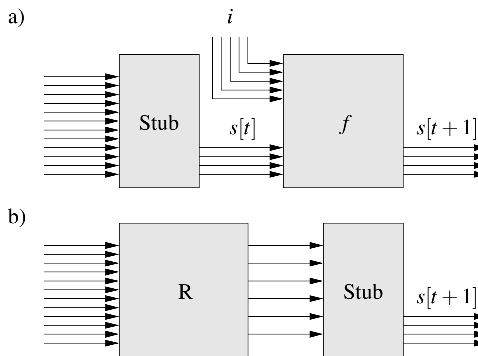


Abb. 6.23. Heraustrennung der Restschaltung vor dem Anhängen des nächsten Zeitschritts [412]

Neben der Reduktion der Schaltungsgröße wird dieses Vorgehen auch zur Detektion des Fixpunktes verwendet [412]. Falls $S_R[t]$ und $S_R[t+T]$ gleich sind, dann muss es auch möglich sein, die iterativen Schaltungsmodelle der Länge t und der Länge $t+T$ so zu zerlegen, dass wertebereichäquivalente Stub-Schaltungen entste-

hen. Der Fixpunkt wird somit bestimmt, indem die Identität der Dekompositionsschritte für verschiedene Zeitschritte gezeigt wird. Allerdings ist die Konstruktion wertebereichäquivalenter Stub-Schaltungen im Allgemeinen ein schweres Problem und eine exakte Bestimmung, auch wenn möglich, für realistische Probleminstanzen nicht anwendbar. Aus diesem Grund werden in der Regel anwendungsspezifische Approximationen verwendet. Im Folgenden wird dies für den Fall der sequentiellen Äquivalenzprüfung gezeigt.

Der Record&Play-Algorithmus

Im Folgenden wird die formale sequentielle Äquivalenzprüfung zweier Schaltwerke betrachtet. Ausgangspunkt ist dabei das iterative Schaltungsmodell der Miter-Schaltung der beiden Schaltwerke.

Der Fixpunkt kann durch identische Dekomposition des iterativen Schaltungsmodells für zwei verschiedene Zeitschritte detektiert werden. Die Dekomposition beruht auf Verfahren, wie sie bereits für die strukturelle kombinatorische Äquivalenzprüfung vorgestellt wurden. Die Dekompositionsschritte werden dabei in sog. *Instruktionsqueues* Q mit einer FIFO-Strategie (engl. *First In, First Out*) gespeichert. In $Q[t]$ stehen diejenigen Instruktionen, die zu Zeitschritt t durchgeführt werden, um die Stub-Schaltung zu generieren. Die Speicherung dieser Instruktionen wird als engl. *record* bezeichnet.

Im darauf folgenden Zeitschritt $t+1$ hofft man, von bereits aufgezeichneten Informationen zu profitieren, indem versucht wird, die selben Instruktionen aus einer der vorherigen Instruktionsqueues wieder anzuwenden. Falls dies möglich ist, wird die anwendbare Instruktion auch in der Instruktionsqueue $Q[t+1]$ aufgenommen. Das Ausprobieren der Instruktionen einer Instruktionsqueue wird als engl. *play* bezeichnet. Falls die gespeicherten Instruktionen nicht ausreichen, um die Stub-Schaltung im Zeitschritt $t+1$ zu erzeugen, werden weitere Äquivalenzen gesucht und die resultierenden Schaltungstransformationen aufgezeichnet.

Durch das ständige Aufzeichnen und Abspielen der Instruktionen wird neue relevante Information in jeder neuen Instruktionsqueue gespeichert. Zu einem Zeitpunkt wird es schließlich möglich sein, eine Instruktionsqueue aus einem der vorherigen Zeitschritte komplett abzuspielen und hierdurch eine Stub-Schaltung zu erzeugen. In diesem Fall ist der Fixpunkt detektiert.

Der hieraus resultierende *Record&Play-Algorithmus* wurde erstmals in [411] vorgestellt und wird im Folgenden diskutiert. Er ist generell für eine Oszillation mit einer Periode T einsetzbar. Um die Beschreibung übersichtlich zu halten, wird hier lediglich der Fall $T = 1$ beschrieben. Der Record&Play-Algorithmus ist in Abb. 6.24 dargestellt.

Zu Beginn wird die Zeit auf $t := 0$ gesetzt und danach der iterative Teil ausgeführt. Zunächst wird jeweils die Schaltung für einen neuen Zeitschritt angehängt und das Abspielen und Aufnehmen alter und neuer Instruktionen durchgeführt. Ist dies abgeschlossen, wird geprüft, ob die Schaltungsausgänge äquivalent sind. Falls die Ausgänge nicht äquivalent sind, so sind auch die beiden Schaltwerke nicht äquivalent. Falls sie äquivalent sind, wird überprüft, ob während der Play-Phase die In-

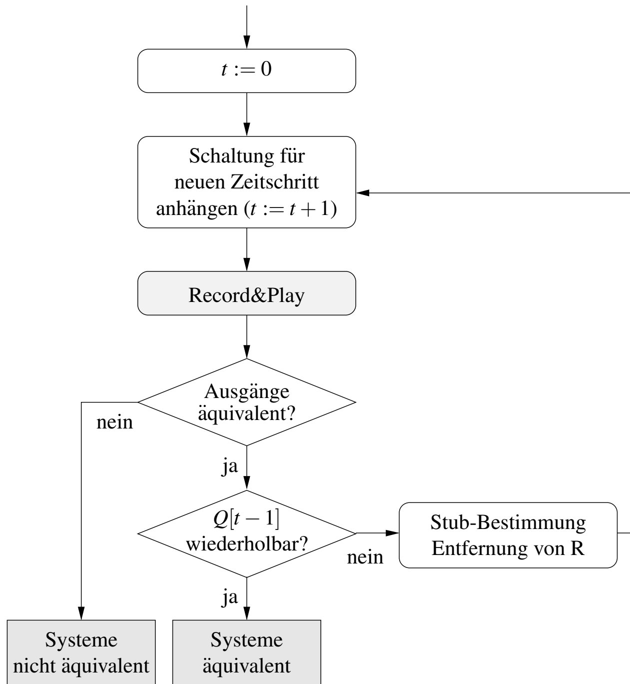


Abb. 6.24. Record&Play-Algorithmus

struktionsqueue abgespielt werden konnte. Ist dies der Fall, sind die Schaltwerke äquivalent. Andernfalls muss eine neue Stub-Schaltung generiert werden und die Restschaltung abgetrennt werden. Anschließend wird ein neuer Zeitschritt begonnen.

Beispiel 6.1.15. Der Record&Play-Algorithmus soll zum Vergleich der Schaltwerke aus Abb. 6.17 auf Seite 260 verwendet werden. Als Anfangszustand wird den drei Registern in beiden Schaltwerken der Wert F zugewiesen. Hierdurch vereinfacht sich die Miter-Schaltung für den ersten Zeitschritt. Dies ist in Abb. 6.25a) zu sehen.

Die Play-Phase des Record&Play-Aufrufs in diesem Zeitschritt wird übersprungen, da $Q[0] = \langle\rangle$ eine leere Sequenz ist. In der Record-Phase lässt sich für diese Schaltung ein Schnittpunkt für die äquivalenten Signale $z_1[1]$ und $z_1'[1]$ bestimmen. Zur Schaltungsreduktion wird das Signal $z_1[1]$ in das Schaltwerk M' übertragen. Die Transformation wird in der Instruktionsqueue

$$Q[1] = \langle(z_1[t] \mapsto z_1'[t])\rangle$$

gespeichert, wobei der Ausdruck $x \mapsto y$ die Substitution des Signals y durch das Signal x beschreibt. Anschließend werden die Ausgänge der Miter-Schaltung auf Erfüllbarkeit überprüft. Dies ist hier nicht möglich. Da in der Play-Phase keine vor-

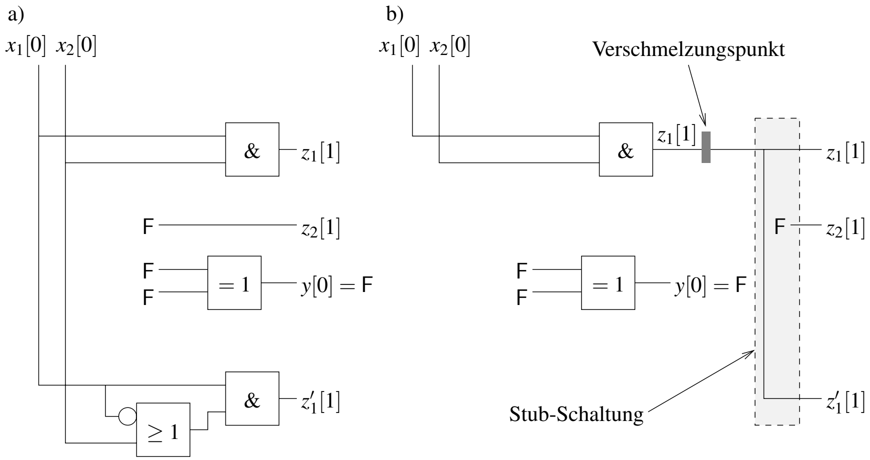


Abb. 6.25. a) Iteratives Schaltungsmodell für den ersten Zeitschritt und b) generierte Stub-Schaltung [412]

herige Instruktionsqueue abgespielt werden konnte, wird die Stub-Schaltung für die nächste Iteration extrahiert.

Hierfür werden die sog. *Verschmelzungspunkte* (engl. *merge points*) bestimmt. Anschaulich handelt es sich dabei um diejenigen Signale, welche die Registerwerte beider Schaltwerke beeinflussen. Man kann die Verschmelzungspunkte bestimmen, indem man von den Zustandsvariablen in der Schaltung rückwärts Signale verfolgt und abbricht, sobald die Pfade der Zustandsvariablen auf gemeinsame Variablen zurückgeführt sind. Die Stub-Schaltung ist dann diejenige Schaltung, die hinter den Verschmelzungspunkten liegt (siehe Abb. 6.25b)). Man beachte, dass es durch die Schaltungspartitionierung zu falschnegativen Ergebnissen kommen kann, was einer Überapproximation der Erreichbarkeitsmenge entspricht.

In der nächsten Iteration wird das Schaltungsmodell für den folgenden Zeitschritt an die zuvor generierte Stub-Schaltung, die die möglichen Anfangszustände für diesen Zeitschritt festlegt, angehängt. Die resultierende Schaltung ist in Abb. 6.26a) zu sehen. Das Abspielen der Instruktionsqueue $Q[1]$ ist für die resultierende Schaltung nicht möglich. Aus diesem Grund wird in der Schaltung nach strukturellen Äquivalenzen gesucht. Diese sind für die Signale $s_1[1]$ und $s_2[1]$ gegeben. Die zugehörige Instruktion in der Instruktionsqueue lautet deshalb:

$$Q[2] = \langle (s_1[t] \mapsto s_2[t]) \rangle$$

Anschließend erfolgt die Überprüfung des Miter-Ausgangs, der nicht erfüllbar ist. Da die vorherige leere Instruktionsqueue nicht abgespielt werden konnte, muss eine neue Stub-Schaltung generiert werden. Die Verschmelzungspunkte liegen diesmal an den Schaltungseingängen, weshalb die gesamte Schaltung als Stub-Schaltung aus-

gewählt wird (Abb. 6.26b)). Allerdings führt dies zu keiner weiteren Reduktion der Schaltung.

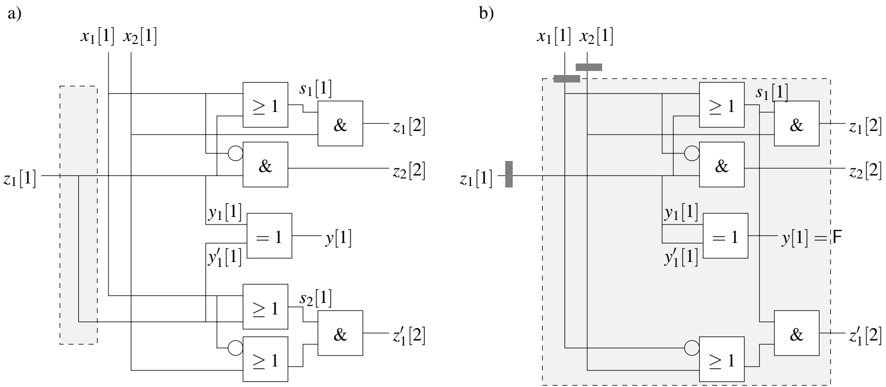


Abb. 6.26. a) Iteratives Schaltungsmodell für den zweiten Zeitschritt und b) generierte Stub-Schaltung [412]

Nun folgt das Anhängen des Schaltungsmodells für den dritten Zeitschritt. Dies ist in Abb. 6.27 dargestellt.

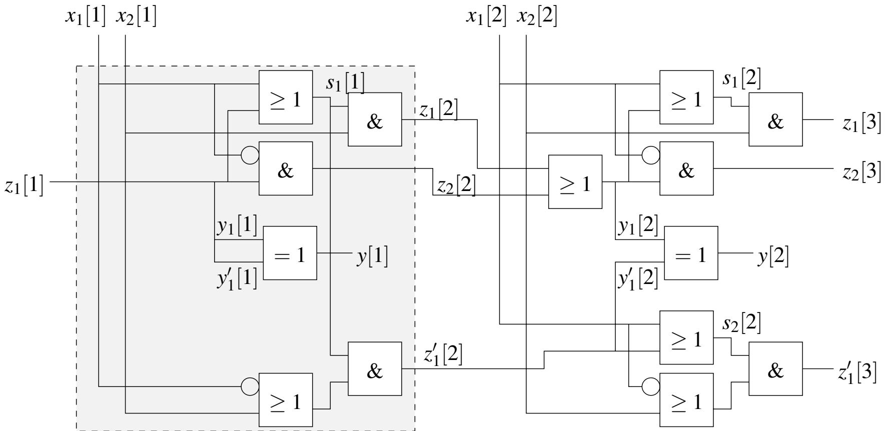


Abb. 6.27. Iteratives Schaltungsmodell für den dritten Zeitschritt [412]

Der Ausgang der Miter-Schaltung ist hier nach wie vor nicht erfüllbar. Durch Abspielen der Instruktionsqueue $Q[2]$ kann strukturell noch nicht die Äquivalenz der

Ausgänge der Schaltwerke gezeigt werden. Hierzu muss zusätzlich der Fall betrachtet werden, dass $y_1[2]$ äquivalent mit $z'_1[2]$ ist. Die neue Instruktionsqueue ergibt sich zu:

$$Q[3] = \langle (y_1[t] \mapsto z'_1[t]), (s_1[t] \mapsto s_2[t]) \rangle$$

Die Stub-Schaltung ergibt sich durch Bestimmung der Verschmelzungspunkte und ist in Abb. 6.28 zu sehen.

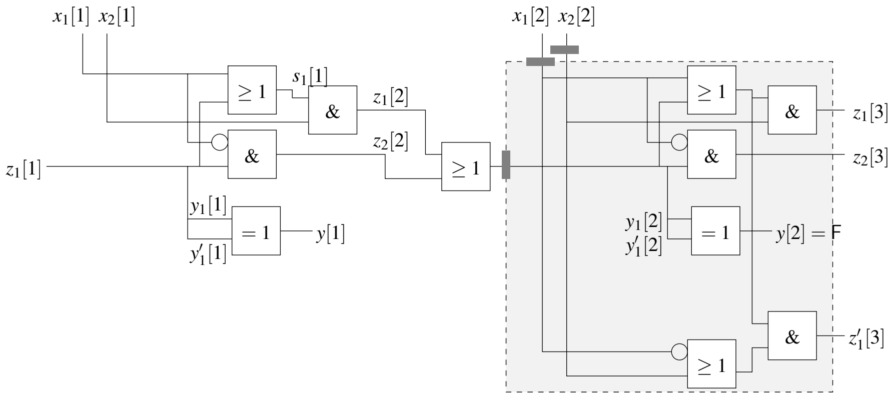


Abb. 6.28. Stub-Schaltung für den dritten Zeitschritt [412]

Man erkennt, dass die im zweiten und dritten Schritt ermittelte Stub-Schaltung identisch ist. Aus diesem Grund wird durch Abspielen von $Q[3]$ nach Anhängen des Schaltungsmodells für den vierten Zeitschritt Äquivalenz erkannt. Da somit der Fixpunkt detektiert ist, und in keinem Zeitschritt der Miter-Ausgang erfüllbar war, sind die beiden Schaltwerke äquivalent.

6.2 Äquivalenzprüfung arithmetischer Schaltungen

Im Folgenden werden Verfahren für die Architekturebene präsentiert, die zur Äquivalenzprüfung arithmetischer Schaltungen geeignet sind. Diese eignen sich weiterhin auch zur Äquivalenzprüfung zwischen Modellen auf Architektur- und Logikebene.

6.2.1 Implizite Äquivalenzprüfung auf der Architekturebene

Auf der Architekturebene (siehe Abb. 6.1 auf Seite 235) ist die Granularität der Operationen in der Verhaltensspezifikation oftmals durch arithmetische Funktionen spezifiziert. Entsprechend sind die Objekte der Strukturmodelle der Implementierung Addierer, Multiplizierer, ALUs (engl. *Arithmetic Logical Units*) etc. In der Synthese werden die arithmetischen Funktionen auf eine Architektur bestehend aus einem

Datenpfad und einem Steuerwerk abgebildet. Ganzzahlige arithmetische Funktionen besitzen die Form $f : \mathbb{Z}^n \to \mathbb{Z}$. Für die implizite Äquivalenzprüfung dieser Funktionen bedarf es einer geeigneten kanonischen Repräsentation.

Eine mögliche kanonische Repräsentation besteht darin, erweiterte Diagramme ähnlich zu ROBDDs zu verwenden. Da bei BDDs allerdings der Definitionsbereich endlich ist, bedarf es einer Einschränkung der Menge der (ganzzahligen) arithmetischen Funktionen. Diese Einschränkung ist durchaus eine gültige Annahme, da bei Hardware-Implementierungen bereits Datentypen mit festen Wortbreiten (Ganzzahl- oder Festpunkt-Datentypen) verwendet werden, diese also bereits eingeschränkt sind. Seien $l \in \mathbb{Z}$ und $u \in \mathbb{Z}$ zwei ganze Zahlen mit $l < u$. Dann beschreibe $\mathbb{Z}_{[l,u]} \subset \mathbb{Z}$ die endliche Menge $\{l, l+1, \ldots, u\}$. Eine Funktion $f : \mathbb{Z}^n_{[l,u]} \to \mathbb{Z}$ lässt sich in einem Entscheidungsdiagramm darstellen, indem jeder Knoten mit einer der n Variablen assoziiert wird und jeder Knoten genau $u - l + 1$ Nachfolger besitzt. Dies bedeutet, dass der Wertebereich $\mathbb{Z}_{[l,u]}$ für jede Variable in $u - l + 1$ Partitionsblöcke aufgeteilt wird, d. h. $\Delta(\mathbb{Z}_{[l,u]}) = \{\Delta_l(\mathbb{Z}_{[l,u]}), \Delta_{l+1}(\mathbb{Z}_{[l,u]}), \ldots, \Delta_u(\mathbb{Z}_{[l,u]})\}$ (siehe Anhang B.1). Bei der Traversierung des Diagramms wird dann eine Kante entsprechend der Variablenbelegung ausgewählt. Der Wertebereich der Funktion selbst muss hierbei nicht beschränkt werden, da der endliche Definitionsbereich in einem endlichen Entscheidungsdiagramm resultiert (siehe Abb. 6.29).

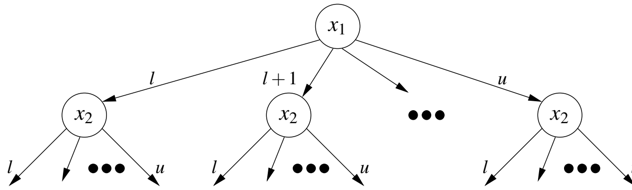


Abb. 6.29. Darstellung einer ganzzahligen arithmetischen Funktion mit endlichem Definitionsbereich als Entscheidungsdiagramm [272]

Eine solche Repräsentation besitzt allerdings den Nachteil, dass sie sowohl exponentiell in der Anzahl der Variablen als auch der Größe des Definitionsbereichs $\left|\mathbb{Z}_{[l,u]}\right|$ wachsen kann [272]. Dies gilt auch, wenn der Definitionsbereich binär codiert wird und ROBDDs zum Einsatz kommen. Viele der für die Hardware-Verifikation auf Architekturebene relevanten Funktionen lassen auf die Klasse der sog. *Pseudo-Booleschen Funktionen* einschränken. Eine Pseudo-Boolesche Funktion ist eine Abbildung $f : \mathbb{B}^n \to \mathbb{Z}$, d. h. nur der Wertebereich umfasst die ganzen Zahlen. Eine *Belegung* ist eine Funktion β, die jeder Variable in f den Wert T oder F zuweist.

Wie bei Booleschen Funktionen gibt es auch unterschiedliche Repräsentationen für Pseudo-Boolesche Funktionen als Entscheidungsdiagramme. Für die Verifikation sind hierbei wieder kanonische Repräsentationen besonders wichtig, also Repräsentationen, die vollständig und eindeutig sind. Eine offensichtliche Repräsentation Pseudo-Boolescher Funktionen basiert auf der Idee von ROBDDs, wobei zu

beachten ist, dass die Menge der Terminalknoten V_T nicht mehr auf die Werte T und F beschränkt ist, sondern ganze Zahlen beinhalten kann, d. h. value : $V_T \rightarrow \mathbb{Z}$ (siehe Anhang B.1). Eine solche kanonische Repräsentation Pseudo-Boolescher Funktionen, basierend auf der Shannon-Zerlegung, heißt *algebraisches Entscheidungsdiagramm* (engl. *Algebraic Decision Diagram, ADD*) oder engl. *Multi-Terminal BDD* (MTBDD) [99, 106]. Das folgende Beispiel stammt aus [66].

Beispiel 6.2.1. Gegeben ist die Pseudo-Boolesche Funktion in Tabelle 6.4. Das entsprechende MTBDD ist in Abb. 6.30 zu sehen. Wie bei ROBDDs wählt eine Belegung β der Variablen einen Pfad im MTBDD vom Quellknoten zu einem Terminalknoten aus. Die Terminalknoten sind mit den entsprechenden Funktionswerten für die gegebene Belegung beschriftet.

Tabelle 6.4. Pseudo-Boolesche Funktion [66]

x_1	x_2	f
F	F	8
F	T	−12
T	F	10
T	T	−6

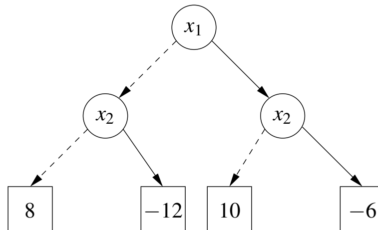


Abb. 6.30. MTBDD der Pseudo-Booleschen Funktion aus Tabelle 6.4

Die Repräsentation von Funktionen durch MTBDDs hat in der Praxis oftmals den Nachteil, dass diese exponentiell mit der Anzahl der Variablen wachsen. Dies kann dadurch begründet werden, dass es aufgrund vieler unterschiedlicher Funktionswerte häufig schwierig ist, die Reduktionsregeln anzuwenden, da die Identifikation isomorpher Teilgraphen nicht mehr trivial ist. Anstatt eine Funktionstabelle direkt in ein MTBDD abzubilden, stellt sich eine *momentenbasierte Zerlegung* als vorteilhaft heraus. Die resultierende Repräsentation wird als *binärer Momentengraph* (engl. *Binary Moment Diagram, BMD*) bezeichnet (siehe u. a. [324]). BMDs wurden erstmalig von Bryant und Chen vorgestellt [65, 66].

Zunächst wird aus der Funktionstabelle der Pseudo-Booleschen Funktion ein linearer Ausdruck gebildet: Die Belegung $\beta(x_i) = \mathsf{T}$ wird als Literal x_i und die Belegung $\beta(x_i) = \mathsf{F}$ wird als Literal $(1 - x_i)$ codiert. Ein Ausdruck für jede Belegung wird durch Multiplikation der entsprechenden Literale gebildet. Dabei wird die Codierung der Belegung noch mit dem Funktionswert gewichtet. Anschließend werden alle Ausdrücke aufsummiert.

Beispiel 6.2.2. Betrachtet wird wiederum die Pseudo-Boolesche Funktion in Tabelle 6.4. Der lineare Ausdruck nach obiger Konstruktionsvorschrift ergibt sich zu:

$$\begin{aligned} f(x_1, x_2) &= 8 \cdot (1 - x_1) \cdot (1 - x_2) - 12 \cdot (1 - x_1) \cdot x_2 + 10 \cdot x_1 \cdot (1 - x_2) - 6 \cdot x_1 \cdot x_2 \\ &= 8 - 20 \cdot x_2 + 2 \cdot x_1 + 4 \cdot x_2 \cdot x_1 \end{aligned}$$

Das zugehörige BMD ist in Abb. 6.31 zu sehen. Das BMD ist dabei wie folgt zu interpretieren: Kanten, die den Wert $\beta(x_i) = \mathsf{T}$ darstellen und über die ein Terminalknoten $v_\mathrm{T} \in V_\mathrm{T}$ erreicht werden kann, zeigen an, dass der Koeffizient value(v_T) von dieser Variablen abhängt. Andernfalls ist der Koeffizient unabhängig von dieser Variablen.

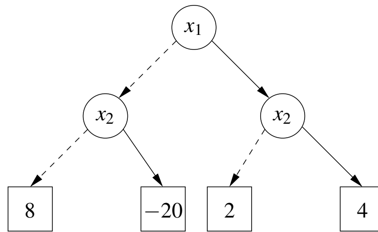


Abb. 6.31. BMD der Pseudo-Booleschen Funktion aus Tabelle 6.4

Aus mathematischer Sicht basieren MTBDDs auf der *Shannon-Zerlegung*. Diese lautet für eine Pseudo-Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{Z}$:

$$f = (1 - x_i) \cdot f|_{x_i:=\mathsf{F}} + x_i \cdot f|_{x_i:=\mathsf{T}} \tag{6.1}$$

Durch Umformung erhält man die *positive Davio-Zerlegung* für eine Pseudo-Boolesche Funktion:

$$\begin{aligned} f &= f|_{x_i:=\mathsf{F}} + x_i \cdot (f|_{x_i:=\mathsf{T}} - f|_{x_i:=\mathsf{F}}) \\ &= f|_{x_i:=\mathsf{F}} + x_i \cdot f|_{\partial x_i} \end{aligned} \tag{6.2}$$

Hierbei wird $f|_{x_i:=\mathsf{F}}$ als *konstantes Moment* und $f|_{\partial x_i}$ als *lineares Moment* bezeichnet. Damit lässt sich ein binärer Momentengraph formal definieren:

Definition 6.2.1 (Binärer Momentengraph). *Gegeben sei eine Pseudo-Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{Z}$ mit den Variablen $x_1, \ldots, x_n$. Der zugehörige* binäre Momentengraph *(*BMD*)* (V, E) *mit Quellknoten v_0 ist ein Entscheidungsdiagramm nach Definition B.1.1 auf Seite 533 mit folgenden Eigenschaften:*

- *Der Quellknoten v_0 repräsentiert die Funktion f, d. h. $\phi(v_0) = f$.*
- *Für jeden Knoten $v \in V$ gilt:*
  - *Falls $v \in V_\mathrm{T}$, dann repräsentiert v die entsprechende konstante Pseudo-Boolesche Funktion* $\mathrm{value}(v)$*, d. h.* $\phi(v) = \mathrm{value}(v)$.
  - *Falls $v \in V_\mathrm{N}$, dann repräsentiert v die Funktion* $\phi(v) = \phi(\mathrm{child}(v,1)) + x_{\mathrm{index}(v)} \cdot \phi(\mathrm{child}(v,2))$.

Ein *geordnetes BMD* (engl. *Ordered BMD, OBMD*) besitzt die Eigenschaft, dass auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten. Ein *reduziertes OBMD* (engl. *Reduced OBMD, ROBMD*) enthält man durch wiederholtes Anwenden der Verschmelzungsregel und der Eliminationsregel für OFDDs (siehe Anhang B.3). ROBMDs sind kanonische Repräsentationen für Pseudo-Boolesche Funktionen und somit für die implizite Äquivalenzprüfung auf Architekturebene geeignet.

In [65, 66] stellen Bryant und Chen eine Erweiterung von BMDs um multiplikative Kantengewichte als Erweiterung additiver Kantengewichte vor, wie sie beispielsweise auch in sog. *EVBDDs* (engl. *Edge Valued Binary Decision Diagrams*) zum Einsatz kommen [281]. Als Abgrenzung zu BMDs werden diese Momentengraphen als *multiplikative BMDs* (engl. *multiplicative BMDs, *BMDs*) bezeichnet. *BMDs verwenden ganzzahlige Faktoren der konstanten und linearen Momente als Kantengewichte, um die Anzahl von Terminalknoten zu verringern und dadurch die Anzahl isomorpher Teilgraphen zu erhöhen. Das Ergebnis sind kompaktere Repräsentanten Pseudo-Boolescher Funktionen.

Für die *Interpretation* eines *BMD wird das Kantengewicht w_e einer eingehenden Kante e als Faktor für die Interpretation des Knotens v verwendet. Somit gilt für *BMDs:

- Der Quellknoten v_0 repräsentiert die Funktion f, d. h. $\phi(w_0, v_0) = f$.
- Für jeden Knoten $v \in V$ gilt:
  - Falls $v \in V_\mathrm{T}$, dann repräsentiert v die konstante Pseudo-Boolesche Funktion $\phi(w_e, v) = w_e \cdot \mathrm{value}(v)$.
  - Falls $v \in V_\mathrm{N}$, dann repräsentiert v die Funktion $\phi(w_e, v) = w_e \cdot (\phi(w_{(v,\mathrm{child}(v,1))}, \mathrm{child}(v,1)) + x_{\mathrm{index}(v)} \cdot \phi(w_{(v,\mathrm{child}(v,2))}, \mathrm{child}(v,2)))$.

Das Ergebnis einer Interpretation wird also immer mit dem entsprechenden Kantengewicht multipliziert. Weiterhin wird angenommen, dass der Quellknoten v_0 eine eingehende Kante mit Kantengewicht w_0 besitzt. Dies ist notwendig, um kanonische *BMDs zu konstruieren (s. u.).

Beispiel 6.2.3. Gegeben ist die Pseudo-Boolesche Funktion $f := 8 - 20 \cdot x_3 + 2 \cdot x_2 + 4 \cdot x_2 \cdot x_3 + 12 \cdot x_1 + 24 \cdot x_1 \cdot x_3 + 15 \cdot x_1 \cdot x_2$. Das zugehörige ROBMD mit Variablenordnung $x_1 < x_2 < x_3$ ist in Abb. 6.32a) zu sehen. Man sieht, dass das BMD reduziert ist, da es keine isomorphen Teilgraphen enthält, und somit die Verschmelzungsregel nicht angewendet werden kann. Weiterhin fehlt der Knoten x_3 im ganz rechten Ast. Dieser besaß eine ausgehende Kante $\mathrm{child}(v,2) = \mathsf{F}$. Somit konnte die Eliminationsregel angewendet werden (siehe Anhang B.3).

Das entsprechende geordnete *BMD (O*BMD) mit der selben Variablenordnung ist in Abb. 6.32b) zu sehen. Die Kantengewichte sind als Quadrate auf die Kanten gezeichnet. Kanten ohne Beschriftung haben das Kantengewicht 1. Man sieht, dass das O*BMD u. a. die mögliche Faktorisierung der Terme $2 \cdot x_2 + 4 \cdot x_2 \cdot x_3$ und $12 \cdot x_1 + 24 \cdot x_1 \cdot x_3$ zu $2 \cdot x_2 \cdot (1 + 2 \cdot x_3)$ bzw. $12 \cdot x_1 \cdot (1 + 2 \cdot x_3)$ ausnutzt. Das O*BMD ist allerdings noch nicht reduziert. Hierzu müssten die Kantengewichte noch wie unten beschrieben angepasst werden.

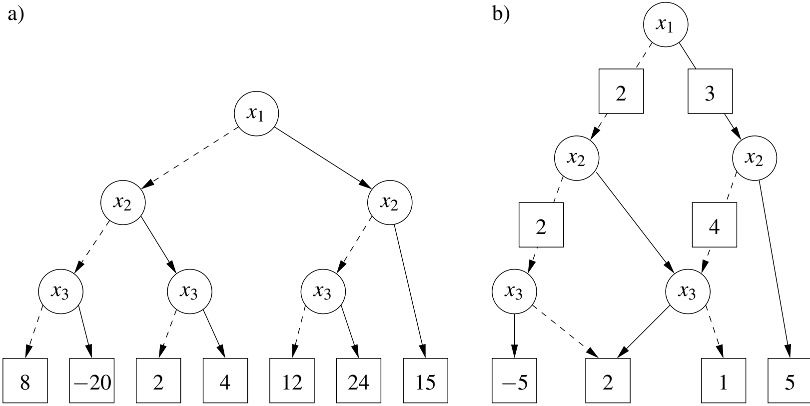


Abb. 6.32. a) BMD und b) entsprechendes *BMD der Pseudo-Booleschen Funktion aus Beispiel 6.2.3 [66]

Damit O*BMDs eine kanonische Repräsentation Pseudo-Boolescher Funktionen sind, müssen die Kantengewichte einige Einschränkungen erfüllen:

1. Ein *BMD enthält höchstens zwei Terminalknoten, die mit $\text{value}(v) = 0$ und $\text{value}(v) = 1$ beschriftet sind.
2. Alle Kantengewichte müssen von 0 verschieden sein.
3. Falls eine Kante $e = (v, v_\text{T})$ auf den Terminalknoten v_T mit Wert $\text{value}(v_\text{T}) = 0$ zeigt, so muss das zugehörige Kantengewicht $w_e = 1$ sein. Falls es sich hierbei um eine F-Kante handelt, d. h. $e = (v, \text{child}(v, 1))$ gilt, muss die entsprechende T-Kante $(v, \text{child}(v, 2))$ ebenfalls das Kantengewicht 1 besitzen.
4. Alle Kantengewichte von F-Kanten müssen nichtnegativ sein.
5. Der größte gemeinsame Teiler der Kantengewichte ausgehender Kanten eines Nichtterminalknotens muss 1 sein, d. h. $\text{gcd}(w_{(v,\text{child}(v,1))}, w_{(v,\text{child}(v,2))}) = 1$.

*BMD-Repräsentation arithmetischer Komponenten

Abbildung 6.33 zeigt das *BMD von drei wichtigen arithmetischen Operationen. Um die Lesbarkeit der graphischen Repräsentationen zu erhöhen, wurden bei der

Darstellung mehr als zwei Terminalknoten zugelassen. Abbildung 6.33a) zeigt die *BMD-Repräsentation einer 3-Bit Ganzzahladdition $x+y$. Dies kann interpretiert werden, als ob die einzelnen Binärstellen gewichtet addiert werden, wobei die i-ten Stellen x_i und y_i mit 2^i gewichtet sind.

Die Multiplikation $x \cdot y$ zweier 3-Bit Zahlen ist in Abb. 6.33b) als *BMD dargestellt. Die Pseudo-Boolesche Funktion lautet $\left(\sum_{i:=0}^{2} x_i \cdot 2^i\right) \cdot \left(\sum_{i:=0}^{2} y_i \cdot 2^i\right)$. Schließlich zeigt Abb. 6.33c) das *BMD für die Berechnung von $2^x = 2^{\sum_{i:=0}^{3} x_i \cdot 2^i}$. Man kann feststellen, dass die Größe dieses Graphen lediglich linear mit der Anzahl der Variablen wächst.

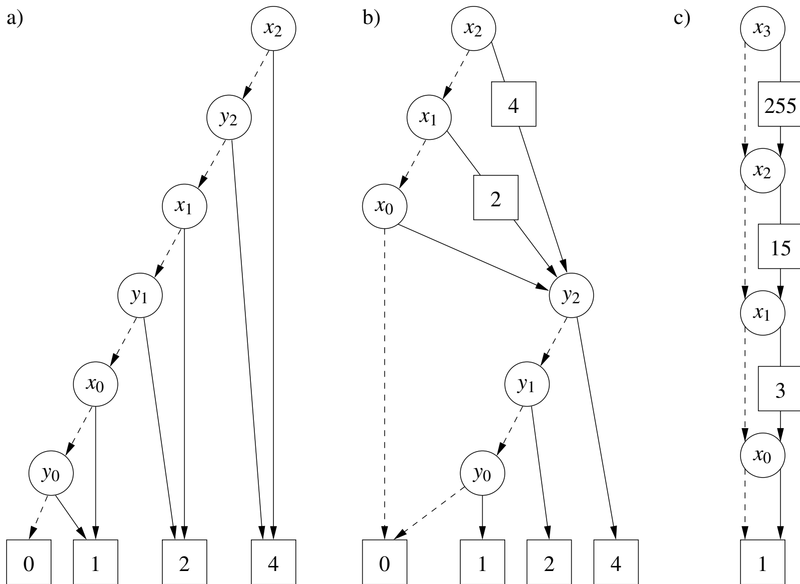


Abb. 6.33. *BMDs von a) Addition, b) Multiplikation und c) 2^x [66]

*BMD-Repräsentation kombinatorischer Schaltungen

Neben der Repräsentation von arithmetischen Operationen können mit *BMDs ebenfalls Boolesche Funktionen repräsentiert werden. Somit stellen reduzierte O*BMDs (RO*BMDs) eine weitere Möglichkeit dar, kombinatorische Schaltungen kanonisch zu repräsentieren. In Abb. 6.34 sind die *BMDs von drei wichtigen Gattern mit vier Eingängen (x_1, x_2, x_3, x_4) dargestellt. Man sieht, dass die Kantengewichte nicht auf die Werte 0 und 1 beschränkt sind. Dennoch ist der Wertebereich der implementierten Funktionen $\{0,1\}$ mit der entsprechenden Interpretation als $\mathbb{B} = \{\mathsf{F}, \mathsf{T}\}$.

Analog zu positiven Davio-Zerlegung existiert ebenfalls eine *negative Davio-Zerlegung* für Pseudo-Boolesche Funktionen:

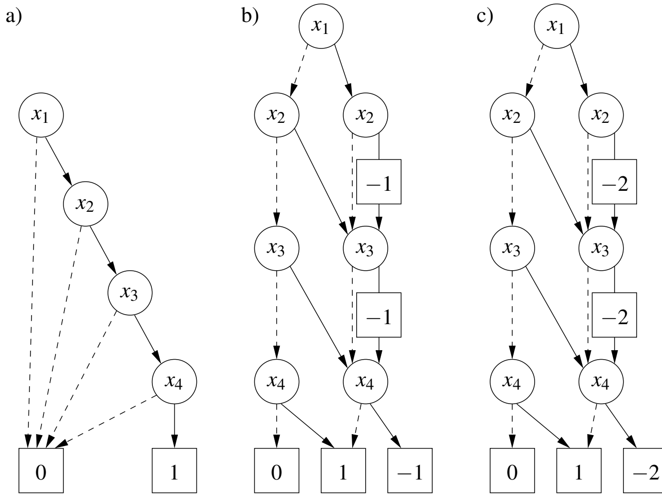


Abb. 6.34. *BMD eines a) OR-Gatters, b) AND-Gatters und c) XOR-Gatters mit jeweils vier Eingängen [66]

$$\begin{aligned} f &= f|_{x_i:=\mathsf{T}} + (1-x_i)\cdot(f|_{x_i:=\mathsf{T}} - f|_{x_i:=\mathsf{F}}) \\ &= f|_{x_i:=\mathsf{T}} + (1-x_i)\cdot f|_{\partial x_i} \end{aligned} \tag{6.3}$$

In [136] stellen Drechsler et al. *Kronecker *BMDs* (K*BMDs) als Verallgemeinerung von *BMDs vor. Wie bei KFDDs können bei K*BMDs für jede Variable der Zerlegungstyp gewählt werden, d. h. entweder Shannon-, positive Davio- oder negative Davio-Zerlegung. Ein Vergleich von Entscheidungsdiagrammen für Boolesche und Pseudo-Boolesche Funktionen findet sich in [134].

6.2.2 Äquivalenzprüfung zwischen Architektur- und Logikebene

Bisher wurde lediglich die Äquivalenzprüfung zweier Schaltungen auf der selben Abstraktionsebene betrachtet. Hierbei wurden implizite Verfahren auf der Logikebene basierend auf ROBDDs oder ROKFDDs diskutiert. Diese lassen sich jedoch nicht gut auf Architekturebene einsetzen, wo sich RO*BMDs als vorteilhaft herausgestellt haben. Möchte man allerdings überprüfen, ob eine Implementierung auf Logikebene, also eine Gatternetzliste, äquivalent zum spezifizierten Verhalten auf Architekturebene ist, bedarf es zusätzlicher Schritte. Ein erster Ansatz wurde in [65, 66] vorgestellt. Die Idee ist in Abb. 6.35 dargestellt.

Die Spezifikation ist durch eine Funktion $f_2 : \mathbb{Z}^2 \to \mathbb{Z}$ gegeben. Hierbei ist davon auszugehen, dass der Definitionsbereich eingeschränkt ist bzw. eingeschränkt werden muss, um eine Implementierung in Hardware zu ermöglichen. Der Einfachheit halber wird in diesem Beispiel davon ausgegangen, dass alle Definitionsbereiche der Eingangsvariablen und der Wertebereich der Funktion identisch sind. Die resultierende Implementierung als Gatternetzliste implementiert die Boolesche Funktion

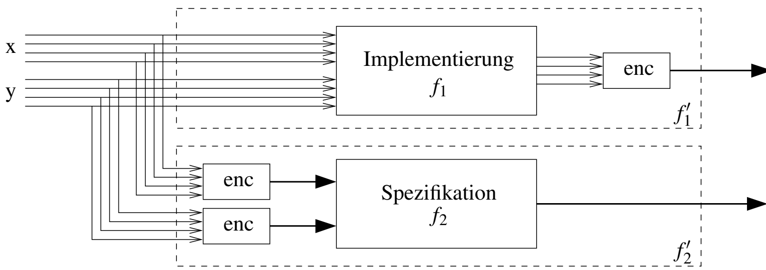


Abb. 6.35. Schematische Darstellung zur Äquivalenzprüfung einer Spezifikation auf Architekturebene und einer Implementierung auf Logikebene [66]

$f_1 : \mathbb{B}^{2n} \to \mathbb{B}^n$ (im Beispiel $n = 4$). Bei der Hardware-Implementierung wird eine Codierung der Bitvektoren gewählt, die es erlaubt, die Ein- und Ausgänge als ganze Zahlen aus dem Definitions- und Wertebereich zu interpretieren. Dies erfolgt in den Blöcken enc, die eine Funktion enc : $\mathbb{B}^n \to \mathbb{Z}$ implementieren. Hieraus ergibt sich, dass die Funktionen f_1' und f_2' Pseudo-Boolesche Funktionen sind.

Typische Codierungen sind *vorzeichenlose binäre Darstellung*, *Vorzeichen-Betrag-Darstellung* und *2er-Komplement-Darstellung*. Um eine Äquivalenzprüfung zwischen unterschiedlichen Abstraktionsebenen durchführen zu können, müssen die gewählten Codierungen berücksichtigt werden. Die Darstellung der drei oben genannten Codierungen als *BMD für vier Binärstellen ist in Abb. 6.36 dargestellt.

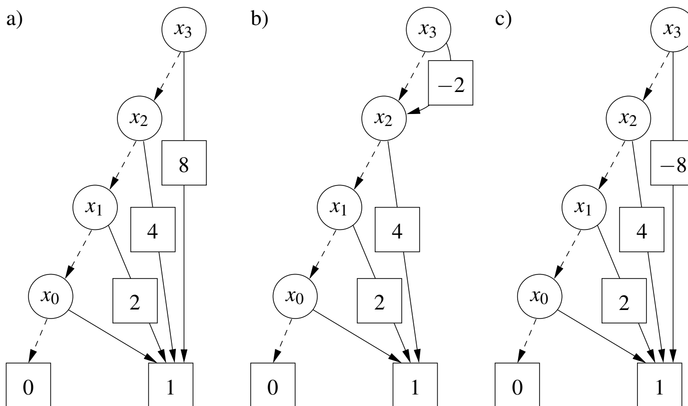


Abb. 6.36. *BMDs der a) vorzeichenlosen binären Darstellung, b) Vorzeichen-Betrag-Darstellung und c) 2er-Komplement-Darstellung [329]

Um eine Äquivalenzprüfung über Abstraktionsebenen hinweg durchführen zu können, wird jede Schaltung zunächst in ihre einzelnen Komponenten zerlegt und jede Komponente gegen ihre Spezifikation geprüft. Anschließend wird die korrekte Verschaltung der einzelnen Komponenten überprüft. Dies wird anhand einer Volladdierer-Schaltung demonstriert [329].

Beispiel 6.2.4. Gegeben ist der Volladdierer aus Abb. A.2 auf Seite 527. Zunächst wird für jeden Ausgang des Volladdierers ein *BMD auf Basis der implementierten Booleschen Funktionen aufgestellt. Der Volladdierer hat die Ausgänge s_1 für die Summe und c_1 für den Übertrag. Die drei Eingänge sind a_1, b_1 und c_0. Die Summe berechnet sich anhand der implementierten Booleschen Funktionen zu:

$$s_1(a_1, b_1, c_0) = a_1 \oplus b_1 \oplus c_0$$

Der Übertrag zu:

$$c_1(a_1, b_1, c_0) = (a_1 \wedge b_1) \vee ((a_1 \oplus b_1) \wedge c_0)$$

Zur Erstellung der beiden *BMD müssen die Booleschen Funktionen in Pseudo-Boolesche Funktionen umgewandelt werden. Hierbei gilt:

$$x_0 \wedge x_1 = x_0 \cdot x_1,$$

$$x_0 \vee x_1 = x_0 + x_1 - x_0 \cdot x_1$$

sowie

$$\begin{aligned}
x_0 \oplus x_1 &= (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) \\
&= (x_0 \cdot (1 - x_1)) \vee ((1 - x_0) \cdot x_1) \\
&= (x_0 - x_0 \cdot x_1) \vee (x_1 - x_0 \cdot x_1) \\
&= (x_0 - x_0 \cdot x_1) + (x_1 - x_0 \cdot x_1) - (x_0 - x_0 \cdot x_1) \cdot (x_1 - x_0 \cdot x_1) \\
&= x_0 + x_1 - 2 \cdot x_0 \cdot x_1 - (x_0 \cdot x_1 - x_0 \cdot x_1 - x_0 \cdot x_1 + x_0 \cdot x_1) \\
&= x_0 + x_1 - 2 \cdot x_0 \cdot x_1,
\end{aligned}$$

wobei $x_0, x_1 \in \{0, 1\}$.

Für die Summen- und Übertragsberechnung ergibt sich dann:

$$\begin{aligned}
s_1(a_1, b_1, c_0) &= a_1 \oplus b_1 \oplus c_0 \\
&= (a_1 + b_1 - 2 \cdot a_1 \cdot b_1) \oplus c_0 \\
&= (a_1 + b_1 - 2 \cdot a_1 \cdot b_1) + c_0 - 2 \cdot (a_1 + b_1 - 2 \cdot a_1 \cdot b_1) \cdot c_0 \\
&= a_1 + b_1 + c_0 - 2 \cdot a_1 \cdot b_1 - 2 \cdot a_1 \cdot c_0 - 2 \cdot b_1 \cdot c_0 + 4 \cdot a_1 \cdot b_1 \cdot c_0
\end{aligned}$$

und

$$c_1(a_1, b_1, c_0) = a_1 \cdot b_1 + a_1 \cdot c_0 + b_1 \cdot c_0 - 2 \cdot a_1 \cdot b_1 \cdot c_0$$

Die resultierenden *BMDs für die Variablenordnung $c_0 < b_1 < a_1$ sind in Abb. 6.37 zu sehen.

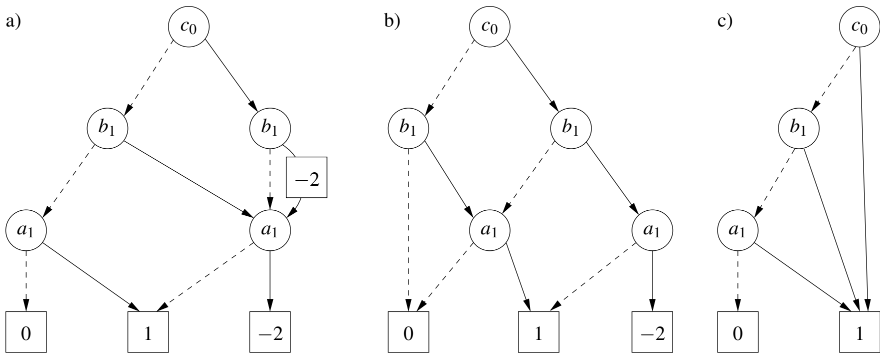


Abb. 6.37. *BMDs für den a) Summen- (s_1) und b) Übertragsausgang (c_1) eines Volladdierers sowie für den c) Volladdierer an sich [329]

Aus den *BMDs für die Funktionen s_1 und c_1 wird nun ein *BMD zur Repräsentation des Volladdierers durch Verwendung der Codierung vorzeichenloser ganzer Zahlen ($2^0 \cdot s_1 + 2^1 \cdot c_1$) konstruiert. Das Ergebnis ist in Abb. 6.37c) zu sehen. Dieses *BMD stimmt mit der Spezifikation der arithmetischen Funktion $a_1 + b_1 + c_0$ für $a_1, b_1, c_0 \in \{0, 1\}$ überein.

Additionen und Multiplikationen können direkt auf *BMDs durchgeführt werden. Die Berechnung erfolgt ähnlich dem ITE-Operator bei ROBDDs. Hier werden die beiden Operationen basierend auf BMDs diskutiert, eine Erweiterung auf *BMDs ist in [329] beschrieben. Für die *Addition* ergibt sich z. B. bei zwei gegebenen BMDs, welche die Pseudo-Booleschen Funktionen f und g repräsentieren:

$$\begin{aligned} f + g &:= (f|_{x_i:=\mathsf{F}} + x_i \cdot f|_{\partial x_i}) + (g|_{x_i:=\mathsf{F}} + x_i \cdot g|_{\partial x_i}) \\ &= (f|_{x_i:=\mathsf{F}} + g|_{x_i:=\mathsf{F}}) + x_i \cdot (f|_{\partial x_i} + g|_{\partial x_i}) \end{aligned}$$

Entsprechend ergibt sich für die *Multiplikation*:

$$\begin{aligned} f \cdot g &:= (f|_{x_i:=\mathsf{F}} + x_i \cdot f|_{\partial x_i}) \cdot (g|_{x_i:=\mathsf{F}} + x_i \cdot g|_{\partial x_i}) \\ &= (f|_{x_i:=\mathsf{F}} \cdot g|_{x_i:=\mathsf{F}}) + x_i \cdot (f|_{x_i:=\mathsf{F}} \cdot g|_{\partial x_i} + f|_{\partial x_i} \cdot g|_{x_i:=\mathsf{F}} + f|_{\partial x_i} \cdot g|_{\partial x_i}) \end{aligned}$$

Auch wenn sich im obigen Beispiel die Verwendung von *BMDs als zweckmäßig herausgestellt hat, kann es schwierig sein, *BMDs für kombinatorische Schaltungen zu konstruieren. So kann die Größe eines *BMD beispielsweise bei der Konstruktion für eine bitgenaue Multipliziererschaltung exponentiell anwachsen.

6.2.3 Äquivalenzprüfung auf der Architekturebene

In diesem Abschnitt wird der Einsatz von *Taylor-Expansions-Diagrammen* (TEDs, siehe Abschnitt 4.1) zur impliziten Äquivalenzprüfung auf der Architekturebene.

Hierbei werden zwei algorithmische Beschreibungen in Form von Polynomfunktionen als TEDs dargestellt und auf ihre Äquivalenz geprüft. TEDs sind kanonische Repräsentationen für Polynomfunktionen, die durch eine endliche Taylor-Reihe dargestellt werden können. Somit bieten TEDs den Zugang zur impliziten Äquivalenzprüfung auf Architekturebene an. Bevor dies an Beispielen demonstriert wird, werden zunächst die wichtigsten Operationen zum Rechnen mit TEDs beschrieben.

Kompositionsoperatoren für TEDs

Ein Taylor-Expansions-Diagramm $G = (V, E)$ mit Knotenmenge V und Kantenmenge E kann aus zwei anderen TEDs zusammengesetzt werden, um komplexe Berechnungen durchzuführen. Hierzu werden im Folgenden die Addition $(+)$ und Multiplikation $(\cdot)$ beschrieben. Die Komposition erfolgt dabei durch rekursive Anwendung von Rechenregeln auf die beiden TEDs: Beginnend mit den beiden Quellknoten der TEDs werden rekursiv alle von null verschiedenen Terme des Ergebnis-TEDs bestimmt und kombiniert. Damit das Ergebnis eindeutig repräsentiert ist, muss das Ergebnis-TED normalisiert und reduziert werden.

Seien v' und v'' die beiden zu kombinierenden TED-Knoten und v der resultierende Knoten im Ergebnis-TED und seien $f = \phi(v')$ und $g = \phi(v'')$ die Funktionen, die durch die Knoten v' und v'' repräsentiert werden. Weiterhin seien $x_{\text{index}(v')}$ und $x_{\text{index}(v'')}$ die beiden mit v' und v'' assoziierten Variablen. Die resultierende Funktion, die durch den Knoten v repräsentiert wird, heiße $h = \phi(v)$.

Zunächst wird die Komposition von v' und v'' bei der Addition betrachtet. Für die Komposition müssen zwei Fälle unterschieden werden:

1. Falls beide Knoten v' und v'' Terminalknoten sind, so wird v als neuer Terminalknoten mit dem Wert $\text{value}(v) = \text{value}(v') + \text{value}(v'')$ angelegt.
2. Falls mindestens ein Knoten ein Nichtterminalknoten ist, werden anhand der Variablenordnung zwei Fälle unterschieden:
   a) Falls die Knoten die selbe Variable repräsentieren ($x_{\text{index}(v')} = x_{\text{index}(v'')}$), wird ein neuer Knoten v mit $x_{\text{index}(v)} := x_{\text{index}(v')}$ angelegt und die Nachfolger von v aus der paarweisen Addition der Nachfolger von v' und v'' gebildet:

$$\begin{aligned} h &:= f + g \\ &= f|_{x:=0} + x \cdot \left.\frac{\partial f}{\partial x}\right|_{x:=0} + \cdots + g|_{x:=0} + x \cdot \left.\frac{\partial g}{\partial x}\right|_{x:=0} + \ldots \\ &= f|_{x:=0} + g|_{x:=0} + x \cdot \left[\left.\frac{\partial f}{\partial x}\right|_{x:=0} + \left.\frac{\partial g}{\partial x}\right|_{x:=0} \right] + \ldots \end{aligned}$$

   b) Falls die Knoten unterschiedliche Variablen repräsentieren, wird ein neuer Knoten v angelegt und es gilt:

$$x_{\text{index}(v)} := \begin{cases} x_{\text{index}(v')} & \text{falls } x_{\text{index}(v')} < x_{\text{index}(v'')} \\ x_{\text{index}(v'')} & \text{sonst} \end{cases}$$

Sei $x_{\text{index}(v')} < x_{\text{index}(v'')}$, dann gilt:

$$
\begin{aligned}
h &:= f + g \\
&= f|_{x:=0} + x \cdot \left.\frac{\partial f}{\partial x}\right|_{x:=0} + \cdots + g \\
&= [f|_{x:=0} + g] + x \cdot \left.\frac{\partial f}{\partial x}\right|_{x:=0} + \ldots
\end{aligned}
$$

Dies bedeutet, dass die Funktion g komplett zum Nachfolger $\text{child}(v', 1)$ von v' hinzu addiert wird, während die weiteren Nachfolger $\text{child}(v', k)$ mit $k > 1$ nicht verändert werden.

Die Fälle a) und b) sind in Abb. 6.38 dargestellt.

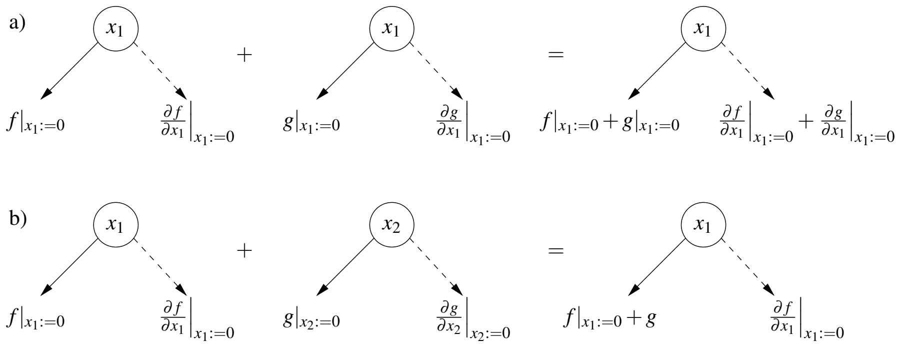


Abb. 6.38. Addition von zwei TED-Knoten mit a) der selben assoziierten Variablen und b) wenn x_1 in der Variablenordnung vor x_2 kommt

Die Kompositionsregeln für die Multiplikation lassen sich auf die gleiche Art herleiten. Wiederum müssen zwei Fälle unterschieden werden:

1. Falls beide Knoten v' und v'' Terminalknoten sind, so wird v als neuer Terminalknoten mit dem Wert $\text{value}(v) = \text{value}(v') \cdot \text{value}(v'')$ angelegt.
2. Falls mindestens ein Knoten ein Nichtterminalknoten ist, werden anhand der Variablenordnung, wie bei der Addition, zwei Fälle unterschieden:
   a) Falls die Knoten die selbe Variable repräsentieren, also $x_{\text{index}(v')} = x_{\text{index}(v'')}$, werden die Nachfolger von dem neuen Knoten v mit $x_{\text{index}(v)} := x_{\text{index}(v')}$ wie folgt gebildet (der Einfachheit halber werden lediglich zwei Nachfolger für v' und v'' betrachtet):

$$
\begin{aligned}
h &:= f \cdot g \\
&= \left(f|_{x:=0} + x \cdot \left.\frac{\partial f}{\partial x}\right|_{x:=0} \right) \cdot \left(g|_{x:=0} + x \cdot \left.\frac{\partial g}{\partial x}\right|_{x:=0} \right) \\
&= (f|_{x:=0} \cdot g|_{x:=0}) + x \cdot \left[\left.\frac{\partial f}{\partial x}\right|_{x:=0} \cdot g|_{x:=0} + f|_{x:=0} \cdot \left.\frac{\partial g}{\partial x}\right|_{x:=0} \right] \\
&\quad + x^2 \cdot \left[\left.\frac{\partial f}{\partial x}\right|_{x:=0} \cdot \left.\frac{\partial g}{\partial x}\right|_{x:=0} \right]
\end{aligned}
$$

Mit anderen Worten: Der Nachfolger $\text{child}(v,1)$ berechnet sich aus dem Produkt der Nachfolger $\text{child}(v',1)$ und $\text{child}(v'',1)$. Der Nachfolger $\text{child}(v,2)$ wird aus der Summe der beiden Kreuzprodukte der Koeffizienten gebildet. Schließlich berechnet sich der Nachfolger $\text{child}(v,3)$ als Produkt der Nachfolger $\text{child}(v',2)$ und $\text{child}(v'',2)$.

b) Falls die Knoten unterschiedliche Variablen repräsentieren, so gilt wiederum für den neuen Knoten v:

$$
x_{\text{index}(v)} := \begin{cases} x_{\text{index}(v')} & \text{falls } x_{\text{index}}(v') < x_{\text{index}}(v'') \\ x_{\text{index}(v'')} & \text{sonst} \end{cases}
$$

Sei $x_{\text{index}(v')} < x_{\text{index}(v'')}$ dann gilt (wiederum werden lediglich zwei Nachfolger für v' und v'' betrachtet):

$$
\begin{aligned}
h &:= f \cdot g \\
&= \left(f|_{x:=0} + x \cdot \left.\frac{\partial f}{\partial x}\right|_{x:=0} \right) \cdot g \\
&= [f|_{x:=0} \cdot g] + x \cdot \left[\left.\frac{\partial f}{\partial x}\right|_{x:=0} \cdot g \right]
\end{aligned}
$$

In diesem Fall wird die Funktion g mit jedem Nachfolger von v' multipliziert.

Die Multiplikation von zwei Knoten ist in Abb. 6.39 zu sehen.

Äquivalenzprüfung mit TEDs

Im Folgenden wird die Äquivalenzprüfung mit Taylor-Expansions-Diagrammen anhand des Beispiels der diskreten Faltung $\text{conv}_{(x_1,x_2,N)}$ von zwei Signalen x_1 und x_2 in einem endlichen Zeitfenster von N Schritten verdeutlicht, die sich wie folgt berechnen lässt:

$$
\text{conv}_{(x_1,x_2,N)}(i) := \sum_{k:=0}^{N-1} x_1(k) \cdot x_2(i-k)
$$

Für $N = 4$ ist das Blockschaltbild in Abb. 6.40a) dargestellt. Die aktuellen Signale $x_1(3)$ und $x_2(3)$ liegen zusammen mit den sechs vorherigen Signalen $x_i(2), x_i(1)$ und $x_i(0)$ an den Eingängen der Schaltung an. Die vier letzten Ergebnisse $y(3), y(2), y(1)$, $y(0)$ werden auf den Ausgängen der Schaltung ausgegeben.

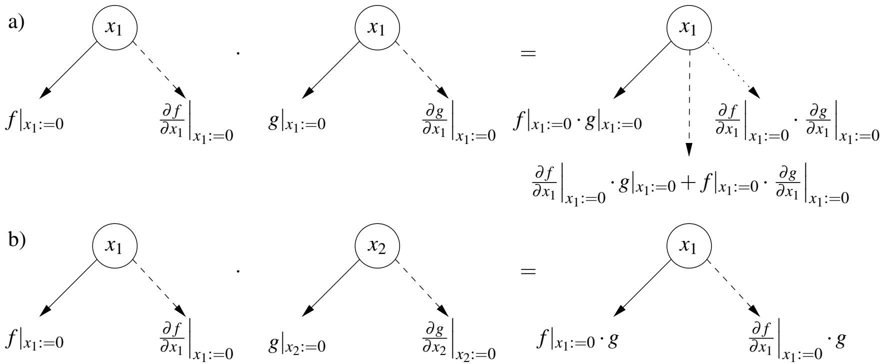


Abb. 6.39. Multiplikation von zwei TED-Knoten a) falls mit beiden Knoten die selbe Variable assoziiert ist und b) falls x_1 in der Variablenordnung vor x_2 kommt

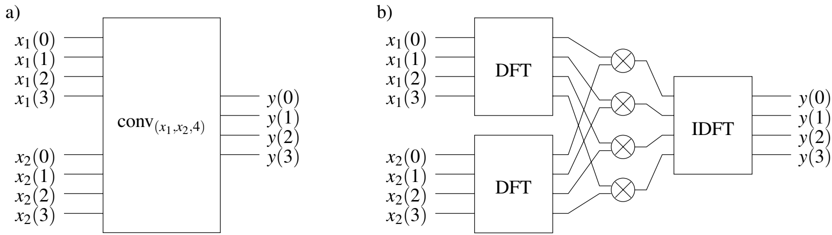


Abb. 6.40. Diskrete Faltung: Berechnung im a) Zeitbereich und b) Frequenzbereich [92]

Um die Berechnungen möglichst einfach zu gestalten, wird im Folgenden davon ausgegangen, dass es sich bei der Faltung um eine sog. *periodische Faltung* handelt, d. h. es gilt $x_1(k+N) = x_1(k)$ und $x_2(k+N) = x_2(k)$. Das Ergebnis der periodischen Faltung lautet somit:

$$\begin{aligned}
y(0) &= x_1(0)\cdot x_2(0) + x_1(1)\cdot x_2(3) + x_1(2)\cdot x_2(2) + x_1(3)\cdot x_2(1) \\
y(1) &= x_1(0)\cdot x_2(1) + x_1(1)\cdot x_2(0) + x_1(2)\cdot x_2(3) + x_1(3)\cdot x_2(2) \\
y(2) &= x_1(0)\cdot x_2(2) + x_1(1)\cdot x_2(1) + x_1(2)\cdot x_2(0) + x_1(3)\cdot x_2(3) \\
y(3) &= x_1(0)\cdot x_2(3) + x_1(1)\cdot x_2(2) + x_1(2)\cdot x_2(1) + x_1(3)\cdot x_2(0)
\end{aligned} \tag{6.4}$$

Ein weitere Möglichkeit die Faltung zu berechnen besteht darin, zunächst die Signale in den Frequenzbereich zu transformieren, dort eine Multiplikation durchzuführen, und das Ergebnis zurück in den Zeitbereich zu transformieren. Dies ist in Abb. 6.40b) dargestellt. Die *Diskrete Fourier-Transformation* (*DFT*) berechnet die Transformation vom Zeit- in den Frequenzbereich. Die inverse Funktion, *Inverse DFT* (*IDFT*), berechnet entsprechend die Transformation vom Frequenz- in den

Zeitbereich. Die DFT ist wie folgt definiert:

$$X(n) := \sum_{k:=0}^{N-1} x(k) \cdot e^{-\frac{2\cdot\pi\cdot\imath\cdot k\cdot n}{N}} \qquad \text{für } 0 \leq n \leq N-1 \tag{6.5}$$

Hierbei bezeichnet $\imath$ die komplexe Zahl, für die $\imath^2 = -1$ gilt. Es gilt weiterhin, dass $e^{\imath x} = \cos(x) + \imath \cdot \sin(x)$ ist. Die Umkehrfunktion IDFT berechnet sich zu:

$$x(n) := \frac{1}{N} \sum_{k:=0}^{N-1} X(k) \cdot e^{\frac{2\cdot\pi\cdot\imath\cdot k\cdot n}{N}} \qquad \text{für } 0 \leq n \leq N-1 \tag{6.6}$$

Zunächst wird das Eingangssignal x_1 mit Hilfe der DFT in den Frequenzbereich transformiert.

$$\begin{aligned}
X_1(0) &= x_1(0) + x_1(1) + x_1(2) + x_1(3) \\
X_1(1) &= x_1(0) - \imath \cdot x_1(1) - x_1(2) + \imath \cdot x_1(3) \\
X_1(2) &= x_1(0) - x_1(1) + x_1(2) - x_1(3) \\
X_1(3) &= x_1(0) + \imath \cdot x_1(1) - x_1(2) - \imath \cdot x_1(3)
\end{aligned}$$

Das Spektrum für das Signal x_2 ergibt sich analog. Die einzelnen Spektralanteile $X_1(0), \ldots, X_1(3)$ lassen sich jeweils in einem TED darstellen. Zu beachten ist hierbei, dass dann die Kantengewichte komplexe Zahlen sind. Für die Variablenordnung wird im Folgenden $x_1(0) < x_1(1) < x_1(2) < x_1(3) < x_2(0) < x_2(1) < x_2(2) < x_2(3)$ angenommen.

Das Spektrum Y des Signals y erhält man durch die paarweise Multiplikation der Spektren X_1 und X_2, wie in Abb. 6.40b) dargestellt. Dies kann durch Komposition der TEDs erfolgen. Die resultierenden TEDs sind in Abb. 6.41 dargestellt. Alle vier TEDs sind bis auf die Kantengewichte isomorph.

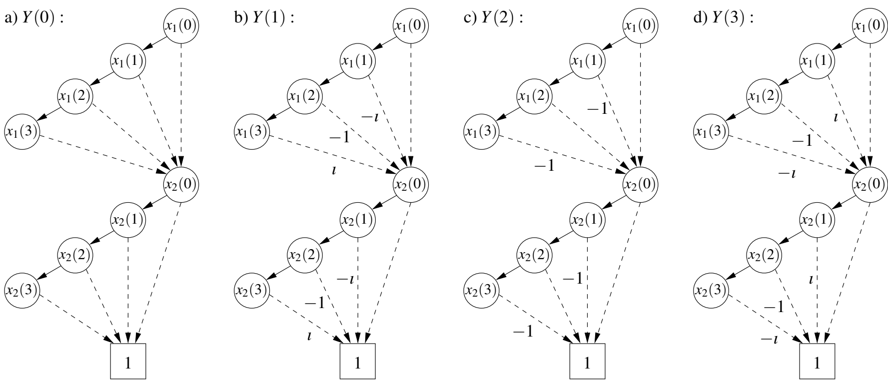


Abb. 6.41. Spektrum Y des Signals y

Im abschließenden Schritt der Faltung wird das Spektrum mit Hilfe der IDFT zurück in den Zeitbereich transformiert. Nach Gleichung (6.6) sind die folgenden Berechnungen durchzuführen:

$$
\begin{aligned}
y(0) &= \frac{1}{4} \cdot (Y(0) + Y(1) + Y(2) + Y(3)) \\
y(1) &= \frac{1}{4} \cdot (Y(0) + \imath \cdot Y(1) - Y(2) - \imath \cdot Y(3)) \\
y(2) &= \frac{1}{4} \cdot (Y(0) - Y(1) + Y(2) - Y(3)) \\
y(3) &= \frac{1}{4} \cdot (Y(0) - \imath \cdot Y(1) - Y(2) + \imath \cdot Y(3))
\end{aligned}
$$

Die Berechnung kann wieder direkt mit den TEDs durchgeführt werden. Als Beispiel ist die Berechnung von $y(1)$ in Abb. 6.42 dargestellt. Das normalisierte, reduzierte und geordnete TED ist in Abb. 6.43 zu sehen. Man erkennt, dass dieses eindeutig das Polynom für $y(1)$ aus Gleichung (6.4) darstellt.

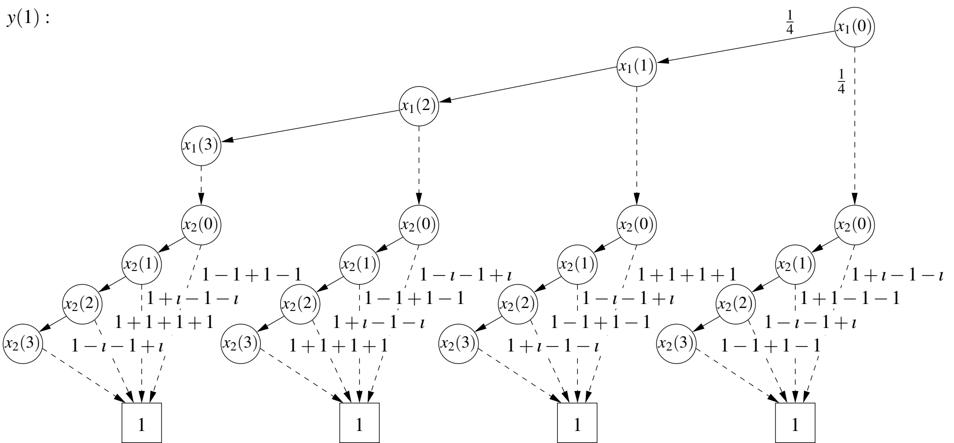


Abb. 6.42. Berechnung der IDFT

Für die anderen Signale erfolgt die Berechnung analog. Auch diese sind äquivalent mit den berechneten Signalen aus Gleichung (6.4), womit die Äquivalenz der beiden Schaltungen gezeigt ist.

Grenzen der Äquivalenzprüfung mit TEDs

Taylor-Expansions-Diagramme sind gut geeignet, um die Äquivalenz von Datentransformationen, die sich als Polynomfunktionen repräsentieren lassen, zu prüfen.

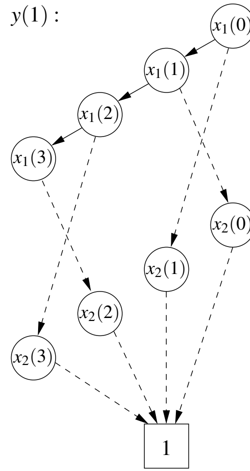


Abb. 6.43. Normalisiertes, reduziertes und geordnetes TED für $y(1)$ in der Faltung

Dafür müssen die Polynomfunktionen durch endliche Taylor-Reihen darstellbar sein. Darüber hinaus gibt es aber weitere Einschränkungen beim Einsatz von TEDs in der Äquivalenzprüfung.

Zunächst sei erwähnt, dass TEDs auch zur Äquivalenzprüfung von Hardware auf der Logikebene einsetzbar sind. Hierfür müssen die ganzzahligen Variablen binär codiert werden. Die Codierung lässt sich wie im Fall der *BMDs als ein Polynom und somit als TED repräsentieren.

Beispiel 6.2.5. Gegeben ist die Variable x, die eine vorzeichenlose ganze Zahl im Wertebereich $0,\ldots,2^n-1$ repräsentiert. Diese lässt sich als Bitvektor $(x_{n-1},\ldots,x_0)$ mit n Bits darstellen. Die Codierungsfunktion $\text{enc}:\mathbb{B}^n\to\mathbb{Z}$ weist dann der Variablen x den durch den Bitvektor $(x_{n-1},\ldots,x_0)$ repräsentierten Wert nach folgender Rechenvorschrift zu:

$$x := \sum_{k:=0}^{n-1} x_k \cdot 2^k$$

Wird lediglich ein einzelnes Bit x_k aus der Repräsentation benötigt, kann x auch als $2^{k+1}\cdot\text{enc}(x_{n-1},\ldots,x_{k+1})+2^k\cdot x_k+\text{enc}(x_{k-1},\ldots,x_0)$ codiert werden. Auf diese Art lässt sich auch elementarer Kontrollfluss in Schaltungen in Form von *if-then-else*-Anweisung mit zugehörigen Vergleichen realisieren. Sollen allerdings alle Bits einzelnen dargestellt werden, wird das TED für die Codierung schnell ineffizient.

Die bisherigen Betrachtungen gelten allerdings nur für die primären Eingänge der Schaltung. Interne Variablen, wie z. B. die Spektren der Eingangssignale der Faltung, oder Ausgänge der Schaltung lassen sich nicht mehr einfach zerlegen, da sich die Decodierung oftmals nicht als Polynomfunktion darstellen lässt. Aus dem selben Grund lassen sich TEDs im Allgemeinen auch nicht auf modulare Arithmetik anwenden, wie sie bei Hardware-Implementierungen oder bei der Implementierung

in Festpunkt-Arithmetik auf digitalen Signalprozessoren auftreten. Dies wird an einem Beispiel illustriert [398].

Beispiel 6.2.6. Gegeben sind zwei primäre Eingangssignale a und b, wobei a durch einen Bitvektor der Länge 12 und b durch einen Bitvektor der Länge 8 repräsentiert wird, d. h. $a = (a_{11}, \ldots, a_0)$ und $b = (b_7, \ldots, b_0)$. Ein Bildverarbeitungssystem führt die folgende Berechnung durch:

$$f := 16384 \cdot (a^4 + b^4) + 64767 \cdot (a^2 - b^2) + a - b + 57344 \cdot a \cdot b \cdot (a - b) \tag{6.7}$$

Eine alternative Implementierung des Systems könnte aber auch lauten:

$$g := 24576 \cdot a^2 \cdot b + 15615 \cdot a^2 + 8192 \cdot a \cdot b^2 + 32768 \cdot a \cdot b + a + 17153 \cdot b^2 + 65535 \cdot b \tag{6.8}$$

Die Genauigkeit beider Berechnungen ist dabei auf 16 Bit beschränkt, d. h. $f = (f_{15}, \ldots, f_0)$ und $g = (g_{15}, \ldots, g_0)$. In der Tat sind diese beiden Systeme beschrieben durch Gleichung (6.7) und (6.8) äquivalent [398].

Formal lässt sich das Problem formulieren als:

$$f(x_1, \ldots, x_d)\%n \overset{?}{\equiv} g(x_1, \ldots, x_d)\%n$$

Mit anderen Worten: Es wird überprüft, ob zwei Funktionen mit endlicher Bitbreite in der Ergebnisdarstellung äquivalent sind. Dieses Problem ist entscheidbar und $\mathcal{NP}$-schwer für $n \geq 2$ [231]. Sei $n = 2^m$, dann lässt sich das Problem in das Null-Äquivalenzproblem umformen in:

$$f\%2^m \equiv g\%2^m \Leftrightarrow (f - g)\%2^m \equiv 0$$

Für die beiden Systeme aus Gleichung (6.7) und (6.8) aus Beispiel 6.2.6 ergibt sich:

$$\left(16384 \cdot (a^4 + b^4) + 32768 \cdot a \cdot b \cdot (a + 1) + 49152 \cdot (a^2 + b^2)\right)\%2^{16} \overset{?}{\equiv} 0 \tag{6.9}$$

Zu beachten ist, dass die Koeffizienten dieses Polynoms von null verschieden sind. Dennoch gilt, dass $\forall a \in \mathbb{B}^{12}, b \in \mathbb{B}^8 : (f - g)\%2^{16} = 0$, d. h. die Funktion $(f - g)$ verschwindet für die Signatur $(f - g) : \mathbb{B}^{12} \times \mathbb{B}^8 \rightarrow \mathbb{B}^{16}$. Man sagt, dass $(f - g)\%2^m$ ein *verschwindendes Polynom* ist. Herauszufinden, ob es sich bei $(f - g)\%2^m$ um ein verschwindendes Polynom handelt, ist ein *NP*-schweres Problem. Aus mathematischer Sicht gehört dieses Problem zur Klasse der Prüfung der *Idealzugehörigkeit*. Dies wird in diesem einführenden Buch nicht weiter verfolgt.

6.3 Formale Verifikation von Prozessoren

Ein Spezialfall der Äquivalenzprüfung auf Architekturebene stellt die Prozessorverifikation dar. Diese überprüft die korrekte Implementierung einer *Instruktionssatzarchitektur* (engl. *Instruction Set Architecture, ISA*) als sog. *Mikroarchitektur*. Die

Mikroarchitektur ist dabei eine Hardware-Beschreibung auf Architekturebene. Es handelt sich hierbei also um die Hardware/Software-Schnittstelle des Prozessors. Die Instruktionssatzarchitektur kann auch als eine sequentielle Mikroarchitektur des Prozessors betrachtet werden, bei der jede Instruktion in der Reihenfolge ihres Auftretens ausgeführt wird und erst nach Beendigung einer Instruktion die nächste Instruktion zur Ausführung kommt.

Beispiel 6.3.1. Ein Beispiel eines Datenpfads einer sequentiellen Mikroarchitektur ist in Abb. 6.44 zu sehen. Der Kontrollpfad ist dabei nicht dargestellt. Die Mikroarchitektur führt Lese/Schreib- und ALU-Operation (ebgl. *Arithmetic Logic Unit*) sowie Sprünge in einem einzelnen Takt aus. Die Instruktionen werden aus dem Instruktionsspeicher (engl. *instruction memory*) geladen. Die Adresse der Instruktion im Speicher, die als nächstes ausgeführt werden soll, wird durch ein Inkrement des PC (engl. *program counter*) oder der Berechnung einer Sprungadresse für den PC ermittelt.

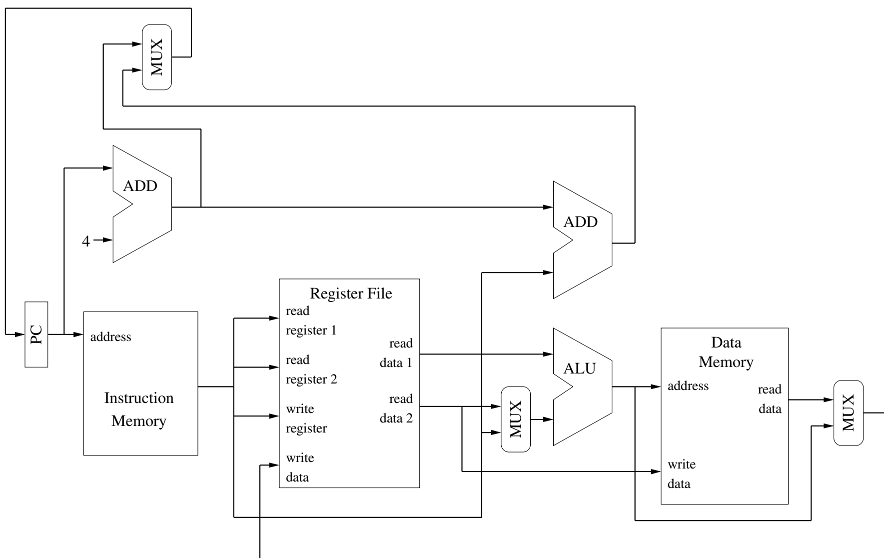


Abb. 6.44. Sequentielle Mikroarchitektur eines Prozessors [355]

In einer Instruktion sind neben der Operation, welche die ALU ausführen soll (in Abb. 6.44 wegen des fehlenden Kontrollpfads nicht zu sehen), auch deren Operanden codiert. Diese können beispielsweise im Registersatz (engl. *register file*) gespeichert sein. Der Registersatz besitzt einen Schreib- (write data) und zwei Leseports (read data 1 und read data 2). Welcher Registerinhalt am jeweiligen Leseport angezeigt wird, hängt von den Adressen read register 1 und read register 2 ab. Welches Register mit den Daten write data überschrieben wird, wird mit der Adresse write register

bestimmt. Alternativ kann ein Operand auch direkt als Konstante in der Instruktion gespeichert sein.

In der ALU wird anschließend die durch der Instruktion codierte Operation berechnet. Das berechnete Ergebnis der ALU kann entweder als Adresse zum Laden oder Speichern von Daten aus bzw. in den Hauptspeicher dienen oder im Registersatz gespeichert werden. Alternativ zu ALU-Operationen kann über einen Addierer (ADD) eine neue Sprungadresse für den PC, also die Adresse der nächsten Instruktion, berechnet werden.

Die Instruktionssatzarchitektur des Prozessors ist durch die Operationen der ALU (typischerweise Addition, Subtraktion und logische Vergleiche), den Lade- und Speicherbefehlen und den implementierten Sprungbefehlen gegeben. Auf der Mikroarchitektur in Abb. 6.44 muss zunächst die Berechnung einer Instruktion abgeschlossen sein, bevor mit Hilfe des PC eine neue Instruktion geladen wird.

Arbeitet die Mikroarchitektur wie die ISA sequentiell, so kann die Verifikation als Automatenäquivalenz formuliert werden. Die Implementierungen von Prozessoren sind allerdings immer komplexer geworden. Die Mikroarchitektur eines modernen Prozessors besteht heutzutage aus mehreren parallelen Pipelines mit Multizyklen-Funktionseinheiten, Sprungvorhersage, spekulativer Ausführung und sogar dynamischer Ablaufplanung von Instruktionen (engl. *out-of-order execution, OOO*). Diese Optimierungen an der Mikroarchitektur zielen darauf ab, den Instruktionsdurchsatz des Prozessors zu erhöhen.

Mit der zunehmenden Komplexität von Prozessoren werden zuverlässige Verifikationsansätze benötigt, die es erlauben, die Äquivalenz von ISA und optimierter Mikroarchitektur zu beweisen. Im Folgenden werden einige wichtige Ansätze zur Äquivalenzprüfung von Prozessoren näher diskutiert. Zunächst werden Verifikationsansätze für Prozessoren mit Fließbandverarbeitung (engl. *pipelining*) diskutiert. Anschließend werden Erweiterungen für Prozessoren mit Funktionseinheiten, deren Berechnungen mehrere Takte dauern, sog. *Multizyklen-Funktionseinheiten*, Mikroarchitekturen mit *Ausnahmebehandlung* und Mikroarchitekturen mit *Sprungvorhersage* betrachtet. Daneben wird in modernen Prozessoren auch ständig die Parallelität der Mikroarchitekturen erhöht. Erweiterungen zur Verifikation von sog. *superskalaren Mikroarchitekturen*, also Architekturen, welche die gleichzeitige Bearbeitung mehrerer Instruktionen erlauben, und Mikroarchitekturen, die eine *dynamische Ablaufplanung von Instruktionen* erlauben, werden zum Schluss behandelt.

6.3.1 Äquivalenzprüfung für Prozessoren mit Fließbandverarbeitung

Das Verhaltensmodell der Spezifikation eines Prozessors entspricht einem funktionalen Modell des Prozessors, wie er von einem Programmierer gesehen wird, d. h. die einzelnen Instruktionen sind in der sog. *Instruktionssatzarchitektur* zusammengefasst. Es handelt sich hierbei um eine Beschreibung, der die Ausführung von Instruktionen einzeln und nacheinander zugrunde liegt..

Die Prozessorimplementierung muss allerdings nicht zwangsläufig alle Instruktionen sequenziell ausführen. Es ist denkbar und gängige Praxis, dass die Ausführung

von Instruktionen verschränkt erfolgt. Dies wird mit Hilfe einer sog. *Fließbandverarbeitung* realisiert.

Beispiel 6.3.2. Betrachtet wird der Prozessor aus Beispiel 6.3.1 in Abb. 6.44. Eine Mikroarchitektur, die eine fünfstufige Pipeline implementiert ist in Abb. 6.45 zu sehen. Die einzelnen Pipeline-Stufen sind durch Pipeline-Register (graue Blöcke) voneinander getrennt. In der ersten Stufe erfolgt das Laden einer neuen Instruktion (engl. *instruction fetch, IF*). In der zweiten Stufe wird die Instruktion decodiert (engl. *instruction decode, ID*). Die eigentliche Berechnung erfolgt in Stufe drei (engl. *execute, EX*). Stufe vier ist den Speicherzugriffen (engl. *memory, MEM*) vorbehalten. Hier werden die entsprechenden Lade- und Speicherbefehle (engl. *load/store*) durchgeführt. In der fünften Stufe wird schließlich das Zurückschreiben (engl. *write back, WB*) des Ergebnisses in den Registersatz ausgeführt. Der Übersichtlichkeit halber wurde wiederum der Kontrollpfad des Prozessors nicht dargestellt.

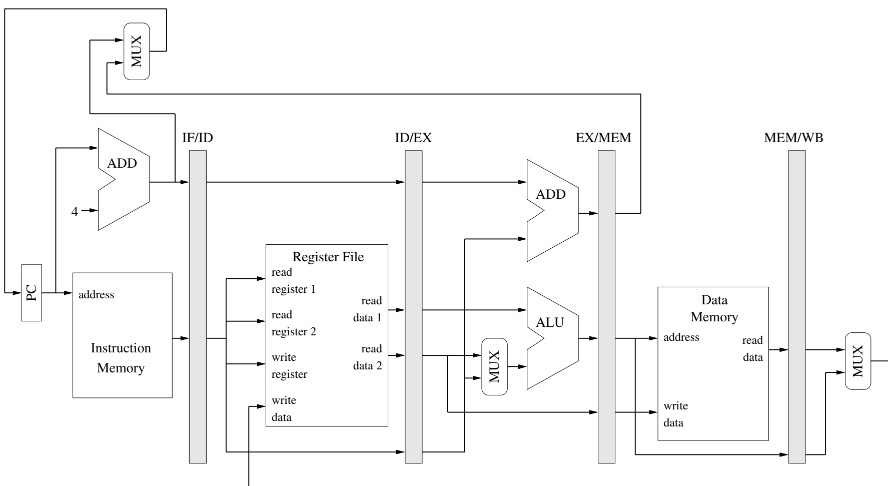


Abb. 6.45. Mikroarchitektur mit fünfstufiger Pipeline des Prozessors aus Abb. 6.44 [355]

Die Aufgabe einer formalen Äquivalenzprüfung zwischen Instruktionssatzarchitektur und Prozessorimplementierung ist, zu zeigen, dass beide die selben Ergebnisse berechnen. Dies unterscheidet sich von der Äquivalenzprüfung für endliche Automaten, da die Mikroarchitektur mit Fließbandverarbeitung mehr Zustände besitzt als die sequentielle Instruktionssatzarchitektur. Deshalb ist es notwendig, beide Architekturen lediglich bezüglich des für den Programmierer sichtbaren Zustands zu vergleichen. Dies wird in der Literatur auch als *Übereinstimmungsproblem* bezeichnet.

Äquivalenzprüfung mit Gleichheit und uninterpretierten Funktionen

Viele Entwurfsfehler bei Prozessoren mit Fließbandverarbeitung entstehen in der Steuerungseinheit des Prozessors. Aus diesem Grund werden im Folgenden ausschließlich Verfahren betrachtet, die diese Art von Entwurfsfehlern aufdecken können. Dies bedeutet, dass davon ausgegangen wird, dass die kombinatorische Logik, die den Datenpfad des Prozessors realisiert (also die eigentlichen Berechnungen durchführt), fehlerfrei ist. Zur Abstraktion des Datenpfads führen Burch und Dill die Theorie der „*Gleichheit und uninterpretierte Funktionen*“ (engl. *Equality and Uninterpreted Functions, EUF*) ein [75].

Korrektheitskriterium

Der Verifikationsprozess geht von einer Verhaltensbeschreibung in der Spezifikation (Instruktionssatzarchitektur) und einer Strukturbeschreibung der Implementierung (Mikroarchitektur) aus. Die Instruktionssatzarchitektur beschreibt, wie sich der für den Programmierer sichtbare Zustand des Prozessors durch die Ausführung einer Instruktion ändert. Dies kann durch eine sequentielle Mikroarchitektur mit Zustandsraum S_u und Eingabealphabet I modelliert werden. Die Implementierung basiert auf dem selben Eingabealphabet I und für den Programmierer sichtbaren Zustandsraum S_u. Darüber hinaus besitzt eine Mikroarchitektur mit Fließbandverarbeitung auch Pipeline-Register, deren Zustandsraum durch S_p modelliert wird.

Sowohl die sequentielle Mikroarchitektur als auch die Mikroarchitektur mit Fließbandverarbeitung lassen sich in *Übergangsfunktionen* f_{spec} bzw. f_{impl} übersetzen. Beide Übergangsfunktionen erhalten als erstes Argument den aktuellen Zustand und als zweites Argument die momentane Eingabe (Instruktion). Die Rückgabewert der Funktionen ist der Folgezustand, d. h.

$$\begin{aligned} f_{spec} &: S_u \times I \to S_u\,, \\ f_{impl} &: (S_u \times S_p) \times I \to (S_u \times S_p)\,. \end{aligned}$$

Um die *Äquivalenz von Mikroarchitektur und ISA* zu beweisen, muss gezeigt werden, dass, ausgehend von einem beliebigen Paar äquivalenter Zustände aus der Spezifikation und Implementierung, die Ausführung einer beliebigen Instruktion zu dem selben Ergebnis führt. Anschließend müssen sich Spezifikation und Implementierung zusätzlich in äquivalenten Zuständen befinden. Dies kann durch ein kommutatives Diagramm dargestellt werden (siehe Abb. 6.46). Dabei wird die Abstraktion von dem momentanen Zustand der Mikroarchitektur (Prozessorzustand) auf den für den Programmierer sichtbaren Zustand durch die Funktion $\text{abs} : (S_u \times S_p) \to S_u$ berechnet. Die Anfangszustände von ISA und Mikroarchitektur sind per definitionem äquivalent.

Um zu überprüfen, ob sich Spezifikation und Implementierung in äquivalenten Zuständen befinden, bietet es sich an, ein *Flushing* der Pipeline durchzuführen. Nahezu alle Prozessoren besitzen spezielle Instruktionen, die eine Weiterverarbeitung von Instruktionen in der Pipeline erlauben, ohne dass neue Instruktionen initiiert

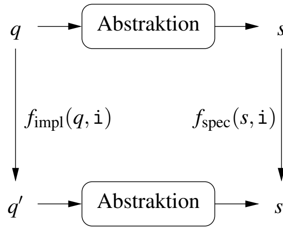


Abb. 6.46. Kommutatives Diagramm zur Visualisierung der Äquivalenzprüfung von Prozessoren [451]

werden. Eine solche spezielle Instruktion $\texttt{i}_{\texttt{stall}}$ wird als engl. *stalling* bezeichnet, d. h. $f_{\text{impl}}(q, \texttt{i}_{\texttt{stall}})$ berechnet den Folgezustand des Prozessors bei einmaliger Ausführung von $\texttt{i}_{\texttt{stall}}$ im Zustand q. Durch eine hinreichende Wiederholung dieser Instruktion können alle Instruktionen in der Pipeline abgearbeitet werden. Dies wird als engl. *Flushing* der Pipeline bezeichnet. Diese Idee führt zu dem kommutativen Diagramm in Abb. 6.47.

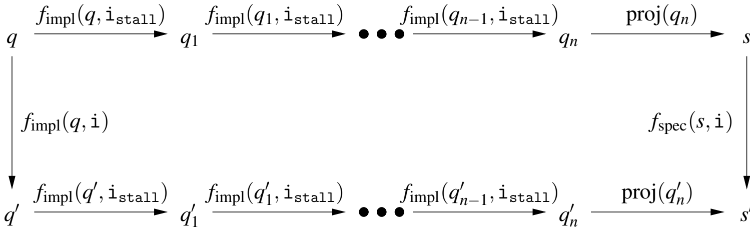


Abb. 6.47. Kommutatives Diagramm mit Flushing der Pipeline [75]

In diesem kommutativen Diagramm wird davon ausgegangen, dass sich der Prozessor in einem beliebigen Zustand q befindet. Um den für den Programmierer sichtbaren Zustand zu extrahieren, wird zunächst ein *Flushing* der Pipeline durchgeführt. Sind keine Instruktionen mehr zur Verarbeitung in der Pipeline, kann von dem Prozessorzustand q_n durch Projektion (Ausblenden des Zustands der Pipeline-Register) auf den für den Programmierer sichtbaren Zustand s geschlossen werden, d. h. es existiert eine *Projektionsfunktion* $\text{proj} : (S_{\text{u}} \times S_{\text{p}}) \to S_{\text{u}}$. Ausgehend von s kann durch Ausführung der Instruktion $\texttt{i}$ der Folgezustand s' in der Spezifikation bestimmt werden.

Der selbe Zustand s' muss als für den Programmierer sichtbaren Zustand erreicht werden, wenn sich der Prozessor im Zustand q befindet, die Instruktion $\texttt{i}$ ausgeführt, und eine Flushing der Pipeline durchgeführt wird. Nach Ausführung von $\texttt{i}$ und dem Flushing der Pipeline befindet sich der Prozessor im Zustand q'_n. Durch Projektion wird der Zustand s' in der ISA erreicht. Vergleicht man dies mit Abb. 6.46, so ergibt

sich:

$$\mathrm{abs}(s_\mathrm{u}, s_\mathrm{p}) := \mathrm{proj}(f^+_\mathrm{impl}((s_\mathrm{u}, s_\mathrm{p}), \underbrace{\langle \mathtt{i_{stall}}, \ldots, \mathtt{i_{stall}} \rangle}_{l})) \tag{6.10}$$

wobei l die Anzahl der Pipeline-Stufen und f^+_impl die erweiterte Übergangsfunktion für die Implementierung nach Gleichung (4.3) auf Seite 142 ist. Hiermit kann das Korrektheitskriterium nach Abb. 6.46 wie folgt definiert werden:

$$\forall s_\mathrm{u}, s_\mathrm{p}, i : \mathrm{abs}(f_\mathrm{impl}(s_\mathrm{u}, s_\mathrm{p}, i)) \stackrel{?}{\equiv} f_\mathrm{spec}(\mathrm{abs}(s_\mathrm{u}, s_\mathrm{p}), i) \tag{6.11}$$

Dabei ist i eine beliebige Instruktion aus dem Instruktionssatz des Prozessors.

Korrektheitsbeweis

Um zu zeigen, dass die Implementierung eines Prozessors äquivalent zu dessen Spezifikation ist, muss gezeigt werden, dass $f_\mathrm{impl}(q,i)$ und $f_\mathrm{spec}(s,i)$ äquivalent für ein beliebiges Paar äquivalenter Zustände (q,s) und eine beliebige Instruktion i sind (siehe Gleichung (6.11)). Beide Funktionen können jeweils als Vektor symbolischer Ausdrücke repräsentiert werden. Jedes Element eines Vektors entspricht dabei einer für den Programmierer sichtbaren Zustandsvariablen. Die Ausdrücke können nacheinander z. B. durch symbolische Simulation der Spezifikation bzw. der Implementierung gewonnen werden. Dabei muss die Implementierung mehrfach stimuliert werden, um das Flushing der Pipeline zu simulieren.

Seien $(q_1, \ldots, q_n)$ und $(s_1, \ldots, s_n)$ Vektoren von Ausdrücken. Um die Äquivalenz der Funktionen f_impl und f_spec zu zeigen, muss die Gleichheit der Ausdrücke q_k und s_k für alle k gezeigt werden, d. h. $\forall 1 \leq k \leq n : q_k = s_k$ gelten.

Zur Abstraktion vom Datenpfad des Prozessors werden die Ausdrücke q_k und s_k mit Hilfe der Theorie „Gleichheit und Uninterpretierte Funktionen“(engl. *Equality with Uninterpreted Functions, EUF*) gebildet. In EUF werden Funktionen nicht ausgewertet, sondern lediglich durch *Funktionssymbole* repräsentiert. Die einzige Annahme, die über Funktionen in EUF getroffen wird, lautet:

$$(x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow f(x_1, x_2) = f(y_1, y_2)$$

Dies bedeutet, dass Funktionen, die durch das selbe Funktionssymbol f repräsentiert werden, und die mit äquivalenten Argumenten aufgerufen werden, das gleiche Ergebnis liefern.

Beispiel 6.3.3. Betrachtet wird der Prozessor aus Beispiel 6.3.2 in Abb. 6.45. Die Einheit zum Inkrement des PC, der Addierer zur Berechnung von Sprungzielen und die ALU können jeweils durch die Funktionssymbole f_inc, f_add bzw. f_alu abstrahiert werden. Ebenfalls kann der Instruktionsspeicher durch ein Funktionssymbol f_im repräsentiert werden, da dieser niemals modifiziert wird. Dies ist in Abb. 6.48 zu sehen.

EUF kann durch die folgende Syntax repräsentiert werden:

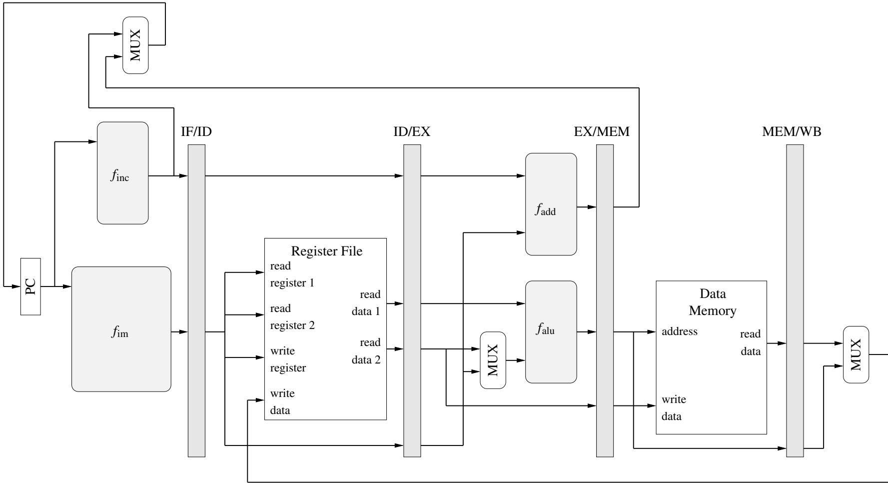


Abb. 6.48. Mikroarchitektur des Prozessors aus Abb. 6.45 mit Funktionssymbolen

$$\begin{aligned} term &::= \text{ITE}(formula, term, term) \\ &\quad \mid function_symbol(term, \ldots, term) \\ formula &::= \mathsf{F} \mid \mathsf{T} \mid \neg formula \\ &\quad \mid (formula \wedge formula) \mid (formula \vee formula) \\ &\quad \mid (term = term) \mid predicate_symbol(term, \ldots, term) \end{aligned}$$

In EUF tragen Formeln ($formula$) entweder den Wert F (*falsch*) oder T (*wahr*). Terme ($term$) können beliebige Werte tragen und werden aus *uninterpretierten Funktionssymbolen* und der Anwendung des ITE-Operators (engl. *if-then-else*) gebildet. Der ITE-Operator wählt dabei zwischen zwei Termen aus, basierend auf dem Wert einer Steuerungsvariablen, d. h. $\text{ITE}(\mathsf{F}, x_1, x_2)$ ergibt x_2, während $\text{ITE}(\mathsf{T}, x_1, x_2)$ das Ergebnis x_1 liefert.

Formeln werden gebildet, indem zwei Terme auf Gleichheit überprüft, ein *uninterpretiertes Prädikatensymbol* ($predicate_symbol$) auf eine Liste von Termen angewendet, oder Formeln mit Hilfe aussagenlogischer Operatoren ($\wedge, \vee$) verknüpft werden. Eine Formel, die zwei Terme auf Gleichheit überprüft, wird als *Gleichung* bezeichnet. Der Begriff *Ausdruck* bezeichnet im Folgenden sowohl Terme als auch Formeln.

Zu jedem Funktionssymbol f gibt es eine sog. *Ordnung* $\text{ord}(f)$, welche die Anzahl der Argumente angibt. Funktionssymbole mit Ordnung null werden als Variablen betrachtet. In diesem Fall wird anstatt von $v()$ auch die kurze Schreibweise v verwendet. Ebenfalls besitzen Prädikatensymbole p eine *Ordnung* $\text{ord}(p)$. Prädikatensymbole der Ordnung null werden als aussagenlogische Variablen bezeichnet. In diesem Fall wird a anstelle von $a()$ geschrieben.

Der Wahrheitsgehalt einer Formel wird relativ zu einer nichtleeren Definitionsmenge $\mathcal{D}$ und einer Interpretation $\mathcal{I}$ der Funktions- und Prädikatensymbole definiert. Eine Interpretation $\mathcal{I}$ weist jedem Funktionssymbol der Ordnung k eine Funktion von $\mathcal{D}^k$ nach $\mathcal{D}$ und jedem Prädikatensymbol der Ordnung k eine Funktion von $\mathcal{D}^k$ nach $\{\mathsf{F},\mathsf{T}\}$ zu. Im Sonderfall der Ordnung null wird dem Funktionssymbol (Prädikatensymbol) eine Variable aus der Definitionsmenge (der Menge $\{\mathsf{F},\mathsf{T}\}$) zugewiesen.

Gegeben sei eine Interpretation $\mathcal{I}$ sowie ein Ausdruck E. Die Bewertung von E unter der Interpretation $\mathcal{I}$, geschrieben als $\mathcal{I}[E]$, kann rekursiv entsprechend Tabelle 6.5 erfolgen.

Tabelle 6.5. Bewertung von EUF Formeln und Termen [67]

Ausdruck E	Bewertung $\mathcal{I}[E]$
F	F
T	T
$\neg F$	$\neg\mathcal{I}[F]$
$F_1 \wedge F_2$	$\mathcal{I}[F_1] \wedge \mathcal{I}[F_2]$
$p(T_1,\ldots,T_k)$	$\mathcal{I}(p)(\mathcal{I}[T_1],\ldots,\mathcal{I}[T_k])$
$T_1 = T_2$	$\mathcal{I}[T_1] = \mathcal{I}[T_2]$
$\text{ITE}(F,T_1,T_2)$	$\text{ITE}(\mathcal{I}[F],\mathcal{I}[T_1],\mathcal{I}[T_2])$
$f(T_1,\ldots,T_k)$	$\mathcal{I}(f)(\mathcal{I}[T_1],\ldots,\mathcal{I}[T_k])$

Falls $\mathcal{I}[F] = \mathsf{T}$ gilt, so sagt man, dass die Formel F unter der Interpretation $\mathcal{I}$ gültig ist. Eine Formel ist gültig in der Domäne $\mathcal{D}$, falls sie für alle Interpretationen über $\mathcal{D}$ gültig ist. Schließlich heißt die Formel F *allgemeingültig*, falls sie über alle Definitionsbereiche $\mathcal{D}$ gültig ist. Es ist bekannt, dass eine gegebene Formel über dem Definitionsbereich $\mathcal{D}$ gültig ist, genau dann, wenn sie über alle anderen Definitionsbereichen mit der selben Kardinalität gültig ist. Weiterhin gilt, dass wenn eine gegebene Formel für eine angemessen große Definitionsmenge gültig ist, so ist sie allgemeingültig [2].

Reduktion von EUF auf Aussagenlogik

Eine Möglichkeit EUF-Formeln auf ihre Gültigkeit zu überprüfen besteht darin, die EUF-Formeln auf aussagenlogische Formeln zu reduzieren. Hierzu müssen die Funktions- und Prädikatensymbole eliminiert werden. Grundlegende Arbeiten zur Elimination von Funktionssymbolen mit Ordnung eins und höher sind in [2] zu finden. Dabei wird jeder Term, der eine Anwendung eines Funktionssymbol beinhaltet, durch eine neue Definitionsbereichsvariable ersetzt und zusätzlich Beschränkungen zu der Formel hinzugefügt, um funktionale Konsistenz zu erreichen.

Ein alternativer Ansatz ist in [67] beschrieben, bei dem jede Anwendung eines Funktionssymbols durch einen ITE-Operator ersetzt wird. Die Idee dabei ist, dass über alle Funktions- und Prädikatensymbole mit Ordnung eins oder höher iteriert wird, und dabei jedes Auftreten des Symbols durch eine ITE-Operation eliminiert wird. Ohne den Algorithmus aus [67] zu wiederholen, wird das Prinzip anhand eines Beispiels demonstriert.

Beispiel 6.3.4. Gegeben ist die EUF-Formel:

$$\neg(x = y) \vee (\mathrm{h}(\mathrm{g}(x), \mathrm{g}(\mathrm{g}(x))) = \mathrm{h}(\mathrm{g}(y), \mathrm{g}(\mathrm{g}(x)))) \tag{6.12}$$

Schematisch lässt sich Gleichung (6.12) wie in Abb. 6.49 dargestellt repräsentieren. Definitionsbereichsvariablen werden dabei als Eingänge dargestellt. Werte des Definitionsbereichs werden als durchgezogene Linien dargestellt. Aussagenlogische Variablen werden als gestrichelte Linien gezeichnet. Der Ausgang repräsentiert die gesamte EUF-Formel.

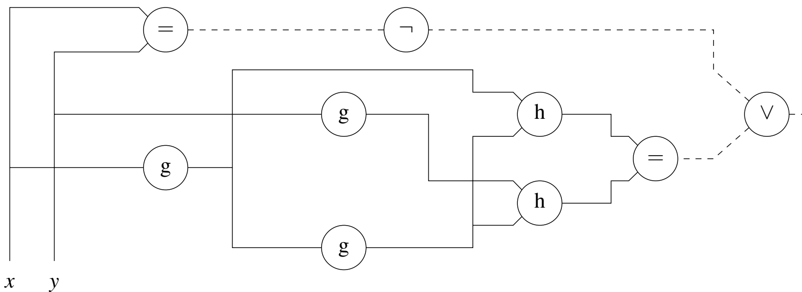


Abb. 6.49. Schematische Repräsentation von Gleichung (6.12) [67]

Um die Funktionssymbole in Gleichung (6.12) durch ITE-Operationen zu ersetzen, müssen die Funktionssymbole nacheinander betrachtet werden. Zunächst wird das Funktionssymbol g betrachtet. Insgesamt wird dieses in drei Termen in Gleichung (6.12) verwendet: $\mathrm{g}(x)$, $\mathrm{g}(y)$ und $\mathrm{g}(\mathrm{g}(x))$. Diese drei Terme werden durch die Bezeichner T_1, T_2 und T_3 identifiziert. Weiterhin werden drei neue Definitionsbereichsvariablen v_{g1}, v_{g2} und v_{g3} eingeführt. Der Term T_1 ergibt sich dann wie folgt:

$$T_1 := v_{g1} \tag{6.13}$$

Dies bedeutet, dass die Anwendung des Funktionssymbols g auf das Argument x mit einer neuen Variablen v_{g1} repräsentiert wird. Die Anwendung des Funktionssymbols g auf das Argument y kann auf ähnliche Art durch die Variable v_{g2} repräsentiert werden. Allerdings kann man an dieser Stelle bereits den Sonderfall $x = y$ mit $\mathrm{g}(x) = \mathrm{g}(y)$ berücksichtigen. Dies ergibt die folgende Definition des Terms T_2:

$$T_2 := \mathrm{ITE}(x = y, v_{g1}, v_{g2}) \tag{6.14}$$

Schließlich wird der Term T_3, der den Ausdruck $\mathrm{g}(\mathrm{g}(x))$ repräsentiert, betrachtet. An dieser Stelle muss die Variable v_{g1}, die den Ausdruck $\mathrm{g}(x)$ repräsentiert, wiederverwendet werden. Im Allgemeinen müssen verschachtelte Anwendungen von Funktionssymbolen stets von Innen nach Außen aufgelöst werden. Wie im Fall von

T_2 in Gleichung (6.14), muss bei T_3 eine Fallunterscheidung vorgenommen werden. Diese ist aber geschachtelt:

$$T_3 := \text{ITE}(v_{g1} = x, v_{g1}, \text{ITE}(v_{g1} = y, v_{g2}, v_{g3})) \tag{6.15}$$

Mit anderen Worten: Ist der Funktionswert von $\text{g}(x) = x$, so muss der Funktionswert von $\text{g}(\text{g}(x))$ ebenfalls x sein. Falls dies nicht zutrifft, der Funktionswert von $\text{g}(x)$ aber gleich y ist, so muss das Ergebnis von $\text{g}(\text{g}(x)) = \text{g}(y)$ sein. Dies wurde bereits in Gleichung (6.14) definiert. Mit der Nebenbedingung, dass $\text{g}(x) \neq x$ ist, ergibt dies v_{g2}. In allen anderen Fällen können keine weiteren Annahmen über den Funktionswert für den Term T_3 getroffen werden, d. h. der Funktionswert wird durch eine neue Variable v_{g3} repräsentiert.

Schematisch lässt sich die Elimination des Funktionssymbols g durch ITE-Operationen in dem DAG aus Abb. 6.49 einzeichnen. Verwendet man zur Repräsentation des ITE-Operators Multiplexer, so kommt man zu der Darstellung in Abb. 6.50a). Man beachte, dass über die Variablen v_{g1}, v_{g2} und v_{g3} keinerlei Annahmen getroffen wurden, weshalb die Reduktion allgemeingültig ist.

Die Elimination des Funktionssymbols h erfolgt analog mit Hilfe der Definitionsbereichsvariablen v_{h1} und v_{h2}. Dabei wird die erste Anwendung des Funktionssymbols durch die Variable v_{h1} ersetzt und die zweite Anwendung durch eine ITE-Operation, welche die Argumente der beiden Funktionsaufrufe vergleicht. Falls diese identisch sind, müssen auch die Funktionswerte identisch sein. Andernfalls muss der neue (unbekannte) Funktionswert durch eine neue Variable v_{h2} repräsentiert werden. Das Ergebnis ist in Abb. 6.50b) zu sehen.

Nach der Elimination von Funktions- und Prädikatensymbolen müssen die Definitionsbereichsvariablen noch geeignet codiert werden, um zu einer aussagenlogischen Formel zu gelangen. Dies wird hier nicht weiter betrachtet.

Effiziente Speichermodellierung

Bei der Prozessorverifikation müssen typischerweise Speicherzugriffe berücksichtigt werden. Der Adressraum wird dabei als unendlich angenommen. Kann die Äquivalenz unter dieser Annahme gezeigt werden, so gilt diese auch unter der Annahme eines endlichen Speichers (Registersatz und Hauptspeicher). Mit der oben beschriebenen Theorie zur „Gleichheit und uninterpretierte Funktionen" lassen sich unter bestimmten Konventionen Speicherzugriffe modellieren. Dies erfolgt durch eine Verschachtelung von ITE-Operatoren zur Repräsentation von Lesezugriffen auf den Speicher. In dieser Verschachtelung ist die Historie aller Schreibzugriffe erfasst, d. h. nach k Schreiboperationen an die Adressen $a_1, \ldots, a_k$ mit den Daten $d_1, \ldots, d_k$ kann der Effekt einer Leseoperation auf Adresse a wie folgt modelliert werden:

$$\text{ITE}(a = a_k, d_k, \text{ITE}(a = a_{k-1}, d_{k-1}, \ldots, \text{ITE}(a = a_1, d_1, f_{\text{init}}(a)) \ldots))$$

Dabei beschreibt $f_{\text{init}}(a)$ eine uninterpretierte Funktion, die den Anfangswert an der Adresse a repräsentiert.

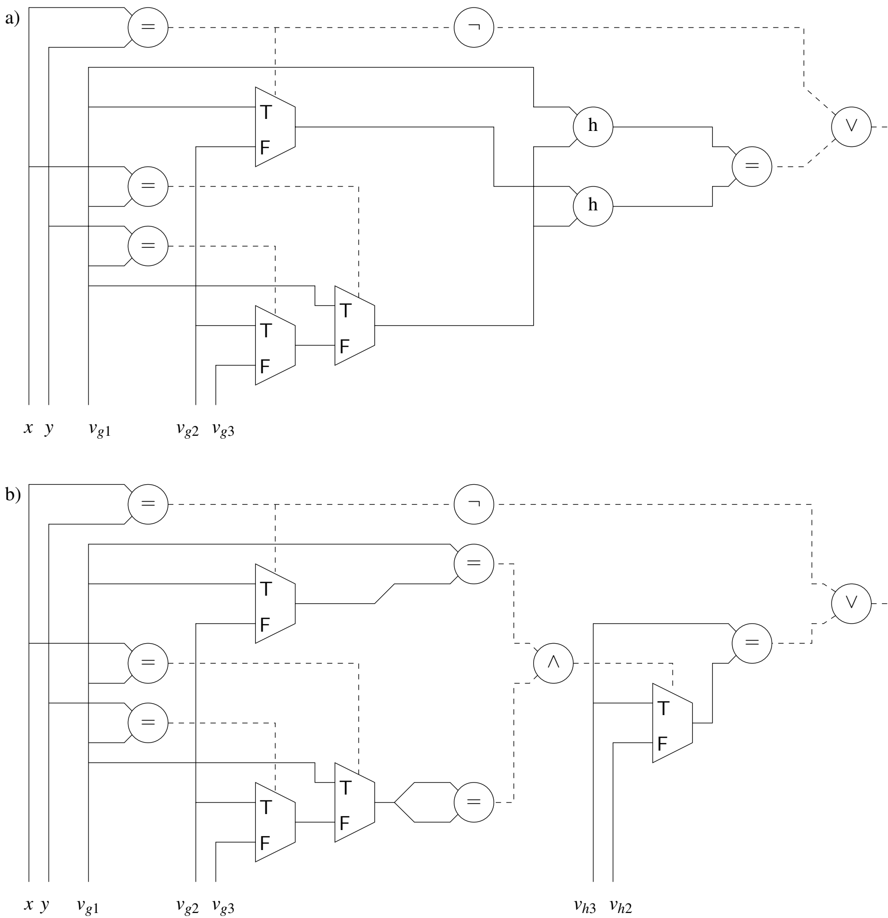


Abb. 6.50. Elimination des a) Funktionssymbols g und b) des Funktionssymbols h [67]

Oftmals ist es jedoch effektiver zwei spezielle Funktionen f_{read} und f_{write} zu verwenden. Die Funktion $f_{\text{write}}(mem, addr, val)$ besitzt drei Argumente: den momentanen Speicherinhalt *mem*, die Schreibadresse *addr* und das Datum *val*. Sie gibt den aktualisierten Speicherinhalt zurück. Die Funktion $f_{\text{read}}(mem, addr)$ besitzt zwei Argumente: den Speicherinhalt *mem* und die Leseadresse *addr*. Die Funktion gibt den an der Adresse *addr* gespeicherten Wert zurück, wobei die folgende Konvention eingehalten wird:

$$f_{\text{read}}(f_{\text{write}}(mem, addr1, val), addr2) = \begin{cases} val & \text{falls } addr1 = addr2 \\ f_{\text{read}}(mem, addr2) & \text{sonst} \end{cases}$$

Einen anderen Ansatz verfolgen Bryant und Velev durch die Einführung eines effizienten Speichermodells [68] (siehe auch Abschnitt 6.4.2). Für die symbolische

Simulation des Speichermodells werden drei Domänen von Ausdrücken unterschieden:

- *Bedingungsausdrücke*: Ausdrücke aus dieser Domäne evaluieren zu F oder T. Sie werden verwendet, um Bedingungen zu repräsentieren.
- *Adressausdrücke*: Ausdrücke aus dieser Domäne repräsentieren Adressen. Adressen können als Bitvektoren codiert werden, wobei im Folgenden davon ausgegangen wird, dass diese aus n Bit bestehen.
- *Datenausdrücke*: Ausdrücke aus dieser Domäne repräsentieren Daten. Daten können ebenfalls als Bitvektoren codiert werden. Im Folgenden haben die Bitvektoren die Breite w.

In jeder Domäne werden symbolische Variablen eingeführt, die verwendet werden, um Ausdrücke in diesen Domänen zu bilden. Weiterhin wird im Folgenden der Begriff *Kontext* für eine Belegung dieser symbolischen Variablen mit Werten verwendet.

Beispiel 6.3.5. Gegeben sind zwei Adressen a_1 und a_2 aus der Domäne der Adressausdrücke. Beide Adressen können durch Bitvektoren der Länge n repräsentiert werden, d. h. $a_1 := (a_{11}, \ldots, a_{1n})$ und $a_2 := (a_{21}, \ldots, a_{2n})$. Der Vergleich der beiden Adressen kann wie folgt durchgeführt werden:

$$(a_1 = a_2) \Leftrightarrow \neg \bigvee_{i=1}^{n} a_{1i} \oplus a_{2i}$$

Die Adressauswahl aufgrund einer Bedingung b kann mit Hilfe des ITE-Operator erfolgen. Für das i-te Adressbit gilt:

$$a_i := \text{ITE}(b, a_{1i}, a_{2i}) \Leftrightarrow a_i := (b \wedge a_{1i}) \vee (\neg b \wedge a_{2i})$$

Die Definition für die Datenausdrücke erfolgt analog, allerdings über die Breite w.

Das Speichermodell

Die Annahme bei der Modellierung von Speicherblöcken ist, dass sich diese durch einen einzelnen Speicher der Größe 2^n mit Lese- und Schreibports modellieren lassen, welche die selbe Anzahl an Adress- (n) und Datenbits (w) verwenden. Dies ist in Abb. 6.51 zu sehen. Das Modell ist äquivalent zu dem Speichermodell aus Abschnitt 6.4.2, erweitert auf mehrere Lese- und Schreibports.

Bei der symbolischen Simulation wird davon ausgegangen, dass die Lese- und Schreibzugriffe auf der steigenden Flanke des enable-Signals stattfinden. Sollten mehrere Ports gleichzeitig auf den Speicher zugreifen, werden die Zugriffe in der Reihenfolge der Prioritäten der Ports durchgeführt.

Die Repräsentation des Speichers erfolgt während der Simulation durch eine geordnete Liste, wobei der Anfang der Liste *head* die niedrigste, das Ende *tail* die höchste Priorität besitzt. Die Einträge der Liste haben die Form (c, a, d), wobei c ein Bedingungsausdruck, a ein Adressausdruck und d ein Datenausdruck ist. Der Boolesche Ausdruck c wird für konditionale Speicherzugriffe verwendet, wobei die Lese-

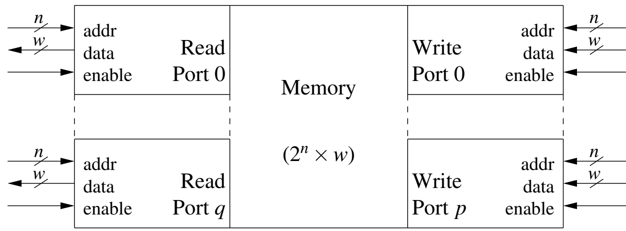


Abb. 6.51. Speichermodell

und Schreiboperationen von weiteren Statusinformationen abhängen. Der Adressausdruck legt eine Stelle im Speicher fest und der zugehörige Datenausdruck repräsentiert den Inhalt dieser Speicherstelle.

Bei einer steigenden Flanke des enable-Signals an einem Speicherport wird zunächst die Bedingung c evaluiert. Nur wenn diese nicht den konstanten Booleschen Wert F darstellt, wird der entsprechende Speicherzugriff durchgeführt. Ist die Bedingung ungleich F, wird der Adressausdruck a und der Datenausdruck d gelesen. Ein Schreibzugriff resultiert dabei im Einfügen eines neuen Eintrags (c,a,d) in die Liste. Ein Lesezugriff gibt den zugehörigen gespeicherten Datenausdruck rd im Falle, dass $c \neq \mathsf{F}$ ist, zurück. Andernfalls wird der eingelesene Datenwert d nicht verändert, d. h. $d := \text{ITE}(c, rd, d)$. Dieser Ausdruck wird auf den Datenleitungen des entsprechenden Leseports ausgegeben und kann in der weiteren symbolischen Simulation verwendet werden.

Die Prioritäten in der Liste können intuitiv so interpretiert werden, dass niederpriore Eintrage weiter in der Vergangenheit liegen, während höherpriore Einträge kürzlich vorgenommene Änderungen des Speichers darstellen. Neue Einträge können aber dennoch am Anfang (PUSH_FRONT) oder am Ende (PUSH_BACK) der Liste eingefügt werden. Schreibzugriffe können dann in Form der folgenden WRITE-Funktion erfolgen:

```
WRITE(list, c, a, d) {
    FOREACH (ec,ea,ed) ∈ list
        IF ((ec ⇒ c) ∧ (a = ea))
            DELETE(list,(ec,ea,ed));
    PUSH_BACK(list, (c,a,d));
}
```

Man beachte, dass ein Funktionsaufruf PUSH_BACK hinreichend ist, um die Schreiboperation zu implementieren. Die FOREACH-Schleife in der WRITE-Funktion dient lediglich der Optimierung. Dabei wird überprüft, ob es bereits einen Speicheränderung an der Adresse a in der Vergangenheit gab. Wenn dieser alte Wert unter keinen anderen Bedingungen als c gelesen werden kann ($ec \Rightarrow c$), kann die Speicherstelle mit dem neuen Wert überschrieben werden. Dies bedeutet, der alte Eintrag kann gelöscht werden.

Die Schreiboperation READ erhält als Argumente die Liste, die den Speicher repräsentiert, einen Bedingungsausdruck c, einen Adressausdruck a und einen Datenausdruck d. READ gibt einen Datenausdruck zurück.

```
READ(list, c, a, d) {
    g := GENERATE_DEXPR();
    RETURN READ_WITH_DEFAULT(list, c, a, g);
}
```

Der eigentliche Lesezugriff ist in der Funktion READ_WITH_DEFAULT implementiert. Die Methode GENERATE_DEXPR dient dazu, während der symbolischen Simulation einen Initialwert auf Anfrage zu bestimmen. Kann in der Funktion READ_WITH_DEFAULT kein aktueller Eintrag für die Speicherstelle ermittelt werden, wird statt dessen der Initialwert zurückgegeben. Damit spätere Zugriffe den selben Initialwert lesen, muss der Initialwert allerdings noch in der Liste (niederprior) eingetragen werden. Die Funktion READ_WITH_DEFAULT ist wie folgt umgesetzt:

```
READ_WITH_DEFAULT(list, c, a, d) {
    rd := d;
    found := F;
    FOREACH (ec, ea, ed) ∈ list
        match := (ec ∧ (a = ea));
        rd := ITE(match, ed, rd);
        found := found ∨ match;
    IF (¬(c ⇒ found))
        PUSH_FRONT(list, (c, a, d));
    RETURN rd;
}
```

Die Funktion READ_WITH_DEFAULT durchläuft die Liste *list*, die einen Speicher repräsentiert, vom Anfang bis zum Ende. So überschreiben aktuellere Werte die älteren. Die Variable *found* zeigt dabei an, ob ein Eintrag gefunden wurde. Ist dies nicht der Fall, so wird der generierte Initialwert am Anfang (niederprior) der Liste eingetragen.

Vergleich der Speicherzugriffe

Für die Äquivalenzprüfung des Prozessors mit der ISA, ist es notwendig zu vergleichen, ob beide Modelle zu dem selben Speicherinhalt nach Ausführung der symbolischen Simulation gelangen, wenn sie mit dem selben Speicherinhalt gestartet wurden. Die Inhalte zweier Speichermodelle, wie sie oben beschrieben sind, lassen sich mit der Funktion COMPARE vergleichen. Bei dem Vergleich der beiden Speichermodelle wird dabei ausgenutzt, dass nur ein kleiner Teil des Speichers tatsächlich verändert wurde. COMPARE gibt T zurück, für den Fall, dass die beiden Speicher den selben Inhalt besitzen, andernfalls F.

```
COMPARE(mem1, mem2) {
    same := T;
    tested := ∅;
    FOREACH (ec,ea,ed) ∈ mem1
        IF (ea ∉ tested)
            g := GENERATE_DEXPR();
            d1 := READ_WITH_DEFAULT(mem1, ec, ea, g);
            d2 := READ_WITH_DEFAULT(mem2, ec, ea, g);
            same := same ∧ (d1 = d2);
        tested := tested ∪ {ea};
    FOREACH (ec,ea,ed) ∈ mem2
        IF (ea ∉ tested)
            g := GENERATE_DEXPR();
            d1 := READ_WITH_DEFAULT(mem1, ec, ea, g);
            d2 := READ_WITH_DEFAULT(mem2, ec, ea, g);
            same := same ∧ (d1 = d2);
        tested := tested ∪ {ea};
    RETURN same
}
```

Man beachte, dass die Menge *tested* lediglich Optimierung ist, um nicht Einträge zweimal zu prüfen.

Da die Initialwerte der Speicher erst auf Anfrage erzeugt werden, muss sichergestellt werden, dass beide symbolische Simulationen die selben Anfangswerte verwenden. Hierfür kann beispielsweise ein sog. *Schattenspeicher* eingesetzt werden. Dieser speichert alle Initialwerte für beide Simulationen. Jede Simulation fragt dann zunächst den Schattenspeicher ab, ob bereits ein Initialwert erzeugt wurde. Falls ja, wird dieser verwendet, andernfalls wird ein neuer Initialwert berechnet und im Schattenspeicher gespeichert. Dies kann z. B. durch die Funktion SHADOW_READ, welche die Funktion READ ersetzt, erfolgen. Dabei ist *shadow* der Schattenspeicher und *mem* das von jeweiligen Simulation verwendete Speichermodell.

```
SHADOW_READ(mem, shadow, c, a, d) {
    g := GENERATE_DEXPR();
    READ_WITH_DEFAULT(shadow, c, a, g);
    RETURN READ_WITH_DEFAULT(mem, c, a, g);
}
```

Äquivalenzprüfung

Auf Basis der Schattenspeicher kann die Äquivalenzprüfung nach Gleichung (6.11) in neun Schritten erfolgen:

1. Lade das Modell der Mikroarchitektur und erzeuge für jeden Speicher ein Speichermodell und ein Modell des Schattenspeichers
2. Simuliere symbolisch einen Schritt der Ausführung einer Instruktion
3. Führe ein Flushing der Pipeline durch

4. Tausche jedes Speichermodell mit dem zugehörigen Schattenspeichermodell
5. Führe ein Flushing der Pipeline durch
6. Tausche die Mikroarchitektur des Prozessors mit der sequentiellen ISA, wobei für alle Speichermodelle für Programmierer sichtbare Speicher erhalten bleiben
7. Simuliere symbolisch die Ausführung der selben Instruktion auf der ISA wie in Schritt 2.
8. Vergleiche das Speichermodell mem_i mit dem Modell des Schattenspeichers $shadow_i$ für jeden der $1 \leq i \leq u$ für den Programmierer sichtbaren Speicher, d. h.
$$equal_i := \text{COMPARE}(mem_i, shadow_i)$$
9. Forme einen Booleschen Ausdruck für die Äquivalenzbedingung:
$$legal_instruction \Rightarrow \bigwedge_{i=1}^{u} equal_i \tag{6.16}$$
wobei $legal_instruction$ ein Boolescher Ausdruck für die verwendete symbolische Instruktion ist.

Repräsentation von Funktionseinheiten durch Speichermodelle

Das oben vorgestellte Speichermodell eignet sich ebenfalls, um Funktionseinheiten des Prozessors zu modellieren. Dies erfolgt mit Hilfe von Modellen für Nur-Lese-Speichern, welche einen einzelnen Leseport besitzen und permanent aktiviert (enable = T) sind. Der Adressausdruck für das Speichermodell wird aus den Operanden und den Steuerungssignalen gebildet. Der Datenausdruck stellt das Ergebnis der Berechnung dar.

Beispiel 6.3.6. Abbildung 6.52a) zeigt als Beispiel einer Funktionseinheit eine ALU. Die Abstraktion als Speichermodell ist in Abb. 6.52b) zu sehen. Der Adressausdruck wird dabei aus den Operanden (Op1 und Op2) und den Steuerungssignalen (Ctrl) gebildet.

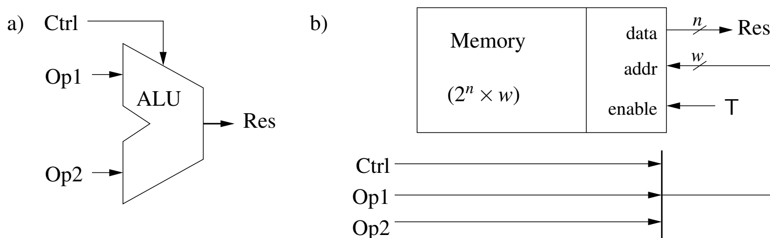


Abb. 6.52. a) ALU und b) Abstraktion als Speichermodell

Da ein solches Speichermodell niemals geschrieben wird, werden stets nur die Initialwerte gelesen, welche die Funktionseinheit implementieren. Durch die Implementierung der SHADOW_READ-Funktion ist sichergestellt, dass die Ergebnisse auch stets identisch sind.

6.3.2 Berücksichtigung von Multizyklen-Funktionseinheiten, Ausnahmebehandlung und Sprungvorhersage

Im Folgenden werden Erweiterungen der Verifikationsmethoden für Multizyklen-Funktionseinheiten im Datenpfad, Mikroarchitekturen mit Ausnahmebehandlung und Mikroarchitekturen mit Sprungvorhersage betrachtet. Die beschriebenen Methoden stammen aus [453].

Multizyklen-Funktionseinheiten

Die bisherigen Ansätze zur Äquivalenzprüfung von ISA und Mikroarchitektur gehen davon aus, dass die Funktionseinheiten im Datenpfad der Mikroarchitektur ihre Ergebnisse innerhalb eines Taktzyklus berechnen. Implementiert der Datenpfad des Prozessors aber komplexere Operationen, ist diese Annahme nicht mehr realistisch. Dies hat zur Folge, dass sog. *Multizyklen-Funktionseinheiten* ihre Ergebnisse erst nach zwei oder mehr Takten berechnet haben. Die Latenz einer Funktionseinheit kann sogar von der selektierten Operation oder den Operanden abhängen. Im Folgenden wird daher eine Modellerweiterung für Multizyklen-Funktionseinheiten vorgestellt.

Konkret werden Funktionseinheiten mit folgenden Eigenschaften betrachtet:

- Die Funktionseinheit enthält keine Verklemmungen und beendet eine Multizyklen-Operation in endlicher Zeit.
- Die Funktionseinheit kann Operanden, die lediglich im ersten Taktzyklus verfügbar sind, speichern.
- Die Funktionseinheit kann das berechnete Ergebnis solange speichern, bis die nachfolgende Pipeline-Stufe das Ergebnis weiterverarbeiten kann.
- Die Funktionseinheit ist in der Lage, eine gestartete Berechnung auf Anweisung abzubrechen und eine neue Berechnung zu starten.

Diese Eigenschaften können beispielsweise mit Modellprüfungsmethoden für einzelne Funktionseinheiten überprüft werden.

Eine Möglichkeit eine Multizyklen-Funktionseinheit zu modellieren, besteht darin, die Funktionseinheit durch ein Funktionssymbol zu repräsentieren. Diesem folgen $n-1$ hintereinander geschaltete Speicherelemente. Dabei wird die Kette aus Speicherelementen verwendet, um das Ergebnis zu verzögern, so dass das Ergebnis in dem für den Programmierer sichtbaren Zustand erst n Takte nach Start der Berechnung sichtbar wird. Dieser Ansatz kann allerdings zu sehr komplexen Modellen führen, möchte man Mikroarchitekturen mit vielen Multizyklen-Funktionseinheiten mit unterschiedlichen Latenzen modellieren. Weiterhin lassen sich so auch keine

Multizyklen-Funktionseinheiten mit Latenzen, die von den Operanden oder Umgebungsbedingungen, wie Speicher- oder Cache-Zustand, abhängen, modellieren.

Velev und Bryant schlagen deshalb einen alternativen Ansatz vor, den sie als *beschleunigtes Flushing* bezeichnen. Hierfür verwenden sie einen Zufallszahlengenerator, der den Endzeitpunkt einer Funktionseinheit bestimmt. Die Idee ist es, wenn die Äquivalenz für einen beliebigen (zufälligen) Endzeitpunkt der Berechnung gezeigt werden kann, gilt das Ergebnis auch für die tatsächliche (evtl. datenabhängige) Latenz. Ein Zufallszahlengenerator kann, wie in Abb. 6.53 zu sehen, modelliert werden.

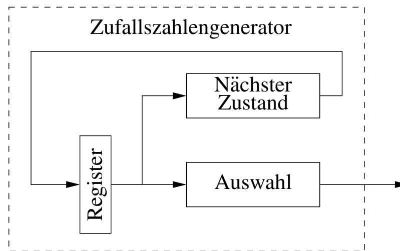


Abb. 6.53. Modell eines Zufallszahlengenerators [453]

Der Zufallszahlengenerator besitzt einen momentanen Zustand, der im Register gespeichert ist. Die uninterpretierte Funktion *Nächster Zustand* bestimmt den Nachfolgezustand. Es sei hier betont, dass weder der momentane Zustand noch die uninterpretierte Funktion zur Berechnung des Folgezustands Teil des Modells zu Äquivalenzprüfung sind. Die Sequenz von Definitionsbereichsvariablen, die durch den momentanen Zustand repräsentiert wird, wird mit Hilfe des uninterpretierten Prädikates *Auswahl* auf eine Boolesche Ausgangsvariable abgebildet. Durch Zuhilfenahme einer weiteren uninterpretierten Funktion kann der momentane Zustand auch auf eine neue Sequenz von Definitionsbereichsvariablen abgebildet werden.

Für die Abstraktion von Multizyklen-Funktionseinheiten wird im Folgenden davon ausgegangen, dass die Berechnung der Funktionseinheit durch das Funktionssymbol f_{ALU} abstrahiert werden kann. Eine Multizyklen-Funktionseinheit kann dann mit Hilfe des Zufallszahlengenerators aus Abb. 6.53, wie in Abb. 6.54 dargestellt, modelliert werden.

Die uninterpretierte Funktion f_{ALU} liefert innerhalb eines Taktes in der symbolischen Simulation das Ergebnis. Dieses wird allerdings nur im ersten Takt (engl. *first cycle*, FC) am Ausgang der Multizyklen-Funktionseinheit sichtbar, wenn die nachfolgende Stufe der Pipeline nicht blockiert ist (engl. *is stalled*, IS) und die Berechnung im ersten Taktzyklus beendet (engl. *Complete*) wurde oder das Signal Flush gesetzt ist. Ob die Berechnung in diesem Takt beendet wurde, zeigt der Zufallszahlengenerator über das Signal NDC (engl. *Non-Determinate Choice*) an. Andernfalls wird das Ergebnis von f_{ALU} in einem Register gespeichert und ausgegeben, sobald

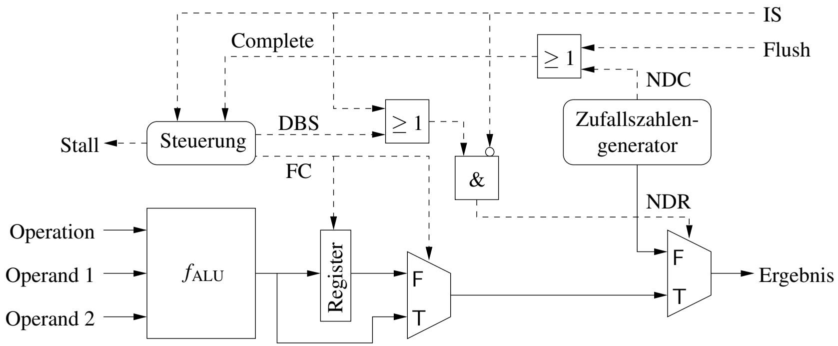
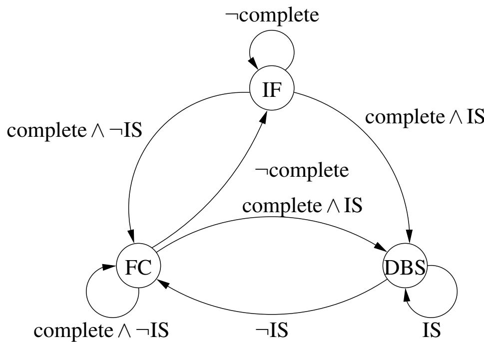


Abb. 6.54. Modell einer Multizyklen-Funktionseinheit [453]

die Berechnung beendet wurde oder das Flushing der Pipeline vorgenommen wird. Allerdings nur, sofern die nachfolgende Stufe der Pipeline nicht blockiert ist.

Solange die Berechnung nicht beendet wurde oder die nachfolgende Stufe der Pipeline blockiert ist, werden Zufallswerte NDR (engl. *non-determinate result*) ausgegeben. Die Steuerung gibt weiterhin der vorherigen Stufe der Pipeline bekannt, ob die Multizyklen-Funktionseinheit gerade blockiert (engl. *stall*) ist. Die Implementierung der Steuerung kann als endlicher Automat dargestellt werden und ist in Abb. 6.55 zu sehen.



Stall := IS ∨ ¬complete ∨ ¬DBS

Abb. 6.55. Steuerung der Multizyklen-Funktionseinheit [453]

Die Steuerung übernimmt dabei die Abstraktion von dem tatsächlichen Zeitverhalten. Bei dem endlichen Automaten handelt es sich um einen Moore-Automaten, d. h. die Ausgabe ist mit dem aktuellen Zustand assoziiert. Insbesondere ist die Aus-

gabe des endlichen Automaten in Abb. 6.55 gleich dem Zustandsnamen, wobei FC für engl. *first cycle*, IF für engl. *in flight* und DBS für engl. *done but stalled* steht. IF modelliert somit den Zustand, dass der erste Taktzyklus bereits beendet wurde, die Berechnung aber noch nicht abgeschlossen ist. FC modelliert entsprechend, dass die Funktionseinheit gerade eine neue Operation und zugehörige Operanden angenommen hat, den ersten Taktzyklus aber noch nicht beendet hat. Schließlich modelliert DBS den Zustand, dass die Berechnung abgeschlossen ist, aber die nachfolgende Stufe der Pipeline blockiert ist. Das Signal ist notwendig, da der Zufallszahlengenerator im nächsten Simulationsschritt nicht mehr die Beendigung der Operation anzeigen muss. Sollten spezielle Operationen der ALU garantiert in einem Takt beendet sein, so kann dies modelliert werden, indem das OR-Gatter, das das Signal Complete erzeugt, um weitere Eingänge erweitert wird. Signale an diesen Eingängen zeigen das Ende von solchen 1-Takt-Operationen an.

Für die Äquivalenzprüfung wird in der ISA die Multizyklen-Funktionseinheit lediglich durch die uninterpretierte Funktion f_{ALU} abstrahiert. Dadurch, dass die Operation selbst ein Argument von f_{ALU} ist, werden unterschiedliche Operationen berücksichtigt und somit in der symbolischen Simulation in jedem Simulationsschritt eine eventuell unterschiedliche Instruktion simuliert. Die Repräsentation von f_{ALU} erfolgt mit Hilfe eines Speichermodells.

Mikroarchitekturen mit Ausnahmebehandlung

Tritt eine Ausnahme auf, wird die momentane Berechnung unterbrochen und eine Ausnahmebehandlung angesprungen. Die Ausnahmebehandlung ist typischerweise in Software implementiert. Somit ist der Sprung zu einer Ausnahmebehandlung durch das Laden eines neuen Wertes für den PC implementiert. Das Verlassen der Ausnahmebehandlung ist entsprechend ein Rücksprung zur unterbrochenen Berechnung. Ausnahmebehandlung ist für den Programmierer sichtbar, d. h. sie ist Bestandteil sowohl der Spezifikation (der Instruktionssatzarchitektur) als auch der Implementierung (der Mikroarchitektur).

Für die Äquivalenzprüfung wird für jede Funktionseinheit, die eine Ausnahme anzeigen kann, ein uninterpretiertes Prädikat in der Spezifikation und der Implementierung eingeführt, welches anzeigt, ob eine Ausnahme aufgetreten ist oder nicht. Kann die Funktionseinheit die Quelle für mehrere unterschiedliche Ausnahmen sein, so wird die Art der Ausnahme auch durch eine uninterpretierte Funktion modelliert. Dies muss wiederum in der Spezifikation und in der Implementierung erfolgen, da das so modellierte Statusregister zu dem für den Programmierer sichtbaren Zustand gehört. Vor dem Rücksprung aus einer Ausnahmebehandlung muss dieses Statusregister immer zurückgesetzt werden.

Durch die Verwendung von uninterpretierten Prädikaten und Funktionen wird jedes mögliche Auftreten einer Ausnahme modelliert. Dies hat zur Folge, dass eine so modellierte Mikroarchitektur, die als äquivalent zu ihre Spezifikation erkannt wurde, auch für jede konkrete Implementierung der Ausnahmebehandlung korrekt ist. Dabei wird davon ausgegangen, dass die Ausnahmebehandlung korrekt implementiert ist.

Somit wird lediglich das korrekte Starten und Beenden von Ausnahmebehandlungen gezeigt.

Mikroarchitekturen mit Sprungvorhersage

Moderne Prozessoren implementieren häufig eine Sprungvorhersage. Diese Sprungvorhersage ist dabei nicht Teil der Instruktionssatzarchitektur (Spezifikation), sondern lediglich in der Mikroarchitektur implementiert. Enthält die nächste Instruktion einen konditionalen (engl. *branch* (B)) oder einen nichtkonditionalen (engl. *jump* (J)) Sprungbefehl, so kann bzw. wird dem PC ein bestimmter Wert zugewiesen, statt diesen sequentiell zu erhöhen. Bei konditionalen Sprüngen werden also dadurch schon Instruktionen spekulativ in der Pipeline hinter der Sprunganweisung bearbeitet. Allerdings ist erst nach Berechnung der entsprechenden Sprunginstruktion in der Execution-Stufe der Pipeline bekannt, ob die Sprungvorhersage korrekt war. Falls nicht, muss ein Flushing der Pipeline durchgeführt werden und der richtige Wert für den PC geladen werden. Eine Implementierung von Sprungvorhersage ist in Abb. 6.56 zu sehen. Dabei ist der Datenpfad und weitere Kontrolllogik nicht dargestellt.

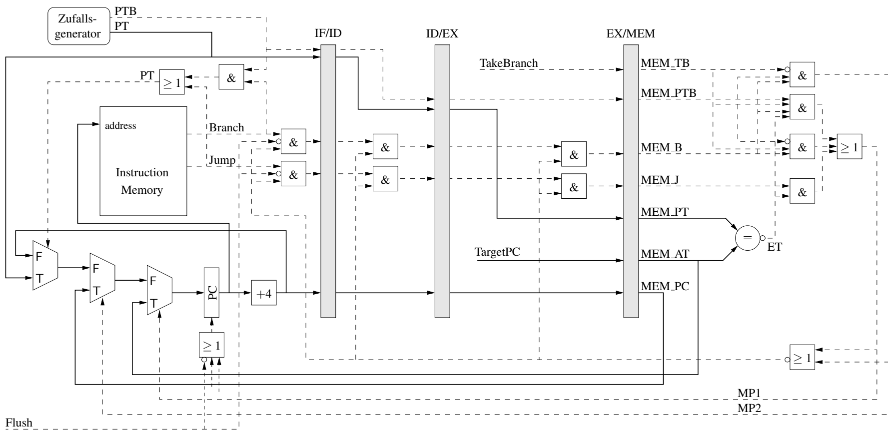


Abb. 6.56. Sprungvorhersage in einem Prozessor mit fünfstufiger Pipeline [453]

Für die Äquivalenzprüfung von Prozessoren mit Sprungvorhersage kann die Sprungvorhersage wie in Abb. 6.56 dargestellt durch einen Zufallszahlengenerator modelliert werden. Dieser liefert zwei Werte: PTB (engl. *predict taken branch*) wird mit Hilfe eines uninterpretierten Prädikates generiert und gibt an, ob einem konditionalen Sprung gefolgt wird. Der Wert von PT (engl. *predicted target*) wird mit Hilfe einer uninterpretierten Funktion gebildet und gibt das Sprungziel an. Die eventuell notwendige Korrektur findet nach der eigentlichen Berechnung der Instruktion statt.

Während der Berechnung wird festgestellt, ob der konditionale Sprung erfolgen sollte, was durch das Signal TakeBranch angezeigt wird. Weiterhin wird der nächste Wert für von PC (TargetPC) bestimmt. Die Berechnung von TakeBranch und TargetPC ist in Abb. 6.56 nicht gezeigt. Aus diesen Werten können zwei Fälle, in denen die Sprungvorhersage falsch war, abgeleitet werden:

- Fehlvorhersage MP1: Hier können drei Fälle eintreten.
  1. Die Instruktion war ein konditionaler Sprung (MEM_B) und der Sprung wurde durchgeführt (MEM_TB), was auch vorhergesagt wurde (MEM_PTB). Allerdings wurde das Sprungziel falsch geraten (MEM_PT $\neq$ MEM_AT (engl. *Actual Target*)).
  2. Die Instruktion war ein konditionaler Sprung (MEM_B) und der Sprung wurde durchgeführt (MEM_TB), was nicht vorhergesagt wurde (MEM_PTB).
  3. Die Instruktion war ein nicht konditionaler Sprung (MEM_J) und das Sprungziel wurde falsch geraten (MEM_PT $\neq$ MEM_ATC).
- Fehlvorhersage MP2: Die Instruktion war ein konditionaler Sprung (MEM_B) und es wurde ein Sprung vorhergesagt (MEM_PTB), welcher nicht durchgeführt wurde (MEM_TB).

Alle Fehlvorhersagen führen dazu, dass bereits falsche Instruktionen geladen und decodiert wurden. Zur Korrektur muss entsprechend ein Flushing der Pipeline und das Laden des korrekten Wert für PC durchgeführt werden. Hierdurch wird sichergestellt, dass nachdem der PC spekulativ geändert wurde und die Sprungvorhersage falsch war, der Prozessor in der Lage ist, diesen Fehler zu korrigieren. Man beachte dabei, dass die Verwendung von uninterpretierten Funktionen garantiert, dass jede konkrete Implementierung der Sprungvorhersage korrekt ist, sofern die Verifikation mit den uninterpretierten Funktionen und Prädikaten erfolgreich war.

6.3.3 Äquivalenzprüfung für Prozessoren mit dynamischer Instruktionsablaufplanung

Um einen höheren Durchsatz zu erreichen, verwenden *superskalare Prozessoren* mehrere, parallel arbeitende Funktionseinheiten. Auf diesen Funktionseinheiten können Instruktionen potentiell parallel abgearbeitet werden. Hängt allerdings eine Instruktion von einer anderen Instruktion ab, d. h. benötigt diese das Ergebnis einer anderen Instruktion, muss diese warten, bis das Ergebnis vorliegt. Durch Warten werden eine oder im Allgemeinen mehrere Funktionseinheiten keine Instruktion berechnen können. Bestehen solche Abhängigkeiten allerdings nicht, können Instruktionen in der Tat parallel berechnet werden. Hierfür ist eine *dynamische Ablaufplanung der Instruktionen* notwendig, die von der sequentiellen Reihenfolge in der Instruktionen auftreten abweichen kann (engl. *out of order execution*). Diese Ausführung muss die verfügbaren Funktionseinheiten sowie die Abhängigkeiten von Instruktionen untereinander berücksichtigen.

Beispiel 6.3.7. Gegeben ist folgender Programmabschnitt:

```
i1:    r0 := r0 * r1
i2:    r0 := r0 * r1
i3:    r1 := r1 + r1
```

In diesem Beispiel hängt Instruktion `i2` von dem Ergebnis von Instruktion `i1` ab. Allerdings braucht `i3` weder auf `i1` noch auf `i2` warten. Dies bedeutet, dass `i3` parallel zu `i1` und `i2` berechnet werden kann.

Enthält der Datenpfad Multizyklen-Funktionseinheiten und wurde `i3` parallel mit `i1` gestartet, kann es sein, dass `i3` vor `i1` die Berechnung beendet, da im Allgemeinen eine Addition schneller berechnet werden kann als eine Multiplikation. In diesem Fall wird `r1` mit einem neuen Wert überschrieben und die Instruktion `i2` würde mit einem falschen Wert aus Register `r1` rechnen. Dies wird als engl. (*write before read*) *hazard* bezeichnet.

Um die oben beschriebenen Hazards zu vermeiden, kann der *Algorithmus von Tomasulo* eingesetzt werden. Die Idee ist, Instruktionen der Reihe nach zuerst in einem reservierten Bereich, die sog. engl. *reservation stations*, zwischenzuspeichern. Eine Instruktion in dem reservierten Bereich kann einer freien Funktionseinheit zugewiesen werden, sobald alle Operanden berechnet sind. Um Hazards zu vermeiden, werden im Algorithmus von Tomasulo zwei weitere Mechanismen umgesetzt:

1. Neben der Instruktion selbst, werden auch die Operanden in dem reservierten Bereich gespeichert. Sollte der Wert eines Operanden noch nicht berechnet sein, ist dies entsprechend gekennzeichnet und ein Zeiger auf denjenigen Eintrag in dem reservierten Bereich gespeichert, der den Wert berechnen wird.
2. In der Speicherphase der Pipeline werden Ergebnisse nicht nur in die Register zurück geschrieben, sondern auch ein Aktualisierung der Operanden im reservierten Bereich vorgenommen, sofern notwendig.

Eine Mikroarchitektur mit dynamischer Instruktionsablaufplanung nach dem Algorithmus von Tomasulo ist in Abb. 6.57 zu sehen. Die einzelnen Speicherplätze im reservierten Bereich sind mit `s1`, ..., `s4` beschriftet. Durch die Verwendung eines Busses zwischen den ALUs und dem Registersatz kann immer nur eine Instruktion in jedem Taktzyklus beendet werden.

Abstraktionsmechanismen

Zur Äquivalenzprüfung von Mikroarchitekturen mit dynamischer Instruktionsablaufplanung mit deren Instruktionssatzarchitektur schlagen Berezin et al. [40] einen Ansatz basierend auf symbolischer Modellprüfung vor. Hierfür diskutieren sie drei mögliche symbolische Repräsentation des Problems.

Die Mikroarchitektur mit dynamischer Instruktionsablaufplanung kann durch einen endlichen Automaten modelliert werden. Dabei wird die Anzahl der Zustände allerdings sehr groß.

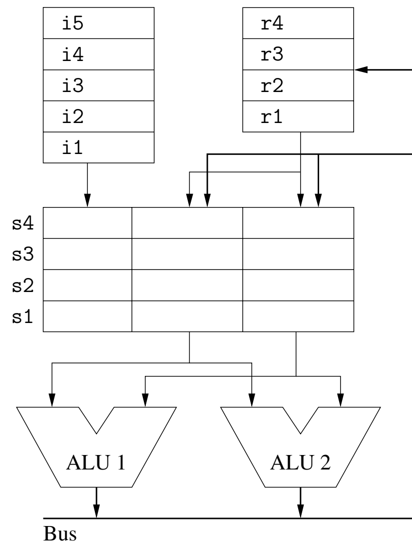


Abb. 6.57. Mikroarchitektur mit dynamischer Instruktionsablaufplanung [40]

Beispiel 6.3.8. Unter der Annahme, dass die Mikroarchitektur r Register mit je w Bit und der reservierte Bereich s Speicherplätze enthält, ergibt sich eine Gesamtspeichergröße von mindestens $w \cdot (r + 2s)$. Dies ergibt sich aus dem Fakt, dass jeder Speicherplatz im reservierten Bereich mindestens zwei Operanden speichern muss. Zur Codierung des entsprechenden Zustandsraums werden somit mindestens $n := w \cdot (r + 2s)$ Bits benötigt. Für einen Prozessor mit $w = 32$, $r = 16$ und $s = 12$ ergibt sich $n \geq 32 \cdot (16 + 24) = 960$.

Eine Verbesserung kann durch die Verwendung uninterpretierter Funktionen erreicht werden.

Beispiel 6.3.9. Die Verwendung uninterpretierter Funktionen ist in Abb. 6.58 zu sehen. Der Prozessor besitzt zwei Register und das Programm besteht aus zwei Instruktionen. Zu Beginn (Abb. 6.58a)) sind die beiden Register mit den symbolischen Werten r_1 und r_2 initialisiert. Die erste Instruktion `i1` addiert die Registerinhalte von `r1` und `r2` und speichert das Ergebnis in `r1`. Der resultierende symbolische Wert ergibt sich zu $r_1 + r_2$ in Abb. 6.58b). In der zweiten Instruktion wird der Registerinhalt von `r1` mit dem Inhalt von Register `r2` multipliziert. Das Ergebnis $((r_1 + r_2) * r_2)$ wird in `r2` gespeichert. Instruktion `i3` beendet das Programm. Man beachte, dass die Funktionssymbole $+$ und $*$ nicht ausgewertet werden.

Zur Codierung wird zu Beginn mit jedem Register eine Symbol assoziiert. Wird eine endliche Anzahl an Instruktionen ausgeführt, so ist auch die Anzahl der symbolischen Ausdrücke, die während der symbolischen Simulation entstehen, endlich. Allerdings führt eine Codierung der symbolischen Ausdrücke zu einer exponentiel-

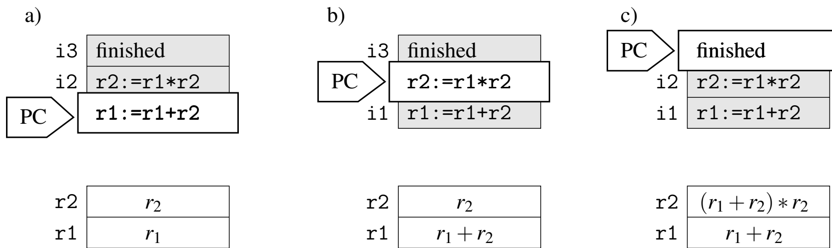


Abb. 6.58. Abstraktion durch uninterpretierte Funktionen [40]

len Anzahl an Bits, die zur Codierung benötigt werden. Aus diesem Grund schlagen Berezin et al. [40] ein anderes Codierungsverfahren vor.

Um zu einer kompakteren Codierung zu gelangen, kann man zwei Tatsachen ausnutzen: 1) In einem einzelnen symbolischen Simulationslauf werden nicht alle möglichen Ausdrücke auftreten und 2) die selben Ausdrücke werden oftmals in unterschiedlichen Registern *referenziert*. So wird z. B. in Abb. 6.58c) der Ausdruck $r_1 + r_2$ in den beiden Registern `r1` und `r2` verwendet. Eine kompaktere Codierung basiert auf einer sog. *Referenzdatei*. Register enthalten entweder konstante Symbole oder verweisen auf Einträge in der Referenzdatei. Jeder Eintrag in der Referenzdatei enthält die Anwendung einer uninterpretierten Funktion. Dabei ist jeder Operand entweder ein Register oder ein Zeiger auf einen anderen Eintrag in der Referenzdatei.

Beispiel 6.3.10. Betrachtet wird wiederum das Programm aus Abb. 6.58. Die selbe Ausführung unter Verwendung einer Referenzdatei ist in Abb. 6.59 zu sehen. Zu Beginn sind die Register auf die konstanten Symbole r_1 und r_2 initialisiert (Abb. 6.59a)). Nach Ausführung der ersten Instruktion `i1` wird die Anwendung des Funktionssymbols $+$ auf die Operanden r_1 und r_2 in der Referenzdatei unter dem Eintrag p_1 gespeichert. Das Register `r1` verweist auf diesen Eintrag (siehe Abb. 6.59b)).

Das Ergebnis der Ausführung der zweiten Instruktion `i2` ist in Abb. 6.59c) zu sehen. Die Anwendung des Funktionssymbols $*$ auf die Operanden p_1 und r_2 ist in der Referenzdatei an Position p_2 gespeichert. Man beachte, dass der Ausdruck aus der Referenzdatei an Position p_1 hierbei wiederverwendet wird. Das Register `r2` verweist auf den Eintrag p_2 in der Referenzdatei. Die Ausführung der dritten Instruktion `i3` beendet das Programm.

Vergleicht man Abb. 6.58 und Abb. 6.59, so sieht man, dass bei der Verwendung der Referenzdatei keine Ausdrücke mehr in den Registern gespeichert werden. Register enthalten nur noch konstante Symbole oder Zeiger auf Einträge in der Referenzdatei. Neue Einträge in der Referenzdatei werden nach jeder Anwendung einer uninterpretierten Funktion angelegt. Dasjenige Register, welches das Ergebnis speichern soll, wird mit einem Zeiger auf diesen Eintrag in der Referenzdatei gefüllt. Die Berechnung der symbolischen Ausdrücke, die mit einem Register assoziiert werden, lässt sich einfach durch Dereferenzierung der Zeiger durchführen.

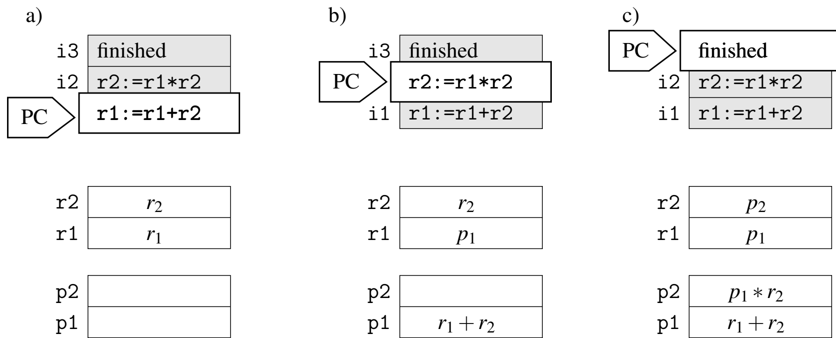


Abb. 6.59. Abstraktion durch uninterpretierte Funktionen und Referenzdatei [40]

Die Verwendung der Codierung basierend auf Referenzdateien für die Modellierung von Mikroarchitekturen mit dynamischer Instruktionsablaufplanung, die den Algorithmus von Tomasulo implementieren, wird anhand eines Beispiels beschrieben.

Beispiel 6.3.11. Betrachtet wird ein Prozessor mit drei Registern, drei Speicherplätzen im reservierten Bereich und zwei Funktionseinheiten. Das betrachtete Programm lautet:

```
i1:   r1 := r1 + r2
i2:   r2 := r1 * r2
i3:   r3 := r1 / r3
```

Die Abarbeitung des Programms auf einer Mikroarchitektur mit dynamischer Instruktionsablaufplanung, die den Algorithmus von Tomasulo implementiert, ist in Abb. 6.60 zu sehen. Zu Beginn sind die Register `r1`, `r2` und `r3` mit den konstanten Symbolen r_1, r_2 und r_3 initialisiert. In einem ersten Schritt wird die Instruktion `i1` in den reservierten Bereich auf Speicherplatz `s1` gespeichert (engl. *dispatch* (D)). Da der richtige Registerinhalt von `r1` von dieser Instruktion abhängt, wird eine Referenz s_1 auf den Speicherplatz `s1` im reservierten Bereich in diesem Register gespeichert.

Im zweiten Schritt wird die Instruktion `i1` auf der Funktionseinheit `f1` zur Ausführung (engl. *execute* (X)) gebracht. Gleichzeitig wird die Instruktion `i2` im reservierten Bereich an Position `s2` gespeichert. Hierdurch wird Register `r2` mit der Referenz s_2 auf diese Position aktualisiert. Weiterhin wird in der gespeicherten Instruktion ein Verweis auf die Instruktion an Position `s1` eingetragen, da erst nach Beendigung der dort gespeicherten Instruktion die Instruktion `i2` zur Ausführung kommen kann.

Im dritten Schritt wird die Berechnung von Instruktion `i1` auf Funktionseinheit `f1` beendet und das Ergebnis in Register `r1` (engl. *write back* (W)) gespeichert. Der resultierende symbolische Ausdruck $r_1 + r_2$ wird jedoch nicht im Register, sondern in der Referenzdatei an Position `p1` abgelegt. Das Register `r1` wird mit dem Zeiger p_1 auf die Position aktualisiert. Ebenfalls aktualisiert wird der Inhalt in Position `s2`

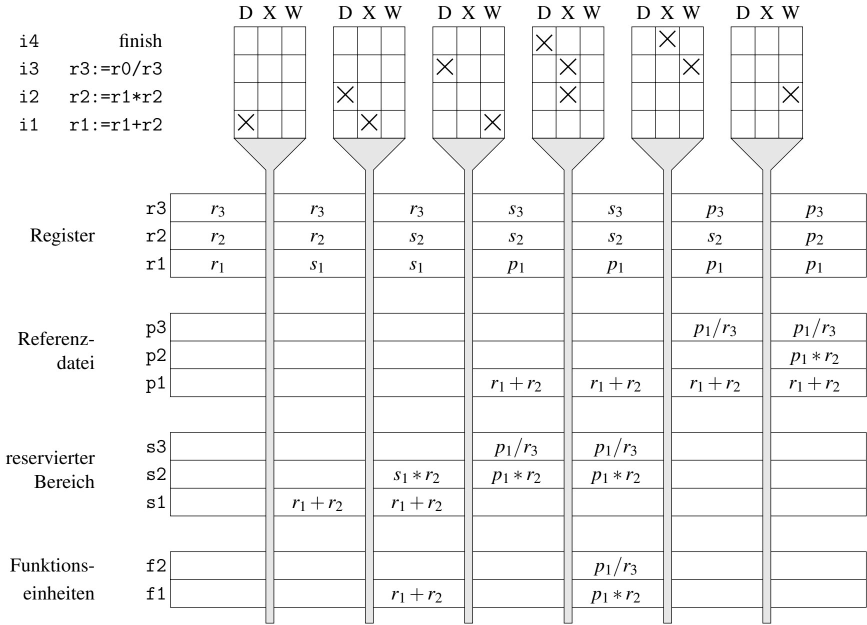


Abb. 6.60. Ausführung des Programms aus Beispiel 6.3.11 [40]

des reservierten Bereichs, da die entsprechende Instruktion auf das Ergebnis (nun an Position `p1`) gewartet hat. Da das Ergebnis aber erst nach Abschluss dieses Taktzyklus in dem Register vorliegt, kann die Instruktion `i2` in diesem Takt nicht ausgeführt werden. Parallel wird noch die Instruktion `i3` im reservierten Bereich an Position `s3` gespeichert. Auch hier wird die Referenz auf den Inhalt in Register `r1` aktualisiert.

Im vierten Schritt kommen die Instruktionen `i2` und `i3` parallel auf den beiden Funktionseinheiten zur Ausführung. Dies ist möglich, da beide Instruktionen voneinander unabhängig sind. Im fünften Schritt werden die Ergebnisse in die Register gespeichert. Da in diesem Modell nur eine Instruktion pro Taktzyklus abgeschlossen werden kann, wird lediglich das Ergebnis der Instruktion `i3` in das Register `r3` geschrieben. Hierbei wird wiederum die Referenzdatei zur effizienteren Codierung verwendet. Im sechsten Schritt wird schließlich auch das Ergebnis von Instruktion `i2` gespeichert.

Die Äquivalenzprüfung

Die Äquivalenzprüfung von Mikroarchitektur mit dynamischer Instruktionsablaufplanung und Instruktionssatzarchitektur, implementiert als sequenzielle Mikroarchitektur, erfolgt durch den Beweis einer Lebendigkeits- und einer Gefahrlosigkeitseigenschaft. Die Lebendigkeitseigenschaft besagt, dass die Mikroarchitektur bei der Abarbeitung einer endlichen Sequenz von Instruktionen irgendwann die Berechnung

beendet. Die Gefahrlosigkeitseigenschaft besagt, dass die Abarbeitung einer beliebigen, endlichen Sequenz an Instruktionen auf der sequentiellen Mikroarchitektur und der Mikroarchitektur mit dynamischer Instruktionsablaufplanung zu dem selben Ergebnis führt.

Damit diese Aussagen für beliebig lange Sequenzen gelten, wird ein Induktionsbeweis über die Länge der Instruktionssequenz durchgeführt. Dabei müssen die Zustände der sequentiellen Mikroarchitektur mit den Zuständen der Mikroarchitektur mit dynamischer Instruktionsablaufplanung verglichen werden. Im Folgenden bezeichnen p und q Zustände der Mikroarchitektur mit dynamischer Instruktionsablaufplanung und s und t Zustände der sequentiellen Mikroarchitektur. Wie in Abschnitt 6.3.1 beschrieben, sind zwei Zustände p und s äquivalent, wenn das Flushing der Mikroarchitektur mit Fließbandverarbeitung im Zustand p zum selben für den Programmierer sichtbaren Zustand s der sequentiellen Mikroarchitektur führt (siehe auch das kommutative Diagramm in Abb. 6.46 auf Seite 296).

Bei einer Mikroarchitektur mit dynamischer Instruktionsablaufplanung bedeutet Flushing, dass alle Instruktionen im reservierten Bereich zur Ausführung gebracht und zu Ende berechnet werden, ohne neue Instruktionen in den reservierten Bereich zu kopieren. Der Induktionsbeweis über die Länge n der Instruktionssequenz $i = \langle \ldots, i_{n-1}, i_n, i_{n+1}, \ldots \rangle$ erfolgt, beginnend in äquivalenten Zuständen p und s, durch Kopieren der Instruktion i_n in den reservierten Bereich der Mikroarchitektur mit dynamischer Instruktionsablaufplanung und Ausführen der selben Instruktion auf der sequentiellen Mikroarchitektur. Die sich ergebenden Zustände q und t müssen wiederum äquivalent sein. Dies wird im Folgenden formal ausgedrückt: Sei $\text{exec}(s,i)$ eine Funktion, welche die Instruktion i in Zustand s auf der sequentiellen Mikroarchitektur ausführt. Entsprechend kopiert $\text{disp}(p,i)$ die Instruktion i in den reservierten Bereich der Mikroarchitektur mit dynamischer Instruktionsablaufplanung im Zustand p. Schließlich führt $\text{flush}(p)$ ein Flushing der Mikroarchitektur durch. Man beachte dabei, dass disp und flush über mehrere Taktzyklen laufen können. Dies ist in Abb. 6.61 zu sehen. Dabei ist OOO die Mikroarchitektur mit dynamischer Instruktionsablaufplanung und SEQ die sequentielle Mikroarchitektur.

Im Folgenden wird der Beweis aus [40] skizziert: Die Induktionsannahme über die Sequenzlänge n lässt sich in folgendem Theorem formulieren:

Theorem 6.3.1. *Für jede Instruktionssequenz* $\langle i_1, \ldots, i_n \rangle$ *und die zugehörigen Zustandssequenzen* $\langle p_0, p_1, \ldots, p_n \rangle$ *und* $\langle s_0, s_1, \ldots, s_n \rangle$ *mit* $p_{k+1} := \text{disp}(p_k, i_k)$ *und* $s_{k+1} := \text{exec}(s_k, i_k)$ *gilt:*

$$(s_0 = \text{flush}(p_0)) \Rightarrow (s_n = \text{flush}(p_n))$$

Der Induktionsanfang mit $n = 0$ ist trivial. Der Induktionsschritt basiert auf folgender Annahme:

$$\forall p, i : \text{exec}(\text{flush}(p), i) = \text{flush}(\text{disp}(p, i)) \tag{6.17}$$

Wenn die Korrektheit von Gleichung (6.17) gezeigt werden kann, ist der Beweis von Theorem 6.3.1 einfach. Allerdings ist dieser Korrektheitsbeweis nicht trivial und tatsächlich der schwierigste Teil in der Verifikation.

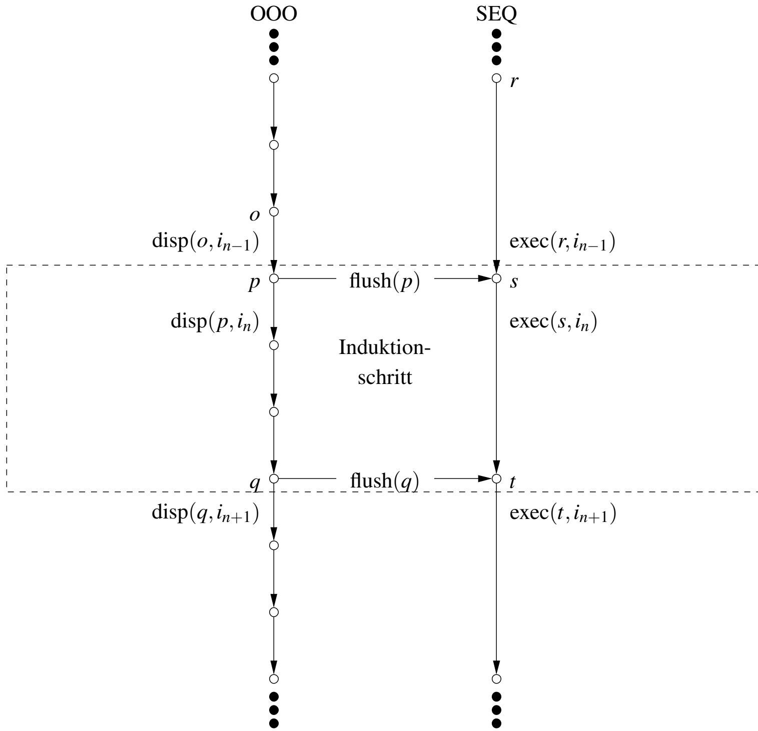


Abb. 6.61. Induktionsbeweis [40]

Eine Beobachtung ist, dass die Anzahl an Instruktionen, die in der Prozessorimplementierung zu einem Zeitpunkt gespeichert ist, endlich und meistens sehr klein ist. Besitzt der reservierte Bereich j Speicherplätze, so können $j+1$ Funktionssymbole (inklusive der Instruktion die gerade neu gespeichert wird) in der Abstraktion verwendet werden. Register enthalten anfangs konstante Symbole, aus denen mit Hilfe von uninterpretierten Funktionen neue symbolische Ausdrücken konstruiert werden. Der Zusammenhang zwischen symbolischen Ausdrücken und dem Prozessorzustand sei durch die *semantische Funktion* $[[\cdot]]$ gegeben. Die Beweisidee lässt sich dann wie in Abb. 6.62 dargestellt skizzieren.

Im Folgenden stellen die Variablen p', q', s', t' und i' symbolische Zustände und symbolische Instruktionen dar. Für den Beweis ist es ausreichend zu zeigen, dass das abstrakte Diagramm (mit den Eckpunkten p', q', s', t' in Abb. 6.62) kommutativ ist, und eine geeignete semantische Funktion $[[\cdot]]$ zur Verfügung zustellen. Dies zeigt in der Tat die Kommutativität des konkreten Diagramms (mit den Eckpunkten p, q, s, t in Abb. 6.62) für die Mikroarchitektur mit dynamischer Instruktionsablaufplanung. Die Kommutativität für das abstrakte Diagramm lässt sich wie folgt formulieren.

$$\forall p, i : \text{exec}(\text{flush}(p'), i)' = \text{flush}(\text{disp}(p', i')) \qquad (6.18)$$

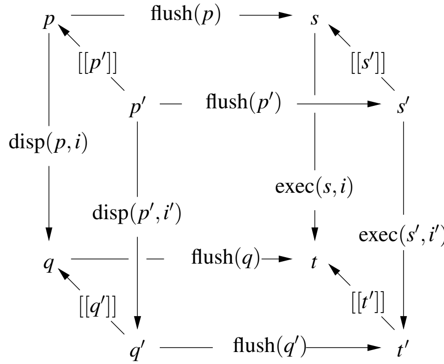


Abb. 6.62. Kommutatives Diagramm für einen konkreten und abstrakten Prozessor sowie deren Zusammenhang [40]

Eine geeignete semantische Funktion lässt sich wie folgt finden: Sei $[[\cdot]]$ eine Abbildung von Symbolen der abstrakten Definitionsbereich auf die Funktionssymbole und konstanten Symbole der Mikroarchitektur, so dass

$$\forall p', i' : \text{disp}([[p']], [[i']]) = [[\text{disp}(p', i')]]$$

und

$$\forall s', i' : \text{exec}([[s']], [[i']]) = [[\text{exec}(s', i')]]$$

gilt. Gilt weiterhin, dass

$$\forall p', i' : \text{exec}(\text{flush}(p'), i') = \text{flush}(\text{disp}(p', i'))\,,$$

so gilt

$$\forall p', i' : \text{exec}(\text{flush}([[p']]), [[i']]) = \text{flush}(\text{disp}([[p']], [[i']]))\,. \tag{6.19}$$

Gleichung (6.18) beschreibt die Kommutativität des abstrakten Diagramms in Abb. 6.62. Gleichung (6.19) beschreibt die Eigenschaften einer geeigneten semantischen Funktion $[[\cdot]]$, die garantiert, dass das konkrete Diagramm in Abb. 6.62 kommutativ ist. Ist die Gültigkeit von Gleichung (6.18) und (6.19) gezeigt, folgt die Gültigkeit von Gleichung (6.17). Was schließlich die Gültigkeit von Theorem 6.3.1 impliziert.

Berezin et al. [40] beweisen Gleichungen (6.17) und (6.19) sowie Theorem 6.3.1 mit Hilfe eines Theorembeweisers. Gleichung (6.18) kann mit Hilfe eines CTL-Modellprüfers bewiesen werden. Die CTL-Formel lautet:

$$\begin{aligned}\text{AG}\quad &(OOO.finished \wedge SEQ.finished \Rightarrow\\ &\quad OOO.regfile = SEQ.regfile \wedge OOO.reffile = SEQ.reffile)\end{aligned}$$

Dabei ist *OOO* die Mikroarchitektur mit dynamischer Instruktionsablaufplanung und *SEQ* die sequentielle Mikroarchitektur (Instruktionssatzarchitektur). *regfile* bzw.

reffile beschreiben den Zustand des Registersatzes bzw. der Referenzdateien. Durch die zusätzliche Lebendigkeitseigenschaft

$$\text{AF } \mathit{OOO.finished}$$

wird sichergestellt, dass das Flushing der Mikroarchitektur terminiert.

Optimierungen

Um reale Prozessoren verifizieren zu können, bedarf es weiterer Optimierungen der oben beschriebenen Äquivalenzprüfung.

- **Symmetriereduktion für Funktionseinheiten:** In dem oben beschriebenen Modell ist es lediglich möglich, dass eine Funktionseinheit zu jedem Zeitpunkt ihr Ergebnis zurück in den Registersatz schreibt. Nimmt man an, dass alle Funktionseinheiten alle Instruktionen ausführen können, kann man aus Symmetriegründen davon ausgehen, dass alle bis auf eine Funktionseinheit inaktiv sein werden. Sie können daher eliminiert werden.
- **Entkoppelte Simulation der Mikroarchitekturen:** Der obige Ansatz zur Äquivalenzprüfung verlangt, dass beide Mikroarchitekturen, die sequentielle und die mit dynamischer Instruktionsablaufplanung, parallel geprüft werden. Ausgehend von äquivalenten Zuständen werden also zwei unabhängige Modelle parallel geprüft. Eine signifikante Effizienzsteigerung kann erzielt werden, wenn zunächst eine Mikroarchitektur und anschließend die zweite Mikroarchitektur geprüft wird.
- **Partialordnungsreduktion des reservierten Bereichs:** Das Modell der Mikroarchitektur mit dynamischer Instruktionsablaufplanung ist unabhängig von den Positionen im reservierten Bereich. Dies bedeutet, dass eine Permutation der Positionen im reservierten Bereich zu dem selben Ergebnis führt. Solch eine Permutation ist in Abb. 6.63 zu sehen. Eine Reduktion der Zustände kann erreicht werden, wenn die Referenzen, die im reservierten Bereich verwendet werden, immer auf Positionen mit niedrigerem Index zeigen (siehe rechte Seite in Abb. 6.63). Weiterhin können alle belegten Positionen im reservierten Bereich auf Positionen mit niedrigem Index geschoben werden, so dass die Position mit hohem Index frei bleiben. Dies kann auch dynamisch erfolgen.

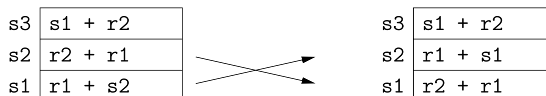


Abb. 6.63. Zustandsraumreduktion durch Permutation der Positionen im reservierten Bereich [40]

6.4 Funktionale Eigenschaftsprüfung

Für die funktionale Eigenschaftsprüfung von Hardware-Komponenten werden heutzutage im Wesentlichen zusicherungsbasierte oder SAT-basierte Verfahren eingesetzt. Beide Ansätze werden im Folgenden näher betrachtet.

6.4.1 Zusicherungsbasierte Eigenschaftsprüfung

Für die zusicherungsbasierte, simulative Eigenschaftsprüfung werden *Zusicherungen* (engl. *assertions*) zunächst in *Monitore* übersetzt (siehe Abschnitt 5.2.3). Für die funktionale Eigenschaftsprüfung von Hardware-Komponenten werden diese Monitore in Schaltungen synthetisiert. Die resultierenden *Monitorschaltungen* können anschließend zusammen mit dem Modell der Schaltung simuliert werden oder zur Überprüfung der Schaltung im späteren Betrieb auch mit gefertigt werden. Im Folgenden wird gezeigt, wie Monitore für PSL-Zusicherungen generiert werden können.

Synthese von Monitoren

Um Monitore zur simulativen Prüfung von Zusicherungen im Hardware-Entwurf einsetzen zu können, müssen Monitore synthetisiert werden. Somit werden Monitore als Komponenten implementiert, die als Eingang das Taktsignal, das Rücksetzsignal, weitere Synchronisationssignale und Signale des SUV (engl. *System Under Verification*), welche für die Überprüfung der Zusicherung beobachtet werden müssen, erhalten. Die Ausgabe der Monitorkomponente zeigt dann an, inwieweit die zu überprüfende Zusicherung momentan erfüllt ist. Eine Zusicherung kann in einer Simulation entweder:

1. *stark erfüllt sein*, wenn die Zusicherung bereits erfüllt ist und unter jedem erdenklichen Ausführungspfad (auch bei Beendigung der Simulation) weiter erfüllt sein wird,
2. *erfüllt sein*, wenn die Zusicherung bereits erfüllt wurde, aber Ausführungspfade denkbar sind, welche die Zusicherung widerlegen,
3. *ausstehend sein*, wenn die Zusicherung bisher weder erfüllt wurde noch widerlegt wurde, aber Ausführungspfade möglich sind, welche die Zusicherung erfüllen oder widerlegen, oder
4. *nicht erfüllt sein*, wenn die Zusicherung bereits widerlegt wurde und weiter keine Vervollständigung des Ausführungspfad möglich ist, auf dem die Zusicherung erfüllt ist.

Die Ausgabe der Monitorkomponenten besteht aus drei Signalen *checking*, *valid* und *pending*, für die folgende Eigenschaften gelten:

- *checking* $= \mathsf{T}$ zeigt an, dass das Signal *valid* im nächsten Takt gültig wird.
- *valid* zeigt das Ergebnis der Überprüfung der Zusicherung an, wobei T die Gültigkeit der Zusicherung und F die Verletzung der Zusicherung signalisiert.

- *pending* = T zeigt an, dass der Monitor gestartet wurde, ein endgültiges Ergebnis allerdings noch aussteht. Dieses Signal wird für die starken Operatoren in PSL benötigt.

Die Schnittstelle eines PSL-Monitors ist in Abb. 6.64 zu sehen.

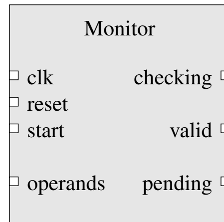


Abb. 6.64. Schnittstelle eines PSL-Monitors [335]

Da komplexe Zusicherungen als Kombinationen von elementaren Zusicherungen aufgebaut werden können, können Monitore für komplexe Zusicherungen auch durch Verschalten von Monitoren für elementare Zusicherungen gewonnen werden. Die Struktur der Verschaltung dieser Monitore entspricht der Struktur der PSL-Formel, die es zu prüfen gilt. Das folgende Beispiel stammt aus [335].

Beispiel 6.4.1. Es soll für die folgende Zusicherung ein Monitor synthetisiert werden.

assert always a -> next![2](b before! c)@(posedge clk);

Diese Zusicherung beschreibt eine Invariante, die garantiert, dass jedes Mal, wenn das Signal a den Wert T annimmt, nach zwei Takten gelten muss, dass das Signal b vor dem Signal c den Wert T annimmt. Ein zugehöriger Monitor hat also eine Schnittstelle, die zu den immer notwendigen Signalen (*clk*, *reset*, *start*. *checking*, *valid* und *pending*) die Signale a, b und c enthält.

Abbildung 6.65 zeigt zwei mögliche Signalverläufe für den Monitor, wobei die PSL-Zusicherung in Abb. 6.65a) nicht erfüllt ist. In Takt 1 wechselt das Signal a von F nach T. Somit muss, um die Zusicherung zu erfüllen, das Signal b ab Takt 3 vor dem Signal c den Wert T annehmen. In Takt 5 wechselt jedoch Signal c von F nach T, ohne dass zuvor Signal b auf T gewechselt hat. Im selben Takt wechselt das Signal *checking* auf T, womit signalisiert wird, dass das Signal *valid* im nächsten Takt das Ergebnis liefert. Im Takt 6 zeigt das Signal *valid* schließlich die Verletzung der Zusicherung an.

In Abb. 6.65b) ist die Zusicherung auf dem dargestellten endlichen Signalverlauf erfüllt. In Takt 2 nimmt das Signal a für einen Takt den Wert T an. Somit muss, um die Zusicherung zu erfüllen, Signal b ab Takt 4 vor Signal c den Wert T annehmen. In Takt 6 wechselt Signal b von F nach T. Da Signal c vorher keinen Wechsel durchgeführt hat, ist die Zusicherung erfüllt, was in Takt 7 mittels des jetzt gültigen Signals *valid* angezeigt wird. Das dieser Wert gültig ist, wird durch den Wechsel

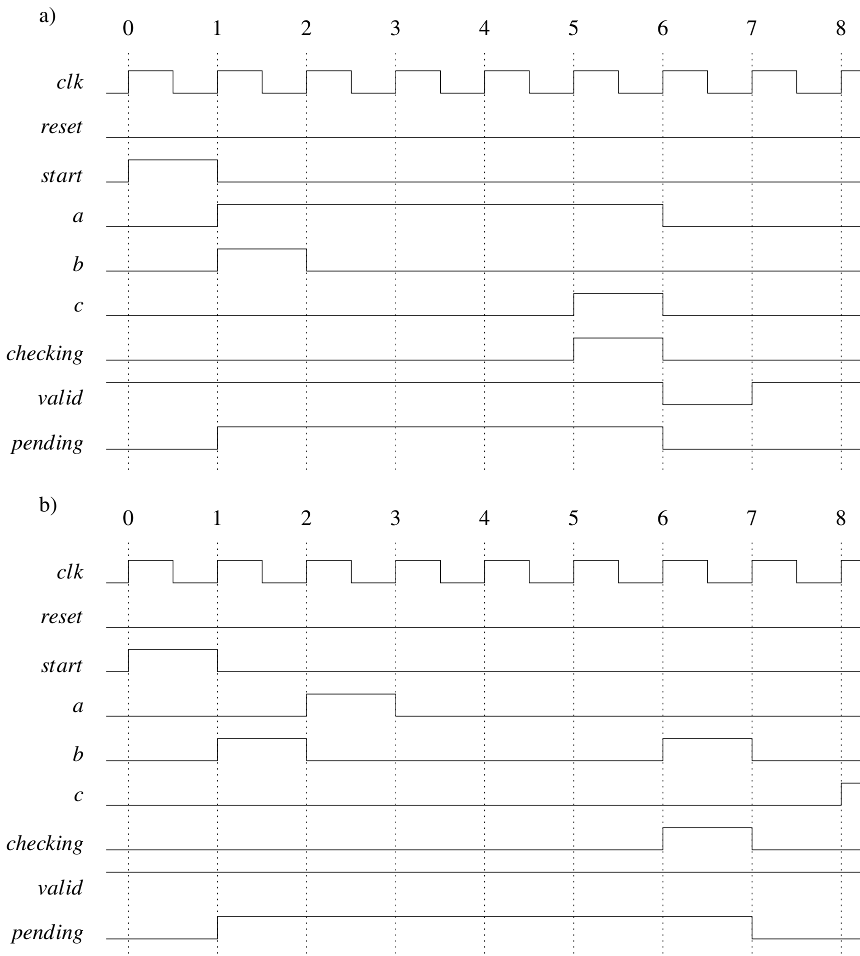


Abb. 6.65. Signalverläufe: a) verletzt die Zusicherung b) erfüllt die Zusicherung [335]

von Signal *checking* im Takt 6 von F nach T signalisiert. Man muss beachten, dass wenn die Beobachtung durch den Monitor in Takt 5 beendet worden wäre, der starke before!-Operator fehlgeschlagen wäre, was durch das Signal *pending* mit dem Wert T angezeigt ist.

Generierung komplexer Monitore durch Komposition

Um komplexe Zusicherungen zu überprüfen, werden Monitore, die elementare Zusicherungen implementieren, zusammen geschaltet. Die Verschaltung erfolgt dabei entsprechend dem Syntaxbaum der PSL-Formel. Der Syntaxbaum zu der PSL-

Zusicherung aus Beispiel 6.4.1 ist in Abb. 6.66 dargestellt. Dabei ist ein Knoten im Syntaxbaum ein PSL-Operator und die Blätter sind Operanden (Signale).

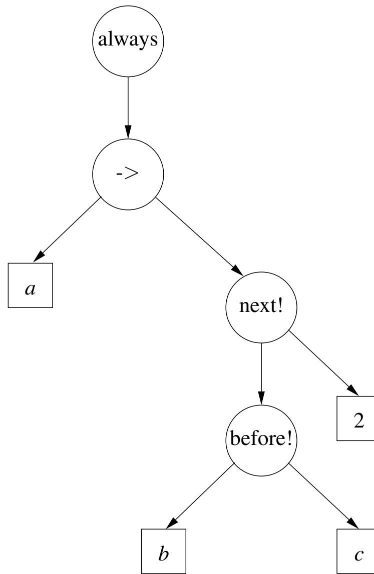


Abb. 6.66. Syntaxbaum für die Zusicherung always a -> next![2](b before! c)

Die Verschaltung erfolgt, indem Operanden mit den einzelnen Operatoren nachfolgender Regeln verbunden werden:

- Für jeden Knoten im Syntaxbaum wird ein entsprechender Monitor instantiiert.
- Die Ergebnisse Boolescher Ausdrücke aus der Schaltung werden direkt mit den Eingängen der Monitore verbunden. Falls ein Operand eine Formel mit temporalen Operatoren ist, so wird der entsprechende Eingang mit einer konstanten T verbunden. Sonst wird der Boolesche Ausdruck direkt mit den Eingängen der Monitore verbunden.
- Für zwei Operatoren $op1$ und $op2$, wobei $op2$ Operand von $op1$ ist, werden die Monitore wie folgt verschaltet:
  - Der Ausgang *checking* vom Monitor für $op1$ wird mit dem Eingang $start$ des Monitors für $op2$ verbunden.
  - Der Ausgang $valid$ vom Monitor für $op1$ wird nicht verwendet.
  - Die Signale *clk* und *reset* werden von beiden Monitoren gemeinsam verwendet.
- Ein Komponente *init* wird erzeugt, welche das Startsignal für den ersten Monitor liefert. Das Startsignal wird einen Takt nachdem das *reset*-Signal von T nach F gewechselt ist generiert.

- Die primären Ausgänge des zusammengesetzten Monitors zeigen an, inwieweit die gesamte PSL-Zusicherung erfüllt ist, wobei das *pending*-Signal als Disjunktion aller *pending*-Signale der starken PSL-Operatoren gebildet wird.

Der komplexe Monitor für die PSL-Zusicherung aus Beispiel 6.4.1 ist in Abb. 6.67 dargestellt.

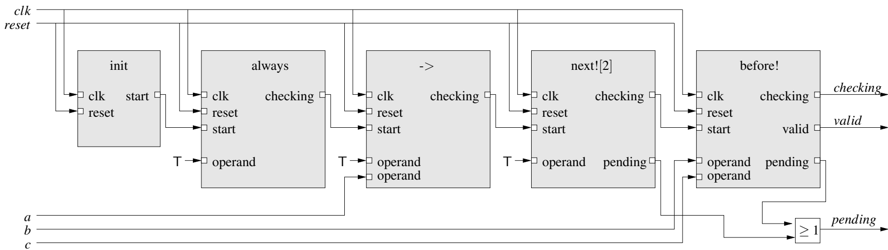


Abb. 6.67. Monitor für die Zusicherung always *a* -> next!$[2](b$ before! $c)$ [335]

Generierung der elementarer Monitore

Die einzelnen primitiven Monitore können entsprechend dem Vorgehen aus Abschnitt 5.2.3 generiert werden. Ein alternativer Ansatz ist in [334] für sequentiell erweitere reguläre Ausdrücke (SEREs) beschrieben. Für die Generierung der Signale *checking*, *valid* und *pending* wird in [333, 335] ein Struktur aus einem Block zur *Zeitfenstergenerierung* und einem Block zur *Evaluierung* vorgeschlagen.

Der Block zur Zeitfenstergenerierung setzt entsprechend der temporalen Operatoren und in Abhängigkeit des Signals *start* das Signal *pending* und *checking* sowie ein internes Signal *check*. Ein Schieberegister kann verwendet werden, wenn die Operatoren eine Überlappung von Zeitfenstern erlauben. Der Block zur Evaluierung beginnt mit der Verarbeitung der Operanden, sobald das interne Signal *check* gesetzt ist und setzt das Ausgangssignal *valid* = F. Ist das Signal *check* = F, so wird das Signal *valid* auf den Wert T gesetzt. Die Verschaltung der beiden Blöcke ist in Abb. 6.68 zu sehen.

Aufstellen von Zusicherungen für die PCI-Spezifikation

Im Folgenden wird anhand der PCI-Spezifikation [356] gezeigt, wie Zusicherungen aus einer umgangssprachlichen Spezifikation abgeleitet werden können. Das Beispiel stammt aus [111].

Peripheral Component Interconnect (PCI) Local Bus ist ein Industriestandard für 32- oder 64-Bit Bus Architekturen mit gemultiplexten Adress- und Datenleitungen. Der Bus wurde im Wesentlichen als kostengünstige aber schnelle Verbindungstechnik für integrierte Peripherie-Controller mit Speicher- und Prozessorsubsystemen entwickelt. Die Schnittstelle eines PCI-Busses ist in Abb. 6.69 zu sehen.

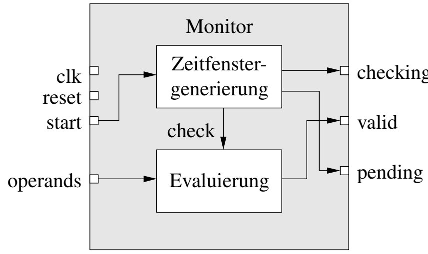


Abb. 6.68. Struktur eines primitiven PSL-Monitors [333]

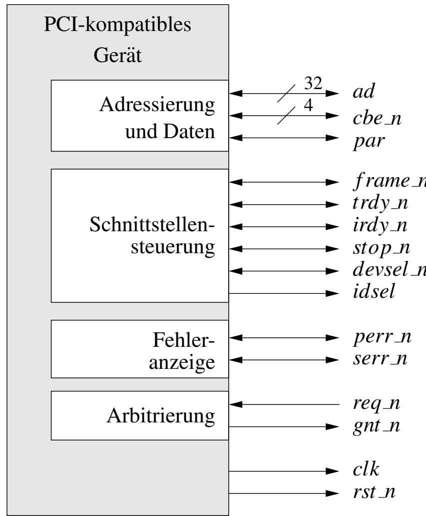


Abb. 6.69. PCI-Bus-Schnittstelle [111]

Eine Transaktion auf dem PCI-Bus besteht aus aus einer *Adressierungs-* und einer oder mehreren folgenden *Datenphasen*. Die Art der Transaktion wird während der Adressierungsphase mittels der Signale $cbe_n[3:0]$ angezeigt. Während der Datenphase dienen diese Signale als Auswahlsignale (engl. *byte enable*). Man beachte, dass das Suffix *_n* ein invers aktives Signal kennzeichnet. Die weiteren Signale werden während der Aufstellung von Zusicherungen nach Bedarf eingeführt.

Beispiel 6.4.2. Die Anforderung an das Zurücksetzen des Busses aus der PCI-Spezifikation soll in einer Zusicherung formuliert werden. Die PCI-Spezifikation gibt vor: Um zu verhindern, dass die Signale $ad[31:0]$, Signal $cbe_n[3:0]$ und Signal *par* beim Zurücksetzen unbestimmte Pegel annehmen, dürfen diese, während das Rücksetzsignal *rst_n* den Wert F besitzt, nicht den Wert T annehmen. In PSL heißt dies:

$$\text{assert always}(rst_n == \mathsf{F})\text{->!}(\{ad\} \mid \{cbe_e\} \mid \{par\})@(\text{posedge } clk);$$

Die Adresse, auf die über den PCI-Bus zugegriffen werden soll, ist durch die Adresssignale $ad[31:2]$ wählbar. Die beiden niederwertigsten Bits $ad[1:0]$ werden vom Bus-Master verwendet, um für einen Burst-Zugriff anzuzeigen, in welcher Reihenfolge Daten übertragen werden. Dabei zeigt die Belegung $ad[1:0] = (\mathsf{F},\mathsf{F})$ eine lineare Inkrementierung der Adresse an. Die Belegung $ad[1:0] = (\mathsf{T},\mathsf{F})$ steht für den sog. *cache wrap mode*. Die Belegungen $ad[1:0] = (\mathsf{F},\mathsf{T})$ und $ad[1:0] = (\mathsf{T},\mathsf{T})$ sind ungenutzt (reserviert).

Beispiel 6.4.3. Aus der obigen Darstellung folgt, dass Signal $ad[0]$ niemals den Wert T während der Adressierungsphase annehmen darf. Um dies in einer Zusicherung zu formulieren, muss zunächst die Adressierungsphase identifiziert werden. Dies erfolgt über einen sequentiell erweiterten regulären Ausdruck (SERE), der beschreibt, dass das Signal $frame_n$ zu Beginn der Adressierungsphase von T nach F wechselt. Die Signale $cbe_n[3:0]$ zeigen zu diesem Zeitpunkt die Transaktionsart an. Die Definition dieser Transaktionsarten kann in PSL mit dem ′define-Befehl erfolgen.

$$
\begin{aligned}
{}'\text{define } mem_cmd\ (&(cbe_n == {}'MEM_READ) \,||\, \backslash \\
&(cbe_n == {}'MEM_WRITE) \,||\, \backslash \\
&(cbe_n == {}'MEM_RD_MULTIP) \,||\, \backslash \\
&(cbe_n == {}'MEM_RD_LINE) \,||\, \backslash \\
&(cbe_n == {}'MEM_WR_AND_INV))
\end{aligned}
$$

Man beachte, dass anstelle der logischen Werte der Signale $cbe_n[3:0]$ Makros verwendet wurden.

Mit der Variablen mem_cmd kann nun der Beginn einer Adressierungsphase als SERE formuliert werden. Dies erfolgt in PSL mittels der sequence-Anweisung.

$$\text{sequence } SERE_MEM_ADDR_PHASE = \{frame_n; !frame_n \,\&\&\, mem_cmd\};$$

Nun kann die Eigenschaft, dass Signal $ad[0]$ in einer Adressierungsphase niemals den Wert F annimmt, formuliert werden:

$$
\begin{aligned}
&\text{property } PCI_VALID_MEM_BURST_ENC = \\
&\quad \text{always } SERE_MEM_ADDR_PHASE \text{ |-> } \{!ad[0]\} \\
&\quad\ \text{abort}!rst_n@(\text{posedge } clk);
\end{aligned}
$$

Schließlich wird diese Eigenschaft als Zusicherung formuliert:

$$\text{assert } PCI_VALID_MEM_BURST_ENC;$$

Beispiel 6.4.4. Eine einfache Lesetransaktion auf dem PCI-Bus besteht aus zwei Phasen: Der Adressierungsphase, bei der eine Adresse innerhalb eines Taktes übertragen wird, und einer Datenphase, die aus einem Datentransfer und keinem, einem oder mehreren Wartezyklen besteht. Die Adressierungsphase beginnt mit dem Wechsel des Signals $frame_n$ von T nach F, d. h.

$$\text{sequence } SERE_RD_ADDR_PHASE = \{frame_n; !frame_n \&\& mem_cmd\};$$

wobei *mem_cmd* wie folgt spezifiziert ist:

$$\begin{aligned} {}'\text{define } mem_cmd \;\; &((cbe_n == {}'IO_READ) \mid\mid \backslash \\ &(cbe_n == {}'MEM_READ) \mid\mid \backslash \\ &(cbe_n == {}'CONFIG_RD) \mid\mid \backslash \\ &(cbe_n == {}'MEM_RD_MULTIP) \mid\mid \backslash \\ &(cbe_n == {}'MEM_RD_LINE)) \end{aligned}$$

Zwischen Adressierungs- und Datenphase muss ein Taktzyklus liegen. Dieser kann durch die folgende SERE erkannt werden:

$$\begin{aligned} &{}'\text{define } ad_turn_around \; (trdy_n \; \& \; !irdy_n) \\ &\text{sequence } SERE_TURN_AROUND = \{ad_turn_around\}; \end{aligned}$$

Der Datentransport wird durch die Signale $irdy_n = \mathsf{F}$, $trdy_n = \mathsf{F}$ und $frame_n = \mathsf{F}$ angezeigt. Die Wartezyklen werden durch die Signale $irdy_n = \mathsf{T}$ oder $trdy_n = \mathsf{T}$ angezeigt. Somit kann der der Datentransport als SERE wie folgt modelliert werden:

$$\begin{aligned} &{}'\text{define } data_transfer \; (!trdy_n \; \&\& \; !irdy_n \; \&\& \; !frame_n) \\ &{}'\text{define } wait_state \; (trdy_n \mid\mid irdy_n) \\ &\text{sequence } SERE_DATA_TRANSFER = \{\{wait_state[*]; data_transfer\}[1 : inf]\}; \end{aligned}$$

Die Datenphase endet entweder, wenn Signal $trdy_n = \mathsf{F}$ oder Signal $stop_n = \mathsf{F}$ und gleichzeitig Signal $irdy_n = \mathsf{F}$ ist. Die Lesetransaktion endet sobald das Signal $frame_n$ den Wert T annimmt. Damit kann das Ende des Datentransfers als SERE definiert werden:

$$\begin{aligned} &{}'\text{define } data_complete \; ((!trdy_n \mid\mid stop_n) \; \&\& \; !irdy_n) \\ &\text{sequence } SERE_END_OF_TRANSFER = \{data_complete \; \&\& \; frame_n\}; \end{aligned}$$

Während der gesamten Datenphase bleiben die Signale $cbe_n[3:0]$ stabil, d. h.

$${}'\text{define } cbe_stable \; (cbe_n == \text{prev}(cbe_n))$$

Die Datenphase kann wie folgt als SERE formuliert werden:

$$\begin{aligned} \text{sequence } \; & SERE_DATA_PHASE = \{ \\ & \{SERE_DATA_TRANSFER; SERE_END_OF_TRANSFER\} \\ & \&\& \; \{cbe_stable\}\}; \end{aligned}$$

Die funktionale Eigenschaft einer gültigen Lesetransaktion sieht dann wie folgt aus:

$$\begin{aligned}&\text{property } PCI_READ_TRANSACTION = \\ &\quad \text{always } SERE_RD_ADDR_PHASE \mid\!=> \\ &\quad\quad \{SERE_TURN_AROUND; SERE_DATA_PHASE\} \\ &\quad\quad\quad \text{abort!} rst_n@(\text{posedge } clk);\end{aligned}$$

Dies kann als Zusicherung ebenfalls formuliert werden:

$$\text{assert } PCI_READ_TRANSACTION;$$

6.4.2 SAT-basierte Modellprüfung

SAT-basierte Modellprüfung hat sich als formales Prüfverfahren für funktionale Eigenschaften im Hardware-Entwurf etabliert. Dies liegt im Wesentlichen an der guten Skalierbarkeit des Verfahren für die Falsifikation von Eigenschaften. Dabei ist das Finden von Gegenbeispielen einer beschränkten Länge von k Zeitschritten im Fokus des Verfahrens. Die zugrundeliegende Idee ist die effiziente Übersetzung der Schaltung und der funktionalen Eigenschaft für eine gegebene Schranke k in eine Instanz des Booleschen Erfüllbarkeitsproblems. Die resultierende Formel ist genau dann erfüllbar, wenn ein Gegenbeispiel der Länge k existiert (siehe Abschnitt 5.3.2).

Die Übersetzung erfordert u. a. das Abrollen der Schaltung über k Zeitschritte. Das Ergebnis ist ein *iteratives Schaltungsmodell*, wie es bereits in Abschnitt 6.1.3 für die Äquivalenzprüfung zweier Schaltwerke definiert wurde. Solch ein Modell ist nochmals in Abb. 6.70 für das Schaltwerk aus Abb. 6.15 auf Seite 258 gezeigt. Der jeweilige Zeitschritt ist hinter den Signalnamen in den eckigen Klammern angegeben.

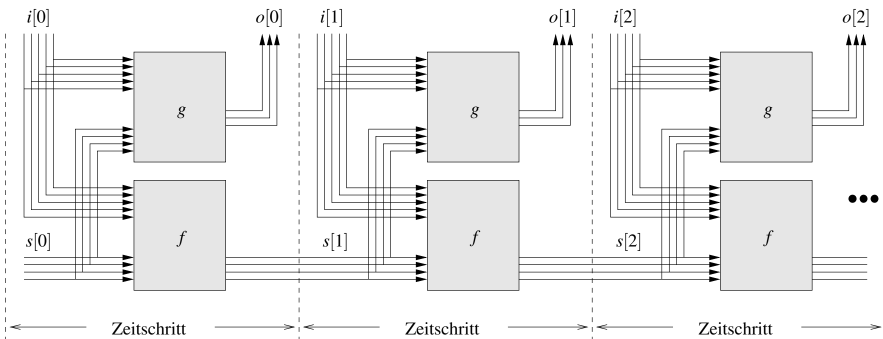


Abb. 6.70. Iteratives Schaltungsmodell des Schaltwerks aus Abb. 6.15

Das Abrollen der Schaltung beschränkt die möglichen Ausführungspfade, so dass bei der späteren Modellprüfung lediglich gültige Pfade, also solche die im Anfangszustand des Systems beginnen, betrachtet werden. Neben dem Abrollen der

Schaltung über k Zeitschritte wird die temporallogische Formel, die eine geforderte funktionale Eigenschaft spezifiziert, auf eine Art übersetzt, so dass für jeden Zeitschritt spezielle Bedingungen entstehen. Das iterative Schaltungsmodell wird mittels symbolischer Simulation ebenfalls in eine aussagenlogische Formel übersetzt. Die Konjunktion beider aussagenlogischer Formeln wird mittels eines SAT-Solvers (siehe Anhang C.2) auf Erfüllbarkeit überprüft. Kann keine erfüllende Variablenbelegung für die aussagenlogische Formel für k Zeitschritte gefunden werden, muss die Schranke k gegebenenfalls inkrementiert werden, um nach Gegenbeispielen größerer Länge zu suchen. Dabei wird mit wachsendem k die Prüfung auf Erfüllbarkeit immer schwieriger. Das folgende Beispiel stammt aus [316].

Beispiel 6.4.5. Ein synchroner, skalierbarer Bus-Arbitrierer hat n Eingänge ($req_0, \ldots, req_{n-1}$) und n Ausgänge ($ack_0, \ldots, ack_{n-1}$) wie in Abb. 6.71b) dargestellt. In jedem Takt können mehrere Bus-Anforderungen $req_i = \mathsf{T}$ gestellt werden. Die Aufgabe des Arbitrierers ist es, genau einem anfragenden Teilnehmer den Bus zuzuteilen $ack_i = \mathsf{T}$. Der Aufbau einer einzelnen Arbitriererzelle ist in Abb. 6.71a) dargestellt.

Hier wird derjenige Teilnehmer mit höherer Priorität (kleinerer Index i) den Bus zugeteilt bekommen. Hierzu erhält jede Arbitriererzelle $i-1$ von der höherprioren Arbitriererzelle i das Signal $grant_in$. Sind sowohl Signal $grant_in_i$ als auch Signal req_i auf den Wert T gesetzt, so wird Signal ack_i auf T gesetzt. Gleichzeitig wird Signal $grant_out_i$ auf F gesetzt. Trotz dieser prioritätsbasierten Arbitrierung garantiert der Arbitrierer, dass jede Bus-Anforderung irgendwann bedient wird und niederpriore Anfragen nicht aushungern. Dies wird mittels einer Marke bewerkstelligt. Diese Marke wird von der Zelle i zur Zelle $i+1$ zyklisch weitergereicht. Hierzu besitzt jede Arbitriererzelle ein Flip-Flop T. Das Flip-Flop T_0 wird zu Beginn auf den Wert T initialisiert, alle anderen Flip-Flops $T_1, \ldots, T_{n-1}$ erhalten den Wert F. Wenn eine Arbitriererzelle i die Marke erhält, d. h. $T_i = \mathsf{T}$ und eine Busanforderung ($req_i = \mathsf{T}$) vorliegt, wird ein spezielles Bit im Flip-Flop W gesetzt. Dies versetzt die Arbitriererzelle in den Zustand „wartend". Die Zelle verbleibt in diesem Zustand, so lange die Bus-Anforderung nicht zurückgenommen wird. Alle Flip-Flops W_i werden zu Beginn mit F initialisiert.

Wenn die Marke zu dieser Arbitriererzelle zurückkehrt, und diese sich weiter im Zustand „wartend" befindet, erhält diese Zelle unverzüglich die höchste Priorität. Dies geschieht, indem die Zelle den Ausgang $override_out = \mathsf{T}$ setzt. Dies sorgt dafür, dass in der höchstprioren Arbitriererzelle das $grant_in$-Signal negiert wird.

Eigenschaften, die für den skalierbaren Arbitrierer gezeigt werden sollen, sind u. a. [206]:

1. *Wechselseitiger Ausschluss:* Keine zwei Ausgänge ack_i und ack_j mit $i \neq j$ dürfen gleichzeitig den Wert T annehmen:

$$\mathrm{G}\left(\bigwedge_{i \neq j}^{n} \neg(ack_i \wedge ack_j)\right)$$

2. *Konservativität:* Signal ack_i wird nur gesetzt, wenn eine Bus-Anforderung req_i erfolgte:

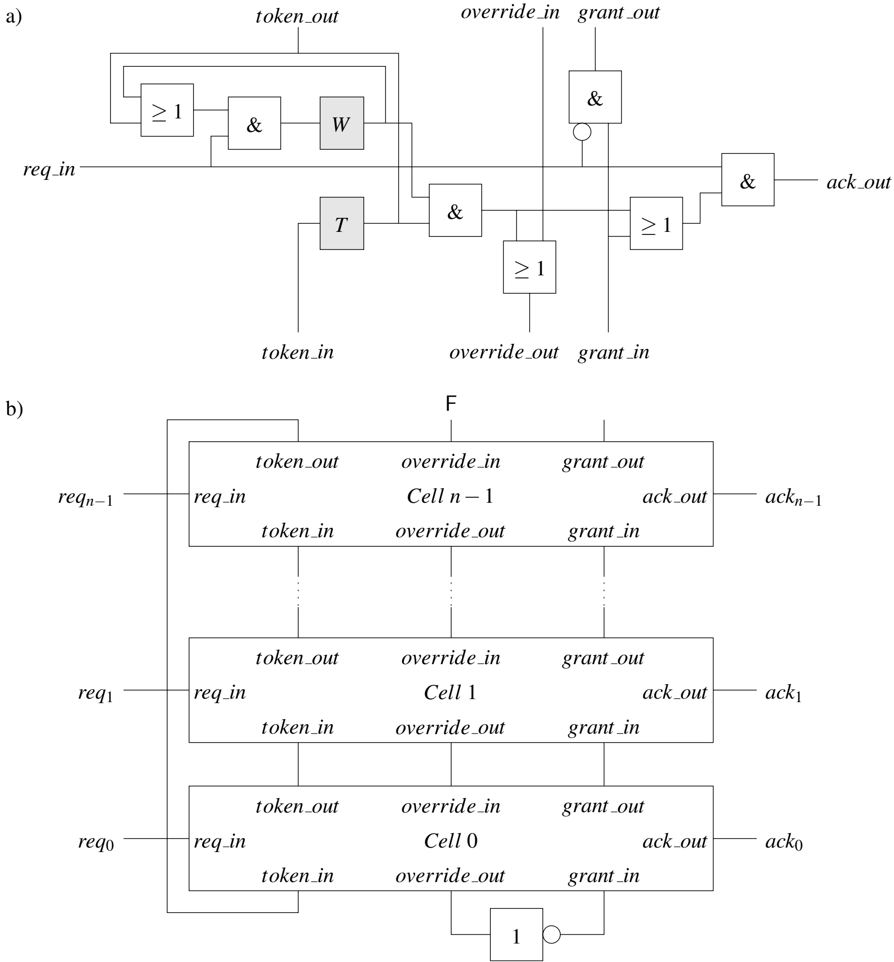


Abb. 6.71. Ein skalierbarer Arbitrierer [206]

$$\mathrm{G}\ (ack_i \Rightarrow req_i)$$

3. *Lebendigkeit:* Jede Anforderung wird innerhalb von $2n$ Zeitschritten durch Signal ack_i bestätigt:

$$\mathrm{G}\ (req_i \Rightarrow \mathrm{F}_{[0,2n-1]}\ ack_i)$$

Für die letzte Eigenschaft wurde das zu berücksichtigende Zeitintervall ($[0,2n-1]$) an den Operator F annotiert. Diese Erweiterung beschränkt die Gültigkeit des Operators auf das gegebene Zeitintervall. Weiterhin lässt sich die Lebendigkeitseigenschaft in der Form $\mathrm{G}\ (req_i \Rightarrow \mathrm{F}_{[0,2n-1]}\ ack_i)$ nicht beweisen. Hierzu ist es notwendig, eine Annahme über die Umgebung zu treffen [206]: Ein Signal req_i muss solange auf T gehalten werden, bis die zugehörige Bestätigung durch Signal ack_i erfolgte. Somit

ergibt sich als Lebendigkeitseigenschaft:

$$\mathrm{G}\left(\mathrm{G}_{[0,2n-1]}\left(req_i \Rightarrow (\neg ack_i \Rightarrow \mathrm{X}\, req_i)\right) \Rightarrow \left(req_i \Rightarrow \mathrm{F}_{[0,2n-1]}\, ack_i\right)\right)$$

Der Operator G gilt entsprechend dem erweiterten Operator F hier nur für das gegebene Zeitintervall.

Große und Drechsler berichten in [206] die Anzahl erreichbarer Zustände des skalierbaren Arbitrierers. Diese sind in Tabelle 6.6 dargestellt.

Tabelle 6.6. Anzahl erreichbarer Zustände des skalierbarer Arbitrierers [206]

Zellen	Zustände	Zellen	Zustände	Zellen	Zustände	Zellen	Zustände
2	8	6	384	10	10.240	50	$5,63 \cdot 10^{16}$
3	24	7	896	11	22.528	100	$1,27 \cdot 10^{32}$
4	64	8	2.048	12	49.152	150	$2,14 \cdot 10^{47}$
5	160	9	4.608	20	$2,10 \cdot 10^{7}$	200	$3,21 \cdot 10^{62}$

Schaltungen auf der Logikebene erlauben eine direkte Übersetzung in endliche Automaten, die sich symbolisch repräsentieren lassen. Somit lässt sich die in Abschnitt 5.3.2 beschriebene SAT-basierte Modellprüfung auch direkt auf Schaltungen auf der Logikebene anwenden. Im Folgenden werden daher lediglich einige Besonderheiten bei der formalen Modellprüfung von Hardware betrachtet.

SAT-basierte Modellprüfung bei mehreren Taktdomänen

Typische eingebettete Computersysteme, sogar wenn auf einem einzelnen Chip integriert, verfügen über mehrere sog. *Taktdomänen*. Der Grund liegt hierfür in der gleichzeitigen Optimierung des Durchsatzes und der Leistungsaufnahme eines Chips. Durch die Verwendung vieler unterschiedlicher Taktsignale können einzelne Bereiche auf dem Chip besonders schnell arbeiten oder besonders verlustleistungseffizient. Dies kann auch bedeuten, dass das Taktsignal für manche Bereiche temporär sogar ganz abgeschaltet wird (engl. *clock gating*), um die Schaltaktivität zu minimieren. Die Verwendung mehrerer Taktdomänen resultiert allerdings in einen gesteigerten Aufwand in der Verifikation.

Formale Modellprüfungsverfahren sind für synchrone Systeme mit einem einzigen Taktsignal entwickelt worden. Um die Modellprüfungsverfahren dennoch für Systeme mit mehren Taktdomänen verwenden zu können, müssen die Systembeschreibungen in ein äquivalentes System mit einer einzelnen Taktdomäne transformiert werden. Handelt es sich bei dem System um ein synchrones System mit mehreren Taktdomänen mit verschiedenen, aber bekannten Frequenzen und bekannten Phasen, kann ein globales Taktsignal mit der Frequenz des kleinsten gemeinsamen Vielfachen aller Frequenzen verwendet werden.

Beispiel 6.4.6. In einer Schaltung werden zwei Taktsignale clk_1 und clk_2 mit Frequenzen $2MHz$ bzw. $3MHz$ ohne Phasenverschiebung eingesetzt. Dann wird ein globales Taktsignal clk_{global} mit Frequenz $6MHz$ verwendet und mit dieser Taktrate das iterative Schaltungsmodell konstruiert. Allerdings ist das zweite, sechste, achte, zwölfte etc. Abrollen (negative Taktflanken mit betrachtet) unnötig, da weder Taktsignal clk_1 noch Taktsignal clk_2 zu diesen Zeitschritten schalten.

Ein analoges Problem entsteht bei der Übersetzung der funktionalen Eigenschaften. Das folgende Beispiel stammt aus [174]:

Beispiel 6.4.7. Die Property Specification Language (PSL) erlaubt die Verwendung mehrerer Taktsignale in der Spezifikation von Zusicherungen. Eine getaktete LTL-Formel in PSL, die von zwei Taktsignalen clk_1 und clk_2 abhängt, ist wie folgt gegeben:

$$\text{eventually! } p \;\&\&\; (\text{next } q@(\text{posedge } clk_1))@(\text{posedge } clk_2);$$

wobei p und q atomare Aussagen darstellen. Bei der Umwandlung in eine ungetaktete LTL-Formel können LTL-Formeln, die von mehreren Taktsignalen abhängen, stark anwachsen. Beispielsweise sieht obige Formel als ungetaktete LTL-Formel wie folgt aus:

$$\begin{aligned}&\text{eventually! } p \;\&\&\; (!clk_2 \text{ until!}(clk_2 \;\&\&\; \text{next}\\&\quad(!clk_2 \text{ until! } (clk_2 \;\&\&\; (!clk_1 \text{ until! } (clk_1 \;\&\&\; q))))));\end{aligned}$$

Im Folgenden werden synchrone Systeme mit mehreren Taktdomänen betrachtet. Ein Ansatz für Systeme mit asynchronen Taktdomänen ist in [105] beschrieben.

Beispiel 6.4.8. Betrachtet wird das synchrone System mit mehreren Taktdomänen aus Abb. 6.72a). FF_1 ist die Menge der Flip-Flops, die positiv taktflankengesteuert sind. FF_2 ist die Menge der Flip-Flops, die negativ taktflankengesteuert sind. Man beachte, dass die Flip-Flops in FF_1 an das Taktsignal clk_1 und die Flip-Flops in FF_2 am Taktsignal clk_2 angeschlossen sind. FF_3 ist ein pegelgesteuertes Flip-Flop. Die Blöcke $f_1, \ldots, f_4$ bezeichnen kombinatorische Logik. Signale i und o sind die primären Ein- bzw. Ausgänge.

Das äquivalente synchrone System mit einem globalen Taktsignal clk_{global} ist in Abb. 6.72b) dargestellt. Dieses System ist derart entworfen, dass alle Änderungen eines Eingangssignals, von internen Signalen oder Speicherelementen nur zusammen mit Änderungen des Taktsignals clk_{global} erfolgen. Um dies zu erreichen, wird vor jedes Flip-Flop ein Multiplexer geschaltet, der von einer Generatorschaltung g_i für $i = 1,2,3$ angesteuert wird. Der Ausgang der Generatorschaltung g_1 nimmt den Wert T an, wenn eine positive Taktflanke von clk_1 detektiert wird. Der Ausgang der Generatorschaltung g_2 nimmt entsprechend den Wert T an, wenn eine negative Taktflanke von clk_2 detektiert wird. Schließlich nimmt der Ausgang von g_3 den Wert T an, wenn die Funktion f_4 den Wert T besitzt. Die Flip-Flops werden über das globale Taktsignal clk_{global} gesteuert und sind sowohl positiv als auch negativ taktflankengesteuert.

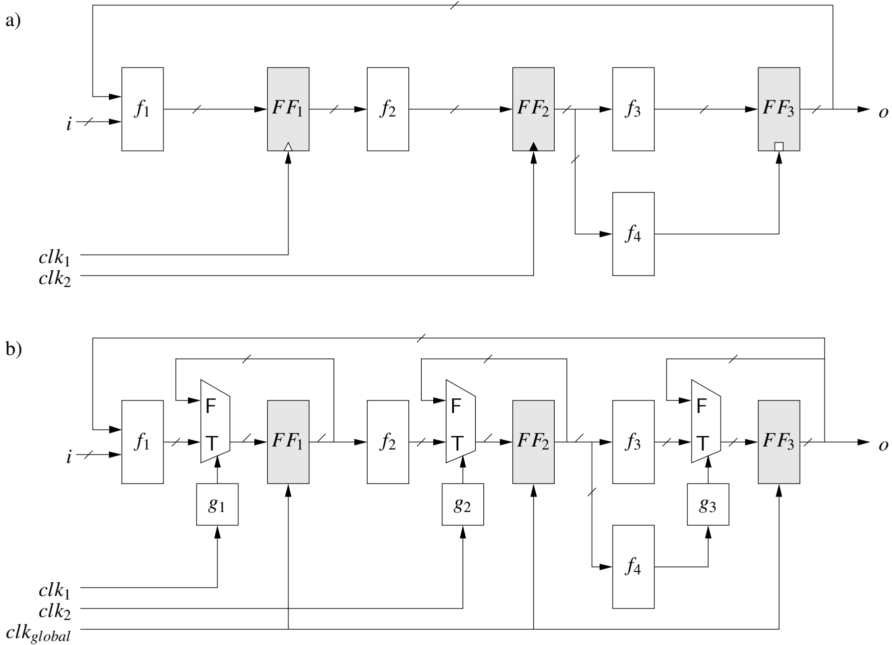


Abb. 6.72. a) System mit mehreren Taktdomänen und b) äquivalentes System mit einem Taktsignal [174]

Die Schaltung aus Abb. 6.72b) kann nun über die Zeitschritte abgerollt werden. Hierzu muss allerdings zunächst das Signal clk_{global} bestimmt werden. Im einfachsten Fall berechnet man das kleinste gemeinsame Vielfache der Frequenzen der Taktsignale clk_1 und clk_2. Dies führt allerdings, wie oben erläutert, zu unnötigen Überprüfungen von Formeln, da keine Schaltaktivität vorliegt.

In einem alternativen Ansatz wird zunächst der sog. *globale Taktzustand* bestimmt. Bei dem globalen Taktzustand handelt es sich hierbei um ein 3-Tupel $S[i] = (\tau[i], c_1[i], c_2[i])$. Dabei bezeichnet i die Position des Ereignisses in der Reihenfolge aller Taktzustände. $\tau[i]$ ist der Zeitpunkt, zu dem das Ereignis eingetreten ist. Die verbleibenden Einträge sind die Werte der ursprünglichen Taktsignale zu dem Zeitpunkt $\tau[i]$. Die globalen Taktzustände werden entsprechend ihrer Reihenfolge in einer Ereignisqueue gespeichert. Mit dieser Ereignisqueue kann abgeleitet werden, zu welchen Zeitschritten ein Abrollen der Schaltung sinnvoll ist.

Beispiel 6.4.9. Betrachtet wird wiederum das synchrone System mit mehreren Taktdomänen aus Beispiel 6.4.8. Die globalen Taktzustände sind durch die folgenden Einträge in der Ereignisqueue gegeben [174]:

$$
\begin{aligned}
S[0] &= (0ns, \mathsf{T}, \mathsf{T}) \\
S[1] &= (4ns, \mathsf{T}, \mathsf{F}) \\
S[2] &= (5ns, \mathsf{F}, \mathsf{F}) \\
S[3] &= (10ns, \mathsf{T}, \mathsf{F}) \\
S[4] &= (12ns, \mathsf{T}, \mathsf{T}) \\
S[5] &= (15ns, \mathsf{F}, \mathsf{T}) \\
S[6] &= (20ns, \mathsf{T}, \mathsf{F}) \\
S[7] &= (25ns, \mathsf{F}, \mathsf{F}) \\
S[8] &= (28ns, \mathsf{F}, \mathsf{T}) \\
S[9] &= (30ns, \mathsf{T}, \mathsf{T})
\end{aligned}
$$

Nur zu den Zeitpunkten $\tau[i] \in \{0ns, 4ns, 5ns, 10ns, 12ns, 15ns, 20ns, 25ns, 28ns, 30ns\}$ ist ein Aufstellen des iterativen Schaltungsmodells und eine Übersetzung in die korrespondierende aussagenlogische Formel sinnvoll. Die Zeitpunkte sind in Abb. 6.73 dargestellt. Dabei werden die ursprünglichen Taktsignale clk_1 und clk_2 derart beschränkt, dass deren Werte mit dem globalen Taktzustand übereinstimmen. Dabei ist in Abb. 6.73 auch zu erkennen, dass die pegelgesteuerten Latches keiner gesonderten Behandlung bedürfen, da sich deren Steuersignal f_4 synchron zu dem vorgeschalteten Flip-Flop ändert.

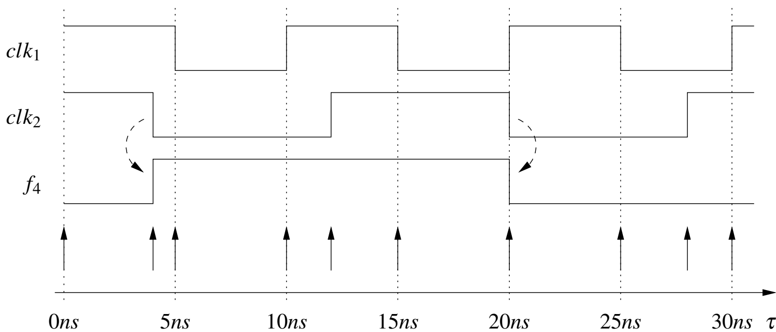


Abb. 6.73. Änderungen im globalen Taktzustand [174]

Obwohl die Anzahl der betrachteten Ereignisse gegenüber dem Ansatz der Bestimmung des kleinsten gemeinsamen Vielfachen der Taktfrequenzen reduziert wurde, können einige der Ereignisse in der Ereignisqueue irrelevant sein. Berücksichtigt man, dass manche Register nur über positive oder negative Taktflanken gesteuert werden, können weitere Zeitschritte bei der Modellprüfung ausgeschlossen werden.

Berücksichtigung von Speicherblöcken

Eingebettete Computersysteme verfügen oft über einen eingebetteten Speicher. Dieser eingebettete Speicher fügt dem Modellprüfungsproblem zusätzliche Komplexität hinzu, da jedes Speicherbit exponentiell zur Zustandsraumvergrößerung beiträgt. Somit wird durch eine explizite Modellierung eingebetteter Speicherblöcke der Zustandsraum oft zu groß, um das Modell zu analysieren. Aus diesem Grund ist eine geeignete Abstraktion von Speicherblöcken notwendig. Allerdings ist es für die formale Modellprüfung hinreichend, die Semantik eines Speichers zu betrachten, ohne jedes Speicherbit explizit zu modellieren [75] (siehe auch Abschnitt 6.3.1).

Betrachtet werden im Folgenden Speicher mit einem einzelnen Lese-/Schreibport, wie in Abb. 6.74 dargestellt. Die Speicherschnittstelle besteht aus einem Adressbus $addr$, einem Datenbus für Schreibzugriffe $data_w$, einem Datenbus für Leseoperationen $data_r$, sowie den beiden Steuersignalen $write_e$ und $read_e$ zur Signalisierung von Schreib- ($write_e = \top$) bzw. Lesezugriffen ($data_e = \top$). Ein Lesezugriff benötigt einen Taktzyklus, d. h. sobald die Adresse $addr$ und das Signal $read_e = \top$ anliegen, legt das Speichermodul die angeforderten Daten auf den Datenbus $data_r$ für Leseoperationen. Ein Schreibzugriff hingegen benötigt zwei Taktzyklen: Im ersten Takt werden die Schreibadresse $addr$ und die zu schreibenden Daten $data_w$ sowie das Signal $write_e = \top$ angelegt. Im zweiten Takt sind die Daten an die adressierte Stelle im Speicher geschrieben, d. h. die Daten sind erst im zweiten Takt aus dem Speicher verfügbar.

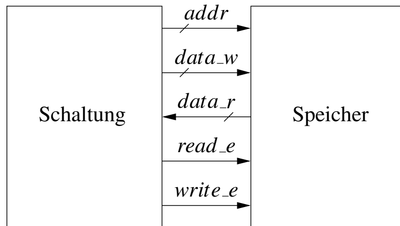


Abb. 6.74. Schaltung mit eingebettetem Speicher [174]

Die Semantik eines Speichers besagt, dass gelesene Daten dem zuletzt an die selbe Adresse geschriebenen Wert entsprechen. Es sei nun angenommen, dass die Schaltung $k+1$ Mal abgerollt wird. Dies ist in Abb. 6.75 dargestellt. Man sieht, dass das Lesen einer Speicherstelle in einem beliebigen Zeitschritt stets nur von dem zuletzt davor liegenden Schreibzugriff auf die selbe Adresse abhängt. Deshalb kann für die SAT-basierte Modellprüfung das explizite Speichermodell durch ein implizites Speichermodell ersetzt werden. Dabei bleibt die Speicherschnittstelle erhalten und das implizite Speichermodell erfüllt die Speichersemantik.

Mathematisch lässt sich dies mittels der binären Variable $c_{i,j} \in \mathbb{B}$ beschreiben, die wie folgt definiert ist:

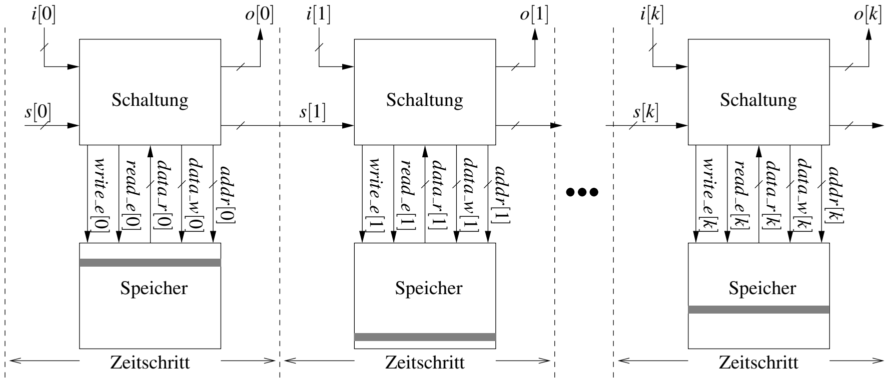


Abb. 6.75. Iteratives Schaltungsmodell mit Speicher [174]

$$c_{i,j} := \begin{cases} \mathsf{T} \text{ falls } addr[i] = addr[j] \\ \mathsf{F} \text{ sonst} \end{cases}$$

Mit $c_{i,j}$ sieht das Speichermodell wie folgt aus:

$$data_r[k] = \begin{cases} data_w[i] & \text{falls } \exists 0 \leq i < k : c_{i,k} \wedge write_e[i] \wedge read_e[k] \wedge \\ & (\forall i < j < k : \neg(c_{j,k} \wedge write_e[j])) \\ d_{init}(addr[i]) \text{ sonst} & \end{cases} \tag{6.20}$$

Mit anderen Worten: Die Daten, die im Zeitschritt k gelesen werden, entsprechen denjenigen Daten, die im Zeitschritt i geschrieben wurden, wenn:

1. die beiden Adressen $addr[i]$ und $addr[k]$ identisch sind,
2. in Zeitschritt i ein Schreib- und in Zeitschritt k ein Lesezugriff erfolgte und
3. in den Zeitschritten zwischen i und k keine weiteren Daten an die selbe Adresse geschrieben wurden.

Wurden allerdings noch keine Daten an die Adresse geschrieben, liefert der Lesezugriff den initialisierten Wert an der Speicherposition zurück. Dabei ist $d_{init}(addr[i])$ der Initialwert der Speicherstelle an Adresse $addr[i]$.

Bei der Übersetzung des iterativen Schaltungsmodells in die aussagenlogische Formel kann die Speichersemantik aus Gleichung (6.20) mittels des ITE-Operators (engl. *if-then-else*, siehe Anhang B.2) beschrieben werden. Hierbei sei die Variable $m_{i,j}$ als $m_{i,j} := c_{i,j} \wedge write_e[i]$ definiert:

$$\begin{aligned} data_r[k] = \text{ITE}(&m_{k-1,k}, data_w[k-1], \\ &\text{ITE}(m_{k-2,k}, data_w[k-2], \ldots, \\ &\text{ITE}(m_{0,k}, data_w[0], d_{init}(addr[i]))\ldots)) \end{aligned} \tag{6.21}$$

Obige Gleichung (6.21) entspricht einer Reihenschaltung von Multiplexern, wie in Abb. 6.76 dargestellt.

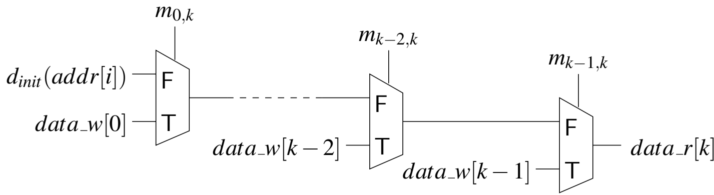


Abb. 6.76. Multiplexer-Reihenschaltung zur Darstellung der Speichersemantik [174]

Bevor ein SAT-Solver das passende Schreib-/Lesepaar $data_r[k] = data_w[i]$ identifiziert hat, muss dieser viele Entscheidungen über Variablenbelegungen treffen, um zu zeigen, dass $m_{i,k} = \mathsf{T}$ und $m_{i+1,k} = m_{i+2,k} = \cdots = m_{k-1,k} = \mathsf{F}$ gilt. Die Prüfung auf Erfüllbarkeit kann deutlich beschleunigt werden, wenn explizite Beschränkungen Schreib-/Lesepaare zwischen dem eigentlichen Schreiben und Lesen invalidieren. Hierfür wird das exklusiv gültige Lesesignal $e_{i,k}$ sowie die internen Signale $im_{i,k}$ definiert:

$$\begin{aligned} im_{k,k} &= read_e[k] \\ \forall 0 \leq i < k : im_{i,k} &= \neg m_{i,k} \wedge im_{i+1,k} \\ e_{-1,k} &= d_{init}(addr[k]) \\ \forall 0 \leq i < k : e_{i,k} &= m_{i,k} \wedge im_{i+1,k} \end{aligned}$$

Man beachte, dass $e_{i,k} = \mathsf{T}$ impliziert, dass $e_{j,k} = \mathsf{F}$ für $i, j < k$ und $i \neq j$.

Damit kann Gleichung (6.20) wie folgt umgeschrieben werden:

$$data_r[k] = (e_{k-1,k} \wedge data_w[k]) \vee \cdots \vee (e_{0,k} \wedge data_w[0]) \vee (e_{-1,k} \wedge d_{init}(addr[k])) \tag{6.22}$$

Initialisierung von Speicher mit unterschiedlichen Werten

Meistens werden Speicherblöcke durchgängig mit dem selben Wert initialisiert. Allerdings kann es passieren, dass die Modellprüfung nicht aus dem Anfangszustand des Systems startet. In diesem Fall müssen die Speicherzellen mit unterschiedlichen Werten belegt werden. Dies kann einfach durchgeführt werden, indem eine notwendige Anzahl an Schreiboperationen in der Vergangenheit als Klauseln zur aussagenlogischen Formel hinzugefügt werden: Angenommen p Speicherstellen $addr_0, \ldots, addr_{p-1}$ besitzen unterschiedliche Anfangswerte. Die Initialisierung der Speicherstellen kann durch p Schreiboperationen in der Vergangenheit, beginnend zum Zeitschritt $t = -p$ modelliert werden. Dann wird zu allen Zeitschritten $t = -p + i$ mit $1 \leq i \leq p - 1$ der Anfangswert an Adresse $addr_i$ geschrieben.

Beispiel 6.4.10. Es sollen zwei Speicher ($p = 2$) initialisiert werden. Adresse $addr_1$ wird mit dem Wert 3, Adresse $addr_2$ mit dem Wert 4 initialisiert. Der restliche Speicher ist mit F initialisiert.

- Zum Zeitschritt $t = -2$ besitzen beide Speicherstellen den Wert F. Zum Initialisieren der Adresse $addr_1$ müssen die Signale $addr[-2] = addr_1$, $data_w[-2] = 3$ und $write_e[-2] = \mathsf{T}$ gesetzt werden.
- Zum Zeitschritt $t = -1$ besitzt die Speicherstelle mit Adresse $addr_1$ den Wert 3 und alle anderen Speicherzellen besitzen den Wert F. Zur Initialisierung der Adresse $addr_2$ müssen die Signale $addr[-1] = addr_2$, $data_w[-1] = 4$ und $write_e[-1] = \mathsf{T}$ gesetzt werden.
- Zum Zeitschritt $t = 0$ besitzen alle Speicherzelle ihren gewünschten Anfangswert.

Effiziente Behandlung von Wortbreiten

Heutzutage werden viele Schaltungsbeschreibungen oft nicht mehr auf der Logikebene erstellt, sondern als Registertransferbeschreibungen auf der Architekturebene. Hardware-Beschreibungssprachen auf dieser Ebene bieten spezielle Datenstrukturen, etwa *Bitvektoren*. Die Verwendung dieser Datentypen erhöht die Lesbarkeit der Schaltungsbeschreibungen.

Auf der anderen Seite arbeiten Verfahren zur symbolischen Modellprüfung von Hardware mit einzelnen Bits als Datentypen. Eine Möglichkeit in der Behandlung von Bitvektoren in der symbolischen Modellprüfung besteht darin, die Information über Bitvektoren zu vernachlässigen und lediglich mit einzelnen Bits weiter zu prüfen, d. h. die Operatoren, die auf Bitvektoren angewendet werden, werden durch Boolesche Formeln ersetzt. Dies wird auch als engl. *bit flattening* bezeichnet. Als Beispiel sei hier die Addition zweier Bitvektoren genannt. Für die Verifikation mit einem Standard-SAT-Solver kann die Addition beispielsweise durch einen Ripple-Carry-Addierer ersetzt werden.

Bei einem solchen Schritt wird allerdings die Anzahl an Variablen in der SAT-Formel massiv erhöht, was in vielen Fällen dazu führt, dass das Modellprüfungsproblem nicht mehr gelöst werden kann. Ein besseres Vorgehen ist somit die Information über Bitvektoren zu erhalten und die Entscheidungsstrategie anzupassen. In diesem Abschnitt wird deshalb ein Ansatz betrachtet, der Formeln mit *Bitvektor-Arithmetik* durch Abstraktion entscheidet [60].

Zunächst wird die Syntax der Bitvektor-Arithmetik beschrieben.

Definition 6.4.1 (Syntax der Bitvektor-Arithmetik). *Die Syntax der* Bitvektor-Arithmetik *kann wie folgt definiert werden:*

$$\begin{aligned}
formula &::= formula \vee formula \mid formula \wedge formula \mid \neg formula \mid atom \\
atom &::= term\ rel\ term \mid id_{\mathbb{B}} \\
rel &::= = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
term &::= term\ op\ term \mid id \mid \sim term \mid const \mid term[i:j] \mid formula\ ?\ term : term \\
op &::= + \mid - \mid * \mid \div \mid \% \mid << \mid >> \mid @ \mid \& \mid | \mid \hat{}
\end{aligned}$$

Formeln $formula$ bezeichnen Ausdrücke mit Booleschem Rückgabewert. Terme *term* bezeichnen Bitvektoren und besitzen einen Datentyp. Der Datentyp gibt Aus-

kunft über die Bitbreite und die verwendete Codierung (z. B. Vorzeichen-Betrag-Darstellung, 2er Komplement). Mittels Relationen *rel* werden Bitvektoren in Boolesche Ausdrücke umgewandelt. Außerdem können Boolesche Ausdrücke von Bezeichnern für binäre Variablen $id_{\mathbb{B}}$ gebildet werden. Terme lassen sich ebenfalls aus Bezeichnern für Bitvektor-Variablen oder -Konstanten bilden. Der Ausdruck $term[i:j]$ ist ein Bereichsoperator und extrahiert die Bits i bis j aus einem Bitvektor. Das Ergebnis ist wiederum ein Bitvektor. Der Ausdruck $formula\,?\,term : term$ ist ein *if-then-else*-Konstrukt, bei dem, abhängig vom Wahrheitswert der Formel $formula$, ein Term $term$ ausgewählt wird. Evaluiert der Ausdruck $formula$ zu T wird der Term vor dem Doppelpunkt ausgewählt. Terme können schließlich über arithmetische, logische Operatoren verknüpft werden, wobei $\div$ die Ganzzahl- und % die Modulo-Division bezeichnet. Die Operatoren &, | und ^ stellen das bitweise logische Und, Oder bzw. Exklusiv-Oder dar. Die Operatoren $<<$ und $>>$ bezeichnen das Links- und Rechtsverschieben eines Bitvektors. Schließlich bezeichnet @ die Konkatenation von Bitvektoren.

Die Semantik der Bitvektor-Arithmetik lässt sich intuitiv definieren (siehe hierzu [59]) und sei hier nicht weiter angegeben. Allerdings muss man beachten, dass die arithmetischen Operatoren nur auf endlichen Wortbreiten definiert sind und die Semantik der Relationen von der Codierung (Vorzeichen-Betrag-Darstellung, 1er oder 2er Komplement) der Bitvektoren abhängt.

Beispiel 6.4.11. Gegeben ist die folgende Formel:

$$(x - y > 0) \Leftrightarrow (x > y)$$

Die Formel ist korrekt, wenn x und y ganze Zahlen in $\mathbb{Z}$ sind. Allerdings ist die Formel nicht mehr richtig, sobald Bitvektoren für x und y verwendet werden und die Subtraktion zu Über- oder Unterläufen führen kann.

Im Folgenden wird nun beschrieben, wie eine Bitvektor-Formel ϕ mit einem Standard-SAT-Solver mittels Abstraktion auf Erfüllbarkeit überprüft werden kann. Die grundlegende Idee ist, dass iterativ Unter- und Überapproximationen von ϕ gelöst werden. Das Vorgehen ist in Abb. 6.77 schematisch dargestellt. Eingabe für diese Entscheidungsstrategie ist die Formel ϕ in Bitvektor-Arithmetik mit den Variablen $x_1, x_2, \ldots, x_n$. Jede Variable x_i ist ein Bitvektor mit Bitbreite w_i.

Die Entscheidungsprozedur basierend auf Abstraktion erfolgt im Wesentlichen in vier Schritten:

1. *Initialisierung:* Für jede Variable x_i wird eine zu betrachtende Codierungsgröße s_i gewählt, wobei $0 \leq s_i \leq w_i$ sein muss.
2. *Generierung der Unterapproximation:* Eine Unterapproximation $\underline{\phi}$ der Formel ϕ wird bestimmt, indem jede Variable x_i auf einen Wertebereich der Kardinalität 2^{s_i} beschränkt wird. Dies kann auf verschiedene Arten erfolgen. Eine Möglichkeit besteht darin, eine Variable x_i lediglich auf den s_i niederwertigsten Bits mittels binärer Variablen zu codieren und die höherwertigen Bits mit F zu initialisieren. Eine andere Möglichkeit besteht darin, eine Vorzeichenexpansion durchzuführen, d. h. dass wiederum die s_i niederwertigsten Bits mittels binärer

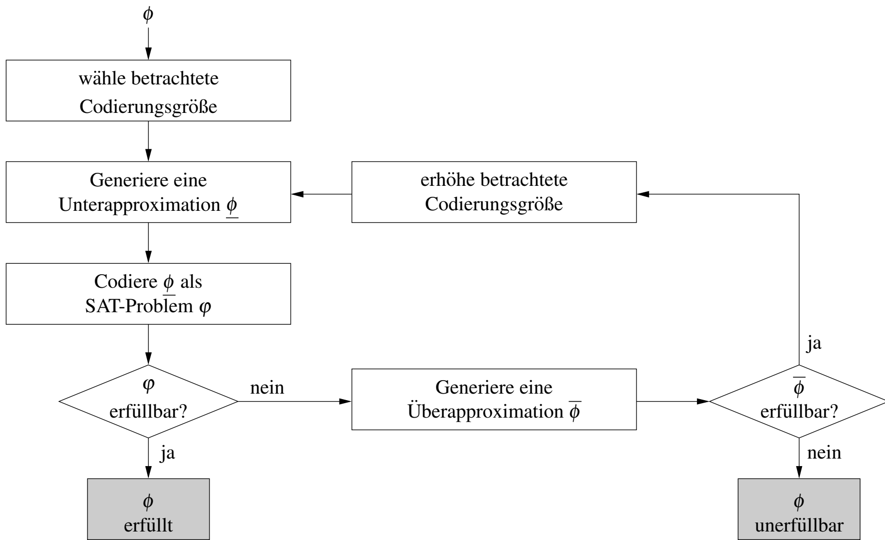


Abb. 6.77. Erfüllbarkeitsprüfung von Bitvektor-Formeln durch Abstraktion [60]

Variablen codiert werden. In diesem Fall werden aber die höherwertigen Bits mit dem selben Wert der binären Variablen an Position $(s_i - 1)$ codiert. Sei beispielsweise $s_i = 2$ und $w_i = 4$. Der Bitvektor, der x_i codiert, ist in diesem Fall $(x_{i1}, x_{i1}, x_{i1}, x_{i0})$, wobei x_{ij} binäre Variablen sind.

3. *Codierung der Unterapproximation als SAT-Formel:* In diesem Schritt wird eine Boolesche Formel φ aus $\underline{\phi}$ berechnet, indem die Bitvektor-Operationen durch Boolesche Funktionen implementiert werden. Die Bitbreiten, für welche die Operationen implementiert werden, sind dabei unverändert aus der Originalformel ϕ übernommen. Lediglich die Codierung der Bitvektoren wurde verändert. Für die Codierung einer Formel ϕ für Bitvektor-Arithmetik in konjunktiver Normalform, wie von den meisten SAT-Solvern benötigt, kann das in [60] beschriebene Verfahren angewendet werden. Hierfür liegt die Formel ϕ als Baum $G_\phi = (V_\phi, E_\phi)$ mit Quellknoten v_0 vor. Blätter $v \in \{v' \in V_\phi \mid \text{outdeg}(v') = 0\}$ repräsentieren die primären Eingänge der Schaltung (Bitvektoren). Die internen Knoten $v \in \{v' \in V_\phi \mid \text{outdeg}(v') > 0\}$ stellen Operatoren dar, wobei die Knoten durch eine konjunktive Normalform als Menge von Klauseln $c(v)$ repräsentiert sind. Die Argumente der Operation (ausgehende Kanten) werden mit dem Ergebnis der Operation (eingehende Kante) konjunktiv verknüpft. Die Konjunktion aller Klauseln stellt zusammen mit der Einerklausel x_0 die konjunktive Normalform φ der Formel ϕ dar, d. h.

$$\varphi = x_0 \wedge \bigwedge_{c \in \{c(v) \mid v \in C_\phi\}} c$$

Die Boolesche Formel φ wird anschließend mit einem Standard-SAT-Solver (siehe Abschnitt C.2) auf Erfüllbarkeit überprüft. Findet der SAT-Solver eine erfüllende Belegung der binären Variablen, so ist diese Belegung ebenfalls eine erfüllende Belegung der Formel ϕ und die Entscheidungsprozedur wird beendet.

4. *Generierung einer Überapproximation:* Ist die Boolesche Formel φ hingegen nicht erfüllbar, so wird eine Überapproximation $\overline{\phi}$ der Bitvektor-Formel ϕ bestimmt. Hierzu muss der SAT-Solver den sog. *unerfüllbaren Kern U* der Formel φ zurückliefern. Der unerfüllbare Kern einer Booleschen Formel φ ist eine nicht erfüllbare Teilmenge der Klauseln in φ und ist der Beweis, dass die Formel nicht erfüllbar ist. Viele SAT-Solver versuchen, möglichst kleine unerfüllbare Kerne zu erzeugen.
Die Überapproximation $\overline{\phi}$ ist wiederum eine Bitvektor-Formel, welche typischerweise aber viel kleiner als ϕ ist. Die Generierung der Überapproximation erfolgt auf Basis des Graphen G_ϕ, der die Formel ϕ repräsentiert. Der folgende Algorithmus erzeugt auf Basis von G_ϕ mit Knoten v und dem unerfüllbaren Kern U den Graphen $G_{\overline{\phi}}$.

```
COMPUTE_OVER_APPROX_GRAPH(G_φ, v, U) {
    IF (outdeg(v) = 0)
        RETURN;
    IF (vars(c(v)) ∩ vars(U) = ∅)
        Ersetze v in G_φ durch eine neue Variable;
        RETURN;
    FOREACH v_s ∈ {ṽ ∈ V | (v, ṽ) ∈ E_φ}
        COMPUTE_OVER_APPROX_GRAPH(G_φ, v_s, U);
}
```

$G_{\overline{\phi}}$ wird durch den Aufruf COMPUTE_OVER_APPROX_GRAPH(G_ϕ, v_0, U) erzeugt, wobei v_0 der Quellknoten von G_ϕ ist. Der Algorithmus ersetzt die assoziierte Variable eines Knoten v durch eine neue (noch nicht verwendete) Variable, genau dann, wenn in der Berechnung $c(v)$ der assoziierten Variablen keine Boolesche Variable ($\text{vars}(c(v))$) vorkommt, die zum unerfüllbaren Kern $\text{var}(U)$ gehört. Andernfalls werden die Nachfolgerknoten von v durchlaufen. Die Ersetzung durch eine neue Variable erfolgt, indem ein neuer Knoten v_n erzeugt wird und die Kante (v_p, v) auf den neuen Knoten umgelenkt wird, d. h. die Kante (v_p, v_n) eingefügt wird.
Eine wesentliche Eigenschaft von $\overline{\phi}$ ist, dass die Übersetzung in eine Boolesche Formel unter Verwendung der selben betrachteten Codierungsgrößen s_i wie bei der Übersetzung von ϕ, nicht erfüllbar ist. Dies liegt darin begründet, dass lediglich diejenigen Variablen, die unabhängig vom unerfüllbaren Kern sind, durch neue Variablen ersetzt wurden. Somit bleibt der unerfüllbare Kern erhalten. Andererseits ist leicht zu erkennen, dass $\overline{\phi}$ eine Überapproximation von ϕ darstellt, da jede erfüllende Variablenbelegung von ϕ ebenfalls $\overline{\phi}$ erfüllt. Dies kann durch Implikationen der neu generierten Variablen gezeigt werden, siehe [60]. Darüber hinaus können aber weitere erfüllende Belegungen existieren. Die Übersetzung der Überapproximation $\overline{\phi}$ in eine Boolesche Formel lässt sich aufgrund dieser

zusätzlichen Einsstellen mit weniger binären Variablen codieren, als die Übersetzung der Originalformel ϕ.
Die Erfüllbarkeit von $\overline{\phi}$ wird nun ohne Begrenzung der Wortbreiten der Bitvektoren überprüft, da durch eine wiederholte wie oben beschriebene Abstraktion wiederum der unerfüllbare Kern aktiviert werden würde. Falls $\overline{\phi}$ nicht erfüllbar ist, kann daraus geschlossen werden, dass ϕ nicht erfüllbar ist. Ist $\overline{\phi}$ hingegen erfüllbar, so bedeutet dies, dass mindestens einer binären Variablen x_{ij} ein Wert zugewiesen wurde, der durch die Codierung von lediglich s_i Bit ausgeschlossen wurde. in diesem Fall müssen somit die betrachteten Codierungsgrößen s_i erhöht werden.

Der Algorithmus benötigt maximal $n \cdot w_{max}$ Iterationen, wobei $w_{max} = \max_i\{w_i\}$ ist. Dies kann leicht gezeigt werden, da in jedem Schritt mindestens ein s_i um 1 erhöht wird.

6.5 Zeitanalyse

Neben der Überprüfung funktionaler Eigenschaften ist auch die Überprüfung nichtfunktionaler Eigenschaften bei Schaltungen von besonderer Bedeutung. Die Eigenschaftsprüfung von Zeitanforderungen besteht aus einer *Zeitanalyse* zur Ermittlung oder Abschätzung zeitlicher Eigenschaften sowie dem Vergleich mit der Spezifikation des Zeitverhaltens der Implementierung. Da die Zeitanalyse dabei die komplexere Aufgabe darstellt, werden im Folgenden Methoden zur Zeitanalyse von Schaltungen vorgestellt. Die mathematischen Grundlagen wurden bereits in Abschnitt 5.4 vorgestellt. Viele der dort vorgestellten Methoden lassen sich auch zur Zeitanalyse von Hardware einsetzen. Hier werden nun spezielle Aspekte zur Zeitanalyse für Hardware betrachtet. Im Folgenden wird zunächst die Zeitanalyse für synchrone Schaltungen diskutiert. Im Anschluss wird die Zeitanalyse für eine spezielle Klasse an Schaltungen mit mehreren Taktdomänen, den sog. *latenzinsensitiven Systemen*, vorgestellt.

6.5.1 Zeitanalyse synchroner Schaltungen

Bei der Äquivalenzprüfung und funktionaler Eigenschaftsprüfung von Schaltungen wurde in den vorangegangen Kapiteln davon ausgegangen, dass die Schaltungen synchron und basierend auf *einem* Taktsignal arbeiten. Die eigentliche Frequenz des Taktsignals war dabei nicht von Bedeutung. Zur Implementierung einer synchronen Schaltung muss diese Frequenz allerdings geeignet gewählt werden. Sie ist dabei abhängig von der gewählten Technologie und der Verschaltung der Logikgatter. Andererseits hat sie Einfluss auf die Ende-zu-Ende-Latenzen von implementierten Funktionen, da diese mehrere Taktzyklen benötigen können.

Die Ermittlung der maximalen Frequenz einer synchronen Schaltung wird typischerweise in einer *statischen Zeitanalyse* (engl. *static timing anaysis*) bestimmt. Beginnend mit einer Netzliste der synchronen Schaltung werden alle Pfade von Registerausgängen zu Registereingängen betrachtet. Auf diesen Pfaden wird das Ausgangssignal des Quellregisters durch verschaltete Logikgatter in die Eingangssignale

der Zielregister transformiert. Für jeden Pfad kann eine Verzögerungszeit bestimmt werden. Der Pfad mit der längsten Verzögerungszeit wird als *kritischer Pfad* bezeichnet und bestimmt die minimale Periode des Taktsignals und somit die maximale Frequenz $f_{\max}$, mit der die synchrone Schaltung betrieben werden kann.

Die Verzögerungszeit eines Pfades wird dabei von den verschiedenen Verzögerungszeiten der Komponenten entlang des Pfades bestimmt. Zunächst treibt das Quellregister nach Auftreten der Taktflanke das Ausgangssignal auf die Ausgangsleitung. Dies kann aufgrund physikalischer Randbedingungen nicht instantan erfolgen, sondern benötigt eine gewisse Zeit, die hier als *Transferzeit* δ_{trans} des Registers bezeichnet wird. Auf dem Weg zum Eingang des Zielregisters wird das Signal auf einem Pfad durch die dazwischenliegende kombinatorische Schaltung transformiert. Diese Verzögerungszeit wird als δ_{comb} bezeichnet. Sie setzt sich selbst wieder aus Verzögerungszeiten der einzelnen Logikgatter auf dem Pfad und den Verzögerungszeiten der Leitungen zwischen diesen Gattern zusammen. Sowohl die Verzögerungszeiten der Gatter als auch die der Leitungen sind technologieabhängig. Schließlich erreicht das transformierte Signal das Zielregister. Damit das Signal korrekt gespeichert werden kann, muss dieses für eine Zeit δ_{set} stabil am Eingang des Registers anliegen. Die Verzögerungszeit vom Ausgang des Quellregisters bis zum Eingang des Zielregisters ergibt sich als Summe der drei Zeiten, d. h. $\delta_{trans} + \delta_{comb} + \delta_{set}$. Für alle Pfade in der synchronen Schaltung muss gelten, dass die Verzögerungszeit des Pfades kleiner der minimalen Periode P_{min} ist, welche die maximal mögliche Taktfrequenz bestimmt:

$$\delta_{trans} + \delta_{comb} + \delta_{set} < P_{min} = \frac{1}{f_{\max}} \tag{6.23}$$

Das Problem der Bestimmung der maximalen Frequenz kann als ein *Längster-Pfad-Problem* formuliert werden. Hierzu wird die kombinatorische Schaltung als gewichteter Graph dargestellt, wobei die Knoten die Logikgatter und die Kanten die Verbindungen repräsentieren. Das Quellregister ist ein ausgezeichneter Quellknoten im Graphen. Die Kanten sind mit Verzögerungszeiten annotiert.

Definition 6.5.1 (Längster Pfad). *Gegeben sei ein zusammenhängender gerichteter Graph $G(V,E)$ und eine Gewichtsfunktion $\delta : E \rightarrow \mathbb{T}$, die jeder Kante $e \in E$ das Kantengewicht $\delta(e)$ zuweist. $\mathbb{T}$ beschreibt dabei die Zeitbasis. Im Folgenden gelte zunächst $\mathbb{T} := \mathbb{R}_{\geq 0}$. Im* Längster-Pfad-Problem mit einem Quellknoten *gibt es einen ausgezeichneten Quellknoten $v_0 \in V$ mit Eingangsgrad* $\text{indeg}(v_0) = 0$. *Gesucht ist der Pfad vom Quellknoten zu einem beliebigen anderen Knoten mit maximalem Pfadgewicht.*

Der längste Graph kann wie folgt berechnet werden:

```
LONGEST_PATH(G) {
    Λ(v0) := 0;
    REPEAT
        Bestimme einen Knoten v ∈ V : indeg(v) = 0;
        FOREACH e = (v, ṽ) ∈ E
```

```
            IF (Λ(ṽ) ≤ Λ(v) + δ(e))
                Λ(ṽ) := Λ(v) + δ(e);
            E := E \ {e ∈ E : e = (v, ṽ)};
        V := V \ {v};
    UNTIL (V = ∅);
    RETURN Λ;
}
```

Die Funktion $\Lambda(v)$ gibt die Länge des längsten Pfades von v_0 zu einem Knoten $v \in V$ an. Die Länge des Pfades von Knoten v_0 zu sich selbst wird mit null initialisiert. Anschließend wird in topologischer Ordnung jedem Knoten $\tilde{v}$ die maximale Summe aus Länge zu einem Vorgängerknoten und Kantengewicht für die Kante von diesem Vorgänger zu $\tilde{v}$ zugewiesen.

Beispiel 6.5.1. Gegeben ist das Schaltwerk in Abb. 6.78a). Die Logikgatter haben alle die selbe Verzögerungszeit von $2ns$. Die Flip-Flops haben beide jeweils eine Transferzeit von $\delta_{trans} = 5ns$ und eine Setup-Zeit von $\delta_{set} = 3ns$. Der zugehörige gewichtete Graph für die Berechnung der maximalen Frequenz ist in Abb. 6.78b) dargestellt. Man beachte, dass bei den Kanten zu z_0' und z_1' die Setup-Zeit der Flip-Flops zu den Kantengewichten hinzugezählt wurde.

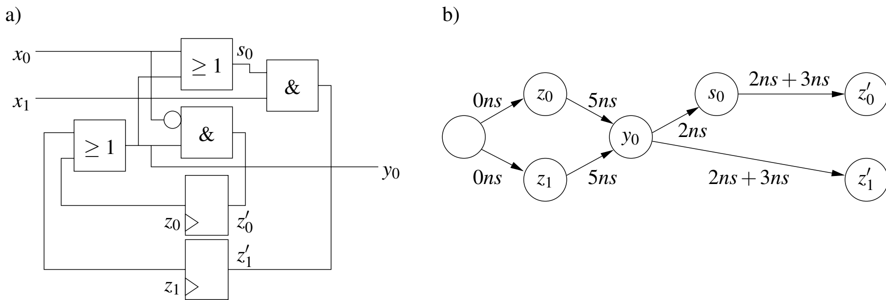


Abb. 6.78. a) Schaltwerk und b) Abhängigkeitsgraph

Die Berechnung des längsten Pfads führt dann zu dem Ergebnis:

$$\begin{aligned} \Lambda(z_0) &= 0ns & \Lambda(z_1) &= 0ns \\ \Lambda(y_0) &= 5ns & \Lambda(s_0) &= 7ns \\ \Lambda(z_0') &= 12ns & \Lambda(z_1') &= 10ns \end{aligned}$$

Es gibt also zwei kritische Pfade, (z_0, y_0, s_0, z_0') und (z_1, y_0, s_0, z_0'), mit maximaler Verzögerungszeit $12ns$. Die maximale Frequenz ergibt sich somit zu: $f_{\max} = \frac{1}{12ns} = 43,33MHz$.

Bei der oben beschriebenen statischen Zeitanalyse geht man davon aus, dass die Transferzeit eines Flip-Flops und Verzögerungszeiten eines Logikgatters konstant

sind. Dies stimmt in der Realität allerdings so nicht. Ändert sich beispielsweise der Ausgang eines Flip-Flops oder Gatters aufgrund der neuen Eingaben nicht, so gibt es auch keine Verzögerungszeit. Weiterhin macht es bei Flip-Flops und Logikgattern einen Unterschied, ob der Ausgang von F nach T oder von T nach F wechselt. All diese Effekte führen dazu, dass die statische Zeitanalyse zu einer Überapproximation bei der Berechnung der längsten Verzögerungszeit tendiert, da die dabei verwendeten Verzögerungszeiten für Gatter und Leitungen maximal sind. Genauere Ergebnisse erzielt man mit einer *dynamischen Zeitanalyse*, welche die oben beschriebenen Effekte berücksichtigt.

Eine dynamische Zeitanalyse basiert dabei auf der Simulation der Schaltung. Dadurch, dass die Schaltung sowohl funktional als auch zeitlich simuliert wird, können unterschiedliche Verzögerungszeiten in Abhängigkeit des Verhaltens bestimmt werden. Allerdings treten bei einer simulativen Zeitanalyse die selben Probleme auf, wie bei einer simulativen funktionalen Verifikation: Die Simulation ist keine vollständige Methode und kann damit nicht garantieren, dass die maximale Verzögerungszeit eines Pfades bereits stimuliert wurde. Somit können für harte Echtzeitanforderungen auch keine definitiven Zusagen getroffen werden. Andererseits liefert die statische Zeitanalyse eine obere Schranke für die maximale Verzögerungszeit eines Pfades. Es ist allerdings nicht klar, ob diese Verzögerung tatsächlich auftreten kann, d. h. ob es eine Anregung der Schaltung gibt, so dass diese Verzögerung auftritt.

Zeitanalyse auf Architekturebene

Die Zeitanalyse auf der Architekturebene für eine gegebene synchrone Schaltung mit einem Taktsignal lässt sich direkt auf dem im Steuerwerk implementierten *Ablaufplan* durchführen. Die Bestimmung von statischen Ablaufplänen in der Architektursynthese ist ausführlich in [426] diskutiert. Hier werden lediglich einige für die Zeitanalyse zentralen Ergebnisse wiederholt. Dabei erfolgt eine Einschränkung auf Implementierungen mit vollstatischer Bindung der Berechnungen auf Hardwarekomponenten.

Latenz aperiodischer Berechnung

Die durch die Schaltung auszuführende Berechnung sei im Folgenden als gerichteter azyklischer Graph, dem sog. *Problemgraph* $G(V,E)$, modelliert. Knoten $v \in V$ stellen elementare Berechnungen dar und Kanten $e \in E$ stellen Datenabhängigkeiten dar. Die Bindung der elementaren Berechnungen $v \in V$ an eine Hardware-Komponente $\tilde{v} \in V_R$ sei mit $\beta : V \to V_R$ bezeichnet. Hardware-Komponenten auf der Architekturebene haben die Granularität von Multiplizieren, Addierern, ALUs (engl. *Arithmetic Logic Units*) etc. Die Verzögerungszeit einer elementaren Berechnung $v \in V$ auf einer Hardware-Komponente $\tilde{v} \in V_R$ sei mit $\delta : V \times V_R \to \mathbb{T}$ bezeichnet. Da hier lediglich synchrone Schaltungen betrachtet werden, sind Verzögerungszeiten ganzzahlige Vielfache von Taktzyklen, d. h. $\mathbb{T} := \mathbb{Z}_{\geq 0}$.

Definition 6.5.2 (Ablaufplan). *Gegeben sei ein Problemgraph* $G(V,E)$. *Ein* statischer Ablaufplan *für G ist eine Funktion* $\tau : V \to \mathbb{T}$, *die jedem Knoten* $v_i \in V$ *die*

Startzeit $\tau(v_i)$ zuordnet, so dass gilt:

$$\forall (v_i, v_j) \in E : \tau(v_j) \geq \tau(v_i) + \delta(v_i, \beta(v_i))$$

Für einen gegebenen statischen Ablaufplan lässt sich die *Latenz* der Schaltung für den Problemgraph $G(V,E)$ bestimmen:

Definition 6.5.3 (Latenz). *Die* Latenz Λ *eines Ablaufplans τ (bei gegebener Bindung β) ist definiert als*

$$\Lambda = \max_{v_i \in V} \{\tau(v_i) + \delta(v_i, \beta(v_i))\} - \min_{v_i \in V} \{\tau(v_i)\}$$

und bezeichnet damit die Anzahl der Zeitschritte des kleinsten Intervalls, das die Ausführungsintervalle aller Knoten $v_i \in V$ einschließt.

Latenz und Periode periodischer Berechnungen

Im Folgenden wird die Berechnung einer Schaltung als *iterativer Problemgraph* $G(V,E,d)$ beschrieben. Die Knoten $v \in V$ repräsentieren elementare Berechnungen. Die Kanten $e \in E$ beschreiben Datenabhängigkeiten, wobei diese über Iterationsgrenzen der Berechnungen hinweg verlaufen können. Die Kantengewichtsfunktion $d : E \rightarrow \mathbb{Z}_{\geq 0}$ ordnet jeder Kante $(v_i, v_j) \in E$ die *Indexverschiebung* $d(v_i, v_j)$ (kurz $d_{i,j}$) zu. Ein iterativer Problemgraph kann Zyklen besitzen.

Die elementaren Berechnungen $v \in V$ sind an Hardware-Komponenten $\tilde{v} \in V_R$ gebunden, beschrieben durch die Funktion $\beta(v) = \tilde{v}$. Die Verzögerungszeit einer elementaren Berechnung v auf Hardware-Komponente $\tilde{v}$ ist gegeben durch $\delta(v, \tilde{v})$. Als *Periode P* (engl. *iteration period* oder *initiation interval*) bezeichnet man die Anzahl von Zeitschritten zwischen dem Beginn der Abarbeitung zweier aufeinander folgender Iterationen einer Operation v_i. Die Periode ist der Kehrwert des Durchsatzes einer Implementierung. Damit lässt sich ein periodischer Ablaufplan wie folgt definieren:

Definition 6.5.4 (Periodischer Ablaufplan). *Ein* periodischer Ablaufplan *(mit Periode P) eines iterativen Problemgraphen $G(V,E,d)$ ist eine Funktion $\tau : V \times \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$, die jedem Knoten $v_i \in V$ die Startzeitpunkte*

$$\forall n \in \mathbb{N} : \tau(v_i, n) := \tau(v_i, 0) + n \cdot P \tag{6.24}$$

zuordnet, so dass für alle Kanten $(v_i, v_j) \in E$ gilt:

$$\tau(v_j, 0) - \tau(v_i, 0) \geq \delta(v_i, \beta(v_i)) - d_{i,j} \cdot P$$

Der Zeitpunkt $\tau_n(v_i) := \tau(v_i, n)$ stellt den Startzeitpunkt der n-ten Iteration von Knoten v_i dar. Dabei gelte $\tau(v_i, n) := 0$ für alle $n < 0$.

Die Latenz eines periodischen Ablaufplans lässt sich dann wie folgt bestimmen:

$$\Lambda = \max_{v_i \in V} \{\tau_0(v_i) + \delta(v_i, \beta(v_i))\} - \min_{v_i \in V} \{\tau_0(v_i)\} \tag{6.25}$$

Beispiel 6.5.2. Bei dem iterativen Algorithmus

$$\begin{array}{ll} x_1[n] := f_1(x_2[n-1], x_3[n-1]) & \forall n \geq 0 \\ x_2[n] := f_2(x_1[n]) & \forall n \geq 0 \\ x_3[n] := f_3(x_2[n-1]]) & \forall n \geq 0 \end{array}$$

besteht bei der Berechnung der Variablen x_1 zu den Variablen x_2 und x_3 eine Indexverschiebung von eins, d. h. es werden die Werte aus dem vorherigen Iterationsschritt verwendet. Der iterative Problemgraph des Algorithmus ist in Abb. 6.79a) dargestellt. Die Verzögerungszeit aller elementaren Berechnungen ist 1. Abbildung 6.79b) zeigt einen periodischen Ablaufplan mit Periode $P = 3$ unter Verwendung von zwei Hardware-Komponenten. Die Latenz des Ablaufplans beträgt $\Lambda = 3$. Abbildung 6.79c) zeigt einen periodischen Ablaufplan mit Periode $P = 2$, ebenfalls unter Verwendung von zwei Hardware-Komponenten. Die Latenz ist auch weiterhin $\Lambda = 3$.

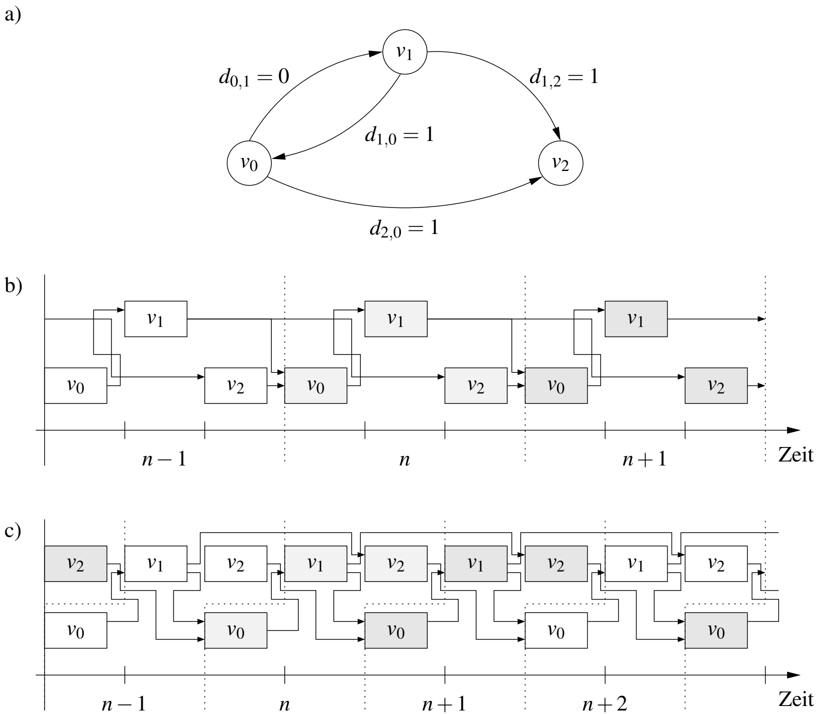


Abb. 6.79. a) iterativer Problemgraph, b) periodischer Ablaufplan mit $P = 3$ und c) $P = 2$

Man beachte, dass die Zeitanalyse auf statischen Ablaufplänen erfolgt. Diese werden während der Architektursynthese bestimmt und optimiert. Dabei können so-

gar Hardware-Komponenten mit Fließbandverarbeitung (engl. *pipelining*) berücksichtigt werden. Solche Komponenten können bereits eine weitere Berechnung starten, ohne das vorherige Berechnungen vollständig abgeschlossen sind.

6.5.2 Zeitanalyse latenzinsensitiver Systeme

Latenzinsensitive Systeme bestehen aus einer Menge von Hardware-Komponenten, die Daten über Kanäle austauschen. Das Protokoll zum Datenaustausch ist latenzinsensitiv. Eine Voraussetzung hierfür ist, dass eine Hardware-Komponente ihre Berechnung für unbestimmte Zeit anhalten (engl. *to stall*) kann, ohne ihren internen Zustand zu verlieren. Dies kann z. B. notwendig sein, wenn die Komponente zur Durchführung einer Berechnung Eingabedaten von mehreren Kanälen benötigt, diese Daten aber zu verschiedenen Zeitpunkten (später) zur Verfügung gestellt werden. Sobald eine Komponente anhält, produziert diese nur noch sog. *nichtinformative Daten* auf den Ausgangskanälen. Entsprechend werden *informative Daten* in den Eingangskanälen, die nicht konsumiert werden können, in den Kanälen zwischengespeichert. Daneben kann eine Hardware-Komponente aufgrund eines vollen Ausgangskanals die Berechnung anhalten. Wenn ein Ausgangskanal keine weiteren informativen Daten speichern kann, kommt es somit zum sog. engl. *back pressure*. Unabhängig von dem Grund, warum Komponenten zwischendurch Berechnungen anhalten müssen, wird durch das latenzinsensitive Protokoll sichergestellt, dass immer das korrekte Ergebnis berechnet wird.

Eine mögliche Implementierung eines latenzinsensitiven Protokolls ist in [309] beschrieben: Ein latenzinsensitives System besteht aus zwei Arten von Blöcken, den Hardware-Komponenten und sog. *Relais-Stationen*. Während die Hardware-Komponenten zur Berechnung der eigentlichen Funktionalität notwendig sind, werden Relais-Stationen zu einem späteren Zeitpunkt in die Schaltung eingefügt, um Inkonsistenzen im Zeitverhalten der Schaltung aufzulösen. Ein Beispiel eines latenzinsensitiven Systems ist in Abb. 6.80 dargestellt. Es besteht aus drei Hardware-Komponenten und einer Relais-Station (schattiert dargestellt).

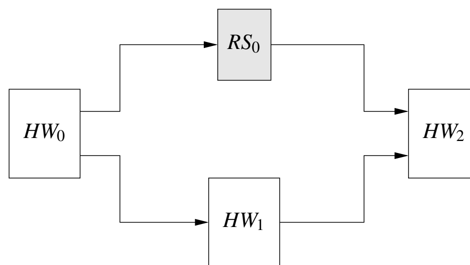


Abb. 6.80. Latenzinsensitives System [309]

Die Verbindung von Hardware-Komponenten und Relais-Stationen basiert auf Queues mit einer Kapazität von 1 oder größer. Dies ist notwendig, da es sonst zu einem Verlust informativer Daten kommen kann: Wenn eine lesende Hardware-Komponente zum Zeitpunkt τ anhält, erhält die schreibende Hardware-Komponente die entsprechende Signalisierung erst zum Zeitpunkt $\tau + 1$. Da das schreibende Modul bereits zum Zeitpunkt τ ein neues informatives Datum generiert haben kann, muss dieses irgendwo zwischengespeichert werden. Im Folgenden wird deshalb eine Queue der Größe 1 als *minimale Queue* bezeichnet.

Die Datenpufferung für eine Relais-Station erfolgt somit in der Eingangsqueue der Relais-Station. Als Folge hieraus produziert eine Relais-Station im ersten Takt ein nichtinformatives Datum. Alle Hardware-Komponenten hingegen produzieren im ersten Takt informative Daten. Ansonsten arbeiten Hardware-Komponenten und Relais-Stationen identisch, gesteuert durch den Zustand der angeschlossenen Kanäle. Dabei wird ein Kanal als *voll* bezeichnet, genau dann, wenn die Queue maximal viele Daten entsprechend ihrer Kapazität momentan gespeichert hat und das schreibende Modul informative Daten produzieren würde. Ein Kanal wird als *leer* bezeichnet, genau dann, wenn die Queue momentan keine Daten speichert und das angeschlossene schreibende Modul nichtinformative Daten produzieren würde. Mit diesen Definitionen gibt es genau zwei Fälle unter denen eine Modul anhält:

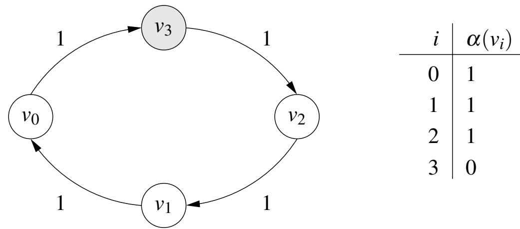
1. Mindestens ein Eingangskanal des Moduls ist leer, d. h. der Kanal kann keine informativen Daten zur Verfügung stellen.
2. Mindestens ein Ausgangskanal des Moduls ist voll, d. h. die Produktion weiterer informativer Daten würde einen Überlauf auf diesem Kanal zur Folge haben.

Zustandsanalyse

Für latenzinsensitive Systeme wird im Folgenden eine Analyse des maximalen Durchsatzes des Systems vorgestellt. Dafür wird angenommen, dass eventuell benötigte Eingabedaten schnell genug zur Verfügung gestellt werden. Für die Zeitanalyse latenzinsensitiver Systeme werden diese zunächst als sog. *LIS-Graph* modelliert:

Definition 6.5.5 (LIS-Graph). *Ein* LIS-Graph $G_{LIS} = (V, E, q, \alpha)$ *ist ein gewichteter, gerichteter Graph mit Knotenmenge V und Kantenmenge E. Knoten $v \in V$ repräsentieren Module, also Hardware-Komponenten und Relais-Stationen, während Kanten $E \subseteq V \times V$ Queues zwischen Modulen repräsentieren. Die Funktion $q : E \rightarrow \mathbb{N}$ weist jeder Queue eine Kapazität zu. Die Funktion $\alpha : V \rightarrow \{0, 1\}$ ist ein Knotengewichtsfunktion der Form, so dass $\alpha(v) = 1$ gilt, falls v eine Hardware-Komponente repräsentiert, und $\alpha(v) = 0$ gilt, falls v eine Relais-Station repräsentiert.*

Beispiel 6.5.3. Für das latenzinsensitive System aus Abb. 6.80 ist der LIS-Graph in Abb. 6.81 dargestellt. Die Knoten v_0, v_1 und v_2 repräsentieren die Hardware-Komponenten HW_0, HW_1 bzw. HW_2. Der Knoten v_3 repräsentiert die Relais-Station RS_0. Dabei wurde angenommen dass alle Queues minimale Queues (mit Kapazität $q(v_i, v_j) = 1$) sind. Dies ist an den Kanten annotiert.



i	$\alpha(v_i)$
0	1
1	1
2	1
3	0

Abb. 6.81. LIS-Graph zu dem latenzinsensitiven System aus Abb. 6.80 mit minimalen Queues [309]

Im Folgenden wird ein System zu diskreten Zeitpunkten betrachtet, d. h. $\mathbb{T} = \mathbb{Z}_{\geq 0}$. Das Verhalten eines einzelnen Moduls $v_i \in V$ lässt sich dann als Sequenz $\tilde{s}_i = \langle s_i(0), s_i(1), s_i(2), \ldots \rangle$ von Zuständen $s_i(\tau)$ des Moduls zu Zeitpunkten $\tau \in \mathbb{T}$ beschreiben. Dabei gilt:

$$s_i(\tau) := \begin{cases} \alpha(v_i) & \text{für } \tau = 1 \\ s_i(\tau - 1) & \text{für } \tau > 1 \text{ und } v_i \text{ hält zum Zeitpunkt } \tau \text{ an} \\ s_i(\tau - 1) + 1 & \text{sonst} \end{cases} \tag{6.26}$$

Mit anderen Worten: $s_i(\tau)$ entspricht der Anzahl produzierter informativer Daten bis zum Zeitpunkt τ. Entsprechend produziert eine Hardware-Komponente zum Zeitpunkt $\tau = 1$ ein Datum ($\alpha(v_i) = 1$), während eine Relais-Station kein Datum produziert ($\alpha(v_i) = 0$). Bei allen Modulen erhöht sich die Anzahl produzierter informativer Daten mit jedem Takt, solange das Modul nicht anhält.

Hiermit lassen sich die in einer Queue (v_i, v_j) gespeicherten Daten $f_{i,j}$ zum Zeitpunkt τ in Abhängigkeit von den Zuständen der Module v_i und v_j berechnen:

$$f_{i,j}(\tau) := \begin{cases} \emptyset & \text{falls } s_i(\tau) = s_j(\tau) - \alpha(v_j) \\ \{s_i(\tau), s_i(\tau) - 1, \ldots, s_j(t) - \alpha(v_j) + 1\} & \text{sonst} \end{cases}$$

Die Anzahl $|f_{i,j}(\tau)|$ der gespeicherten informativen Daten $f_{i,j}(\tau)$ in einer Queue (v_i, v_j) zum Zeitpunkt τ ist somit:

$$|f_{i,j}(\tau)| = s_i(\tau) - s_j(\tau) + \alpha(v_j) \leq q(v_i, v_j) + 1 \tag{6.27}$$

Diese muss kleiner der Kapazität der Queue plus eins sein, d. h. der Kanal kann keine weiteren informativen Daten speichern und das Modul v_i möchte informative Daten produzieren. Mit anderen Worten: Der Kanal ist voll, wenn die Kapazität der Queue um eins überschritten wurde.

Basierend auf obiger Zustandsdefinition und den Kanalbeschränkungen in Gleichung (6.27) können nun die Zustandsänderungen in einem latenzinsensitiven System beschrieben werden. Betrachtet wird das Modul v_i mit Eingangskanal (v_j, v_i). Falls Kanal (v_j, v_i) nicht genügend informative Daten bereit hält, hält Modul v_i an. In diesem Fall gilt $|f_{j,i}(\tau)| = s_j(\tau) - s_i(\tau) + \alpha(v_i) = 0$, d. h. $s_i(\tau + 1) = s_i(\tau) =$

$s_j(\tau)+\alpha(v_i)$. Falls der Kanal allerdings nicht leer ist, gilt $|f_{j,i}(\tau)| = s_j(\tau) - s_i(\tau) + \alpha(v_i) \geq 0$ und somit $s_i(\tau+1) = s_i(\tau)+1$ und $s_i(\tau+1) \leq s_j(\tau)+\alpha(v_i)$. Zusammengefasst ergibt dies: $s_i(\tau+1) \leq s_j(\tau)+\alpha(v_i)$. Da dies für alle Eingangskanäle gelten muss, bestimmt der langsamste Kanal, ob das Modul v_i anhält, d. h.

$$s_i(\tau+1) \leq \min_{(v_j,v_i)\in E} \{s_j(\tau)+\alpha(v_i)\} \tag{6.28}$$

Betrachtet wird das Modul v_i und der Ausgangskanal (v_i,v_j). Ist der Kanal (v_i,v_j) voll, so kann das Modul v_i keine weiteren informativen Daten produzieren und hält an, d. h.:

$$s_i(\tau+1) = s_i(\tau) = |f_{i,j}(\tau)| + s_j(\tau) - \alpha(v_j) = s_j(\tau) + q(v_i,v_j) + 1 - \alpha(v_j)$$

Falls der Kanal allerdings nicht voll ist, gilt: $s_i(\tau+1) = s_i(\tau)+1$. Dies kann höchstens dazu führen, dass anschließend der Kanal voll ist, d. h. $s_i(\tau)+1 \leq s_j(\tau) + q(v_i,v_j) + 1 - \alpha(v_j)$. Fasst man nun beide Fälle zusammen erhält man: $s_i(\tau+1) \leq s_j(\tau) + q(v_i,v_j) + 1 - \alpha(v_j)$, da hierbei alle Ausgangskanäle betrachtet werden müssen, ist der langsamste Kanal ausschlaggebend:

$$s_i(\tau) \leq \min_{(v_i,v_j)\in E} \{s_j(\tau) + q(v_i,v_j) + 1 - \alpha(v_j)\} \tag{6.29}$$

Unter Verwendung der Max-Plus-Algebra $(\mathbb{R}\cup\{\infty\}, \min, +)$ (siehe auch Seite 230) und des Vektors $s(\tau) = (s_0(\tau), \ldots, s_{|V|-1}(\tau))^\top$ lassen sich die Gleichungen (6.28) und (6.29) für alle $s_i(\tau)$ zusammenfassen:

$$s(\tau+1) \leq A_{LIS} \otimes s(\tau) \tag{6.30}$$

Dabei ist $\otimes$ die Multiplikation der Max-Plus-Algebra mit $a \otimes b := \min\{a,b\}$. Die Matrix A_{LIS} ist für einen gegebenen LIS-Graphen G_{LIS} wie folgt definiert:

$$a_{i,j} := \begin{cases} \alpha(v_i) & \text{falls } (v_j,v_i) \in E \\ q(v_i,v_j)+1-\alpha(v_j) & \text{falls } (v_i,v_j) \in E \wedge (v_j,v_i) \notin E \\ 1 & \text{falls } i=j \\ \infty & \text{sonst} \end{cases} \tag{6.31}$$

Beispiel 6.5.4. Betrachtet wird wiederum das latenzinsensitive System aus Abb. 6.80 auf Seite 351 mit dem zugehörigen LIS-Graphen G_{LIS} aus Abb. 6.81. Die Matrix A_{LIS} ergibt sich zu:

$$A_{LIS} = \begin{pmatrix} 1 & 1 & \infty & 2 \\ 1 & 1 & 1 & \infty \\ \infty & 1 & 1 & 1 \\ 0 & \infty & 1 & 1 \end{pmatrix}$$

Zeitanalyse

Die Bestimmung des Durchsatzes eines latenzinsensitiven Systems erfolgt über die Bestimmung des Durchsatzes eines einzelnen Moduls. Der Durchsatz eines Moduls

$v_i \in V$ ist definiert zu $\Theta(v_i) := \lim_{\tau\to\infty} \frac{s_i(\tau)}{\tau}$. Um zu dem Durchsatz des Systems zu gelangen, werden nun zwei verbundene Module v_i und v_j mit $(v_i, v_j) \in E$ betrachtet. Man beachte, dass das Module v_j lediglich Daten lesen kann, die bereits von v_i produziert worden sind, d. h. $s_i(\tau) \geq s_j(\tau) - \alpha(v_j)$. Mit Gleichung (6.27) ergibt dies:

$$s_j(\tau) - \alpha(v_j) \leq s_i(\tau) \leq s_j(\tau) + q(v_i, v_j) + 1 - \alpha(v_j)$$

Für den Durchsatz der beiden Module bedeutet dies:

$$\Theta(v_j) := \lim_{\tau\to\infty} \frac{s_j(\tau) - \alpha(v_j)}{\tau} \leq \Theta(v_i) \leq \lim_{\tau\to\infty} \frac{s_j(\tau) + q(v_i, v_j) + 1 - \alpha(v_j)}{\tau} = \Theta(v_j)$$

Mit anderen Worten: Der Durchsatz der beiden Module ist identisch, d. h. $\Theta(v_i) = \Theta(v_j)$. Somit gilt für alle Module eines latenzinsensitiven Systems, dass der Durchsatz aller Module gleich ist, sofern der LIS-Graph G_{LIS} zusammenhängend ist.

Die Berechnung des maximalen Durchsatzes, also desjenige Durchsatzes, der erzielt werden kann, wenn auf keine externen Eingaben gewartet werden muss, kann auf Basis des folgenden Theorems [309] berechnet werden:

Theorem 6.5.1. *Sei $G_{LIS} = (V, E, q, \alpha)$ ein LIS-Graph mit Matrix A_{LIS}. Der maximale Durchsatz ist $\Theta(G_{LIS}) = \lambda$, wobei λ der Eigenwert der Matrix A_{LIS} ist.*

Das obige Ergebnis beruht auf folgendem Theorem [112]:

Theorem 6.5.2. *Sei $X \in \mathbb{R}^{n\times n}$ die Adjazenzmatrix eines stark zusammenhängenden markierten Graphen G, dann gilt*

$$\exists q, p : \forall k \geq q : X^{\otimes^{k+p}} = \lambda^{\otimes^p} \otimes X^{\otimes^p}$$

Dies bedeutet, dass die Sequenz $\langle X^{\otimes^1}, X^{\otimes^2}, \dots \rangle$ zyklisch mit einer Periode p ist.

Beispiel 6.5.5. Betrachtet wird das latenzinsensitive Systeme aus Abb. 6.80 auf Seite 351. Die zugehörige Matrix A_{LIS} wurde bereits in Beispiel 6.5.4 bestimmt. Bestimmt man $A_{LIS}^{\otimes^k}$ für $k = 1, 2, \dots$, so sieht man, dass $A_{LIS}^{\otimes^6} = 3 \otimes A_{LIS}^{\otimes^2}$ ist. Mit Theorem 6.5.2 erhält man:

$$A_{LIS}^{\otimes^{k+4}} = A_{LIS}^{\otimes^6} \otimes A_{LIS}^{\otimes^{k-2}} = 3 \otimes A_{LIS}^{\otimes^2} \otimes A_{LIS}^{\otimes^{k-2}} = 3 \otimes A_{LIS}^{\otimes^k} \text{ für } k > 2$$

Damit ergibt sich $\lambda^{\otimes^4} = 3$ und somit $\lambda = \Theta(G_{LIS}) = \frac{3}{4}$ als maximaler Durchsatz.

Zur Interpretation des Ergebnisses bietet es sich an, die Matrix A_{LIS} als einen markierten Graphen darzustellen. Der zu dem in Abb. 6.80 auf Seite 351 dargestellten latenzinsensitiven System mit minimalen Queues äquivalente markierte Graph ist in Abb. 6.82 dargestellt. Dieser wurde direkt aus der Matrix A_{LIS} aus Beispiel 6.5.4 konstruiert. Die Verzögerungszeiten der Aktoren $a \in A$ sind mit $\delta(a) := 1$ angenommen. Es handelt sich um eine synchrone Schaltung. Darin sieht man, dass jede Verbindung im Originalsystem gepuffert ist und entsprechend im markierten Graph

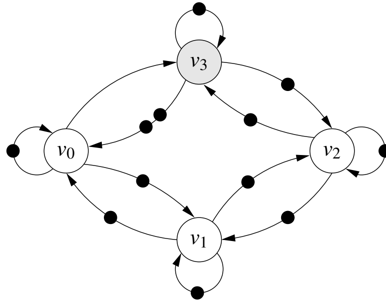


Abb. 6.82. Markierter Graph für das latenzinsensitive System in Abb. 6.80 mit minimalen Queues

anfangs mit einer Marke belegt ist. Die Relais-Station fügt allerdings keine zusätzliche Verzögerung ein, weshalb die entsprechende Kante (v_0, v_3) nicht markiert ist.

Weiterhin sieht man, dass jede Hardware-Komponente und jede Relais-Station erst eine Berechnung beenden muss, um die nächste Berechnung zu starten. Dies ist durch die Selbstschleifen mit je einer Anfangsmarkierung im markierten Graphen dargestellt. Schließlich sind die beschränkten Kanäle durch Rückkanten modelliert. Somit kann ein latenzinsensitives System auch als die Implementierung eines markierten Graphen mit beschränkten Kanalkapazitäten gesehen werden. Sobald das latenzinsensitive System als markierter Graph modelliert ist, lässt sich der Durchsatz auch über Theorem 5.4.1 auf Seite 231 ermitteln.

6.6 Literaturhinweise

Die Äquivalenzprüfung kombinatorischer und sequentieller Schaltungen ist ausführlich in dem Buch von Molitor und Mohnke [329] diskutiert. Entscheidungsdiagramme für implizite Äquivalenzprüfung auf der Logikebene sind in [135] beschrieben. Einen methodischen Vergleich von Entscheidungsdiagrammen findet man in [35]. Die implizite Äquivalenzprüfung zwischen Architektur- und Logikebene wird in [65, 66] vorgestellt. Eine hierzu alternative Methode der *Rückwärtstraversierung* wurde in [214] präsentiert. Einen Überblick über Entscheidungsdiagramme und deren Einsatz zur impliziten Äquivalenzprüfung auf Architekturebene findet sich in [224].

Verfahren zur expliziten Äquivalenzprüfung werden häufig auf der Logikebene eingesetzt. Hierbei kommen zwei verwandte Verfahren, die automatische *Testfallgenerierung* (engl. *Automatic Test Pattern Generation*, *ATPG*) oder SAT-Solver zum Einsatz. Der Hauptunterschied beider Verfahren liegt darin, dass SAT-Solver überwiegend auf aussagenlogischen Formeln in konjunktiver Normalform arbeiten, während ATPG-Ansätze Boolesche Netzwerke als Datenstruktur verwenden.

Weiterhin wurden SAT-Solver vor dem Hintergrund des automatischen Theorem-Beweisens entwickelt [129, 128], während ATPG-Ansätze im Kontext von Tests für die Chipfabrikation entstanden sind [1]. Ein Vergleich und eine Übersicht beider Verfahren findet sich in [50, 137].

Der Einsatz von *symbolischer Simulation* zur Verifikation auf Logikebene ist ausführlich in [45] beschrieben. Symbolische Simulation wurde erstmals im Jahr 1976 vorgestellt [260]. Obwohl zunächst zur Analyse von Software-Programmen gedacht, wurde das Potential für die Logiksimulation schnell erkannt. 1979 präsentierte IBM den ersten symbolischen Simulator für Hardware [86], mit dem Ziel Mikrocode für ihre Prozessoren zu verifizieren. Der erste symbolische Simulator mit Namen MOSSYM basierend auf einer eigenen Repräsentation für Boolesche Ausdrücke. MOSSYM wurde Mitte der 1980er Jahre von Bryant vorgestellt [61]. 1987 wurde eine Erweiterung von MOSSYM mit Namen COSMOS vorgestellt [64]. COSMOS ist der erste symbolische Simulator der auf BDD-Repräsentationen arbeitet. In [117] wurde schließlich ein Verfahren vorgestellt, wie die Booleschen Formeln, die während der symbolischen Simulation hergeleitet wurden, direkt verwendet werden können, um die Erreichbarkeitsmenge in sequentiellen Schaltungen zu bestimmen. Zur Speicherung der Erreichbarkeitsmenge werden typischerweise BDDs verwendet. Diese können allerdings sehr groß werden, weshalb Verfahren entwickelt wurden, um die Größe der BDDs zu beschränken [371, 81] und die Komplexität der Bildberechnung zu reduzieren [80, 331]. Ein Verfahren, welches direkt auf Bitvektoren arbeitet und somit ohne rechenintensive Konstruktion von BDDs auskommt, ist in [199] vorgestellt.

Die Ausnutzung struktureller Ähnlichkeiten während der Äquivalenzprüfung auf der Logikebene basiert auf dem dekompositionalen Verfahren von Berman und Trevillyan [43]. In diesem Verfahren werden interne Signale auf ihre Äquivalenz geprüft. In [58] ist ein Verfahren zur Reduktion der Miter-Schaltung durch Signalsubstitution vorgestellt. Bei diesem Verfahren geht durch die Signal-Substitution keine Information verloren. Dies gilt allerdings nicht bei der Schaltungspartitionierung, welche auf der Verwendung von Schnittpunkten als Eingänge basiert [273, 274]. Die Wahl der Schnittpunkte ist dabei nicht trivial [314, 145]. Verfahren zur strukturellen sequenziellen Äquivalenzprüfung sind in [411, 146, 412] beschrieben.

Eine Prozessorverifikation auf Basis der Theorie „Gleichheit und uninterpretierte Funktionen"(engl. *Equality and Uninterpreted Functions, EUF*) wurde erstmals in [75] vorgestellt. Die Erweiterung auf die Theorie „Positive Gleichheit und uninterpretierte Funktionen"(engl. *Positive EUF, PEUF*) wird in [67] ausgiebig diskutiert. Hierbei wird eine Unterscheidung von Gleichungen in *positive* und *generelle Gleichungen* vorgenommen. Positive Gleichungen dürfen nicht negiert auftreten, d. h. sie dürfen insbesondere nicht als Bedingung in ITE-Operationen auftreten. Positive Gleichungen lassen sich bei der Reduktion auf Aussagenlogik speziell behandeln und führen zu einfacheren Strukturen der aussagenlogischen Formeln, was in der Verifikation ausgenutzt werden kann.

Die Verifikation superskalarer Prozessoren ist in [71] beschrieben. Die grundlegende Idee ist die Dekomposition des kommutativen Diagramms in drei kommutative Diagramme, was es erleichtert, die symbolische Simulation der Implementierung

und Spezifikation aufeinander abzustimmen. Weiterführende Arbeiten zur Äquivalenzprüfung superskalarer Prozessoren mit [453] und ohne Sprungvorhersage [452] basieren auf PEUF. Die Verwendung eines effizienten Speichermodells in [68] erlaubt es, die Äquivalenz anhand von Speicherzugriffen der Mikroarchitektur und der ISA in der symbolischen Simulation zu prüfen. Die Erweiterung, diese Speichermodelle auch für die funktionalen Einheiten zu verwenden, ist in [451] vorgestellt.

In [403] ist die Erweiterung des Ansatzes aus [75] für Mikroarchitekturen mit dynamischer Instruktionsablaufplanung beschrieben. Ein weitergehender Ansatz, der in [40] beschrieben ist, basiert auf Modellprüfung. Dieser Ansatz verwendet Ergebnisse aus [319] zur Verifikation des Algorithmus von Tomasulo.

Funktionale Eigenschaftsprüfung für Hardware ist heutzutage überwiegend simulativ oder SAT-basiert. Die Synthese von Monitoren aus PSL-Zusicherungen für die Hardware-Verifikation ist in [333] für die schwachen PSL-Operatoren und in [335] für die starken PSL-Operatoren beschrieben. In letzterem Ansatz erfolgt auch die Strukturierung eines elementaren PSL-Monitors in einen Block zur Zeitfenstergenerierung und einen Block für die Evaluierung. Dabei zeigen die Autoren die Korrektheit des Ansatzes mittels eines Theorembeweisers. Einen alternativen Ansatz, basierend auf sog. *Sequenzautomaten*, haben Boulé und Zilic in [53, 55] beschrieben. Sequenzautomaten können, im Gegensatz zu herkömmlichen endlichen Automaten, ebenfalls die starken PSL-Operatoren behandeln. Die Synthese für die Hardware-Verifikation ist in [54] dargestellt.

Verschiedene kommerzielle Werkzeuge unterstützen Zusicherungssprachen als Eingabe für funktionale Eigenschaftsprüfung: RuleBase von der Firma IBM ist ein Modellprüfer, welcher PSL unterstützt [384]. Incisive Formal Verifier von der Firma Cadence unterstützt neben PSL auch SystemVerilog Assertions [238]. Solidify, ein Produkt der Firma Averant, unterstützt ebenfalls PSL und SVA [404]. Daneben unterstützen viele RTL-Simulatoren ebenfalls Zusicherungssprachen zur simulativen Überprüfung von Zusicherungen. VCS, ein Simulator der Firma Synopsys, unterstützt SVA als Eingabesprache [450]. ModelSim von der Firma Mentor Graphics [328] unterstützt PSL. Incisive Design Team Simulator von der Firma Cadence wiederum unterstützt beide Möglichkeiten [237]. Schließlich erlaubt das Werkzeug FoCs [167] von der Firma IBM die automatische Generierung von Monitoren in VHDL, Verilog oder SystemC aus PSL-Zusicherungen.

Die SAT-basierte Modellprüfung wurde erstmals im Jahr 1999 von Biere et al. vorgestellt [49, 48]. Deren Anwendung auf Schaltungen ist ausführlich in [174] dargestellt. Darin werden im Wesentlichen drei Verfahren vorgeschlagen, um die Effizienz des Standard-Verfahrens zu verbessern: 1) *Dynamische Schaltungsvereinfachungen* helfen, das iterative Schaltungsmodell möglichst klein zu halten und somit die Variablenanzahl bei der Übersetzung in eine aussagenlogische Formel zu verkleinern [52, 465, 173]. 2) *Iterative Lernverfahren* verkürzen die Zeit zur Verifikation durch Wiederverwendung von Ergebnissen von vorherigen Läufen des SAT-Solvers für kleinere Schranken [401, 463]. 3) Die Übersetzung von Schaltung und temporaler Formel in eine aussagenlogische Formel erfolgt typischerweise monolithisch, d. h. die gesamte aussagenlogische Formel wird für eine gegebene Schranke k erzeugt und auf Erfüllbarkeit unter Verwendung eines Standard-SAT-

Solvers überprüft. Die *inkrementelle Übersetzung von LTL-Formeln* hilft, Teile der bereits erstellten aussagenlogischen Formel wiederzuverwenden [176]. Eine Erweiterung der SAT-basierten Modellprüfung zur Behandlung von Speichermodulen ist in [175, 177] beschrieben. Dort werden neben den eingebetteten Speichern mit einem Lese-/Schreibport auch Erweiterungen für Speicher mit mehreren Lese-/Schreibports vorgestellt.

Der vorgestellte Ansatz zur effizienten Behandlung von Wortbreiten basiert auf den Arbeiten von Bryant et al. [60]. Eine Vielzahl weiterer Ansätze für Entscheidungsprozeduren für Bitvektor-Arithmetik sind in der Literatur bekannt. Das engl. *bit blasting* ist das weit verbreitetste, bei dem die Bitvektor-Operationen durch Boolesche Formeln ersetzt werden. Ein Beispiel hierfür ist der Cogent-Ansatz [114]. Verbesserungen werden in CVC-Lite erzielt, indem vor dem eigentlichen *bit blasting* eine Normalisierungsschritt erfolgt [179]. Ein analoger Ansatz wird in [460, 459] beschrieben, bei dem Schaltungen normalisiert mittels Partialproduktgeneratoren, Additionsnetzwerken und Komparatoren repräsentiert werden. Der Normalisierungsschritt hilft dabei, die später zu generierende SAT-Instanz möglichst kein zu halten. STP [82] verwendet Array-Optimierung in Kombination mit logischen und arithmetischen Vereinfachungen. Frühere Arbeiten basieren auf dem Ansatz von Shostak. Beispiele hierfür sind [124, 31]. Ansätze, welche die Behandlung von Modulo-Arithmetik betrachten, finden sich in [230, 59, 353].

Die Zeitanalyse für synchrone Schaltungen erfolgt auf der Logikebene typischerweise durch eine statische Zeitanalyse. Aufgrund der technologiebedingten Asymmetrien bei Verzögerungszeiten beim Wechsel der Ausgänge von Flip-Flops und Logikgattern von logisch T zu F bzw. umgekehrt, kann eine genauere Abschätzung auf Basis einer Simulation erfolgen, die als dynamische Zeitanalyse bezeichnet wird. Diese liefert allerdings keinerlei Garantien für harte Echtzeitanforderungen und ist darüber hinaus auch sehr zeitintensiv. Diskussionen technologiespezifischer Verzögerungszeiten finden sich z. B. in [438, 311].

Die Zeitanalyse auf Architekturebene besteht im Wesentlichen darin, den implementierten statischen Ablaufplan im Steuerwerk der Schaltung zu analysieren. Verfahren zur Optimierung der Ablaufpläne bezüglich Latenz und Durchsatz kann man in [426] finden. Eine spezielle Klasse synchroner Schaltungen auf Architekturebene sind sog. *latenzinsensitive Systeme*. Ein erstes latenzinsensitives Systeme wurde unter diesem Namen in [83] vorgestellt. Die Theorie zu latenzinsensitiven Systemen wurde erstmals in [84] vorgestellt. Um ein latenzinsensitives System zu realisieren, werden für jeden Kommunikationskanal zwischen Hardware-Komponenten zwei neue Verbindungen zwischen diesen Komponenten eingefügt. Eine Verbindung besitzt die selbe Richtung wie der Kommunikationskanal und dient zur Anzeige, ob die Daten im Kanal informativ oder nichtinformativ sind. Die zweite Verbindung verläuft in entgegengesetzter Richtung und zeigt *back pressure* an. Eine erste Zeitanalyse für latenzinsensitive Systeme wurde in [85] vorgestellt, vernachlässigte aber *back pressure*. Eine Erweiterung zur Berücksichtigung dieser Effekte ist in [309] vorgestellt.

7

Software-Verifikation

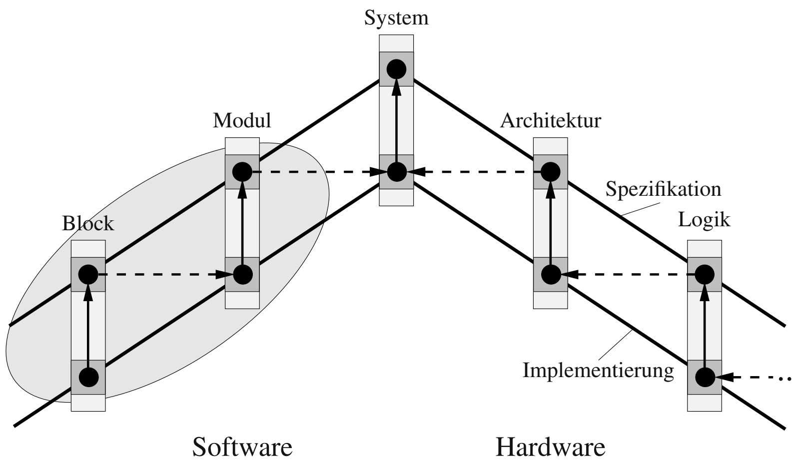


Abb. 7.1. Software-Verifikation

In diesem Kapitel werden wichtige Methoden zur Verifikation von Software beschrieben. Zunächst werden Verfahren zur Äquivalenzprüfung auf Blockebene präsentiert. Dabei wird die Verifikation sowohl von Assembler- als auch von C-Programmen, die typischen Einschränkungen für eingebettete Software unterliegen, betrachtet. Danach werden Verfahren zur Testfallgenerierung für simulative Verifikationsmethoden für Software zusammen mit den zugehörigen Überdeckungsmaßen vorgestellt. Anschließend werden formale Methoden zur funktionalen Eigenschaftsprüfung von Software präsentiert. Schließlich folgen Verfahren zur Verifikation des Zeitverhaltens eingebetteter Software. Dabei wird zunächst die Abschätzung des Zeitverhaltens auf Blockebene und anschließend der Einfluss dynamischer Ablaufplanungsverfahren auf die Antwortzeit von Prozessen betrachtet.

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_7, © Springer-Verlag Berlin Heidelberg 2010

7.1 Formale Äquivalenzprüfung eingebetteter Software

Das Problem der Äquivalenzprüfung von zwei Programmen ist im Allgemeinen nicht entscheidbar. Aus diesem Grund wird ein Großteil der Äquivalenzprüfung von Software simulativ durchgeführt, wobei ein Programm, welches durch Transformation aus einem Referenzprogramm entstanden ist, mit dem Referenzprogramm verglichen wird. Simulative Äquivalenzprüfung ist allerdings unvollständig, weshalb im Allgemeinen lediglich die Anwesenheit von Fehlern, nicht aber deren Abwesenheit gezeigt werden kann. Die Erzeugung geeigneter Testfälle wird in Kapitel 7.2 diskutiert.

Während die Äquivalenz von Programmen im Allgemeinen nicht gezeigt werden kann, kann durch Einschränkungen eine Klasse an Programmen definiert werden, für welche dies noch formal möglich ist. Eingebettete Software unterliegt oftmals genau solchen Einschränkungen, weshalb die formale Äquivalenzprüfung eingebetteter Software in den letzten Jahren neue Aufmerksamkeit erhalten hat. Einige wichtige Ansätze werden im Folgenden betrachtet.

Ähnlich wie Hardware, wird eingebettete Software starken Optimierungen unterzogen. Dabei muss die Software Anforderungen an ihr Laufzeitverhalten entsprechen, aber auch weitere nichtfunktionale Eigenschaften, z. B. bezüglich der maximalen Leistungsaufnahme und ihres Speicherbedarfs, erfüllen. Ein Programm, welches lediglich ein paar Bytes mehr an Speicher benötigt als eine Alternative, erfordert eventuell die Verwendung eines größeren und teureren Speichers. Auch ein Programm, welches nur ein bisschen langsamer ist als die Vorgabe, kann inakzeptabel sein. Aus diesem Grund wird eingebettete Software häufig einer sehr starken Optimierung unterzogen. Das kann soweit gehen, dass sogar compilierte Programme nochmals manuell verbessert werden.

Daneben wird eingebettete Software häufig für Spezialprozessoren übersetzt. Diese wiederum sind selbst ebenfalls hoch optimiert, z. B. im Hinblick auf Leistungsaufnahme, Kosten und Geschwindigkeit. Spezialprozessoren besitzen dabei meist Spezialinstruktionen und mehrere parallel arbeitende funktionale Einheiten im Datenpfad. Die Implementierungsdetails sind dabei nicht immer vor dem Programmierer verborgen und müssen berücksichtigt werden. Dies bedeutet einerseits, dass eingebettete Software stark optimiert werden kann, andererseits bedeutet dies große Herausforderungen bei der Codegenerierung, -optimierung und -verifikation.

Schließlich sei noch erwähnt, dass Fehler in eingebetteter Software weniger toleriert werden als in Desktop-Software. Dies liegt zum einen daran, dass das Einspielen einer neuen Software-Version im Betrieb sehr kostspielig oder auch unmöglich sein kann. Zum anderen können die Folgen eines Fehlers bei eingebetteten Systemen, die stark mit ihrer Umwelt interagieren, katastrophale Folgen haben. Aus diesen Gründen gewinnt die Verifikation eingebetteter Software zunehmend an Bedeutung.

7.1.1 Äquivalenzprüfung von Assemblerprogrammen

Im Folgenden wird ein Ansatz zur Äquivalenzprüfung von Assemblerprogrammen von Currie et al. [122] basierend auf *symbolischer Simulation* vorgestellt. Das Ver-

fahren eignet sich lediglich dazu, kleinere Programmsegmente miteinander zu vergleichen. Aber selbst für kleinere Programmsegmente gilt, dass deren Äquivalenz nicht offensichtlich sein muss, weshalb sich eine entsprechende Prüfung und deren Automatisierung an dieser Stelle lohnt.

Beispiel 7.1.1. Betrachtet wird das folgende Assemblerprogramm für einen digitalen Signalprozessor aus [123]:

```
1         msm dx, a0, a1
2         bge OK
3         add dx, cx
4 OK:     mov (x0++1), dx
```

Die erste Instruktion multipliziert den Inhalt von Register a0 mit dem Inhalt des Registers a1. Das Ergebnis wird zu dem Inhalt in Register dx addiert und dort gespeichert. Die Instruktion setzt dabei Status-Register. Damit verzweigt die folgende Instruktion (bge) zu der Marke OK, wenn die vorherige Multiplikation zu einem nichtnegativen Ergebnis geführt hat. Die add-Instruktion addiert den Inhalt von Register cx zu dem Inhalt in Register dx. Das Ergebnis steht anschließend im Register dx. Schließlich wird der Inhalt des Registers dx in den Speicher geschrieben. Die verwendete Adresse ergibt sich dabei aus dem Inhalt des Indexregisters x0, der inkrementiert wird, um auf die nächste Speicheradresse zu zeigen. Der Kontroll-Datenflussgraph des Assemblerprogramms ist in Abb. 7.2 zu sehen.

Symbolische Simulation und uninterpretierte Funktionen

Das hier vorgestellte Verfahren basiert auf symbolischer Simulation. Ausgehend von zwei Programmsegmenten, werden durch symbolische Simulation für diese Repräsentanten erstellt. Basierend auf den beiden resultierenden Repräsentanten wird anschließend die Äquivalenz geprüft.

Wie bei der symbolischen Simulation von Hardware (Boolesche Netzwerke) kann auch Software symbolisch simuliert werden. Anstatt jedes Signal der Mikroarchitektur, die das Assemblerprogramm ausführt, mit einem Wert zu belegen, werden bei der symbolischen Simulation von Assemblerprogrammen in jeder Programmzeile die neuen Werte der Register und Speicher der Mikroarchitektur bestimmt. Um die aus der Äquivalenzprüfung für Hardware bekannte symbolische Simulation verwenden zu können, ist es möglich, die Registerinhalte durch symbolische Bitvektoren zu repräsentieren. Der Vorteil hierbei wäre, dass der Datenpfad des verwendeten Prozessors mit in der Verifikation berücksichtigt wird. Der Nachteil ergibt sich aus der exponentiell wachsenden Größe der Repräsentation, die hierbei für die Booleschen Funktionen entsteht. Dies gilt insbesondere für Software, die intensiv Operationen wie Multiplikation und Division durchführt.

Aus diesem Grund erfolgt die symbolische Simulation auf Basis beliebiger Datentypen. Dies erfolgt mit Hilfe eines symbolischen ISA-Simulators (engl *Instruction Set Architecture*). Einige Funktionen, wie Addition oder Sprungbefehle, können durch arithmetische und logische Operatoren erfasst werden. Kommen komplexere Funktionen hinzu, so bietet es sich an, Symbole für sog. *uninterpretierte Funktionen*

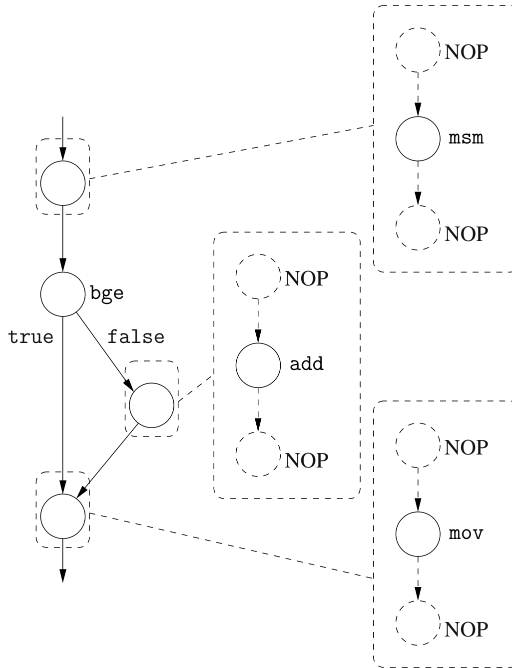


Abb. 7.2. Kontroll-Datenflussgraph des Assemblerprogramms aus Beispiel 7.1.1

zu erzeugen. Wie der Name bereits sagt: Über uninterpretierte Funktionen werden keine Annahmen gemacht, außer, dass es sich um Funktionen im mathematischen Sinne handelt: Für die selbe Eingabe wird also stets das selbe Ergebnis berechnet, d. h. sei $a = b$ und $c = d$, dann gilt für die uninterpretierte Funktion f, dass $f(a,c) = f(b,d)$ ist.

Uninterpretierte Funktionen bieten die Möglichkeit zur Abstraktion. Ob ein Multiplizierer im Datenpfad des Prozessors richtig multipliziert, muss z. B. auf einer höheren Abstraktionsebene nicht überprüft werden. Allein die Aussage, dass bei jeder Multiplikation mit den selben Operanden das selbe Ergebnis berechnet wird, kann für die Äquivalenzprüfung zweier Programmsegmente ausreichend sein.

Beispiel 7.1.2. Zunächst werden Kontrollstrukturen vernachlässigt. Durch symbolische Simulation ergeben sich für das Programmsegment in Beispiel 7.1.1 ohne die `add`-Operation die folgenden Ausdrücke:

$$\begin{aligned} \texttt{dx} &:= init_dx + f_{\text{mult}}(init_a0, init_a1) \\ \texttt{x0} &:= init_x0 + 1 \\ \texttt{mem} &:= f_{\text{write}}(init_mem, init_x0 + 1, init_dx + f_{\text{mult}}(init_a0, init_a1)) \end{aligned}$$

Dabei wurden zwei uninterpretierte Funktionen verwendet f_{mult} und f_{write}. Die Symbolnamen mit Präfix *init_* repräsentieren die Initialwerte von Registern und Speichern.

Man beachte, dass die Abstraktion durch uninterpretierte Funktionen sicher ist, d. h. zwei nicht äquivalente Programmsegmente werden nicht durch die Verwendung von uninterpretierten Funktionen fälschlicherweise für äquivalent gehalten. Andererseits kann die Abstraktion zu konservativ sein, was bedeutet, dass zwei äquivalente Programmsegmente für nicht äquivalent gehalten werden. Ein Beispiel hierfür ist die Multiplikation mit Zwei. Diese kann neben der "echten" Multiplikation auch als Linksshift der Bitvektor-Repräsentation in nur einem der Programmsegmente implementiert werden. Dass dann die Äquivalenz nicht erkannt wird, liegt an den unterschiedlichen verwendeten Symbolen $f_{\text{mult}}(x,2)$ und $f_{\text{shift}}(x)$ in der symbolischen Simulation.

Ein weiteres Problem, dass sich aus der Nichtinterpretation der Multiplikation ergibt, ist, dass die Symbole $f_{\text{mult}}(x,2)$ und $f_{\text{mult}}(2,x)$ nicht als identisch erkannt werden. Allgemein gilt, dass durch die Verwendung uninterpretierter Funktionen Wissen über Kommutativität und Assoziativität von Operationen verloren geht. Auf der anderen Seite kann aufgrund von Rundungsoperationen im Allgemeinen nicht davon ausgegangen werden, dass die implementierte Multiplikation kommutativ ist.

Äquivalenzprüfung

Durch die symbolische Simulation werden prädikatenlogische Formeln für zwei Programmsegmente gebildet. Um die *Äquivalenz* der beiden Programmsegmente zu zeigen, muss die Äquivalenz der Formeln bewiesen werden. Da bei der Bildung der prädikatenlogischen Formeln uninterpretierte Funktionen verwendet werden, erfolgt die Äquivalenzprüfung mit Hilfe eines SMT-Solvers (siehe Anhang C.3).

Ein SMT-Solver arbeitet im Wesentlichen wie ein SAT-Solver: Anfangs werden atomare prädikatenlogische Formeln durch Boolesche Variablen abstrahiert. Die resultierende aussagenlogische Formel wird mit einem SAT-Solver gelöst (siehe Anhang C.2). Findet der SAT-Solver eine konsistente Belegung der Booleschen Variablen, wird ein spezialisierter Theorielöser gestartet, der eine konsistente Variablenbelegung für die zu erfüllenden atomaren prädikatenlogischen Formeln (ausgewählt durch die Booleschen Variablen mit der Belegung $\top$) sucht. Wird eine solche konsistente Belegung gefunden, ist die gesamte prädikatenlogische Formel erfüllt. Findet der Theorielöser jedoch keine solche Belegung, so muss der SAT-Solver eine Zurückverfolgung durchführen. Ist der SAT-Solver nicht in der Lage eine konsistente Variablenbelegung für die aussagenlogische Formel zu finden, so ist auch die ursprüngliche prädikatenlogische Formel nicht erfüllbar.

Ein SMT-Solver ist heutzutage deshalb typischerweise als SAT-Solver realisiert, der einen spezialisierten Theorielöser verwendet. Da dabei nicht die Erfüllbarkeit der prädikatenlogischen Formel bezüglich aller möglichen Theorien, sondern lediglich bezüglich einer ausgewählten sog. *Hintergrundtheorie* überprüft wird, spricht man von Erfüllbarkeit modulo Theorien (engl. *Satisfiability Modulo Theories, SMT*).

Für das obige Beispiel der Äquivalenzprüfung zweier Programmsegmente ist die verwendete Hintergrundtheorie „Gleichheit und uninterpretierten Funktionen“(engl. *Equality and Uninterpreted Functions, EUF*).

Für die EUF-Theorie gibt es zwei mögliche Lösungsansätze: (1) Der Kongruenzabschluss und (2) die Reduktion auf Aussagenlogik (siehe auch Abschnitt 6.3.1). Beim Kongruenzabschluss für uninterpretierte Funktionen überwacht der Theorielöser alle Terme der prädikatenlogischen Formel und bildet *Äquivalenzklassen* für diejenigen Terme, für welche die Äquivalenz gezeigt wurde. Zwei Terme basierend auf uninterpretierten Funktionen sind genau dann gleich, wenn sie das selbe Funktionssymbol verwenden und die verwendeten Funktionsargumente äquivalent sind.

Beispiel 7.1.3. Abbildung 7.3 zeigt den Kongruenzabschluss bei uninterpretierten Funktionen. Es soll gezeigt werden, dass unter der Annahme $a = b$ und $b = c$ auch gilt, dass $f(a) = f(c)$ ist. Hierzu muss zunächst gezeigt werden, dass $a = b$ ist, was bereits durch die Annahme erfüllt ist. Somit können a und b der selben Äquivalenzklasse zugeordnet werden. Als nächstes wird gezeigt, dass $b = c$ ist. Dies ist wiederum durch die Annahme gegeben, weshalb auch b und c der selben Äquivalenzklasse zugeordnet werden.

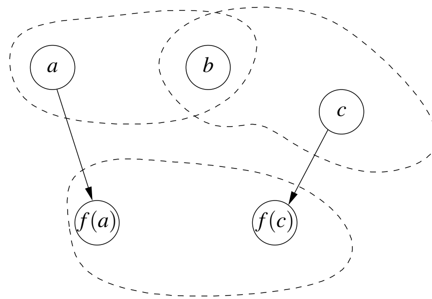


Abb. 7.3. Kongruenzabschluss [122]

Da die Äquivalenzrelation transitiv ist, bedeutet dies, dass ebenfalls a und c in der selben Äquivalenzklasse liegen. Zum Schluss soll nun gezeigt werden, dass $f(a) = f(c)$. Dies kann aus $a = c$ und der Verwendung des selben Funktionssymbols f direkt geschlossen werden, weshalb $f(a)$ und $f(c)$ der selben Äquivalenzklasse zugeordnet werden.

Neben der Hintergrundtheorie „Gleichheit und uninterpretierte Funktionen“müssen weitere Hintergrundtheorien zur Äquivalenzprüfung von Assemblerprogrammen unterstützt werden. Die wohl wichtigste ist die Theorie zu „Arrays und Speichern“(siehe auch Abschnitt 6.3.1 und 6.4.2). Diese ermöglicht es, Systeme mit Speichern zu analysieren, ohne den Zustandsraum des Speichers selbst abzubilden.

Die Theorie zu „Arrays und Speichern“ besteht aus zwei Funktionen: $f_{\text{read}}(mem, addr)$, die den Wert im Speicher *mem* an Adresse *addr* liefert, und $f_{\text{write}}(mem, addr,$

val), die in den Speicher *mem* an Adresse *addr* den Wert *val* speichert und den so veränderten Speicherinhalt zurück gibt. Es gilt:

$$f_{\text{read}}(f_{\text{write}}(mem, addr1, val), addr2) = \begin{cases} val & \text{falls } addr1 = addr2 \\ f_{\text{read}}(mem, addr2) & \text{sonst} \end{cases}$$

Behandlung von konditionalen Sprüngen

Bisher wurden lediglich Folgen von Instruktionen eines Assemblerprogramms behandelt. Um darüber hinaus auch konditionale Sprünge behandeln zu können, muss der symbolische Simulator diese unterstützen. Eine Möglichkeit dieser Unterstützung besteht darin, dass der symbolische Wert, der in einem Register gespeichert ist, von Bedingungen abhängt. In diesem Fall würde man einen konditionalen Ausdruck über alle möglichen Wertebelegungen eines Registers bestimmen.

Beispiel 7.1.4. Für das Programmsegment aus Beispiel 7.1.1 ergeben sich zwei mögliche Pfade zur Programmausführung. Bei der Auswertung bestimmt der symbolische Simulator die Funktionen zur Wertebelegung des Register sowie die Bedingungen, die hierzu führen. Dies resultiert in dem folgenden Ausdruck für die Belegung des Registers `dx`:

$$\texttt{dx} := \begin{cases} \text{IF} & (init_dx + f_{\text{mult}}(init_a0, init_a1) \geq 0) \\ \text{THEN} & init_dx + f_{\text{mult}}(init_a0, init_a1) \\ \text{ELSE} & init_dx + f_{\text{mult}}(init_a0, init_a1) + init_cx \end{cases}$$

An diesem Beispiel erkennt man bereits, dass die konditionalen Ausdrücke für realistische Programme sehr groß werden können. Dies gilt zumindest, wenn die symbolischen Sprungbedingungen nicht direkt zu T oder F evaluieren, was die Auswahl des Sprungziels eindeutig macht. Alternativ kann man auch paarweise mögliche Ausführungspfade der Programme auf Äquivalenz prüfen. Hierzu ist es allerdings notwendig, dass die beiden zu vergleichenden Programme identische Kontrollstrukturen enthalten. Dies muss aber nicht immer der Fall sein.

Reduktion der Anzahl falschnegativer und falschpositiver Ergebnisse

Das hier vorgestellte Verfahren orientiert sich stark an der Äquivalenzprüfung von kombinatorischen Schaltungen. Ausgehend von zwei Programmsegmenten wird mit Hilfe von symbolischer Simulation ein Repräsentant für jedes Programmsegment erzeugt. Basierend auf den beiden Repräsentanten wird anschließend gezeigt, dass die beiden Programmsegmente für die selben Eingaben auch das selbe Ergebnis berechnen. Damit das Verfahren anwendbar ist, muss bei der Erstellung der Repräsentanten an einigen Stellen approximiert werden. Beispielsweise werden die verwendeten Datenformate nicht bitgenau dargestellt. Hierdurch kann es bei dem Verfahren sowohl zu falschnegativen als auch falschpositiven Ergebnissen kommen, d. h. die beiden Programmsegmente werden als nicht äquivalent erkannt, sind aber äquivalent, bzw.

die beiden Programmsegmente werden als äquivalent erkannt und sind es nicht. Letzterer Fall tritt nur sehr selten auf, weshalb das Verfahren als nahezu sicher bezeichnet werden kann. Im Folgenden wird die Reduktion von falschen Ergebnissen genauer betrachtet.

Reduktion der Anzahl falschnegativer Ergebnisse

Die Hauptursache für *falschnegative Ergebnisse* liegt in der Verwendung von uninterpretierten Funktionen. Die Theorie von „Gleichheit und uninterpretierten Funktionen" reicht nicht aus, um alle Eigenschaften auszudrücken, die für den Äquivalenzbeweis notwendig sind. Ein häufiges Problem ist etwa die Compiler-Optimierung, welche die Multiplikation mit einer Konstanten, die eine Zweierpotenz ist, durch eine Schiebe-Operation ersetzt. Daneben basieren viele Compiler-Optimierungen auf Kommutativität und Assoziativität von Berechnungen.

Um solche Compiler-Optimierungen zu unterstützen, müssen dem SMT-Solver weitere entscheidbare Hintergrundtheorien in Form von Axiomen zur Verfügung gestellt werden. Ein mögliches Axiom ist somit die Kommutativität der Multiplikation, die wie folgt ausgedrückt werden kann:

$$f_{\text{mult}}(arg1, arg2) = (f_{\text{mult}}(arg1, arg2) \vee f_{\text{mult}}(arg2, arg1))$$

Trifft der symbolische Simulator auf eine Multiplikationsinstruktion, so wird diese durch beide möglichen Ersetzungen repräsentiert.

Reduktion der Anzahl falschpositiver Ergebnisse

Die wesentliche Quelle für *falschpositive Ergebnisse* liegt in der Abstraktion der verwendeten Datentypen. Um die symbolischen Ausdrücke nicht unnötig zu vergrößern, wird auf eine bitakkurate Repräsentation der Datentypen verzichtet. Da die Mikroarchitektur des Prozessors jedoch auf endlichen Zahlendarstellungen arbeitet, kann es passieren, dass zwei Assemblerprogramme für äquivalent gehalten werden, obwohl sie dies nicht sind.

Als Beispiel dienen hier die beiden Berechnungen $(x+y)-z$ und $(x-z)+y$. Bei Verwendung unbeschränkter Zahlendarstellungen sind diese beiden Ausdrücke in der Tat äquivalent. Werden die beiden Berechnungen auf einem Prozessor mit beschränkter Zahlendarstellung berechnet, kann es allerdings aufgrund von internen Rundungen oder Überläufen bei einigen Eingaben zu unterschiedlichen Ergebnissen kommen. Von diesem Problem sind nicht zwangsläufig alle Hintergrundtheorien betroffen.

7.1.2 Strukturelle Äquivalenzprüfung von Assemblerprogrammen

Analog zur strukturellen Äquivalenzprüfung von kombinatorischen Schaltungen, kann man *strukturelle* Verfahren auch zur *Äquivalenzprüfung* von Assemblerprogrammen einsetzen. Die grundlegende Frage lautet dabei: Wie lassen sich geeignete

Schnittpunkte in Assemblerprogrammen definieren? Im Folgenden wird ein Ansatz nach Feng und Hu näher betrachtet [160].

In kombinatorischen Schaltungen werden Datenwerte entlang von Verbindungen von den Schaltungseingängen zu den -ausgängen transportiert. Die Logikgatter transformieren diese Datenwerte. Entsprechend werden in Software Datenwerte von Programmzustand zu -zustand (Programmzähler, Registerinhalte etc.) transportiert und durch Instruktionen manipuliert. Somit muss ein Schnittpunkt ein Teil eines Programmzustands sein, der äquivalent zu einem Schnittpunkt in dem Vergleichsprogramm ist.

Zur strukturellen Äquivalenzprüfung in Hardware wird der Schnittpunkt durch einen neuen primären Eingang ersetzt. Analog hierzu kann bei der symbolischen Simulation der beiden Programme der symbolische Wert, der den Schnittpunkt repräsentiert, gelöscht und durch eine neue symbolische Variable repräsentiert werden. Kann nun die Äquivalenz der verbleibenden Programmen mit Hilfe der neuen symbolischen Variablen gezeigt werden, so sind auch die ursprünglichen Programme äquivalent.

Während in Hardware jede Leitung für jede Belegung einen Wert zugewiesen bekommt, ist es in Software aufgrund des möglicherweise verzweigten Kontrollflusses nicht notwendig, dass für jede Eingabe auch alle Instruktionen durchlaufen werden. Bei Verzweigung können Instruktionen gar nicht oder auch mehrfach ausgeführt werden. Damit ist aber auch der Programmzustand nicht für jeden Punkt in einem Programm definiert. Aus diesem Grund werden Schnittpunkte in Software *dynamisch* anhand des aktuellen Ausführungspfads, und nicht anhand des statischen Quelltextes, bestimmt.

Beispiel 7.1.5. Es wird eine Schleife betrachtet, die einen Speicherblock der Größe von 1024 Wörtern mit null initialisiert. Ohne Berücksichtigung der Schleifenvariablen sieht das zugehörige Assemblerprogramm wie folgt aus:

```
   1      STW .D1 A0, *A4++
   2      STW .D1 A0, *A4++
          ...
1024      STW .D1 A0, *A4++
```

Davor wurde das Register `A0` mit null initialisiert und das Register `A4` enthält die Speicheradresse des Blockanfangs. Durch symbolische Simulation von i Instruktionen des Assemblerprogramms erhält man den folgenden symbolischen Ausdruck für den Speicher:

$$f_{\text{write}}(\ldots f_{\text{write}}(f_{\text{write}}(init_mem, init_a4, 0), init_a4 + 4, 0) \ldots, init_a4 + 4(i-1), 0)$$

wobei $init_mem$ und $init_a4$ die Initialwerte des Speichers und des Register `A4` bezeichnen.

Vergleicht man nun das Schleifenprogramm mit sich selbst (also zwei Kopien des selben Programms), so stellt man fest, dass die Programmzustände nach jeder Instruktion identisch sind. Somit kann nach jeder Instruktion ein neuer Schnittpunkt eingeführt werden, welcher sich durch den momentanen Speicherzustand mem_i und den den momentanen Adresswert $a4_i$ repräsentieren lässt. Somit kann

die Äquivalenzprüfung für den nächsten Schritt auf den symbolischen Ausdruck $f_{\text{write}}(mem_i, a4_i, 0)$ für beide Assemblerprogramme reduziert werden.

Bereits an diesem simplen Beispiel kann man die interessanten Fragen erkennen, die es zu beantworten gilt, um eine strukturelle Äquivalenzprüfung für Assemblerprogramme zu konstruieren:

- *Wo und wie feingranular sollte nach Schnittpunkten gesucht werden?* Im obigen Beispiel ist nach jeder Instruktion der Zustand beider Prozessoren identisch. Somit kann der gesamte Prozessorzustand als Schnitt verwendet werden. Für das obige Beispiel würde dies in der Tat funktionieren, allerdings würden für komplexere Beispiele falschnegative Ergebnisse erzeugt werden. Andererseits könnte jedes Registerpaar oder jedes Speicherzeilenpaar überprüft werden, ob es sich für einen Schnittpunkt eignet. Hierbei würde allerdings die Menge an Möglichkeiten explodieren.
- *Wie findet man Schnittpunkte?* In dem obigen Beispiel konnte für beide Programme je eine Instruktion ausgeführt werden und anschließend konnten geeignete Schnittpunkte gefunden werden. Allerdings werden im Allgemeinen Instruktionen umgeordnet oder aufgrund von Optimierungen eliminiert.
- *Wie kann der Schnitt durchgeführt werden?* Wenn ein Schnittpunkt gefunden wurde, wird dieser in beiden Programmen durch eine neue symbolische Variable ersetzt. Allerdings kann es vorkommen, dass zu dem Zeitpunkt, zu dem ein Schnittpunkt identifiziert wurde, der symbolische Simulator bereits weiter simuliert hat. Hierbei kann er bereits symbolische Werte des Schnittpunktes verwendet haben. Wie kann also die neu eingeführte symbolische Variable verwendet werden? Müssen alle bereits berechneten Werte gelöscht werden?
- *Wie kann die Anzahl falschnegativer Ergebnisse reduziert werden?* Die Beantwortung der ersten drei Fragen hat großen Einfluss auf die Rate, mit der falschnegative Ergebnisse erzeugt werden. Welche Möglichkeiten gibt es dann, die Anzahl an falschnegativer Ergebnisse zu reduzieren?

Ein Ansatz, der die obigen Fragen beantwortet, ist in [160] vorgestellt. Hierbei werden Schnittpunkte lediglich im Speicher gesucht, d. h., die Register des Prozessors werden nicht berücksichtigt. Dabei wird der gesamte Speicher als Schnittpunkt interpretiert, d. h. lediglich, wenn die Inhalte des gesamten Speichers bei zwei Programmausführungen identisch sind, handelt es sich tatsächlich um einen Schnittpunkt. Dieser Ansatz hilft, die symbolischen Ausdrücke für die Speicher klein zu halten, da jedes Mal, wenn ein Schnittpunkt gefunden wurde, dieser durch neue symbolische Variablen repräsentiert wird. Register können potentiell auch Schleifenvariablen halten. Ist dies der Fall, und eine Schleifenvariable ist Teil eines Schnittpunktes, kann das Terminierungskriterium der Schleife nach der Substitution durch eine neue symbolische Variable nicht mehr überprüft werden.

Zur Identifikation von Schnittpunkten muss jetzt nur noch der Speicher nach jedem Schreibbefehl überprüft werden. Dazu werden beispielsweise die letzten k symbolischen Ausdrücke, die den Speicher repräsentiert haben, gespeichert. In diesem Fall können die beiden zu vergleichenden Programme simuliert werden, bis jedes

k Schreibbefehle durchgeführt hat. Anschließend werden Schnittpunkte in den k^2 Kombinationen gesucht. Wird das k erhöht, können im Fall von Instruktionsumsortierung größere Bereiche bezüglich möglicher Schnittpunkte betrachtet werden. Hierdurch verlangsamt sich allerdings auch die Verifikation.

Die Berücksichtigung der Umsortierung von Instruktionen führt auch zu Problemen: So kann es passieren, dass der symbolische Ausdruck für den Speicher eines Programms mit dem symbolischen Ausdruck für den Speicher des anderen Programms von vor k Schreibbefehlen übereinstimmt. In diesem Fall werden die beiden Speicher durch die selbe neue symbolische Variable repräsentiert. Allerdings muss das Programm, welches bereits k Schreibbefehle (und eine nicht bekannte Anzahl an Instruktionen) weiter gelaufen ist, auf diesen Zeitpunkt zurückgesetzt werden und die symbolische Simulation mit der neuen Variable neu gestartet werden. Hierfür ist es notwendig, dass mit jedem Ausdruck, der den Speicher repräsentiert, auch der Prozessorzustand nach diesem Schreibbefehl als sog. *Kontrollpunkt* (engl. *check point*) gespeichert wird. Mit diesem Kontrollpunkt kann der Prozessor neu initialisiert und die symbolische Simulation neu gestartet werden.

Eine Alternative besteht in der Weitergabe des neuen symbolischen Wertes durch die symbolischen Ausdrücke der Programmausführung bis zum zuletzt simulierten Schreibbefehl. Da dies aber eine unbestimmte Anzahl an Instruktionen (nicht Schreibbefehle) beinhalten kann, müsste für jede dieser Instruktionen der Prozessorzustand gespeichert werden. Dies ist im Allgemeinen unrealistisch. Weiterhin würde dies nicht ausreichen, da durch die Einführung einer neuen symbolischen Variablen im Allgemeinen auch weitere Kontrollpfade zu berücksichtigen sind. Zusammenfassend scheint die Speicherung von Kontrollpunkten nach jedem Schreibzugriff effizienter.

Reduktion falschnegativer Ergebnisse

Bei der oben beschriebenen Methode enstehen *falschnegative Ergebnisse* durch Schnittpunkte auf dem Speicher, wenn es zur Umsortierung von unabhängigen Speicherzugriffen kommt:

```
1      LDW .D1 *A3++, A1
2      NOP 4
3      STW .D1 A0, *A4++
```

kann bei Unabhängigkeit von `A3` und `A4` zu

```
1      STW .D1 A0, *A4++
2      LDW .D1 *A3++, A1
3      NOP 4
```

umsortiert werden. Die in Abschnitt 7.1.1 beschriebene Äquivalenzprüfung würde ohne Probleme die Äquivalenz der beiden Programme zeigen. Durch die Einführung von Schnittpunkten ist es allerdings nicht mehr möglich, diese Äquivalenz zu zeigen. Der Grund hierfür liegt in den Berechnungen nach dem Schnittpunkt mem_1, der nach beiden `STW`-Instruktionen erkannt wird. Im ersten Programm ergibt sich der symbolische Wert von Register `A1` aus $f_{\text{read}}(init_mem, init_a3)$. Für das zweite Programm errechnet sich der Wert für `A1` zu $f_{\text{read}}(mem_1, init_a3)$. Da nach der

Theorie von „Gleichheit und uninterpretierte Funktionen" $f_{\text{read}}(init_mem, init_a3) \neq f_{\text{read}}(mem_1, init_a3)$ wegen $init_mem \neq mem_1$ ist, kann auch die Äquivalenz der beiden Programme nicht gezeigt werden. Es wurde ein falschnegatives Ergebnis erzeugt.

Eine Möglichkeit diese Art von falschnegativen Ergebnissen zu eliminieren, besteht darin, alle verwendeten Adressen bei Schreibzugriffen auf den Speicher aufzuzeichnen. Für jeden Lesezugriff wird anschließend versucht, zu zeigen, dass hierbei keine der Schreibadressen verwendet wird. Ist dies der Fall, so liest diese Leseoperation vom Initialspeicher *init_mem*.

Eine andere Möglichkeit besteht darin, die Entstehung von jedem Schnittpunkt zu speichern. In obigen Beispiel wäre dies $mem_1 := f_{\text{write}}(init_mem, init_a4, init_a0)$. Hierdurch ist die neue symbolische Variable mem_1 nicht unabhängig, d. h. welche Werte diese annehmen kann, hängt von der Funktion f_{write} ab. Im Vergleich zur strukturellen Äquivalenzprüfung kombinatorischer Schaltungen kann man sich dies so vorstellen, als ob der Schnittpunkt in einer Schaltung durch den Schnittpunkt der anderen Schaltungen substituiert wird.

Limitierungen der formalen Äquivalenzprüfung von Assemblerprogrammen

Die beschriebenen Verfahren basieren auf einer Reihe von Annahmen, die eine formale Äquivalenzprüfung von zwei Assemblerprogrammen erlauben. Diese Annahmen müssen erfüllt sein, da die Äquivalenzprüfung von uneingeschränkten Programmen im Allgemeinen nicht entscheidbar ist. Die wesentlichen Annahmen werden hier noch einmal zusammengefasst:

- Die Prüfung eignet sich lediglich zum Vergleich von zwei Assemblerprogrammsegmenten, d. h. insbesondere dürfen diese keine Prozeduren aufrufen.
- Beide Assemblerpogramme müssen den selben Kontrollfluss besitzen. Andernfalls müssen für alle Register konditionale symbolische Werte gespeichert werden. Da diese Ausdrücke sehr schnell anwachsen, würde dies einen praktischen Einsatz der Verfahren verhindern.
- Um den Kontrollfluss bestimmen zu können, ist es notwendig, dass die Programme nicht den Programmzähler oder gar das Programm selbst verändern.
- Weiterhin sind unbeschränkte Schleifen und unbeschränkte Rekursionen verboten, da diese das Äquivalenzprüfungsproblem unentscheidbar machen können.

Neben diesen Einschränkungen an die Assemblerprogramme gibt es auch Einschränkungen an die Verifikationsmethode selbst. Die wohl größte Einschränkung ist die Abstraktion von bitakkuraten Datentypen, was zum einen die Verifikation realistischer Systeme erst ermöglicht, andererseits aber auch durch Rundungen in der Mikroarchitektur zu *falschpositiven Ergebnissen* führen kann. Dies bedeutet, dass die Äquivalenzprüfung fälschlicherweise zwei Assemblerprogramme für äquivalent halten wird, obwohl diese es bei Ausführung nicht sind.

7.1.3 Äquivalenzprüfung von C-Programmen

Die symbolische Simulation kann auch auf eingebettete Software angewendet werden, welche als C-Programm vorliegt. Hierzu muss das C-Programm jedoch den folgenden Restriktionen unterliegen:

- Das C-Programm darf keine dynamische Speicherallokation und keine Zeiger verwenden.
- Das C-Programm darf keine rekursiven Funktionsaufrufe beinhalten.
- Alle Schleifen im C-Programm sind abgerollt.

Einige dieser Einschränkungen können teilweise aufgehoben werden, indem z. B. eine Analyse von Zeigern durchgeführt wird und diese ersetzt werden.

Bei der symbolischen Simulation von C-Programmen wird jede Variable durch einen symbolischen Wert repräsentiert. Wird eine Anweisung im C-Programm symbolisch simuliert, so wird für den resultierenden symbolischen Ausdruck der geschriebenen Variablen eine *Äquivalenzklasse* gebildet. Wird für zwei Variablen aus verschiedenen Äquivalenzklassen gezeigt, dass diese äquivalent sind, so werden die beiden Äquivalenzklassen zu einer zusammengefasst. Nur wenn nach Beendigung des Programms zwei Variablen in der selben Äquivalenzklasse enthalten sind, sind diese beiden Variablen auch äquivalent. Bei konditionalen Sprüngen müssen jeweils beide Fälle symbolisch simuliert werden. Bei der symbolischen Simulation von C-Programmen können wiederum uninterpretierte Funktionen eingesetzt werden. Dies wird an einem simplen Beispiel [315] gezeigt.

Beispiel 7.1.6. Die folgenden zwei Programme sollen auf Äquivalenz geprüft werden:

```
      Programm 1          Programm 2
1     a = b + c;          tmp1 = b;
2                         tmp2 = c;
3                         a = tmp1 + tmp2;
```

Die Äquivalenz der beiden Programme lässt sich zeigen, indem gezeigt wird, dass die Variablen a nach Terminierung beider Programme in der selben Äquivalenzklasse sind.

Im Folgenden wird eine Variable x mit i, j indiziert, wobei i das $(i+1)$-te Auftreten der Variablen im Programm und $j \in \{1,2\}$ das jeweilige Programm bezeichnet. Nach der Initialisierung zu Beginn der Simulation gibt es zwei Äquivalenzklassen:

$$E_1 := \{b_{0,1}, b_{0,2}\}$$
$$E_2 := \{c_{0,1}, c_{0,2}\}$$

Nach Ausführung der ersten beiden Instruktionen aus Programm 2 müssen auch die Variablen tmp1 und tmp2 berücksichtigt werden:

$$E_1 := \{b_{0,1}, b_{0,2}, tmp1_{0,2}\}$$
$$E_2 := \{c_{0,1}, c_{0,2}, tmp2_{0,2}\}$$

Wird nun die Instruktion 1 aus Programm 1 und die Instruktion 3 aus Programm 2 symbolisch simuliert, erhält man:

$$\begin{aligned}
E_1 &:= \{b_{0,1}, b_{0,2}, tmp1_{0,2}\} \\
E_2 &:= \{c_{0,1}, c_{0,2}, tmp2_{0,2}\} \\
E_3 &:= \{a_{0,1}, b_{0,1} + c_{0,1}\} \\
E_4 &:= \{a_{0,2}, tmp1_{0,2} + tmp2_{0,2}\}
\end{aligned}$$

Mit den Äquivalenzklassen E_1 und E_2 kann die Äquivalenz von E_3 und E_4 gezeigt werden. Aus diesem Grund werden Äquivalenzklassen E_3 und E_4 vereinigt:

$$E_{3'} := \{a_{0,1}, a_{0,2}, b_{0,1} + c_{0,1}, tmp1_{0,2} + tmp2_{0,2}\}$$

Da dadurch $a_{0,1}$ und $a_{0,2}$ in der selben Äquivalenzklasse sind, sind die beiden Programme in der Tat äquivalent.

Äquivalenzprüfung mittels Programmabhängigkeitsgraphen

Für große C-Programme stößt die symbolische Simulation schnell an ihre Grenzen. Aus diesem Grund bietet es sich an, zunächst diejenigen Stellen in den Programmen zu identifizieren, bei denen es sich lohnt die Äquivalenz bzw. Nichtäquivalenz von Variablen zu zeigen. Hierzu wird in [315] ein Verfahren vorgestellt, bei dem zunächst Unterschiede in den Quelltexten der C-Programme identifiziert werden. Diese Quelltext-Unterschiede definieren die sog. *Verifikationsbereiche*. Nur für die Verifikationsbereiche wird versucht, deren Äquivalenz zu zeigen. Ist dies nicht möglich, wird versucht, den Verifikationsbereich zu erweitern. Gelingt auch dieses nicht, sind die Programme nicht äquivalent. Wird hingegen für alle Verifikationsbereiche gezeigt, dass diese äquivalent sind, so sind die beiden C-Programme äquivalent. Der gesamte Ablauf ist in Abb. 7.4 zu sehen.

Es kann aufgrund von Optimierungen vorkommen, dass die beiden zu vergleichenden Programme an manchen Stellen keine zeilenweise Korrespondenz zeigen, was für die im Folgenden vorgestellte Bestimmung der Verifikationsbereiche allerdings notwendig ist. Wird dies bei der Bestimmung der Quelltext-Unterschiede festgestellt, werden in den Programmen neutrale Erweiterungen vorgenommen, also Erweiterungen, die keinen Einfluss auf das Ergebnis der Verifikation haben: Handelt es sich bei einem festgestellten Unterschied um eine Zuweisung an die Variable x, so kann in dem Programm, das diese Zuweisung nicht enthält, die Anweisung $x := x$ hinzugefügt werden. Dies wird an einem Beispiel verdeutlicht.

Beispiel 7.1.7. Gegeben sind die beiden Ausschnitte aus zwei Programmen:

	Programm 1	Programm 2
1	`x = a + c;`	`x = a;`
2	`y = b - c;`	`y = b - c;`
3	`x = x;`	`x = x + c;`

Hierbei wurde in Programm 1 die dritte Zeile hinzugefügt, um eine zeilenweise Korrespondenz zwischen den Programmen zu schaffen.

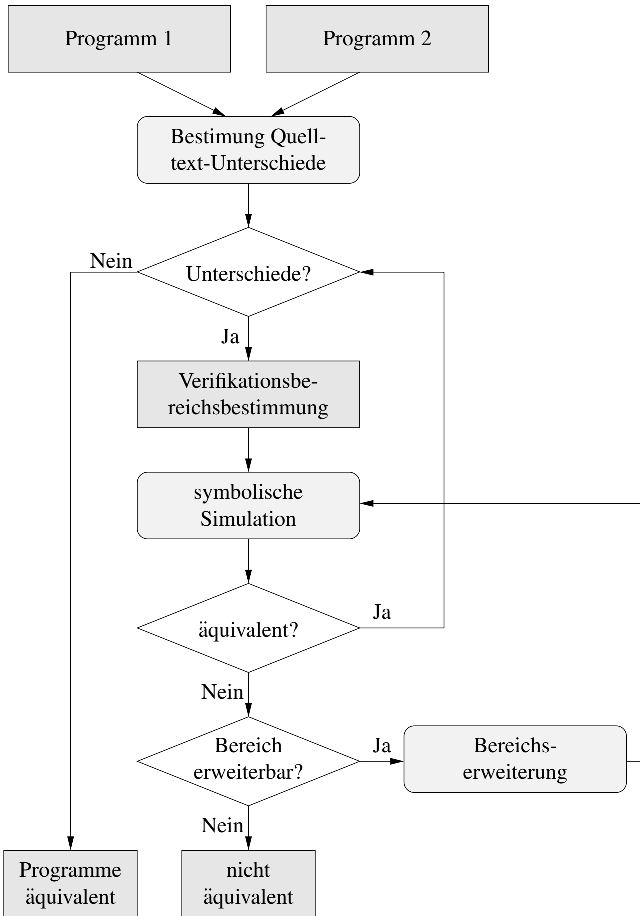


Abb. 7.4. Äquivalenzprüfung von C-Programmen basierend auf Quelltext-Unterschieden [315]

Fehlen in einem Programm Kontrollanweisungen, so kann es um diese Anweisung erweitert werden. In jedem der beiden folgenden Ausführungszweige werden dann zum anderen Programm korrespondierende Zuweisungen von Variablen an sich selbst vorgenommen. Basierend auf den so erweiterten Programmen, können nun die Verifikationsbereiche bestimmt werden.

Für die Bestimmung der Verifikationsbereiche bietet sich eine spezielle Datenstruktur an, der sog. *Programmabhängigkeitsgraph* (engl. *Program Dependence Graph, PDG*) [226].

Definition 7.1.1 (Programmabhängigkeitsgraph (PDG)). *Ein* Programmabhängigkeitsgraph *(PDG) ist ein gerichteter Graph* $G_P(V,E)$*, wobei die Knoten* $v \in V$ *Zuweisungen oder Prädikate von konditionalen Sprüngen repräsentieren. Es existiert*

ein ausgezeichneter Startknoten $v_0 \in V$. Die Menge der Kanten $E = V \times V$ ist bipartitioniert, d. h. $E = E_C \cup E_D$ mit $E_C \cap E_D = \emptyset$.

Die Kontrollflusskanten $e \in E_C$ stellen Kontrollabhängigkeiten dar und sind mit dem Wert F *oder* T *beschriftet. Kontrollflusskanten $e = (v_i, v_j) \in E_C$ beginnen entweder bei einem Knoten v_i, der ein Prädikat repräsentiert, oder dem ausgezeichneten Startknoten $v_i = v_0$. Eine Kontrollflusskante, die mit* T *(*F*) beschriftet ist, stellt den Kontrollfluss für den Fall dar, dass das Prädikat, das durch v_i repräsentiert ist, erfüllt (nicht erfüllt) ist. Die Datenflusskanten $e \in E_D$ stellen Datenabhängigkeiten dar. Eine Datenflusskante $e = (v_i, v_j)$ zwischen Knoten v_i und v_j existiert genau dann, wenn:*

- *In der Zuweisung, die durch v_i repräsentiert wird, die Variable x geschrieben wird,*
- *in der Anweisung, die durch v_j repräsentiert wird, die Variable x gelesen wird und*
- *ein Ausführungspfad von v_i nach v_j existiert, d. h. ein Pfad im Kontrollflussgraph des Programms existiert, auf dem x nicht ein weiteres Mal geschrieben wird.*

Ein PDG kann zu einem *Systemabhängigkeitsgraph* (engl. *System Dependence Graph, SDG*) erweitert werden, in dem Funktionen und Prozeduren ebenfalls als PDG dargestellt werden und Kontroll- und Datenflusskanten die PDGs der Funktionen und Prozeduren geeignet mit dem Programmabhängigkeitsgraph des Programms verknüpft [226].

Beispiel 7.1.8. Gegeben ist das folgende C-Programm mit Eingaben für die Variablen x und y [315]:

```
1     if (x >= y)
2       z = x - y;
3     else
4       z = y -x;
5     return z;
```

Der zugehörige Programmabhängigkeitsgraph ist in Abb. 7.5 zu sehen. Kontrollflusskanten sind gestrichelt dargestellt. Die Eingaben für x bzw. y sind x_in bzw. y_in.

Mit Hilfe der PDGs und den Quelltext-Unterschieden zweier C-Programme lassen sich nun die Verifikationsbereiche definieren. Ein Verifikationsbereich umfasst den Graphen, der durch die Knoten der PDGs, bei denen Unterschiede aufgetreten sind, induziert wird. Für solche Verifikationsbereiche können nun *Eingabe-* und *Ausgabevariablen* definiert werden:

- *Eingabevariablen eines Verifikationsbereichs* sind durch die in den Verifikationsbereich eingehende Datenflusskanten repräsentiert.
- *Ausgabevariablen eines Verifikationsbereichs* sind durch die aus dem Verifikationsbereich ausgehende Datenflusskanten repräsentiert.

Lediglich für die Ausgabevariablen des Verifikationsbereichs, die in beiden Programmen vorkommen, muss Äquivalenz gezeigt werden. Kann für alle Ausgabevariablen deren Äquivalenz gezeigt werden, so sind auch die beiden Programmabschnitte des Verifikationsbereichs *äquivalent*. Kann die Äquivalenz allerdings nicht

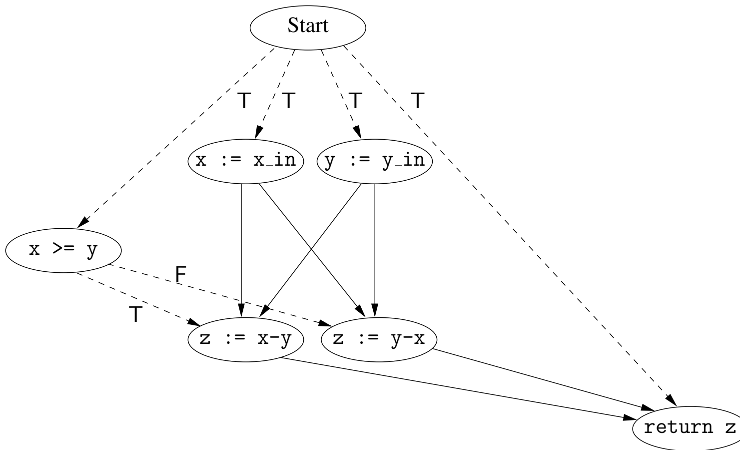


Abb. 7.5. Programmabhängigkeitsgraph [226]

gezeigt werden, so muss der Verifikationsbereich erweitert werden. Hierbei gibt es prinzipiell drei Möglichkeiten:

1. *Rückwärtserweiterung*: Bei der Rückwärtserweiterung wird ein Vorgängerknoten im PDG, der über eine eingehende Datenflusskante verbunden ist, zu dem Verifikationsbereich hinzugenommen. Wenn verschiedene Zuweisungen für den Vorgängerknoten auf unterschiedlichen Kontrollflusspfaden existieren, so müssen alle Knoten, die diese Zuweisungen repräsentieren, inklusive des kontrollierenden Knoten mit dem zugehörigen Prädikat, dem Verifikationsbereich zugeordnet werden.
2. *Vorwärtserweiterung über eine Datenabhängigkeit*: Hierbei wird ein direkter Nachfolgerknoten, der über eine ausgehende Datenflusskante verbunden ist, zu dem Verifikationsbereich hinzugenommen.
3. *Vorwärtserweiterung über eine Kontrollflusskante*: Hierbei werden alle direkten Nachfolgerknoten, die über eine ausgehende Kontrollflusskante eines Knoten im Verifikationsbereich verbunden sind, zu dem Verifikationsbereich hinzugenommen.

Die Frage, welche Erweiterung am sinnvollsten ist, hängt stark von dem gegebenen Programm ab. Allerdings lassen sich Terminierungskriterien spezifizieren, die eine weitere Erweiterung des Verifikationsbereichs überflüssig machen:

- Falls die Äquivalenz der neu zugefügten Knoten bereits gezeigt wurde, so ist eine Rückwärtserweiterung nicht mehr sinnvoll.
- Falls die zugefügten Knoten den Anfang oder das Ende des Programms beschreiben, ist eine Erweiterung über diese Knoten hinaus nicht möglich.

Der erste Fall trifft auch für Eingabevariablen des Verifikationsbereichs zu. Zwei Eingabevariablen sind äquivalent, wenn diese nicht von anderen Verifikationsberei-

chen abhängen, für welche die Äquivalenz noch nicht gezeigt wurde, oder für die beiden Variablen wurde bereits Äquivalenz gezeigt. Die Äquivalenzprüfung mittels Programmabhängigkeitsgraphen wird an einem Beispiel gezeigt [315].

Beispiel 7.1.9. Gegeben sind die beiden Programme:

```
        Programm 1              Programm 2
 1      if (in1 > in2) {        if (in1 > in2) {
 2        a = in1 + in2;          a = in1 + in2;
 3        b = in1 * 3;            b = in1 * 3;
 4        c = in2 * 5;            c = in2 * 5;
 5      } else {                } else {
 6        a = in2 - in1;          a = in2 - in1;
 7        b = in1 * 5;            b = in1 * 5;
 8        c = in2 * 3;            c = in2 * 3;
 9      }                       }
10      x = a + c;              x = a;
11      y = b - c;              y = b - c;
12      x = x;                  x = x + c;
13      out = x + y;            out = x + y;
14      return out;             return out;
```

Die Eingabevariablen der beiden C-Programmen sind `in1` und `in2`. Die Ausgabevariable in beiden Programmen heißt `out`. Man beachte, dass das Programm 1 in Zeile 12 um die Anweisung `x = x` erweitert wurde, um eine zeilenweise Korrespondenz zwischen den Programmen zu erreichen.

Die beiden Programmabhängigkeitsgraphen sind in Abb. 7.6 zu sehen. Der erste Quelltext-Unterschied ist in Zeile 10 der beiden C-Programme. Entsprechend wird der Verifikationsbereich A in Abb. 7.6 als erstes betrachtet. Da für die Eingabevariablen des Verifikationsbereichs keine Äquivalenzen bekannt sind, kann auch nicht die Äquivalenz der Variablen `x` gezeigt werden.

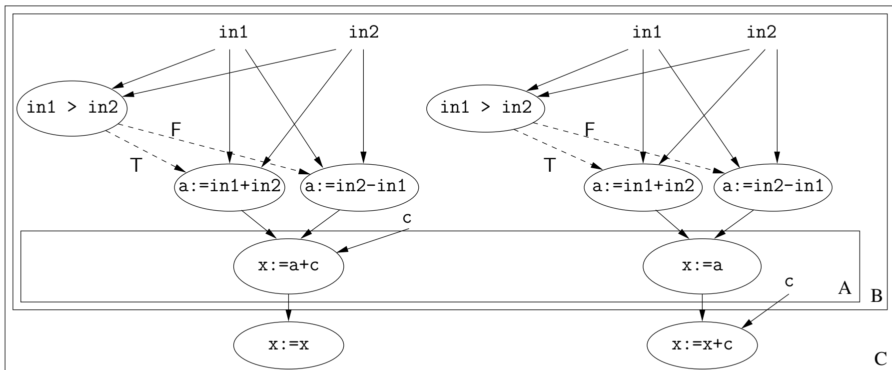


Abb. 7.6. Programmabhängigkeitsgraphen und Verifikationsbereiche [315]

Die entsprechenden Äquivalenzklassen lauten:

$$\begin{aligned}
E_1 &:= \{a_{0,1}\}\\
E_2 &:= \{a_{0,2}, x_{0,2}\}\\
E_3 &:= \{c_{0,1}\}\\
E_4 &:= \{x_{0,1}, a_{0,1} + c_{0,1}\}
\end{aligned}$$

Um den Verifikationsbereich zu erweitern, bietet sich ein Rückwärtserweiterung für die Variable `a` an. Da `a` in einem konditionalen Kontrollpfad berechnet wird, muss auch die Zuweisung im alternativen Pfad sowie das Prädikat selbst mit zur Erweiterung gehören. Somit ist der neue Verifikationsbereich der Bereich B. Die Eingabevariablen für Verifikationsbereich B sind `in1`, `in2` und `c`. Die Eingabevariable `in1` bzw. `in2` in Programm 1 ist äquivalent mit `in1` bzw. `in2` in Programm 2. Dennoch kann aufgrund der Unterschiede in der Zuweisung keine Äquivalenz der Programme für den Verifikationsbereich B gezeigt werden. Die entsprechenden Äquivalenzklassen lauten:

$$\begin{aligned}
E_1 &:= \{in1_{0,1}, in1_{0,2}\}\\
E_1 &:= \{in2_{0,1}, in2_{0,2}\}\\
E_1 &:= \{a_{0,1}, a_{0,2}, x_{0,2}\}\\
E_3 &:= \{c_{0,1}\}\\
E_4 &:= \{x_{0,1}, a_{0,1} + c_{0,1}\}
\end{aligned}$$

Es folgt schließlich eine Vorwärtserweiterung über die Datenabhängigkeit von `x`. Der neue Verifikationsbereich ist der Bereich C in Abb. 7.6. In diesem Fall wird auch die Variable `c` in Programm 2 berücksichtigt. Außerdem ist bekannt, dass die Variable `c` in Programm 1 äquivalent zu Variable `c` in Programm 2 ist. Somit kann auch die Äquivalenz der Variable `x` in Programm 1 und Programm 2 gezeigt werden. Damit ist gleichzeitig auch gezeigt, dass die beiden Programme äquivalent sind. Die abschließenden Äquivalenzklassen lauten:

$$\begin{aligned}
E_1 &:= \{in1_{0,1}, in1_{0,2}\}\\
E_1 &:= \{in2_{0,1}, in2_{0,2}\}\\
E_1 &:= \{a_{0,1}, a_{0,2}\}\\
E_3 &:= \{c_{0,1}, c_{0,2}\}\\
E_4 &:= \{x_{1,1}, x_{1,2}, a_{0,1} + c_{0,1}, a_{0,2} + c_{0,2}\}
\end{aligned}$$

Äquivalenzprüfung ohne Abrollen der Schleifen

Die bis jetzt vorgestellten Verfahren zur Äquivalenzprüfung für C-Programme setzen Einschränkungen voraus, die sich teilweise lockern lassen. In [395] wird beispielsweise eine Äquivalenzprüfung vorgestellt, welches das Abrollen der Schleifen nicht verlangt. Geprüft werden zwei Programme, die durch Transformationen auseinander

hervorgegangen sind. Dabei werden im Folgenden nur Transformationen betrachtet, die zum Ziel haben, die Speicherzugriffe zu minimieren. Im Wesentlichen lassen sich zwei Transformationen aus diesem Bereich unterscheiden:

- *Globale Schleifentransformationen* werden verwendet, um `for`-Schleifen umzuordnen und neu zu strukturieren, wodurch sich die temporale und räumliche Lokalität der Daten verbessert. Hierdurch werden Zugriffe auf den Hauptspeicher minimiert.
- *Globale Datenflusstransformationen* werden verwendet, um beispielsweise wiederholte Berechnungen zu entfernen oder einen Flaschenhals, der durch die Berechnung von Datenabhängigkeiten hervorgerufen wird, aufzulösen.

Die Transformationen basieren dabei auf dem Propagieren von Ausdrücken (engl. *expression propagation*), was temporäre Variablen erzeugt oder eliminiert. Daneben wurden globale algebraische Transformationen eingesetzt, die auf algebraischen Eigenschaften beruhen. Da diese Transformationen fehleranfällige Änderungen an den Berechnungen der Variablenindizes durchführen können, ist hier eine Unterstützung durch eine formale Äquivalenzprüfung besonders hilfreich.

Beispiel 7.1.10. Betrachtet wird das folgende C-Programm [396]:

```
 1            Programm 1
 2            void prog1(int a[], int b[], int c[]) {
 3              int k, tmp1[256], tmp2[256], tmp3[256];
 4
 5              for (k=0; k<256; k++)
 6      s1:       tmp1[k] = a[2*k] + f(B[k+1]);
 7              for (k=10; k<138; k++)
 8      s2:       tmp2[k] = b[k-8];
 9              for (k=10; k<266; k++) {
10                if (k >= 138)
11      s3:         tmp2[k] = b[k-8];
12      s4:       tmp3[k-10] = f(a[2*k - 19]) + tmp2[k];
13              }
14              for (k=255; k>=0; k--)
15      s5:       c[3*k] = tmp1[k] + tmp3[k];
16            }
```

Ferner ist das folgende C-Programm gegeben, welches durch Propagieren von Ausdrücken, Schleifen- und algebraischen Transformation aus Programm 1 erhalten wurde [396]:

```
 1            Programm 2
 2            void prog2(int a[], int b[], int c[]) {
 3              int k, tmp4[256], tmp5[256];
 4
 5              for (k=0; k<256; k++) {
 6      t1:       tmp4[k] = f(a[2*k+1]) + a[2*k];
 7      t2:       tmp5[k] = b[k+2] + tmp4[k];
```

```
 8    t3:     c[3*k] = f(b[k+1]) + tmp5[k];
 9          }
10        }
```

Vernachlässigt man potentielle Wertebereichsüberläufe, berechnen die beiden Funktionen die selben Ausgaben für die selben Eingaben, d. h. sie sind *Eingabe/Ausgabe-äquivalent*.

Es werden Programme mit den folgenden Eigenschaften betrachtet:

- *Dynamische Einmalzuweisung* (engl. *dynamic single assignment, DSA*): Das Programm ist in dynamischer Einmalzuweisung geschrieben. Hierbei wird jede Variable lediglich einmal geschrieben. Im Gegensatz zur *statischen Einmalzuweisung* (engl. *static single assignment, SSA*) [125], welche lediglich auf dem Umbenennen einzelner Variablen basiert, wird bei DSA auch der dynamische Kontrollfluss mit berücksichtigt, so dass auch in Schleifenrümpfen Variablen lediglich einmal geschrieben werden [158].
- *Stückweise affine Ausdrücke*: Indizes für Arrays innerhalb von Schleifenrümpfen sind stückweise affin. Auch sind die Operatoren `mod`, `div`, `min`, `max`, `floor` und `ceil` zugelassen. Hierdurch können die Relationen zwischen Arrayvariablen als affine Ungleichungen beschrieben werden.
- *Statischer Kontrollfluss*: Die Programme enthalten keine datenabhängigen Schleifen. Es wird angenommen, dass datenabhängige Schleifen durch Schleifen mit konstanten Grenzen und einer `if`-Anweisung im Schleifenrumpf zum datenabhängigen Abbruch der Schleife ersetzt werden.
- *Zeiger und Rekursionen*: Zeiger und Rekursionen sind in den Programmen nicht gestattet.

ADDGs

Array-Datenabhängigkeitsgraphen (engl. *Array Data Dependency Graph, ADDG*) werden im Folgenden verwendet, um die Datenabhängigkeiten in Programmen mit den obigen Eigenschaften zu erfassen. Skalare Variablen können als einelementige Arrays angesehen werden. Der Index einer zu schreibenden Variablen in einer Anweisung sowie die Indizes der dabei gelesenen Variablen können von Schleifenvariablen abhängen. Diese Abhängigkeiten können als Punkte in einem ganzzahligen mehrdimensionalen Raum angesehen werden. Man erhält somit eine geometrische Repräsentation.

Zur Definition eines ADDG wird von der folgenden Anweisung `s` ausgegangen:

$$\begin{aligned}\texttt{s}: \; & \texttt{v}[f_{i_1}([k_1,\ldots,k_d])],\ldots,[f_{i_n}([k_1,\ldots,k_d])] \\ & = \texttt{exp}(\ldots,\texttt{u}[f_{j_1}([k_1,\ldots,k_d])],\ldots,\texttt{u}[f_{j_m}([k_1,\ldots,k_d])],\ldots);\end{aligned}$$

Die Anweisung `s` wird für verschiedene Iterationen berechnet, wobei diese durch Elemente in dem sog. *Iterationsbereich* D gegeben sind. Der Iterationsbereich $D := \{k = [k_1,\ldots,k_d]\}$ definiert, für welche Belegungen von Schleifenvariablen $k_1,\ldots,k_d$ die Anweisung ausgeführt wird. Aus dem Iterationsbereich lässt sich der sog. *Definitionsbereich* $W_{\texttt{v}}$ einer Arrayvariablen `v` definieren:

Definition 7.1.2 (Definitionsbereich). *Der* Definitionsbereich $W_{\mathtt{v}}$ *einer Variable* `v` *in der Anweisung* `s` *mit Iterationsbereich D ist die Menge der Belegungen der Indizes* $[i_1,\ldots,i_n]$*, d. h.*

$$W_{\mathtt{v}} := \left\{[i_1,\ldots,i_n] \mid \left(\bigwedge_{r=1}^{n} i_r = f_{i_r}([k_1,\ldots,k_d])\right) \wedge [k_1,\ldots,k_d] \in D\right\}$$

Analog lässt sich der sog. *Operandenbereich* R_u einer in Anweisung `s` verwendeten Variablen `u` definieren.

Definition 7.1.3 (Operandenbereich). *Der* Operandenbereich $R_{\mathtt{u}}$ *einer Variablen* `u` *in der Anweisung* `s` *mit Iterationsbereich D ist die Menge der Belegungen der Indizes* $[j_1,\ldots,j_m]$*, d. h.*

$$R_{\mathtt{u}} := \left\{[j_1,\ldots,j_m] \mid \left(\bigwedge_{r=1}^{n} j_r = f_{j_r}([k_1,\ldots,k_d])\right) \wedge [k_1,\ldots,k_d] \in D\right\}$$

Schließlich kann mit Hilfe der Iterations-, Definitions- und Operandenbereiche eine sog. *Abhängigkeitsabbildung* $M_{\mathtt{v},\mathtt{u}}$ der Variablen `v` und `u` in Anweisung `s` definiert werden:

Definition 7.1.4 (Abhängigkeitsabbildung). *Für eine Anweisung* `s` *lässt sich eine* Abhängigkeitsabbildung $M_{v,u}$*, welche die Abhängigkeit der Variable* `v` *von der Variablen* `u` *beschreibt, wie folgt berechnen:*

$$M_{\mathtt{v},\mathtt{u}} := \left\{[i_1,\ldots,i_n] \to [j_1,\ldots,j_m] \mid \left(\bigwedge_{r=1}^{n} i_r = f_{i_r}([k_1,\ldots,k_d])\right)\right.$$
$$\left. \wedge \left(\bigwedge_{r=1}^{n} j_r = f_{j_r}([k_1,\ldots,k_d])\right) \wedge [k_1,\ldots,k_d] \in D\right\}$$

Beispiel 7.1.11. Für die Anweisung `s4` in Programm `Programm 1` aus dem Beispiel 7.1.10 ergibt sich der Iterationsbereich D zu:

$$D = \{[k] \mid 10 \leq k < 266 \wedge k \in \mathbb{Z}\}$$

Der Definitionsbereich für Variable `tmp3` und die Operandenbereiche für die Variablen `a` und `tmp2` ergeben sich zu:

$$\begin{aligned} W_{\mathtt{tmp3}} &= \{[d] \mid d = k - 10 \wedge k \in D\} \\ R_{\mathtt{a}} &= \{[d] \mid d = 2 \cdot k - 19 \wedge k \in D\} \\ R_{\mathtt{tmp2}} &= \{[d] \mid d = k \wedge k \in D\} \end{aligned}$$

Hieraus ergeben sich die Abhängigkeitsabbildungen $M_{\mathtt{tmp3,a}}$ und $M_{\mathtt{tmp3,tmp2}}$ zu:

$$\begin{aligned} M_{\mathtt{tmp3,a}} &= \{[d_1] \to [d_2] \mid d_1 = k - 10 \wedge d_2 = 2 \cdot k - 19 \wedge k \in D\} \\ M_{\mathtt{tmp3,tmp2}} &= \{[d_1] \to [d_2] \mid d_1 = k - 10 \wedge d_2 = k \wedge k \in D\} \end{aligned}$$

Eine Datenabhängigkeit zwischen Anweisung `s1` und `s2` besteht, wenn `s1` Variablenwerte schreibt, die in Anweisung `s2` gelesen werden, d. h., wenn `s1` den Definitionsbereich $W_{\mathtt{v}}$ besitzt und Anweisung `s2` den Operandenbereich $R_{\mathtt{v}}$ und gilt: $W_{\mathtt{v}} \cap R_{\mathtt{v}} \neq \emptyset$. Die Menge an Datenabhängigkeiten kann als Array-Datenabhängigkeitsgraph (ADDG) repräsentiert werden.

Definition 7.1.5 (ADDG). *Ein* ADDG *ist ein gerichteter Graph $G = (V, E)$, wobei die Knoten die Variablen und Operationen eines Programms repräsentieren. Die Kanten $e \in E$ stellen Datenabhängigkeiten dar. Kanten mit Operationen als Quellknoten werden mit der Position des jeweiligen Operanden (Senke) und Kanten mit Variablen als Quellknoten werden mit der jeweiliger Anweisung im Programm beschriftet.*

Während Datenabhängigkeitsgraphen in Compilern verwendet werden, um Datenabhängigkeiten von Operationen zu modellieren, werden in ADDGs die Abhängigkeiten von vielen Werten, die in einem Array gespeichert sind, modelliert. Die Datenabhängigkeitskanten in ADDGs sind dabei rückwärts gerichtet. Für die beiden Programme aus Beispiel 7.1.10, sind die ADDGs in Abb. 7.7 zu sehen. Man sieht, dass die Funktion `f` nicht interpretiert wird.

Eine Variable `v` heißt im Folgenden

- *interne Variable*, falls $\bigcup W_{\mathtt{v}} = \bigcup R_{\mathtt{v}}$ gilt, d. h. jeder produzierte Wert auch konsumiert wurde.
- *Eingabevariable*, falls $\bigcup W_{\mathtt{v}} = \emptyset$ ist, d. h. kein Element produziert wurde.
- *Ausgabevariable*, falls $\bigcup R_{\mathtt{v}} \subset W_{\mathtt{v}}$ gilt, d. h. nicht alle produzierten Werte auch konsumiert wurden.

Beispiel 7.1.12. Es wird wiederum das `Programm 1` aus Beispiel 7.1.10 betrachtet. Die Variablen `a` und `b` sind Eingabevariablen und Variable `c` ist eine Ausgabevariable. Die Variablen `tmp1`, `tmp2` und `tmp3` sind interne Variablen.

Neben der Abhängigkeitsabbildung aus Definition 7.1.4, die sich direkt aus den Anweisungen ablesen lässt, können Datenabhängigkeiten auch über mehrere Variablen hinweg als sog. *transitive Abhängigkeitsabbildung* definiert werden.

Definition 7.1.6 (Transitive Abhängigkeitsabbildung). *Sei $\langle v_0, v_1, \ldots, v_n \rangle$ ein Pfad im ADDG beginnend an Variablenknoten v_0 und endend in Variablenknoten v_n mit $n \geq 0$. Die* transitive Abhängigkeitsabbildung *$M^*_{v_0,v_n}$ ergibt sich zu:*

$$M^*_{v_0,v_n} := \begin{cases} I \text{ (Die Identität)} & \text{für } n = 0 \\ M_{v_0,v_1} & \text{für } n = 1 \\ M_{v_0,v_1} \circ M_{v_1,v_2} \circ \cdots \circ M_{v_{n-1},v_n} & \text{sonst} \end{cases}$$

Dabei ist $\circ$ die Konkatenation für zwei Relationen, d. h. $x \to z \in F \circ G \Leftrightarrow \exists y : x \to y \in F \wedge y \to z \in G$.

Die transitive Abhängigkeitsabbildung von einer Ausgabevariable zu einer Eingabevariable wird als *Ausgabe-zu-Eingabe-Abbildung* bezeichnet. Die Menge der Ausgabe-zu-Eingabe-Abbildungen definiert den Datenfluss des Programms.

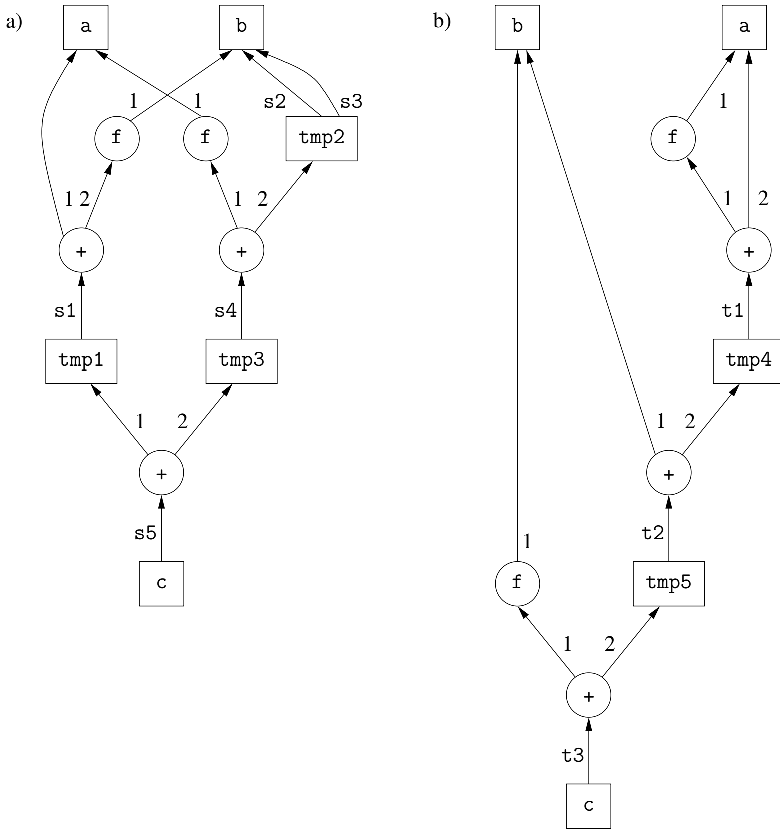


Abb. 7.7. ADDG für a) `Programm 1` und b) `Programm 2` aus Beispiel 7.1.10

Beispiel 7.1.13. Für `Programm 1` aus Beispiel 7.1.10 ergibt sich die Ausgabe-zu-Eingabe-Abbildung für die Variablen `c` und `b` unter ausschließlicher Berücksichtigung des rechten Pfades im ADDG in Abb. 7.7a) zu:

$$\begin{aligned} M^*_{\texttt{c},\texttt{b}} &= M_{\texttt{c},\texttt{tmp3}} \circ M_{\texttt{tmp3},\texttt{tmp2}} \circ M_{\texttt{tmp2},\texttt{b}} \\ &= \{[d_1] \rightarrow [d_2] \mid d_1 = 3 \cdot k \wedge d_2 = k + 2 \wedge 128 \leq k < 256 \wedge k \in \mathbb{Z}\} \end{aligned}$$

Äquivalenzprüfung

Zur Äquivalenzprüfung von zwei C-Programmen muss gezeigt werden, dass beide Programme die selben Ausgaben für die selben Eingaben erzeugen. Unter der Annahme, dass beide Programme die selben Eingabe- und Ausgabevariablen besitzen, kann die Prüfung in zwei Schritten durchgeführt werden. Dazu müssen die folgenden zwei Bedingungen auf ihre Gültigkeit überprüft werden:

1. Die Menge der Ausgabe-zu-Eingabe-Abbildungen ist die selbe für beide Programme.

2. Die Berechnungen der beiden Programme sind identisch.

Zusammen garantieren diese beiden Bedingungen, dass die Ausgaben beider Programme durch die selben Berechnungen auf den selben Eingabedaten entstehen. Da die ADDGs der beiden Programme Abstraktionen der Berechnungen der Programme darstellen, kann die Äquivalenzprüfung auch mit Hilfe der ADDGs erfolgen. Dabei werden die ADDGs der beiden Programme synchron von den Ausgabevariablen hin zu den Eingabevariablen traversiert. Die gesamte Äquivalenzprüfung von zwei C-Programmen ist nochmals in Abb. 7.8 zu sehen.

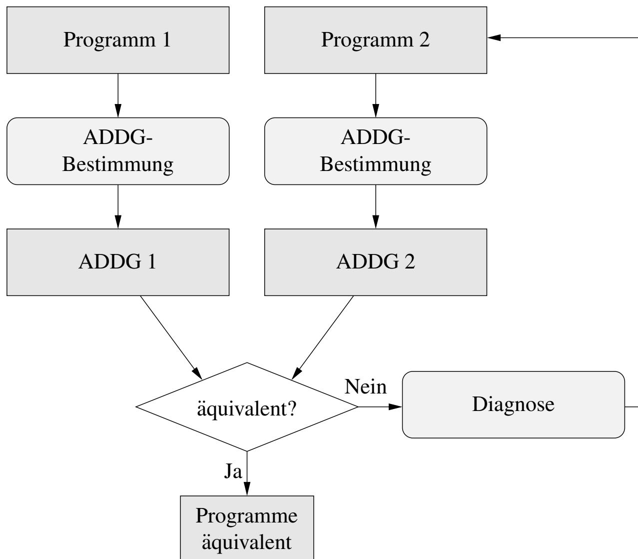


Abb. 7.8. Äquivalenzprüfung zweier C-Programme mittels ADDGs [395]

Zunächst wird die Traversierung der ADDGs für Programme gezeigt, die lediglich ein Propagieren von Ausdrücken und Schleifentransformationen zur Optimierung verwendet haben. Die synchrone Traversierung der ADDGs beginnt an korrespondierenden Knoten der ADDGs, welche die selben Ausgabevariablen repräsentieren. Die Abhängigkeitsabbildung wird für beide Knoten mit der Identitätsfunktion initialisiert. Anschließend werden die beiden ADDGs synchron von den Ausgabevariablen zu den Eingabevariablen traversiert. Wenn dabei in einem ADDG eine interne Variable entdeckt wird, so wird diese eliminiert, d. h. die transitive Abhängigkeitsabbildung der Ausgaben wird mit der Abhängigkeitsabbildung der internen Variablen konkateniert. Falls in einem ADDG ein Knoten erreicht wird, der eine Operation repräsentiert, so muss ein Knoten mit der gleichen Operation als nächstes auch im anderen ADDG erreicht werden. Dies kann allerdings auch erst nach einer Reihe weiterer interner Variablen erfolgen.

Jedes Mal, wenn sich ein Pfad in einem ADDG verzweigt, müssen die passenden Pfade der beiden ADDGs bestimmt werden. Entspringt die Verzweigung einem Operator-Knoten, so werden die Pfade mit der selben Beschriftung der beiden ADDGs synchron weiterverfolgt. Entspringt die Verzweigung allerdings einem Variablenknoten, erfolgt die Zuordnung anhand der bisher berechneten transitiven Abhängigkeitsabbildung.

Zu jedem Zeitpunkt der Traversierung ist sichergestellt, dass auf den beiden korrespondierenden Pfaden der ADDGs die selben Operationen berechnet wurden. Die zweite Bedingung für die Gültigkeit der Äquivalenzprüfung ist bereits erfüllt, sobald ein Pfad bei einer Eingabevariable endet. Was noch zu zeigen ist, ist dass die Ausgabe-zu-Eingabe-Abbildungen der beiden Programme identisch sind. Dies ist automatisch gezeigt, wenn erschöpfend für alle Pfade gezeigt wurde, dass die Ausgabe-zu-Eingabe-Abbildungen identisch sind. Damit ist bekannt, dass die selben Berechnungen auf den selben Eingabedaten zu den selben Ausgabedaten geführt haben. Somit sind die beiden Programme äquivalent.

Beispiel 7.1.14. Gegeben ist das folgende Schleifenprogramm [395]:

```
 1          Programm 1
 2          void prog1(int a[], int b[], int c[]) {
 3            int k, tmp[1024], buf[2048];
 4
 5            for (k=0; k<1024; k++)
 6      s1:     tmp[k] = b[2*k] + b[k];
 7            for (k=1024; k>=1; k--)
 8      s2:     buf[2*k-2] = a[2*k-2] + a[k-1];
 9            for (k=0; k<1024; k++)
10      s3:     c[k] = tmp[k] + buf[2*k];
11          }
```

Durch Propagieren von Ausdrücken erhält man `Programm 2`:

```
 1          Programm 2
 2          void prog2(int a[], int b[], int c[]) {
 3            int k, tmp[1024], buf[1024];
 4
 5            for (k=0; k<512; k++)
 6      t1:     tmp[k] = b[2*k] + b[k];
 7            for (k=0; k<1024; k++) {
 8      t2:     buf[k] = a[2*k] + a[k];
 9              if (k<512)
10      t3:       c[k] = tmp[k] + buf[k];
11              else
12      t4:       c[k] = b[2*k] + b[k] + buf[k];
13            }
14          }
```

Die resultierenden ADDGs der beiden Programme sind in Abb. 7.9 zu sehen. Im ADDG von `Programm 1` gibt es insgesamt vier Pfade. Dem gegenüber stehen acht

Pfade im ADDG von `Programm 2`, welche aus der Aufteilung der Zuweisungen an die Variable `c` auf zwei Anweisungen `t3` und `t4` folgen. Nummeriert man die Pfade von links nach rechts, so gilt, dass Pfad 1 in Abb. 7.9a) mit den Pfaden 1 und 5 in Abb. 7.9b) korrespondiert. Die transitive Abhängigkeitsabbildung ergibt sich zu:

$$M^*_{\mathtt{c},\mathtt{b}} = \{[k] \to [2k] \mid 0 \leq k < 512 \wedge k \in \mathbb{Z}\}$$

Weiterhin korrespondiert Pfad 2 in Abb. 7.9a) mit den Pfaden 2 und 6 in Abb. 7.9b), usw.

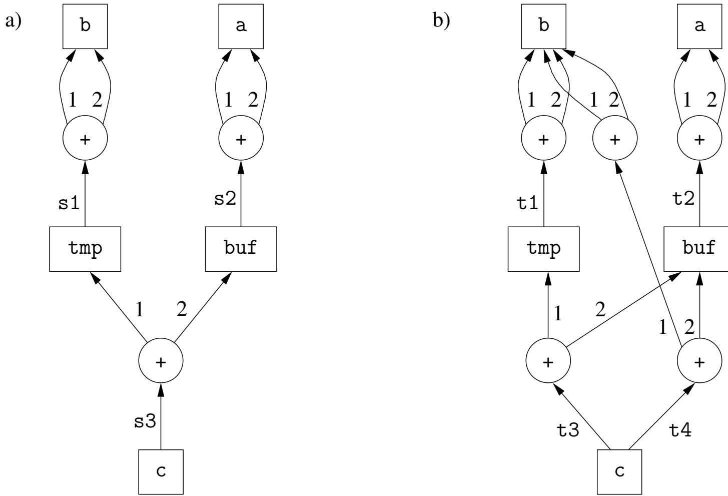


Abb. 7.9. ADDG von a) `Programm 1` und b) `Programm2` aus Beispiel 7.1.14 [395]

Äquivalenzprüfung mit algebraischen Transformationen

Im Folgenden werden C-Programme betrachtet, die zusätzlich zum Propagieren von Ausdrücken und Schleifentransformationen durch algebraische Transformationen entstanden sind. Algebraische Datenflusstransformationen nutzen Eigenschaften von Operationen und benutzerdefinierten Funktionen aus, um das Programm zu optimieren, ohne die Funktionalität zu verändern. Bei algebraischen Transformationen handelt es sich um globale Transformationen, da diese nicht auf einzelne Anweisungen beschränkt sind. Viele dieser Transformationen basieren auf der Assoziativität bzw. Kommutativität von Operationen.

Die Auswirkung von algebraischen Transformationen sind in Abb. 7.10 zu sehen. Wird eine kommutative Operation $\otimes$ berechnet, können die Operanden permutiert werden (Abb. 7.10a)). Abbildung 7.10b) zeigt die mögliche Gruppierung von assoziativen Operationen $\oplus$. Ist eine Operation $\odot$ kommutativ und assoziativ, kann die Berechnung durch jeden möglichen Baum aus der Operation gepaart mit jeder möglichen Permutation der Operanden berechnet werden (Abb. 7.10c)).

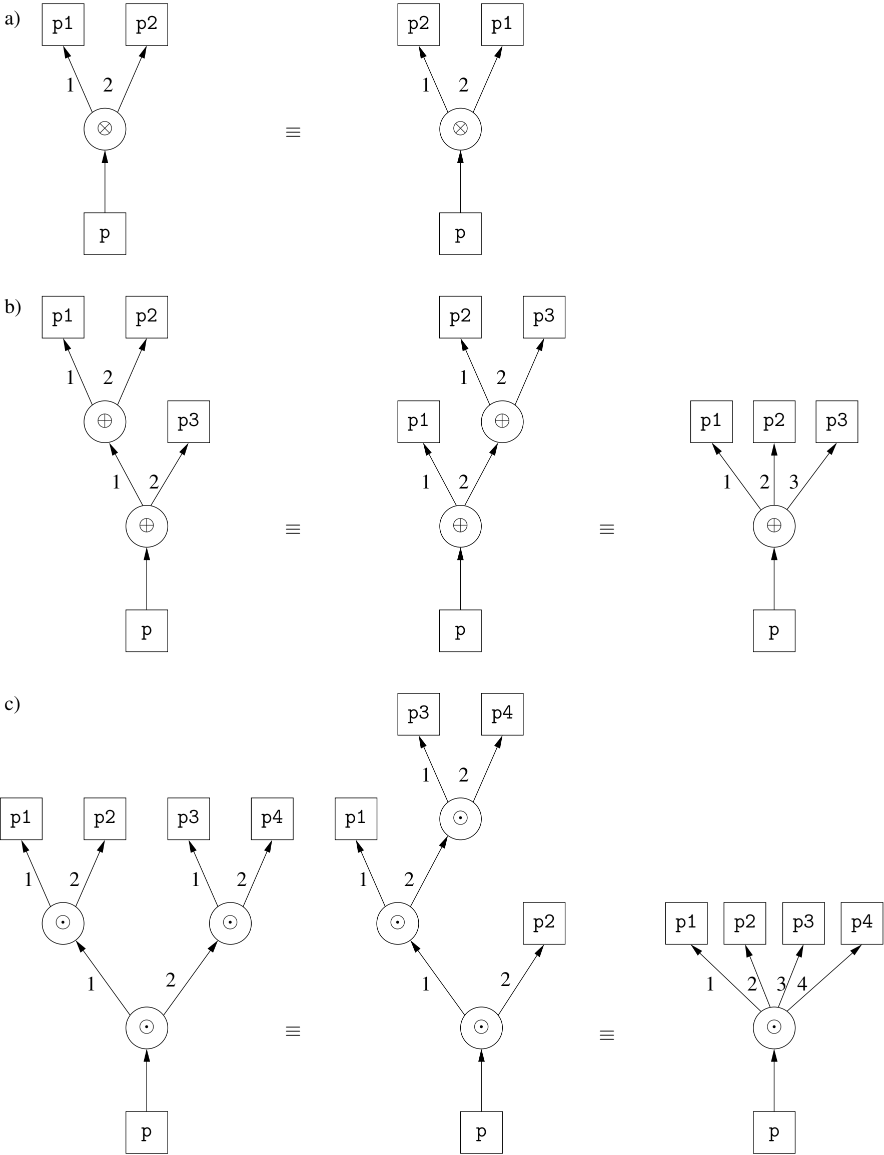


Abb. 7.10. Einfluss algebraischer Transformationen auf einen ADDG [395]

Beispiel 7.1.15. Betrachtet wird wiederum `Programm 1` aus Beispiel 7.1.14. Durch Anwendung algebraischer Transformationen erhält man `Programm 3` [395]:

```
1            Programm 3
2            void prog3(int a[], int b[], int c[]) {
3              int k, buf[2048];
4
5              for (k=0; k<1024; k++)
6      u1:       buf[k] = a[k] + b[k];
7              for (k=1024; k<=2046; k+=2) {
8      u2:       buf[k] = a[k] + b[k];
9              for (k=0; k<1024; k++)
10     u3:       c[k] = buf[k] + buf[2*k];
```

Abbildung 7.11 zeigt den zu `Programm 3` gehörigen ADDG.

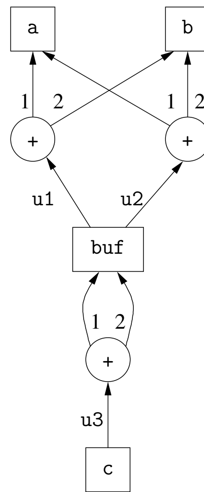


Abb. 7.11. ADDG von `Programm 3` aus Beispiel 7.1.15 [395]

Hier funktioniert der simple Traversierungsalgorithmus für die ADDGs aus Abb. 7.9a) und Abb. 7.11 nicht mehr. Um dennoch eine Traversierung und somit den Äquivalenzvergleich zu ermöglichen, müssen die algebraischen Transformationen berücksichtigt werden, d. h. jedes Mal, wenn ein Operator erreicht wird, der algebraische Transformationen zulässt, muss eine Normalform den ADDG bestimmt werden.

Eine solche Normalisierung kann durch die beiden Operationen *Entfaltung* und *Zuordnung* erfolgen. Wird ein assoziativer Operator erkannt, wird eine Entfaltung vorgenommen, wird hingegen ein kommutativer Operator erkannt, wird eine Zuordnung vorgenommen. Ist ein Operator assoziativ und kommutativ, wird die Entfaltung gefolgt von der Zuordnung durchgeführt.

Die beiden Operationen werden nun genauer beschrieben:

- Entfaltung: Wird bei der Traversierung ein assoziativer Operator $\oplus$ erkannt, wird eine Voraustraversierung durchgeführt, d. h. es wird eine geordnete Liste aus Nachfolgerknoten aufgebaut, so dass die Knoten entweder Eingabevariablen oder Operationen repräsentieren, die als erstes auf $\oplus$ folgen. Dadurch wird der ADDG derart reduziert, dass die internen Variablen zwischen $\oplus$ und den direkt folgenden Operationen eliminiert wurden.
- Zuordnung: Wird ein kommutativer Operator $\otimes$ in beiden ADDGs erreicht, kann die Beschriftung der ausgehenden Kanten nicht zur Zuordnung verwendet werden, da jede mögliche Permutation der Operanden gültig ist. Für die Zuordnung wird in jedem ADDG für jeden Teil-ADDG, der Operator $\otimes$ als Quellknoten besitzt, eine Traversierung durchgeführt, bis entweder die Eingabevariablen oder Folgeoperationen erreicht wurden. Interne Variablen werden dabei eliminiert. Auf diese Art entsteht für jeden ADDG eine Liste von Nachfolgerknoten. Sind diese eindeutig, können sie paarweise eins-zu-eins zugeordnet werden. Handelt es sich bei mehrdeutigen Knoten um Eingabevariablen, erfolgt die Zuordnung über Ausgabe-zu-Eingabe-Abbildungen. Handelt es sich um Operationsknoten, wird eine Voraustraversierung durchgeführt.

Beispiel 7.1.16. Bei dem Vergleich von `Programm 1` und `Programm 3` aus den Beispielen 7.1.14 und 7.1.15 wird zuerst die Addition in den beiden ADDGs erkannt. Aus diesem Grund muss eine Entfaltung vorgenommen werden. Die beiden resultierenden ADDGs sind in Abb. 7.12 zu sehen.

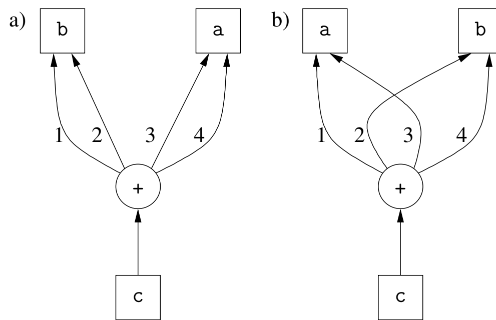


Abb. 7.12. ADDG von a) `Programm 1` aus Beispiel 7.1.14 und b) `Programm 3` aus Beispiel 7.1.15 nach der Entfaltung [395]

Die anschließende Zuordnung muss mit nicht eindeutigen Eingabevariablen umgehen. Dies bedeutet, dass für jeden Pfad die Ausgabe-zu-Eingabe-Abbildung berechnet werden muss. Diese ergeben sich wie folgt:

$$
\begin{array}{lll}
\texttt{Prog1} & & \texttt{Prog3} \\
M^{*(1)}_{\mathtt{c,b}} & \Leftrightarrow & M^{*(4)}_{\mathtt{c,b}} = \{[k] \rightarrow [2k] \mid k \in D\} \\
M^{*(2)}_{\mathtt{c,b}} & \Leftrightarrow & M^{*(2)}_{\mathtt{c,b}} = \{[k] \rightarrow [k] \mid k \in D\} \\
M^{*(3)}_{\mathtt{c,a}} & \Leftrightarrow & M^{*(3)}_{\mathtt{c,a}} = \{[k] \rightarrow [2k] \mid k \in D\} \\
M^{*(4)}_{\mathtt{c,a}} & \Leftrightarrow & M^{*(1)}_{\mathtt{c,a}} = \{[k] \rightarrow [k] \mid k \in D\}
\end{array}
$$

Der Iterationsbereich ist gegeben als $D = \{[k] \mid 0 \leq k \leq 1024 \wedge k \in \mathbb{Z}\}$. Weiterhin ist an den transitiven Abhängigkeitsabbildungen in Klammern immer der jeweilige Pfad angegeben.

7.2 Testfallgenerierung zur simulativen Eigenschaftsprüfung

Für allgemeine Programme ist das Problem der Äquivalenzprüfung nicht entscheidbar. Per Simulation kann jedoch ein Programm mit einem Referenzprogramm verglichen werden. Dabei ist das Ziel nicht, die Äquivalenz der Programme zu zeigen, sondern deren unterschiedliches Verhalten aufzudecken. Dies kann, wie in Abschnitt 4.2 beschrieben, mit diversifizierenden Methoden, z. B. auf Basis des Pfadbereichstest erfolgen.

In diesem Abschnitt werden nun weitere Verfahren diskutiert, die zur Generierung von Testfällen verwendet werden können. Die Testfälle lassen sich über die Äquivalenzprüfung hinaus zur funktionalen Eigenschaftsprüfung von Programmen einsetzen. Bei der Testfallgenerierung werden *funktionsorientierte*, *kontrollflussorientierte* und *datenflussorientierte* Testfälle unterschieden. Die beiden letzteren sind Spezialformen der *strukturorientierten Testfälle*. Die Verfahren zur Testfallgenerierung werden im Folgenden zusammen mit den zugehörigen *Überdeckungsmaßen* eingeführt, die es erlauben, den Fortschritt der simulativen Verifikation zu messen. Die Darstellung erfolgt dabei in Anlehnung an [305].

7.2.1 Funktionsorientierte Testfälle

Grundlage für den funktionsorientierten Test bildet die Spezifikation. Diese wird herangezogen, um Testfälle zu entwickeln, und um zu überprüfen, ob die Verifikation vollständig ist, d. h. alle Funktionen auf deren korrekte Implementierung geprüft sind. Um dies quantitativ zu messen, können beispielsweise funktionale Eigenschaften des Systems als *Zusicherungen* formuliert werden. Bei der anschließenden Simulation mit den generierten Testfalleingaben wird dann gezählt, ob jede Zusicherung mindestens einmal simuliert und dabei erfüllt wurde. In diesem Zusammenhang spricht man auch davon, dass eine Zusicherung *überdeckt* wurde. Der prozentuale Anteil aller überdeckten Zusicherungen wird als *Zusicherungsüberdeckung* bezeichnet.

Oftmals liegen die funktionalen Anforderungen allerdings nicht als formale Zusicherungen vor. In diesem Fall können andere funktionsorientierte Testfälle generiert werden.

Funktionsorientierte Äquivalenzklassenbildung

Die obige Darstellung zur Zusicherungsüberdeckung ist sehr vereinfacht dargestellt. In realen Implementierungen werden komplexe Datentypen verwendet, die eine Überprüfung für jede mögliche Variablenbelegung unmöglich machen. Aus diesem Grund muss eine Auswahl geeigneter Testfälle vorgenommen werden, die repräsentativ für alle möglichen Variablenbelegungen ist. Neben dem Kriterium, dass diese Testfälle gute Repräsentanten aller möglichen Testfälle darstellen, sollten diese darüber hinaus so gestaltet sein, dass zu erwartende Fehler zuverlässig aufgedeckt werden. Ein ähnliches Vorgehen wurde bereits bei der Testfallgenerierung für kombinatorische Schaltungen vorgestellt (siehe Abschnitt 6.1.2), bei denen ein zugrundeliegendes Fehlermodell angenommen wurde.

Ein Verfahren zur Testfallgenerierung nach dem Prinzip „Teile und Herrsche" ist die sog. *funktionsorientierte Äquivalenzklassenbildung* (siehe auch Abschnitt 4.2.1). Die funktionsorientierte Äquivalenzklassenbildung versucht, durch wiederholte Fallunterscheidungen für Eingabe- und Ausgabebedingungen die Komplexität zu reduzieren. Werte aus den resultierenden Äquivalenzklassen sollten dann zu einem äquivalenten Verhalten der Implementierung führen.

Zur Bildung von Äquivalenzklassen unterscheidet man *gültige* und *ungültige Äquivalenzklassen*. Gültige Äquivalenzklassen kennzeichnen Werte, die laut Spezifikation zugelassen sind. Ungültige Äquivalenzklassen repräsentieren dementsprechend Werte, die laut Spezifikation nicht auftreten dürfen. Wird das System mit Werten aus ungültigen Äquivalenzklassen stimuliert, so liegt ein Fehlerfall vor und das System muss mit einer entsprechenden Fehlerbehandlung reagieren. Erfolgt die Ermittlung der Äquivalenzklassen anhand der Ausgabewerte, müssen anschließend die Testfälle so konstruiert werden, dass die Stimulation des Systems mit den zugehörigen Testfalleingaben zur Ausgabe in den entsprechenden Äquivalenzklassen führt.

Die Auswahl der eigentlichen Testfälle kann nach Bildung der Äquivalenzklassen bezüglich unterschiedlicher Kriterien erfolgen. Ein übliches Vorgehen basiert auf der Grenzwertanalyse, bei der Werte direkt auf oder neben einer Grenze von Äquivalenzklassen getestet werden. Die Idee hierbei ist, dass Fehler besonders häufig in der Nähe von Grenzen auftreten (siehe auch Abschnitt 4.2.1). Andere Möglichkeiten zur Auswahl der Testfälle können beispielsweise auch zufallsbasiert sein.

Ursache-Wirkungs-Analyse

Die unabhängige Betrachtung von Eingabebereichen bzw. Ausgabebereichen ist in komplexen Systemen häufig nicht ausreichend. Vielmehr stehen Äquivalenzklassen typischerweise in Wechselwirkung miteinander. Dies bedeutet, dass Fehler häufig nicht durch alleinige Betrachtung einer einzelnen Eingabe erzeugt werden können, sondern bestimmte Wertekombinationen über alle Eingänge zu einem Fehlverhalten führen. Allerdings ist es in der Praxis äußerst schwierig diese Wechselwirkungen festzustellen.

Ein systematisches Verfahren zur Erzeugung von funktionsorientierten Testfällen unter Berücksichtigung von Wechselwirkungen ist unter dem Namen *Ursache-Wirkungs-Analyse* (engl. *cause-effect graphing*) bekannt geworden. Dabei werden zu-

nächst die Beziehungen zwischen Äquivalenzklassen graphisch formuliert und anschließend in einem tabellarischen Verfahren geeignete Testfälle ausgewählt. Die Ursache-Wirkungs-Analyse verläuft in vier Schritten [305]:

1. In einem anfänglichen Schritt wird die Spezifikation in handhabbare Teile zerlegt, da die Betrachtung von Wertekombinationen den Suchraum explodieren lässt.
2. Für jede Teilspezifikation werden Ursachen und Wirkungen identifiziert. Dabei stellen Ursachen und Wirkungen Prädikate dar und besitzen somit den Wert T oder F. Ursachen sind dabei einzelne Eingabebedingungen oder eine Äquivalenzklasse, z. B. die Variable $x \geq 10$. Eine Wirkung ist eine Ausgabebedingung.
3. Erstellung eines *Ursache-Wirkungs-Graphen*, der die logischen Beziehungen von Ursachen und Wirkungen darstellt. Es handelt sich hierbei im Wesentlichen um ein Boolesches Netzwerk, das durch Transformation der Spezifikation gewonnen wird. Es werden dabei aber auch Abhängigkeiten zwischen den Ursachen und/oder Wirkungen, die sich aus Umgebungsbedingungen ergeben, berücksichtigt.
4. Der Ursache-Wirkungs-Graph wird in eine sog. *Entscheidungstabelle* umgeformt und für jede Spalte der Tabelle eine Testfalleingabe erzeugt.

Der Ursache-Wirkungs-Graph kann als Boolesches Netzwerk betrachtet und repräsentiert werden. Es wird hier dennoch die in [305] verwendete, kompakte Repräsentation verwendet. In Ursache-Wirkungs-Graphen stehen Ursachen links und Wirkungen rechts. Die wesentlichen Beziehungen zwischen Ursachen und/oder Wirkungen sind in Abb. 7.13 dargestellt.

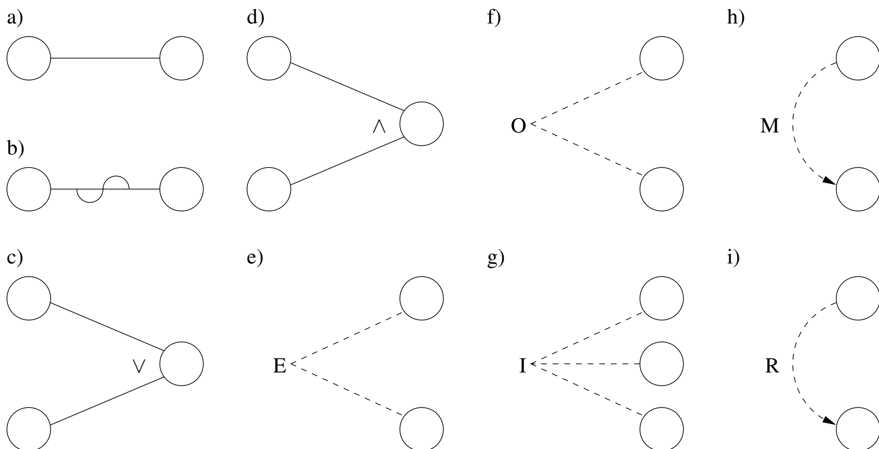


Abb. 7.13. Beziehungen zwischen Ursachen und/oder Wirkungen [305]

Falls eine Ursache und eine Wirkung stets den selben Wert besitzen, so besteht zwischen Ursache und Wirkung eine Identität (siehe Abb. 7.13a)). Falls Ursache und Wirkung jedoch stets invertierte Werte zueinander besitzen, so ist deren Relation eine Negation (Abb. 7.13b)). Eine Disjunktion (Konjunktion) liegt vor, wenn eine Wirkung eintritt, sofern eine (alle) der verbundenen Ursachen eintreten (Abb. 7.13c) und d)). Eine exklusive Beziehung zwischen zwei Ursachen besteht, wenn die Anwesenheit einer Ursache die Anwesenheit der anderen Ursache ausschließt (Abb. 7.13e)). Die O-Abhängigkeit (engl. *one, and only one*) ist gegeben, wenn immer exakt eine Ursache erfüllt ist (Abb. 7.13f)). Abbildung 7.13g) zeigt eine inklusive Beziehung zwischen Ursachen. Dies bedeutet, dass mindestens immer eine Ursache erfüllt ist. Die R- und M-Abhängigkeit (engl. *requires* und *masks*), dargestellt in Abb. 7.13h) und i), liegen vor, falls die Anwesenheit einer Ursache die Anwesenheit der anderen Ursache voraussetzt, bzw. die Anwesenheit der Ursache durch die Anwesenheit der anderen Ursache maskiert wird. Das folgende Beispiel stammt aus [305].

Beispiel 7.2.1. Betrachtet wird ein Programm, das Zeichen von einer Tastatur solange einliest, wie große Vokale und große Konsonanten eingegeben werden oder ein Zähler kleiner ist als der Maximalwert vom Typ `int`. Der Zähler wird beim Lesen eines großen Vokals oder Konsonanten inkrementiert. Er zählt somit die gelesenen Zeichen. Im Falle, dass das gelesene Zeichen ein großer Vokal ist, wird ein zweiter Zähler für die Vokale ebenfalls inkrementiert. Ist das gelesene Zeichen weder ein großer Vokal noch ein großer Konsonant, so wird das Programm beendet. Das Programm sieht wie folgt aus:

```
 1     Programm
 2       char c;
 3       int count = 0;
 4       int voc_count = 0;
 5       std::cin >> c;
 6       while((c >= 'A') && (c <= 'Z')
 7              && (count < MAX_INT)) {
 8         count++;
 9         if ((c == 'A') || (c == 'E') || (c == 'I')
10              || (c == 'O') || (c == 'U'))
11           voc_count++;
12         std::cin >> c;
13       }
```

Die Ursachen in diesem Programm lauten:

- u_1: das gelesene Zeichen ist ein großer Konsonant
- u_2: das gelesene Zeichen ist ein großer Vokal
- u_3: der Zähler für die gelesenen Zeichen ist kleiner dem maximalen Wert vom Typ `int`

Die Wirkungen in diesem Programm lauten:

- w_1: der Zähler für die gelesenen Zeichen wird inkrementiert
- w_2: der Zähler für die gelesenen Vokale wird inkrementiert

- w_3: das Zeichen wird gelesen
- w_4: das Programm wird beendet

Der Ursache-Wirkungs-Graph für dieses Programm ist in Abb. 7.14 zu sehen.

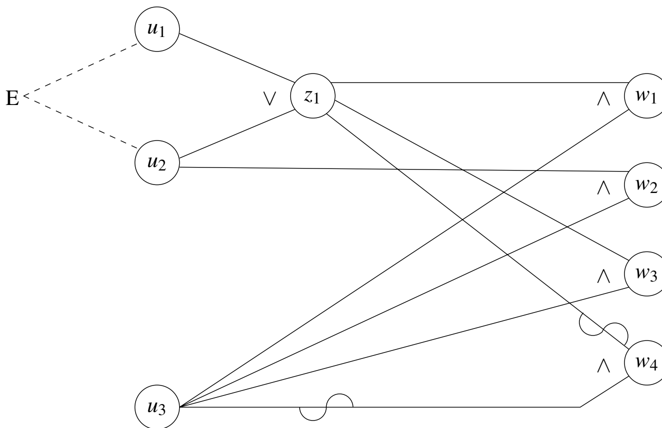


Abb. 7.14. Ursache-Wirkungs-Graph für das Programm aus Beispiel 7.2.1 [305]

Man erkennt, dass sich die Ursachen u_1 und u_2 gegenseitig ausschließen. Durch die Disjunktion beider Ursachen wird eine Zwischenursache z_1 generiert, die den Wert T annimmt, sobald ein großer Konsonant oder großer Vokal eingegeben wurde. Weiterhin sieht man, dass die Wirkungen w_1 und w_3 von exakt den selben Ursachen abhängen. Die Wirkung w_4 tritt immer genau dann ein, wenn w_1 bzw. w_3 nicht auftreten und umgekehrt. Aus diesem Grund ist es ausreichend, Testfalleingaben für die Wirkungen w_1 und w_2 zu erzeugen. Diese Testfälle überprüfen automatisch auch die Wirkungen w_3 und w_4.

Nach Erstellung des Ursache-Wirkungs-Graphen erfolgt die Umwandlung in eine *Entscheidungstabelle*. Dies geschieht durch die folgenden vier Schritte [305]:

1. Auswahl einer Wirkung im Ursache-Wirkungs-Graphen.
2. Identifikation von Kombinationen von Ursachen, welche die Wirkung erzeugen bzw. nicht erzeugen.
3. Für jede gefundene Kombination von Ursachen wird eine Spalte in der Entscheidungstabelle erstellt, die ebenfalls alle Werte der übrigen Wirkungen enthält.
4. Entfernung mehrfach vorkommender Einträge aus der Entscheidungstabelle

Die Identifikation der Kombinationen von Ursachen erfolgt durch Bestimmung möglicher Eingabewerte anhand der Ausgabewerte von (Zwischen-)Ursachen. Besitzt beispielsweise der Ausgang einer Konjunktion den Wert T, so müssen alle Eingänge auf den Wert T gesetzt werden. Analog müssen bei einer Disjunktion mit Ausgabe F alle Eingänge auf F gesetzt werden. Komplizierter ist es, wenn der

Ausgang der Disjunktion den Wert T trägt. In diesem Fall müssen alle Eingaben berücksichtigt werden, wobei genau ein Eingang den Wert T trägt und alle anderen Eingänge den Wert F. Durch diese Regel werden Fehlermaskierungen vermieden. Schließlich müssen bei einer Konjunktion mit Ausgabe F alle Eingaben berücksichtigt werden, bei denen genau ein Eingang auf F gesetzt ist und alle anderen Eingänge auf T. Durch diese Regel sollen wiederum Fehlermaskierungen vermieden werden.

Beispiel 7.2.2. Betrachtet wird das Programm aus Beispiel 7.2.1. Der Ursache-Wirkungs-Graph ist in Abb. 7.14 zu sehen. Man kann sehen, dass die Wirkungen w_1, w_3 und w_4 gemeinsam betrachtet werden können. Repräsentativ hierfür wird die Wirkung w_1 betrachtet. Die Wirkung w_1 tritt ein, wenn die Ursache u_3 und die Zwischenursache z_1 auftreten. Damit die Zwischenursache z_1 auftritt, muss Ursache u_1 oder Ursache u_2 erfüllt sein. Aufgrund der oben genannten Regel (aber auch aufgrund der Exklusivität der beiden Ursachen) sind die zu betrachtenden Belegungen $(u_1 = \mathsf{T}) \wedge (u_2 = \mathsf{F})$ sowie $(u_1 = \mathsf{F}) \wedge (u_2 = \mathsf{T})$.

Das Nichteintreten von Wirkung w_1 resultiert zunächst in den Belegungen $(z_1 = \mathsf{T}) \wedge (u_3 = \mathsf{F})$ sowie $(z_1 = \mathsf{F}) \wedge (u_3 = \mathsf{T})$. Aus $z_1 = \mathsf{F}$ folgt direkt, dass die beiden Ursachen u_1 und u_2 in diesem Testfall nicht erfüllt sein dürfen. Für den Fall $z_1 = \mathsf{T}$ ergeben sich wiederum die beiden Kombinationen $(u_1 = \mathsf{T}) \wedge (u_2 = \mathsf{F})$ sowie $(u_1 = \mathsf{F}) \wedge (u_2 = \mathsf{T})$. Nach den selben Regeln können nun noch die Testfalleingaben für $w_2 = \mathsf{F}$ und $w_2 = \mathsf{T}$ erzeugt werden. Die resultierende Entscheidungstabelle ist in Tabelle 7.1 dargestellt.

Tabelle 7.1. Entscheidungstabelle für das Programm aus Beispiel 7.2.1 [305]

		Testfälle					Kommentar
		1	2	3	4	5	
	u_1	T	F	F	F	F	großer Konsonant
Ursache	u_2	F	T	F	T	F	großer Vokal
	u_3	T	T	T	F	F	< MAX_INT
	z_1	T	T	F	T	T	
	w_1	T	T	F	F	F	Zähler wird inkrementiert
	w_2	F	T	F	F	F	Vokalzähler wird inkrementiert
Wirkung	w_3	T	T	F	F	F	Zeichen wird gelesen
	w_4	F	F	T	T	T	Programm wird beendet

Zustandsorientierte Testfälle

Die beschriebene Äquivalenzklassenbildung arbeitet alleinig auf den Wertebereichen der Ein- oder Ausgaben eines Systems. Wenn unter Verwendung von Äquivalenzklassen für Ausgabewertebereiche Testfalleingaben generiert werden sollen, stellt sich allerdings die Frage, ob das System zustandsbehaftet ist, oder nicht. Spezielle

Verfahren zur Erstellung funktionsorientierter Testfälle, die auf Verhaltensspezifikationen als endliche Automaten basieren, werden als Verfahren zur Erstellung *zustandsorientierter Testfälle* bezeichnet. Da der endliche Automat das Verhalten (die Funktion) des Systems spezifiziert, handelt es sich hierbei um funktionsorientierte Testfälle.

Die einfachste Möglichkeit zustandsorientierte Testfälle zu erstellen, besteht darin, Eingaben so zu wählen, dass jeder Zustand mindestens einmal besucht wird. Es wird somit eine 100%-ige *Zustandsüberdeckung* gefordert. Ein Auswahlkriterium für zustandsbasierte Testfälle, das zu einer höheren Verifikationsvollständigkeit führt, ist die Forderung, jeden Zustandsübergang mindestens einmal zu gehen. Das Ziel ist somit eine 100%-ige *Übergangsüberdeckung* zu erreichen. Man beachte, dass 100 Prozent Übergangsüberdeckung 100% Zustandsüberdeckung implizieren. Übergangsbedingungen an Zustandsübergängen können aus disjunktiv verknüpften Ereignissen bestehen. Da eine 100%-ige Übergangsüberdeckung nicht garantiert, dass jedes Ereignis einmal getestet wurde, kann ein weiteres Auswahlkriterium für Testfälle sein, eine möglichst hohe *Ereignisüberdeckung* zu erreichen. Man beachte, dass wiederum eine 100%-ige Ereignisüberdeckung eine 100%-ige Übergangsüberdeckung und somit eine 100%-ige Zustandsüberdeckung impliziert. Diese drei Überdeckungsarten für zustandsorientierte Testfälle bilden somit eine Hierarchie, die in Abb. 7.15 zu sehen ist. Die Überdeckungsmaße für zustandsorientierte Testfälle wird anhand eines Beispiels aus [305] verdeutlicht.

Abb. 7.15. Überdeckungshierarchie für zustandsorientierte Testfälle [305]

Beispiel 7.2.3. Es wird der Verbindungsaufbau und -abbau beim Telefonieren betrachtet. Der endliche Zustandsautomat ist in Abb. 7.16 dargestellt. Die einzelnen Zustände haben darin die folgende Bedeutung:

Zustand	Bedeutung
s_0	Verbindung ist getrennt
s_1	Wählend
s_2	Verbindung hergestellt
s_3	Ungültige Nummer gewählt
s_4	Timeout aufgetreten

Der Anfangszustand ist der Zustand s_0. Sobald der Wählvorgang durch Abheben (Ereignis e_{ab}) begonnen wurde, dieser aber noch nicht beendet wurde, befindet sich

das Telefon im Zustand s_1. In diesem Zustand kann der Benutzer Ziffern wählen (Ereignisse e_0 bis e_9) und diese werden an die Rufnummer angehängt (e_z) und auf Plausibilität geprüft (e_p). Wurde eine gültige Rufnummer gewählt (e_{rg}), so wird die Verbindung aufgebaut (e_{vauf}). Dies wird durch den Wechsel in Zustand s_2 dargestellt. Ist die gewählte Rufnummer hingegen ungültig (e_{ru}), so erfolgt der Übergang in Zustand s_3. Das Auflegen (e_{auf}) führt in jedem Zustand zum vollständigen Abbruch des Telefonats und dem Zurücksetzen des Automaten (Zustand s_0 und Ereignis e_r). Im Zustand s_2 führt es zusätzlich dazu, dass die Verbindung abgebaut wird (e_{vab}). Schließlich kann es noch in den Zustanden s_1 und s_3 zu der Situation kommen, dass der Benutzer keinerlei Eingaben mehr tätigt. Ein Timeout e_{to} wird verwendet, um das Telefonat zu beenden. Hierzu wird der Zwischenzustand s_4 betreten.

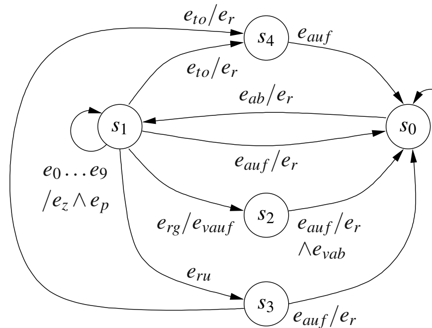


Abb. 7.16. Endlicher Zustandsautomat für den Verbindungsaufbau und - abbau

An diesem Beispiel kann man erkennen, dass eine 100%-ige Zustandsüberdeckung keine 100%-ige Übergangsüberdeckung garantiert. Durch die beiden Testfalleingaben $\langle e_{ab}, e_{rg} \rangle$ und $\langle e_{ab}, e_{ru}, e_{to} \rangle$ werden zwar alle Zustände besucht, aber lediglich eine Übergangsüberdeckung von $\frac{4}{10} = 40\%$ erreicht. Eine Übergangsüberdeckung von 100% könnte beispielsweise durch die folgenden sechs Testfalleingaben erzielt werden:

$$\langle e_{ab}, e_{auf} \rangle$$
$$\langle e_{ab}, e_{to}, e_{auf} \rangle$$
$$\langle e_{ab}, e_{to}, e_{auf} \rangle$$
$$\langle e_{ab}, e_0, e_{rg}, e_{auf} \rangle$$
$$\langle e_{ab}, e_8, e_{ru}, e_{auf} \rangle$$
$$\langle e_{ab}, e_8, e_{ru}, e_{to}, e_{auf} \rangle$$

Hierdurch ist allerdings noch keine 100%-ige Ereignisüberdeckung erzielt worden. Um dies zu erreichen, müsste im Zustand s_1 jede Ziffer mindestens einmal gewählt werden und somit den Zustandsübergang zurück zu s_1 auslösen.

Der in Abb. 7.16 dargestellte endliche Automat ist nicht vollständig spezifiziert, d. h. nicht alle Reaktionen auf jedes Ereignis in jedem Zustand wurden spezifiziert. Der Automat beschreibt somit lediglich das erwartete Verhalten des Telefons. Dies bedeutet aber auch, dass die zustandsorientierten Testfälle lediglich die Normalfälle überprüfen. Fehlersituationen werden ignoriert.

Zur Erzeugung zustandsorientierter Testfälle für den Fehlerfall bietet es sich an, den endlichen Automaten als *Automatentabelle* zu repräsentieren. In einer Automatentabelle wird mit jeder Zeile ein Zustand und mit jeder Spalte ein Ereignis assoziiert. Bei den Ereignissen kann es sich dabei auch um zusammengesetzte Ereignisse handeln. In den Feldern der Tabelle, die Kreuzungspunkte zwischen Zuständen und Ereignissen darstellen, werden der Folgezustand und die Ausgabe eingetragen, wenn in dem gegeben Zustand das angegebene Ereignis auftritt. Ist der zugrundeliegende endliche Automat nicht vollständig spezifiziert, ist diese Tabelle ebenfalls unvollständig, da Felder leer bleiben.

Um eine vollständige Ereignisüberdeckung auch für Fehlerfälle zu erreichen, ist es notwendig, jedes Tabellenfeld in einem Testfall zu berücksichtigen. Man kann dabei annehmen, dass das Auftreten nicht spezifizierter Ereignisse in einem Zustand zu einem Verhalten führt, in dem der endliche Automat in dem selben Zustand verbleibt und keine Ausgabe erzeugt. Unter dieser Annahme können die leeren Tabellenfelder gefüllt und somit ein vollständig spezifizierter endlicher Automat geschaffen werden. Werden nun Testfalleingaben so erzeugt, dass jedes Tabellenfeld berücksichtigt wird, so kann man eine 100%-ige Ereignisüberdeckung für die Normal- und Fehlerfälle erzielen.

Beispiel 7.2.4. Betrachtet wird der Zustandsautomat aus Abb. 7.16 aus Beispiel 7.2.3. Die vollständige Automatentabelle ist in Tabelle 7.2 gezeigt. Die Einträge in den Tabellenfelder sind von der Form *Folgezustand*/*Ausgabe*.

Tabelle 7.2. Vollständige Automatentabelle zum endlichen Automaten aus Abb. 7.16 [305]

	e_{ab}	e_{auf}	e_0	...	e_9	e_{to}	e_{rg}	e_{ru}
s_0	s_1/e_r	$s_0/$	$s_0/$		$s_0/$	$s_0/$	$s_0/$	$s_0/$
s_1	$s_1/$	s_0/e_r	$s_1/e_z \wedge e_p$		$s_1/e_z \wedge e_p$	s_4/e_r	s_2/e_{vauf}	$s_3/$
s_2	$s_2/$	$s_0/e_r \wedge e_{vab}$	$s_2/$		$s_2/$	$s_2/$	$s_2/$	$s_2/$
s_3	$s_3/$	s_0/e_r	$s_3/$		$s_3/$	s_4/e_r	$s_3/$	$s_3/$
s_4	$s_4/$	$s_0/$	$s_4/$		$s_4/$	$s_4/$	$s_4/$	$s_4/$

Im Allgemeinen wird man bei der Generierung zustandsbasierter Testfälle drei Arten von Ereignissen unterscheiden [305]:

1. Ereignisse, die, falls sie in einem Zustand auftreten, einen Zustandswechsel und/oder eine Ausgabe bewirken,
2. Ereignisse, die, falls sie in einem Zustand auftreten, ignoriert werden und

3. Ereignisse, die, falls sie in einem Zustand auftreten, einer Fehlerbehandlung erforderlich machen.

Letztere kann man beispielsweise durch einen speziellen Fehlerzustand s_e behandeln.

Beispiel 7.2.5. Für das Telefon aus Beispiel 7.2.3 können im Anfangszustand s_0 die Ereignisse $e_0, \ldots, e_9$, die das Drücken einer Ziffer signalisieren, und das Auflegen (e_{auf}) ignoriert werden. Ein Auftreten eines Timeouts (e_{to}) oder das Auftreten der Ereignisse e_{rg} und e_{ru}, die eine gültige bzw. ungültige Rufnummer repräsentieren, ist als Fehlerfall zu werten. Die vollständige Automatentabelle (ohne Zustandsübergänge in Zustand s_2) ist in Tabelle 7.3 zu sehen.

Tabelle 7.3. Vollständige Automatentabelle zum endlichen Automaten aus Abb. 7.16 mit Fehlerzustand s_e [305]

	e_{ab}	e_{auf}	e_0	$\ldots$	e_9	e_{to}	e_{rg}	e_{ru}
s_0	s_1/e_r	$s_0/$	$s_0/$		$s_0/$	$s_e/$	$s_e/$	$s_e/$
s_1	$s_1/$	s_0/e_r	$s_1/e_z \wedge e_p$		$s_1/e_z \wedge e_p$	s_4/e_r	s_2/e_{vauf}	$s_3/$
s_2	$s_2/$	$s_0/e_r \wedge e_{vab}$	$s_2/$		$s_2/$	$s_e/$	$s_e/$	$s_e/$
s_3	$s_3/$	s_0/e_r	$s_3/$		$s_3/$	s_4/e_r	$s_e/$	$s_e/$
s_4	$s_4/$	$s_0/$	$s_4/$		$s_4/$	$s_4/$	$s_e/$	$s_e/$

7.2.2 Kontrollflussorientierte Testfälle

Neben funktionsorientierten Testfällen, die sich an der Spezifikation eines Systems orientieren, gibt es eine zweite große Klasse an Testfällen, die sog. *strukturorientierten Testfälle*. Wie der Name bereits suggeriert, werden diese anhand von Strukturmodellen, also Implementierungen, erstellt. Die Klasse der strukturorientierten Testfälle lässt sich wiederum in zwei Unterklassen einteilen, die sog. *kontrollflussorientierten* und die sog. *datenflussorientierten Testfälle*. Die Verwendung von strukturorientierten Testfällen in der simulativen Verifikation hat allerdings zum Nachteil, dass spezifizierte, aber nicht implementierte Funktionen nicht entdeckt werden können. Hierzu ist es notwendig auf funktionsorientierte Testfälle zurückzugreifen.

Zunächst werden die kontrollflussorientierten Testfälle näher betrachtet. Diese arbeiten auf Kontrollflussgraphen, die aus Programmen extrahiert werden. Für die Erstellung der Testfälle werden unterschiedliche Aspekte der Programmabarbeitung verwendet, z. B. Anweisungen, Zweige, Bedingungen, Schleifen oder Pfade. Ziel bei der Erstellung kontrollflussorientierter Testfälle ist es immer, eine möglichst hohe *Überdeckung* dieser Aspekte zu erzielen. Aus diesem Grund werden Überdeckungsmaße für diese Aspekte als *Vollständigkeitskriterium* der mit diesen Testfalleingaben durchgeführten simulativen Verifikation verwendet. Simulative Verifikationsverfahren, die ein Überdeckungsmaß als Vollständigkeitskriterium beinhalten, werden aus

historischen Gründen als *Überdeckungstests* bezeichnet. Die Erstellung datenflussorientierter Testfälle wird anschließend in Abschnitt 7.2.3 beschrieben.

Anweisungsüberdeckungstest

Die einfachsten kontrollflussorientierten Testfälle werden so entwickelt, dass eine möglichst hohe *Anweisungsüberdeckung* (engl. *statement coverage*) erzielt wird. Somit ist das Ziel, jede Anweisung in einem Programm mindestens einmal auszuführen. Diese Vorgehensweise ist einleuchtend, da sie sicherstellt, dass das Programm keinen unerreichbaren Quelltext (engl. *dead code*) enthält. Die Anweisungsüberdeckung $\mathcal{C}_S$ ist wie folgt definiert:

$$\mathcal{C}_S = \frac{|\{ausgeführte\ Anweisungen\}|}{|\{Anweisungen\}|} \tag{7.1}$$

Es wird also der Anteil ausgeführter Anweisungen aller Anweisungen bestimmt. Das Ziel ist es, 100% Anweisungsüberdeckung zu erzielen. Die zugehörige simulative Verifikation wird als *Anweisungsüberdeckungstest* bezeichnet.

Beispiel 7.2.6. Gegeben ist das folgende Fragment eines C-Programms:

```
 1     a = b + c;
 2     x = y;
 3     if (a > x) {
 4       y = b;
 5       x = x + 2;
 6     } else
 7       b = b + c;
 8     if (x == y)
 9       c = y;
10     else
11       y = a;
```

Dieses Programm enthält neun Anweisungen (die Zeilen 6 und 10 enthalten keine Anweisungen). Betrachtet wird die Testfalleingabe $(b = 1) \wedge (c = 1) \wedge (y = 1)$. Durch Simulation kann man bestimmen, dass die Anweisungen $\langle 1,2,3,4,5,8,11 \rangle$, also sieben der neun vorhandenen Anweisungen, ausgeführt werden. Die durch diese Testfalleingabe erzielte Anweisungsüberdeckung beträgt somit $\mathcal{C}_S = \frac{7}{9}$. Um 100% Anweisungsüberdeckung zu erreichen, wird eine zweite zusätzliche Testfalleingabe benötigt. Dies könnte beispielsweise $(b = 0) \wedge (c = 0) \wedge (y = 1)$ sein. In der Simulation dieses Testfalls werden die Anweisungen $\langle 1,2,3,7,8,9 \rangle$ ausgeführt. Die alleinige Anweisungsüberdeckung dieses Testfalls beträgt somit $\mathcal{C}_S = \frac{6}{9}$. Beide Testfalleingaben zusammen führen allerdings 100% der Anweisungen aus, weshalb $\mathcal{C}_S = 1$ ist.

Sicherlich ist eine 100%-ige Anweisungsüberdeckung ein notwendiges Kriterium, um Fehler, die in Anweisungen enthalten sind, zu prüfen. Allerdings handelt es sich nicht um ein hinreichendes Kriterium, um Fehler in Anweisungen zu erkennen, da ein Fehler eventuell nur unter bestimmten Stimuli auftritt.

Beispiel 7.2.7. Angenommen das Programm in Beispiel 7.2.6 enthalte in Zeile 3 einen Fehler: Anstelle von `(a > x)` muss es korrekterweise `(a >= x)` heißen. Keiner der beiden Testfalleingaben aus Beispiel 7.2.6 ($(b=1) \wedge (c=1) \wedge (y=1)$ und $(b=0) \wedge (b=0) \wedge (y=1)$) ist in der Lage diesen Fehler aufzudecken, obwohl sie zusammen eine 100%-ige Anweisungsüberdeckung hervorrufen. Eine geeignete Testfalleingabe, um diesen Fehler zu entdecken, wäre beispielsweise $((b=1) \wedge (c=0) \wedge (y=1)$, da dies zu einer identischen Wertzuweisung von a und x in Zeile 3 des Programms führt.

Man muss allerdings auch bei diesem Beispiel berücksichtigen, dass der Fehler nicht nur steuerbar, sondern auch beobachtbar sein muss, so dass dieser entdeckt werden kann. Der Fehler muss sich also in der Ausgabe des Programms niederschlagen.

Zweigüberdeckungstest

Der Anweisungsüberdeckungstest hat zum Ziel, 100% Anweisungsüberdeckung zu erreichen. Dies ist gleichbedeutend mit der Aussage, dass in dem Kontrollflussgraphen alle Knoten mindestens einmal durchlaufen werden. Ein wesentlich wichtigeres Maß ist allerdings die *Zweigüberdeckung* (engl. *branch coverage*), die misst, wie viele Kanten des Kontrollflussgraphen durchlaufen werden. Der zugehörige Zweigüberdeckungstest, der eine 100%-ige Zweigüberdeckung zum Ziel hat, subsumiert den Anweisungsüberdeckungstest, d. h. eine Zweigüberdeckung von 100% impliziert eine 100%-ige Anweisungsüberdeckung. Die Zweigüberdeckung $\mathcal{C}_B$ lässt sich als Quotient aus durchlaufenen Kanten zur Gesamtanzahl aller Kanten im Kontrollflussgraphen ausdrücken:

$$\mathcal{C}_B = \frac{|\{\textit{ausgeführte Kontrollflusskanten}\}|}{|\{\textit{Kontrollflusskanten}\}|} \tag{7.2}$$

Der Zweigüberdeckungstest hat zwei Ziele: Zum einen sollen Kontrollflusskanten identifiziert werden, die nicht durchlaufen werden können. Zum anderen sollen Kontrollflusskanten identifiziert werden, die sehr häufig durchlaufen werden. Letztere können für die Programmoptimierung verwendet werden. Werden Kontrollflusskanten nicht durchlaufen, kann dies prinzipiell zwei Gründe haben: Entweder kann der Zweig des Programms nicht durchlaufen werden, d. h. es existieren keine Testfalleingaben, die bewirken, dass der Programmzweig durchlaufen wird, oder es konnte bisher keine geeignete Testfalleingabe konstruiert werden, obwohl eine existiert. Letzteres tritt besonders häufig ein, wenn Umgebungswerte in dem eingebetteten System verarbeitet werden, die sich nicht direkt steuern lassen. Existieren allerdings keine Testfalleingaben, um einen Programmzweig zu durchlaufen, so liegt ein Entwurfsfehler vor.

Beispiel 7.2.8. Betrachtet wird das Programm aus Beispiel 7.2.1 auf Seite 394 zum Zählen der großen Konsonanten und Vokale in einer Eingabe. Der Kontrollflussgraph

ist in Abb. 7.17 dargestellt. Eine Testfalleingabe, die 100% Zweigüberdeckung erzeugt, wäre beispielsweise die Eingabesequenz $\langle$'A','B','1'$\rangle$. Die durchlaufene Sequenz an Zuständen in dem Kontrollflussgraphen lautet bei Simulation mit dieser Testfalleingabe $\langle v_1, v_2, v_3, v_4, v_5, v_2, v_3, v_5, v_2, v_6 \rangle$.

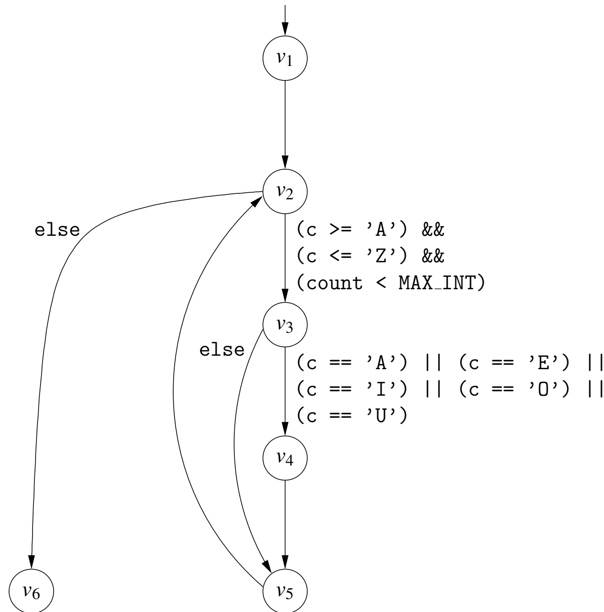


Abb. 7.17. Kontrollflussgraph für das Programm aus Beispiel 7.2.1 [305]

Die unkritische Benutzung des Zweigüberdeckungsmaßes aus Gleichung (7.2) hat zur Folge, dass die Testaktivität überschätzt wird. Der Grund hierfür liegt in der Feststellung, dass häufig viele Zweige in einem Programm zu mehreren Programmpfaden gehören. Da eine Testfalleingabe stets die Ausführung eines ganzen Pfades stimuliert, werden in den ersten Testfällen überproportional viele Zweige überdeckt, während es in späteren Testfällen immer schwieriger wird, neue Zweige zu erschließen [91]. Um dies zu verhindern, können die Abhängigkeiten zwischen einzelnen Zweigen analysiert werden. Dabei wird ein Zweig in der Zweigüberdeckung nicht berücksichtigt, wenn dieser immer genau dann ausgeführt wird, wenn ein anderer Zweig ausgeführt wird. Zweige ohne diese Eigenschaft werden als *primitive Zweige* bezeichnet. Verwendet man lediglich primitive Zweige in einem verbesserten Zweigüberdeckungsmaß $\mathcal{C}_{B^*}$, so sind Aussagen über den Verifikationsfortschritt aussagekräftiger. Das neue Zweigüberdeckungsmaß definiert sich wie folgt:

$$\mathcal{C}_{B^*} = \frac{|\{\textit{ausgeführte primitive Zweige}\}|}{|\{\textit{primitive Zweige}\}|} \tag{7.3}$$

Bedingungsüberdeckungstest

Der Zweigüberdeckungstest berücksichtigt zwei wesentliche Faktoren nicht: Zum einen können Entscheidungen, die das Durchlaufen von Zweigen steuern, aus mehreren Teilbedingungen zusammengesetzt sein. Jeder dieser Teilbedingungen ist eine potentielle Fehlerquelle und sollte deshalb überprüft werden. Zum anderen bilden mehrere Zweige einen Pfad in der Programmausführung, weshalb die Überprüfung einzelner Zweige nicht sehr aussagekräftig ist. Insbesondere können Zweige innerhalb von Schleifen mehrfach durchlaufen werden. Für beide Aspekte wurden eigene Überdeckungstests entwickelt, die im Folgenden betrachtet werden. Zunächst werden die sog. *Bedingungsüberdeckungstests* betrachtet. Die sog. *Pfadüberdeckungstests* werden anschließend vorgestellt.

Einfacher Bedingungsüberdeckungstest

Um eine 100%-ige Zweigüberdeckung zu erzielen, ist es lediglich notwendig, dass jede Entscheidung in dem zu prüfenden Programm einmal den Wert `F` und einmal den Wert `T` annimmt. Dabei werden Teilbedingungen und deren logische Verknüpfung nicht weiter betrachtet. Im Gegensatz dazu haben Bedingungsüberdeckungstests darüber hinaus das Ziel, Entscheidungen möglichst gründlich zu überprüfen.

Die einfachste Form eines Bedingungsüberdeckungstest ist der sog. *einfache Bedingungsüberdeckungstest*. Die Annahme ist hierbei, dass jede Entscheidung aus logisch verknüpften Teilbedingungen aufgebaut ist. Diese Teilbedingungen selbst können wiederum aus verknüpften Teilbedingungen konstruiert sein. Teilbedingungen, die nicht aus anderen Teilbedingungen aufgebaut sind, werden als *atomare Bedingungen* bezeichnet. Der einfache Bedingungsüberdeckungstest fordert, dass jede atomare Bedingung einmal den Wert `F` und einmal den Wert `T` annimmt.

Ein starres Vorgehen bei diesem Schema führt dazu, dass der einfache Bedingungsüberdeckungstest den Zweigüberdeckungstest nicht subsumiert.

Beispiel 7.2.9. Das Programm aus Beispiel 7.2.1 auf Seite 394 enthält zwei Entscheidungen:

1. `((c >= 'A') && (c <= 'Z') && (count <= MAX_INT))`
2. `((c == 'A') || (c == 'E') || (c == 'I') ||`
   `(c == 'O') || (c == 'U'))`

Die erste Entscheidung enthält drei atomare Bedingungen, während die zweite Entscheidung fünf atomare Bedingungen beinhaltet. Drei mögliche Testfälle sind in Tabelle 7.4 zu sehen, die alle im Zustand v_6 des Kontrollflussgraphen aus Abb. 7.17 enden. Der erste Testfall prüft alle möglichen Belegungen der atomaren Bedingungen der zweiten Entscheidung, die zur Evaluierung `T` führen. Die letzte Eingabe ('1') führt allerdings schon dazu, dass die erste Entscheidung fehlschlägt und somit

die zweite Entscheidung nicht mehr erreicht wird, weshalb die Belegung der atomaren Bedingungen den Wert – (don't care) trägt. Hierbei hat jede atomare Bedingung der zweiten Entscheidung genau einmal den Wert T und mehrfach den Wert F angenommen.

Tabelle 7.4. Einfacher Bedingungsüberdeckungstest des Programms aus Beispiel 7.2.1 [305]

	1.						2.	3.
`count`	0	1	2	3	4	5	0	`MAX_INT`
`c`	'A'	'E'	'I'	'O'	'U'	'1'	'a'	'D'
`c >= 'A'`	T	T	T	T	T	F	T	T
`c <= 'Z'`	T	T	T	T	T	T	F	T
`count < INT_MAX`	T	T	T	T	T	T	T	F
`c == 'A'`	T	F	F	F	F	–	–	–
`c == 'E'`	F	T	F	F	F	–	–	–
`c == 'I'`	F	F	T	F	F	–	–	–
`c == 'O'`	F	F	F	T	F	–	–	–
`c == 'U'`	F	F	F	F	T	–	–	–

Ebenfalls haben alle atomaren Bedingungen der ersten Entscheidung mindestens einmal jede der beiden möglichen Belegungen angenommen. Wie man aber sehen kann, ist die Kontrollflusskante (v_3, v_5) im einfachen Bedingungsüberdeckungstest nicht erhalten.

Bedingungs-/Entscheidungsüberdeckungstest

Das Problem, dass der einfache Bedingungsüberdeckungstest nicht den Zweigüberdeckungstest subsumiert, wird im *Bedingungs-/Entscheidungsüberdeckungstest* dadurch gelöst, dass zusätzlich zur einfachen Bedingungsüberdeckung auch die Zweigüberdeckung gefordert wird. Dies resultiert darin, dass nicht nur jede atomare Bedingung mindestens einmal jeden Wert T oder F annehmen muss, sondern zusätzlich auch jede Entscheidung mindestens einmal erfüllt sein muss und einmal fehlschlägt.

Beispiel 7.2.10. Betrachtet wird die Entscheidung `(((u == 0) || (x > 5)) && ((y < 6) || (z == 0)))`. Zwei Testfalleingaben, die zu einer 100%-igen *Bedingungs-/Entscheidungsüberdeckung* (engl. *condition/decision coverage*), lauten:

	`(u == 0)`	`(x > 5)`	`(y < 6)`	`(z == 0)`
1.	F	T	F	F
2.	T	F	T	T

Für den ersten Testfall evaluiert die Entscheidung zu F, für den zweiten Testfall zu T. Somit wurde jede atomare Bedingung einmal mit jeder möglichen Belegung belegt sowie die Entscheidung einmal erfüllt und einmal nicht. Man beachte jedoch, dass die Teilbedingungen `(((u == 0) || (x > 5))` in beiden Testfalleingaben den Wert T annimmt.

Minimaler Mehrfach-Bedingungsüberdeckungstest

Wie im Beispiel 7.2.10 dargestellt, fordert der Bedingungs-/Entscheidungsüberdeckungstest lediglich, dass alle atomaren Bedingungen und alle Entscheidungen mindestens einmal erfüllt sind und mindestens einmal nicht erfüllt sind. Der *minimale Mehrfachüberdeckungstest* fordert darüber hinaus, dass auch alle Teilbedingungen jeweils mindestens einmal jeden Wert annehmen. Somit subsumiert der minimale Mehrfach-Bedingungsüberdeckungstest den Bedingungs-/Entscheidungsüberdeckungstest und damit auch den Zweigüberdeckungstest.

Beispiel 7.2.11. Betrachtet wird wiederum die Entscheidung `(((u == 0) || (x > 5)) && ((y < 6) || (z == 0)))` aus Beispiel 7.2.10. Die beiden folgenden Testfalleingaben führen zu 100% *minimaler Mehrfach-Bedingungsüberdeckung*:

	(u == 0)	(x > 5)	(y < 6)	(z == 0)
1.	F	F	F	F
2.	T	T	T	T

da sich hierbei für die Teilbedingungen die folgenden Werte ergeben:

	(u == 0) || (x > 5)	(y < 6) || (z == 0)
1.	F	F
2.	T	T

Die Entscheidung selbst nimmt auch wiederum beide Werte an.

Bei der Erstellung der Testfälle sollte die hierarchische Struktur der Entscheidungen berücksichtigt werden. Das oben beschriebene Vorgehen im minimalen Mehrfach-Bedingungsüberdeckungstest ist allerdings nur dann korrekt, wenn die Evaluierungen der Entscheidungen im Programm vollständig erfolgen, d. h. alle Teilbedingungen auch ausgewertet werden. Oftmals führen Compiler eine Optimierung durch, bei der die Auswertung auf unvollständigen Belegungen beruht. Dies kann beispielsweise geschehen, wenn eine Eingabe einer Konjunktion (Disjunktion) den Wert F (T) hat. Bei einer solchen frühzeitigen Auswertung werden nicht zwangsläufig alle Teilbedingungen bestimmt, wodurch Fehler in der Implementierung dieser Teilbedingungen übersehen werden können. Verwendet ein Programm eine frühzeitige Auswertung, müssen die Testfälle entsprechend angepasst werden (siehe [305]).

Mehrfach-Bedingungsüberdeckungstest

Der *Mehrfach-Bedingungsüberdeckungstest* verwendet als Überdeckungsmaß denjenigen Anteil an getesteten Wertzuweisungen an atomare Bedingungen zu der Anzahl aller möglichen Wertzuweisungen an die atomaren Bedingungen. Dieser subsumiert somit den minimalen Mehrfach-Bedingungsüberdeckungstest. Eine 100% *Mehrfach-Bedingungsüberdeckung* (engl. *multiple condition coverage*) ist allerdings nur erreichbar, wenn alle Wertzuweisungen simuliert werden. Dies ist in der Praxis oft nicht möglich, da diese Anzahl mit der Anzahl atomarer Bedingungen exponentiell ansteigt.

Pfadüberdeckungstest

Der *Pfadüberdeckungstest* hat zum Ziel, alle möglichen Berechnungspfade in einem Programm zu prüfen. Als Vollständigkeitskriterium wird die sog. *Pfadüberdeckung* $\mathcal{C}_P$ (engl. *path coverage*) verwendet:

$$\mathcal{C}_P = \frac{|\{ausgeführte\ Pfade\}|}{|\{Pfade\}|} \tag{7.4}$$

Der Pfadüberdeckungstest ist somit eine umfassende kontrollflussorientierte Verifikationsmethode. Er subsumiert den Zweigüberdeckungstest und damit den Anweisungsüberdeckungstest. Allerdings werden die Bedingungsüberdeckungstests nicht subsumiert. Um auch diese mit abzudecken, bleibt lediglich die Möglichkeit, eine erschöpfende Simulation durchzuführen.

Beispiel 7.2.12. Betrachtet wird das C-Programm aus Beispiel 7.2.6 auf Seite 401. Das Programm enthält vier Pfade. Unter Verwendung der Testfalleingabe $(b = 1) \wedge (c = 1) \wedge (y = 1)$, werden die Anweisungen $\langle 1,2,3,4,5,8,11 \rangle$ ausgeführt. Diese Testfalleingabe führt zu einer Pfadüberdeckung von $\mathcal{C}_P = \frac{1}{4}$. Die Pfadüberdeckung kann also lediglich durch weitere Testfälle erhöht werden. Die Testfalleingaben $(b = 0) \wedge (c = 0) \wedge (y = 1)$ und $(b = 3) \wedge (c = 0) \wedge (y = 1)$ resultieren in den Anweisungssequenzen $\langle 1,2,3,7,8,9 \rangle$ und $\langle 1,2,3,4,5,8,9 \rangle$. Damit ist die Pfadüberdeckung auf $\mathcal{C}_P = \frac{3}{4}$ angestiegen. Für den vierten Pfad lässt sich allerdings keine geeignete Testfalleingabe konstruieren. Der Grund hierfür liegt darin, dass x und y in Zeile 3 stets den selben Wert besitzen. Ist gleichzeitig $a \leq x$, wird der `else`-Zweig (Zeile 8) ausgeführt, in dem weder x noch y geändert werden. Als Ergebnis besitzen x und y in Zeile 8 immer den selben Wert, wenn zuvor Zeile 7 ausgeführt wurde.

An diesem Beispiel sieht man bereits, dass die Konstruktion von Testfällen sehr komplex werden kann. Im Allgemeinen lässt sich mit dem Pfadüberdeckungstest keine 100%-ige Pfadüberdeckung erzielen.

Ein Pfad ist eine Sequenz von Knoten des Kontrollflussgraphen, wobei zwei Pfade genau dann identisch sind, wenn sie die selben Knoten in der selben Reihenfolge durchlaufen. Enthält ein Programm Schleifen, so kann die Anzahl unterschiedlicher Pfade unendlich sein. Diese resultieren aus Schleifen, die keine feste Anzahl an Wiederholungen besitzen. Wie das folgende Beispiel aus [305] zeigt, kann die Anzahl der Pfade selbst bei Programmen, die lediglich Schleifen mit konstanter Anzahl an Schleifendurchläufen besitzen, so groß sein, dass ein sinnvoller Test mit dem Ziel 100% Pfadüberdeckung nicht möglich ist:

Beispiel 7.2.13. Betrachtet wird das Programm aus Beispiel 7.2.1 auf Seite 394 zum Zählen von großen Konsonanten und großen Vokalen. Der Kontrollflussgraph des Programms ist in Abb. 7.17 auf Seite 403 zu sehen. Jeder Pfad in diesem Programm beginnt mit den Sequenzen $\langle v_1, v_2 \rangle$. Diesem Teilpfad folgen i ($i \geq 0$) Teilpfade, die entweder die Form $\langle v_3, v_4, v_5, v_2 \rangle$ oder $\langle v_3, v_5, v_2 \rangle$ besitzen. Schließlich endet jeder Pfad mit dem Teilpfad $\langle v_6 \rangle$.

Die Anzahl der Schleifendurchläufe ist in dem Programm durch die Bedingung `count < MAX_INT` begrenzt. Man sieht also, dass in diesem Beispiel eine obere Grenze für die Anzahl an Schleifendurchläufen angegeben werden kann. Somit handelt es sich hierbei um ein eher gutmütiges Beispiel im Vergleich zu Programmen, die eine unbegrenzte Anzahl an Durchläufen zulassen.

In diesem Beispiel lässt sich die zu prüfende Pfadanzahl angeben: Es gibt exakt einen Pfad, der die Schleife niemals durchläuft, und zwei unterschiedliche Pfade, welche die Schleife jeweils einmal ausführen. Vier unterschiedliche Pfade enthalten zwei Schleifendurchläufe. Führt man diese Rechnung fort, sieht man, dass 2^n unterschiedliche Pfade mit genau n Schleifenausführungen existieren. Somit ist die Gesamtanzahl unterschiedlicher Pfade für dieses Programm $1+2+4+\cdots+2^{MAX_INT} = 2^{MAX_INT+1}-1$. Geht man davon aus, dass MAX_INT den Wert 32.767 besitzt, so gibt es ungefähr $1{,}41 \cdot 10^{9.864}$ Pfade. Unter der Annahme, dass 1.000 Pfade pro Sekunde simuliert werden, würde die gesamte Verifikationszeit $4{,}5 \cdot 10^{9.853}$ Jahre dauern. Im Vergleich dazu beträgt das geschätzte Alter der Erde 4,5 Milliarden Jahre, also $4{,}5 \cdot 10^9$ Jahre.

Strukturierter Pfadüberdeckungstest

Die extrem hohe Anzahl von Pfaden in Programmen in Gegenwart von Schleifen verlangt nach Verfahren, die geeignete repräsentative Pfade bestimmen, die überprüft werden sollen. Bei dem *strukturierten Pfadüberdeckungstest* werden deshalb lediglich Pfade betrachtet, die höchstens k Schleifendurchläufe enthalten. Am häufigsten wird dabei ein k der Größe 2 gewählt. Somit erfolgt ausschließlich eine Betrachtung von Pfaden, die eine Schleife gar nicht durchlaufen, genau einmal durchlaufen oder genau zweimal durchlaufen. Die zweite Klasse von Testfalleingaben stimuliert also Pfade, welche die Schleife zwar betreten, nicht aber wiederholen. Die dritte Klasse an Testfällen, repräsentiert schließlich alle Pfade, welche die Schleife betreten und wiederholen.

Bei Verwendung des strukturierten Pfadüberdeckungstests kann das Vollständigkeitskriterium aus Gleichung (7.4) in eine *strukturierte Pfadüberdeckung* $\mathcal{C}_{P,k}$ geändert werden.

$$\mathcal{C}_{P,k} = \frac{|\{ausgeführte\ Pfade\}|}{|\{Pfade[k]\}|} \tag{7.5}$$

wobei *Pfade[k]* diejenigen Pfade, die höchstens k Schleifendurchläufe durchführen, repräsentieren.

Beispiel 7.2.14. Betrachtet wird wiederum das Programm aus Beispiel 7.2.1 auf Seite 394 zum Zählen von großen Konsonanten und großen Vokalen. Der Kontrollflussgraph des Programms ist in Abb. 7.17 auf Seite 403 dargestellt.

1. Die erste Klasse an Testfalleingaben ($k = 0$) führt die Schleife nicht aus. Es gibt in diesem Fall lediglich einen Pfad $\langle v_1, v_2, v_6 \rangle$ mit dieser Eigenschaft. Die Stimuli für diesen Testfall müssen die Form besitzen, dass *count* bereits den Wert *INT_MAX* besitzt.

2. Die zweite Klasse an Testfällen durchläuft die Schleife genau einmal. Zwei verschiedene Pfade müssen dabei unterschieden werden: $\langle v_1, v_2, v_3, v_4, v_5, v_2, v_6 \rangle$ und $\langle v_1, v_2, v_3, v_5, v_2, v_6 \rangle$. Die Eingabe für den ersten Testfall ($k = 1$) wäre beispielsweise $\langle$`'A','1'`$\rangle$. Für den zweiten Testfall wäre eine geeignete Eingabe $\langle$`'B','1'`$\rangle$. Für beide Testfälle wird *count* zu Beginn auf *count* := 0 gesetzt.
3. Die dritte Klasse an Testfällen ($k = 2$) durchläuft die Schleife genau zweimal. Es existieren insgesamt vier Pfade mit *count* := 0:
   a) $\langle v_1, v_2, v_3, v_4, v_5, v_2, v_3, v_4, v_5, v_2, v_6 \rangle$ bei Eingabe $\langle$`'A','A','1'`$\rangle$.
   b) $\langle v_1, v_2, v_3, v_4, v_5, v_2, v_3, v_5, v_2, v_6 \rangle$ bei Eingabe $\langle$`'A','B','1'`$\rangle$.
   c) $\langle v_1, v_2, v_3, v_5, v_2, v_3, v_4, v_5, v_2, v_6 \rangle$ bei Eingabe $\langle$`'B','A','1'`$\rangle$.
   d) $\langle v_1, v_2, v_3, v_5, v_2, v_3, v_5, v_2, v_6 \rangle$ bei Eingabe $\langle$`'B','B','1'`$\rangle$.

Mit den angegeben sieben Testfalleingaben kann eine 100%-ige strukturierte Pfadüberdeckung $\mathcal{C}_{P,2}$ erreicht werden.

Eine Limitierung des strukturierten Pfadüberdeckungstests besteht darin, dass manche Pfade nicht ausführbar sind. Der Grund hierfür ist, dass manche Pfade erst nach mehreren ($n > k$) Schleifendurchläufen erreichbar sind.

Beispiel 7.2.15. Gegeben ist folgendes Programm [305]:

```
1     i = 0;
2     while (i < n) {
3       if (i < 4)
4         do_this();
5       else
6         do_that();
7       i++;
8     }
```

Der Kontrollflussgraph ist in Abb. 7.18 dargestellt. Die Pfade, die sich nach dem strukturierten Pfadüberdeckungstest mit $k = 2$ ergeben, lauten:

$$\begin{aligned}
k = 0 &: \langle v_1, v_2, v_7 \rangle \\
k = 1 &: \langle v_1, v_2, v_3, v_4, v_6, v_2, v_7 \rangle \\
& \langle v_1, v_2, v_3, v_5, v_6, v_2, v_7 \rangle \\
k = 2 &: \langle v_1, v_2, v_3, v_4, v_6, v_2, v_3, v_4, v_6, v_2, v_7 \rangle \\
& \langle v_1, v_2, v_3, v_4, v_6, v_2, v_3, v_5, v_6, v_2, v_7 \rangle \\
& \langle v_1, v_2, v_3, v_5, v_6, v_2, v_3, v_4, v_6, v_2, v_7 \rangle \\
& \langle v_1, v_2, v_3, v_5, v_6, v_2, v_3, v_5, v_6, v_2, v_7 \rangle
\end{aligned}$$

Von diesen Testfällen sind allerdings der zweite Pfad für $k = 1$ und die letzten drei Pfade für $k = 2$ nicht ausführbar, da der Kontrollflussknoten v_5 erst im fünften Schleifendurchlauf erreichbar ist.

Schließlich sei noch erwähnt, dass Programme im Allgemeinen mehr als eine Schleife enthalten. Diese können entweder sequentiell oder verschachtelt auftreten.

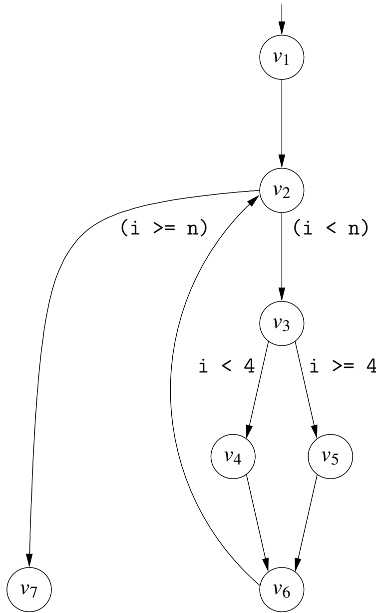


Abb. 7.18. Kontrollflussgraph für das Programm aus Beispiel 7.2.15 [305]

In jedem Fall führt die Gegenwart mehrerer Schleifen zu einem erhöhten Aufwand in der Simulation, der bereits bei Wahl eines sehr kleinen k, z. B. $k := 2$, enorm sein kann.

7.2.3 Datenflussorientierte Testfälle

Neben kontrollflussorientierten Testfällen gibt es eine zweite Klasse von strukturorientierten Testfällen, die sog. datenflussorientierten Testfälle. Die Verifikationsvollständigkeit bei Verwendung datenflussorientierter Testfälle wird anhand von Datenzugriffen gemessen. Dabei werden drei Arten unterschieden:

1. *def*: Auf eine Variable wird schreibend zugegriffen, d. h. sie wird definiert.
2. *c-use*: Auf eine Variable wird lesend innerhalb einer Berechnung (engl. *computation*) zugegriffen
3. *p-use*: Auf eine Variable wird lesend innerhalb einer Entscheidung zugegriffen, d. h. sie wird innerhalb eines Prädikates verwendet.

Die Konstruktion datenflussorientierter Testfälle erfolgt anhand eines sog. *datenflussattributierten Kontrollflussgraphen*. In einem datenflussattributierten Kontrollflussgraphen werden die Kanten mit *p-uses* und Knoten mit *defs* und *c-uses* attributiert. Weiterhin wird der Kontrollflussgraph um einen Ein- und einen Ausgabeknoten v_{in} und v_{out} erweitert, um die Übergabe von Argumenten bzw. Rückgabewerten zu modellieren, was wiederum durch *defs* bzw. *c-uses* geschieht.

Beispiel 7.2.16. Gegeben ist das folgende Programm aus [305] zur Bestimmung des Minimum/Maximum von zwei Zahlen:

```
1     minMax(int& min, int& max) {
2       if (min > max) {
3         int tmp = min;
4         min = max;
5         max = tmp;
6       }
7     }
```

Der datenflussattributierte Kontrollflussgraph ist in Abb. 7.19 dargestellt. Im Knoten v_{in} werden die Argumente `min` und `max` definiert. Die `if`-Anweisung in Zeile 2 ist auf Zustand v_1 abgebildet, wobei die ausgehenden Kontrollflusskanten mit den *p-uses* für die beiden Variablen attributiert sind. In Zustand v_2 werden die insgesamt drei Anweisungen (Zeilen 3, 4 und 5) auf drei *defs* und drei *c-uses* abgebildet. Der Knoten v_{out} ist schließlich noch mit den beiden *c-uses* der Variablen `min` und `max` beschriftet, um die Rückgabe dieser Werte zu modellieren.

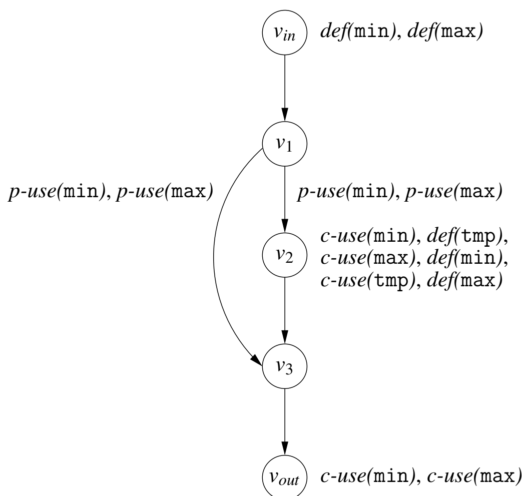


Abb. 7.19. Datenflussattributierter Kontrollflussgraph zu dem Programm aus Beispiel 7.2.16 [305]

Während Kontroll-Datenflussgraphen den lokalen Datenfluss lediglich innerhalb von Grundblöcken beschreiben, dient der datenflussattributierte Kontrollflussgraph der Ermittlung des globalen Datenflusses. Der globale Datenfluss bildet die Grundlage zur Erstellung datenflussorientierter Testfälle.

Ein Variablenzugriff auf Variable x innerhalb einer Berechnung (*c-use(x)*) ist *lokal*, wenn im selben Grundblock eine Definition dieser Variablen erfolgt, d. h. *def(x)*.

Andernfalls ist dieser Zugriff *global*. Der *c-use* von Variable `tmp` in Zustand v_2 in Abb. 7.19 ist beispielsweise lokal, da diesem *c-use* ein *def(*`tmp`*)* vorangeht. Die *c-uses* der Variablen `min` und `max` im selben Zustand sind hingegen global, da die Definitionen erst nach den *c-uses* erfolgen. Zentral für die Bestimmung des globalen Datenflusses ist der Begriff des *definitionsfreien Pfades*:

Definition 7.2.1 (Definitionsfreier Pfad). *Ein Pfad* $\langle v_1, v_2, \dots, v_n \rangle$ *in einem datenflussattributierten Kontrollflussgraphen heißt* definitionsfrei *bezüglich einer Variablen x, wenn x in keinem Knoten des Pfades definiert wird.*

Es existiert nun ein Datenfluss von Zustand v_i zu Zustand v_j, wenn eine Variable x in v_i definiert (*def(x)*) und in v_j global berechnend benutzt wird (*c-use(x)*) und ein Pfad $\langle v_i, v_{i+1}, \dots, v_{j-1}, v_j \rangle$ existiert, bei dem der Teilpfad $\langle v_{i+1}, \dots, v_{j-1} \rangle$ definitionsfrei bezüglich x ist. Man sagt auch, die Definition der Variablen x in Knoten v_i erreicht die berechnende Benutzung von x in v_j. Analog erreicht eine Definition der Variablen x in Knoten v_i die prädikative Benutzung (*p-use*) auf Kante (v_{j-1}, v_j), wenn der Pfad $\langle v_{i+1}, \dots, v_{j-1} \rangle$ definitionsfrei bezüglich x ist. Erreicht eine Definition *def(x)* in Knoten v_i die global berechnende Benutzung in v_j bzw. eine prädikative Benutzung auf Kante (v_{j-1}, v_j), so ist die *Variablendefinition global*. Eine Variablendefinition heißt *lokal*, wenn einer Definition in Knoten v_i eine berechnende Verwendung in Knoten v_i folgt, ohne dass weitere Definitionen der selben Variablen dazwischen erfolgen. Definitionen, die weder lokal noch global sind, sind redundant.

Für die Erstellung datenflussorientierter Testfälle ist es notwendig, die global definierten *def(*v_i*)*, global berechnend benutzten *c-use(*v_i*)* und prädikativ benutzen Variablen *p-use(*v_i, v_{i+1}*)* in jedem Knoten v_i und auf jeder Kante (v_i, v_{i+1}) des datenflussattributierten Kontrollflussgraphen zu bestimmen.

Beispiel 7.2.17. Für das Programm aus Beispiel 7.2.16 mit datenflussattributierten Kontrollflussgraphen aus Abb. 7.19 ergeben sich folgende globale Definitionen, globale berechnende Benutzungen und prädikative Benutzungen:

Knoten/ Kante	*def(*v_i*)*	*c-use(*v_i*)*	*p-use(*v_i, v_{i+1}*)*
v_{in}	{*def(*`min`*)*, *def(*`max`*)*}	$\emptyset$	-
v_1	$\emptyset$	$\emptyset$	-
v_2	{*def(*`min`*)*, *def(*`max`*)*}	{*c-use(*`min`*)*, *c-use(*`max`*)*}	-
v_3	$\emptyset$	$\emptyset$	-
v_{out}	$\emptyset$	{*c-use(*`min`*)*, *c-use(*`max`*)*}	-
(v_{in}, v_1)	-	-	$\emptyset$
(v_1, v_2)	-	-	{*p-use(*`min`*)*, *p-use(*`max`*)*}
(v_1, v_3)	-	-	{*p-use(*`min`*)*, *p-use(*`max`*)*}
(v_2, v_3)	-	-	$\emptyset$
(v_3, v_{out})	-	-	$\emptyset$

Der globale Datenfluss lässt sich schließlich mit den Mengen *dcu* und *dpu* für jeden Knoten v_i in dem datenflussattributierten Kontrollflussgraphen beschreiben:

Definition 7.2.2 (dcu). *Sei v_i ein Knoten in einem datenflussattributierten Kontrollflussgraphen und x eine Variable mit $x \in def(v_i)$, dann ist dcu(x, v_i) die Menge aller Knoten v_j mit $x \in c\text{-}use(v_j)$, für die ein Pfad $\langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ existiert, bei dem der Teilpfad $\langle v_{i+1}, \ldots, v_{j-1} \rangle$ definitionsfrei bezüglich x ist.*

Definition 7.2.3 (dpu). *Sei v_i ein Knoten in einem datenflussattributierten Kontrollflussgraphen und x eine Variable mit $x \in def(v_i)$, dann ist dpu(x, v_i) die Menge aller Kanten (v_{j-1}, v_j) mit $x \in p\text{-}use(v_{j-1}, v_j)$, für die ein Pfad $\langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ existiert, bei dem der Teilpfad $\langle v_{i+1}, \ldots, v_{j-1} \rangle$ definitionsfrei bezüglich x ist.*

Beispiel 7.2.18. Für das Programm in Beispiel 7.2.16 mit dem datenflussattributierten Kontrollflussgraphen in Abb. 7.19 ergibt sich somit:

Variable x	Knoten v_i	*dcu(x, v_i)*	*dpu(x, v_i)*
`min`	v_{in}	$\{v_2, v_{out}\}$	$\{(v_1, v_2), (v_1, v_3)\}$
`max`	v_{in}	$\{v_2, v_{out}\}$	$\{(v_1, v_2), (v_1, v_3)\}$
`min`	v_2	$\{v_{out}\}$	$\emptyset$
`max`	v_2	$\{v_{out}\}$	$\emptyset$

defs/uses-Überdeckungstest

Der ermittelte globale Datenfluss dient als Grundlage zur Generierung datenflussorientierter Testfälle. Das Ergebnis sind Überdeckungstests, die unterschiedliche Überdeckungsmaße basierend auf Variablendefinitionen und -verwendungen definieren. Die folgende Darstellung erfolgt in Anlehnung an [305].

all defs-Überdeckungstest

Zur Erreichung einer 100%-igen *all defs*-Überdeckung ist es notwendig, Testfalleingaben so zu generieren, dass zu jeder globalen Variablendefinition mindestens ein *c-use* oder ein *p-use* getestet wird. Somit müssen die Testfälle das folgende Kriterium erfüllen: Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu einem Knoten in *dcu(x, v_i)* oder einer Kante in *dpu(x, v_i)* getestet werden.

Beispiel 7.2.19. Betrachtet wird das Programm aus Beispiel 7.2.16 auf Seite 411. Die Knoten v_{in} und v_2 des datenflussattributierten Kontrollflussgraphen aus Abb. 7.19 enthalten jeweils die globalen Definitionen von `min` und `max`. Die Testfalleingabe $(v_{in}, v_1, v_2, v_3, v_{out})$ erfüllt das Kriterium zum Erreichen einer *all defs*-Überdeckung von 100%. Die Definitionen in Knoten v_{in} werden in den *p-uses* der Kante (v_1, v_2) prädikativ verwendet. Die Definitionen in Knoten v_2 werden im Knoten v_{out} berechnend verwendet.

Man sieht, dass der *all defs*-Überdeckungstest weder den Zweig- noch den Anweisungsüberdeckungstest subsumiert.

all p-uses-Überdeckungstest

Im *all p-uses*-Überdeckungstest wird gefordert, dass für jede Entscheidung und für jede darin verwendete Variable jede Kombination mit deren Definitionen, welche die Entscheidung erreichen, geprüft wird. Die Testfälle müssen also das folgende Kriterium erfüllen: Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu jeder Kante in *dpu(x, v_i)* getestet werden.

Beispiel 7.2.20. Betrachtet wird das Programm aus Beispiel 7.2.16. Um 100% *all p-uses*-Überdeckung zu erzielen, sind zwei Testfalleingaben notwendig: $(v_{in}, v_1, v_3, v_{out})$ und $(v_{in}, v_1, v_2, v_3, v_{out})$. Damit werden die beiden Kanten (v_1, v_3) und (v_1, v_2), die mit *p-uses* attributiert sind, getestet.

Somit subsumiert der *all p-uses*-Überdeckungstest den Zweigüberdeckungstest.

all c-uses-Überdeckungstest

Analog zum *all p-uses*-Überdeckungstest wird im *all c-uses*-Überdeckungstest gefordert, dass für jeden globalen berechnenden Zugriff und für jede darin verwendete Variable jede Kombination mit deren Definitionen, die den Zugriff erreichen, geprüft wird. Die Testfälle müssen somit das folgende Kriterium erfüllen: Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu jedem Knoten in *dcu(x, v_i)* getestet werden.

Beispiel 7.2.21. Für das Programm aus Beispiel 7.2.16 führt die Testfalleingabe $(v_{in}, v_1, v_2, v_3, v_{out})$ zu einer 100%-igen *all c-uses*-Überdeckung.

Der *all c-uses*-Überdeckungstest subsumiert weder Zweig-, Anweisungs- noch einen anderen *defs/uses*-Überdeckungstest.

all c-uses/some p-uses-Überdeckungstest

Der *all c-uses*-Überdeckungstest prüft offensichtlich lediglich Variablendefinitionen, die in berechnenden Zugriffen münden. Variablen, die ausschließlich prädikativ verwendet werden, werden somit nicht getestet. Der *all c-uses/some p-uses*-Überdeckungstest erweitert den *all c-uses*-Überdeckungstest dahingehend, dass für ausschließlich prädikativ verwendete Variablen ebenfalls Testfälle gefordert werden. Somit müssen die Testfälle das folgende Kriterium erfüllen:

- Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu *jedem* Knoten in *dcu(x, v_i)* getestet werden.
- Ist *dcu(x, v_i)* leer, so muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu *einer* Kante in *dpu(x, v_i)* getestet werden.

Beispiel 7.2.22. Für das Programm aus Beispiel 7.2.16 fordert der *all c-uses/some p-uses*-Überdeckungstest die Testfalleingabe $(v_{in}, v_1, v_2, v_3, v_{out})$. Die Definitionen in v_{in} und v_2 erfordern den Test der berechnenden Zugriffe in v_2 und v_{out}. Da damit bereits alle definierten Variablen getestet wurden, müssen keine weiteren Testfälle hinzugenommen werden, obwohl der Zweig (v_1, v_3) noch nicht getestet wurde.

Der *all c-uses/some p-uses*-Überdeckungstest subsumiert den *all defs*-Überdeckungstest. Er subsumiert aber weder den Zweig- noch den Anweisungsüberdeckungstest.

all p-uses/some c-uses-Überdeckungstest

Analog zum *all c-uses/some p-uses*-Überdeckungstest existiert der *all p-uses/some c-uses*-Überdeckungstest. Bei diesem wird zusätzlich zu den Testfällen des *all p-uses*-Überdeckungstests gefordert, dass für eine Variable, die ausschließlich berechnend verwendet wird, also in keiner prädikativen Benutzung auftaucht, weitere Testfälle generiert werden müssen. Das Kriterium für die Testfälle lautet somit:

- Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss *mindestens* ein definitionsfreier Pfad bezüglich x von Knoten v_i zu jeder Kante in *dpu(x, v_i)* getestet werden.
- Ist *dpu(x, v_i)* leer, so muss *mindestens* ein definitionsfreier Pfad bezüglich x von Knoten v_i zu einem Knoten in *dcu(x, v_i)* getestet werden.

Beispiel 7.2.23. Betrachtet wird das Programm aus Beispiel 7.2.16 auf Seite 411. Für eine 100%-ige *all p-uses/some c-uses*-Überdeckung sind die beiden Testfalleingaben $(v_{in}, v_1, v_3, v_{out})$ und $(v_{in}, v_1, v_2, v_3, v_{out})$ notwendig. Die Definitionen von `min` und `max` in v_{in} erfordern den Test der prädikativen Verwendung in (v_1, v_3) und (v_1, v_2). Zusätzlich wird aber durch die Definitionen in Knoten v_2 gefordert, dass der berechnende Zugriff in v_{out} getestet wird. Dies ist allerdings bereits im zweiten Testfall enthalten.

Der *all p-uses/some c-uses*-Überdeckungstest subsumiert Zweig, Anweisungs- und *all p-uses*-Überdeckungstest.

all uses-Überdeckungstest

Die Kombination aus den *all c-uses/some p-uses*- und *all p-uses/some c-uses*-Überdeckungstests führt zu dem *all uses*-Überdeckungstest. Es wird dabei gefordert, dass für jede globale Definition jede erreichbare berechnende und prädikative Verwendung getestet wird. Das Kriterium für die Testfälle lautet somit: Für jeden Knoten v_i im datenflussattributierten Kontrollflussgraph und jeder Variablen $x \in defs(v_i)$ muss mindestens ein definitionsfreier Pfad bezüglich x von Knoten v_i zu jedem Knoten in *dpu(x, v_i) und* zu jeder Kante in *dpu(x, v_i)* getestet werden. Der *all uses*-Überdeckungstest subsumiert somit den *all c-uses/some p-uses*- und den *all p-uses/some c-uses*-Überdeckungstest.

Die Subsumierungsrelationen der in diesem Abschnitt vorgestellten Verfahren zur Generierung strukturorientierter Testfälle ist zusammenfassend in Abb. 7.20 dargestellt.

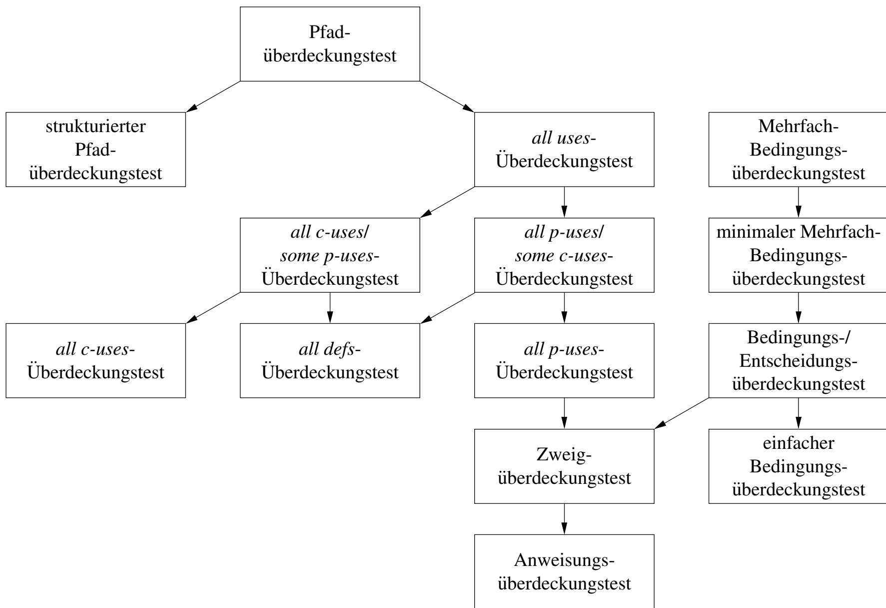


Abb. 7.20. Subsumierungsrelation zwischen den Verfahren zur Generierung strukturorientierter Testfälle

7.3 Formale funktionale Eigenschaftsprüfung von Programmen

Die automatischen Verfahren zur formalen funktionalen Eigenschaftsprüfung von Programmen haben in den vergangenen Jahren enorme Fortschritte verzeichnet. Einen guten Überblick hierüber gibt die Veröffentlichung [140]. Im Folgenden werden in Anlehnung an [140] einige Techniken näher betrachtet.

7.3.1 Statische Programmanalyse

Der Begriff *statische Programmanalyse* beschreibt Techniken, mit deren Hilfe Informationen über das Verhalten von Programmen ermittelt werden können, ohne diese auszuführen. Viele dieser Techniken wurden für die Optimierungsphase in Compilern konzipiert. Dennoch ist deren Anwendungsgebiet nicht auf diese Aufgabe beschränkt. Hier wird statische Programmanalyse im Kontext der formalen Eigenschaftsprüfung von Programmen betrachtet.

Ein grundlegendes Problem in der Programmverifikation ist, dass viele Verifikationsfragen unentscheidbar oder zu rechenintensiv sind, um sie zu beantworten. Deshalb berechnen Techniken zur statischen Programmanalyse typischerweise lediglich Approximationen für Eigenschaften. Diese müssen allerdings Korrektheitsgarantien liefern, d. h. nicht zu Fehlschlüssen verleiten.

Beispiel 7.3.1. Ein statisches Verfahren zur Detektion von „Division durch Null“-Fehlern in einem gegeben Programm muss sämtliche Werte eines Divisors, die zur Laufzeit des Programms auftreten können, berücksichtigen. Da eine Aufzählung aller potentiellen Werte in der Regel aus Zeitgründen nicht möglich ist, arbeiten Verfahren zur statischen Programmanalyse typischerweise mit Teil- bzw. Obermengen. Verwendet die Analyse eine *Teilmenge* aller möglichen Werte und findet dabei keine Fehler, kann keine endgültige Aussage über die Abwesenheit von „Division durch Null“-Fehlern gemacht werden. Liefert das Analyseverfahren dennoch dieses Ergebnis, so kann diese Aussage falsch sein, da die verwendet Approximation inkorrekt ist.

Verwendet das Verfahren hingegen eine *Obermenge* aller möglichen Werte eines Divisors, so ist die Approximation korrekt, und die Aussage, dass keine „Division durch Null“-Fehlern existieren, gerechtfertigt. Allerdings kann eine solche Überapproximation zu *falschnegativen Ergebnissen* führen, d. h. der Fehler tritt lediglich unter Verwendung von Werten auf, die zwar in der Obermenge, nicht aber in der ursprünglichen Menge möglicher Werte des Divisors liegt. In diesem Zusammenhang spricht man auch von *unzulässigen Gegenbeispielen*.

Im Gegensatz zu falschnegativen Ergebnissen können bei der statischen Programmanalyse auch falschpositive Ergebnisse auftreten. Hierbei handelt es sich um Testfälle, die als fehlerfrei angesehen werden, es aber in Wirklichkeit gar nicht sind. Aufgrund der Unentscheidbarkeit statischer Analyseprobleme kann nicht garantiert werden, dass eine Methode sowohl keine falschnegativen als auch keine falschpositiven Ergebnisse erzeugt.

Statische Programmanalyse beruht auf der Idee der Fixpunktberechnung (siehe auch Anhang C.4). Dabei werden Wertemengen solange durch ein Programm propagiert und angepasst, bis diese Mengen sich nicht weiter ändern. Dies wird anhand des Beispiels aus [140] illustriert:

Beispiel 7.3.2. Gegeben ist folgender Ausschnitt eines C-Programms:

```
1      int i = 0;
2      do {
3        assert(i <= 10);
4        i = i+2;
5      } while (i < 5);
```

Es sollen nun alle Werte, welche die Variable `i` annehmen kann, ermittelt werden. Dies ist beispielsweise sinnvoll, wenn `i` als Index für Arrays verwendet wird und ein Zugriff außerhalb des deklarierten Array-Bereichs verhindert werden soll. Der Kontroll-Datenflussgraph zu dem obigen Programmsegment ist in Abb. 7.21 zu sehen. Der Zustand v_2 repräsentiert dabei die Programmzeilen 2 und 3. Der Zustand v_5 repräsentiert das Verlassen des Programmabschnitts und der Zustand v_e den Fehlerzustand, der durch die `assert`-Anweisung erreicht wird.

Links in Abb. 7.21 sind für die ersten beiden Iterationen für jeden Knoten des Kontrollflussgraphen die zugehörige Wertemenge für die Variable `i` angegeben. Diese enthält diejenigen Werte, welche die Variable `i` bei Erreichen des repräsentierten

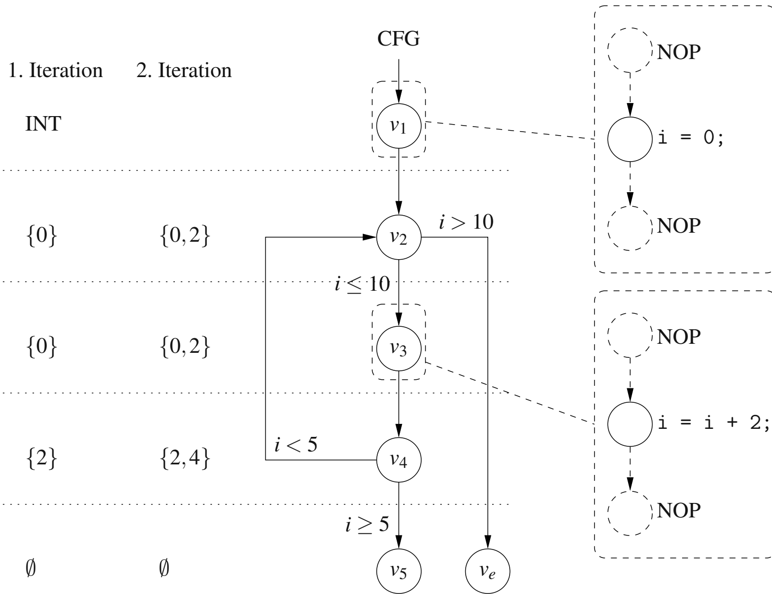


Abb. 7.21. Kontroll-Datenflussgraph zu dem Programmsegment aus Beispiel 7.3.2

Zustands tragen kann. Zu Beginn der ersten Iteration im Zustand v_1 hält die Variable `i` einen beliebigen Wert. Die Wertemenge ist somit die Menge aller Werte in INT, die eine Variable von Typ `int` in der Programmiersprache C annehmen kann. Für die statische Programmanalyse werden nun die Wertemengen entlang der Kontrollflusskanten propagiert und entsprechend der Operationen im Datenflussgraphen manipuliert.

Die beiden ersten Iterationen laufen wie folgt ab:

1. Zu Beginn ist die Variable `i` nicht initialisiert und nimmt somit einen beliebigen Wert aus der Menge INT an. Bei Erreichen von Zustand v_2 wurde `i` bereits initialisiert und besitzt den Wert 0. Somit ergibt sich die Wertemenge $\{0\}$ mit lediglich einem Element. Die `assert`-Anweisung in Zeile 3 ändert an dieser Wertebelegung nichts. Bei Erreichen von Zustand v_4 wurde die Variable `i` bereits um 2 erhöht, weshalb die Wertemenge als einziges Element die 2 enthält. Zustand v_5 und v_e werden in der ersten Iteration aufgrund der Kontrollanweisungen $2 < 5$ bzw. $0 \leq 10$ nicht erreicht.
2. Die Wertemenge $\{2\}$ wird nun von Zustand v_4 zu Zustand v_2 propagiert. Da v_2 mehrere Eingangskanten besitzt, kann sich in diesem Knoten die Wertemenge vergrößern, da die Vereinigung aller ankommenden Wertemengen gebildet werden muss. Das Ergebnis ist somit die Wertemenge $\{0,2\}$, wobei das Element 0 noch aus der ersten Iteration stammt. Da beide Werte kleiner gleich zehn sind, wird die Wertemenge nicht an Zustand v_e propagiert. Man beachte aber auch, dass der Knoten v_1 in diesem Durchlauf nicht wieder betrachtet werden muss,

da sich die dort ermittelte Wertemenge nicht geändert hat. Die Menge $\{0,2\}$ wird nun unverändert weiter an Knoten v_3 propagiert. Dort werden die Elemente um zwei erhöht, weshalb sich für Zustand v_4 die Wertemenge $\{2,4\}$ ergibt. Da beide Elemente echt kleiner fünf sind, wird die Wertemenge nicht an v_5 jedoch aber an v_2 propagiert.

Die Iteration wird solange fortgesetzt, bis ein Fixpunkt erreicht ist, d. h. keine Änderungen in den Wertemengen auftreten.

Die eben gerade beschriebene Programmanalyse wird als *konkrete Interpretation* bezeichnet. Dies liegt darin begründet, dass Elemente aus INT und beliebigen Teilmengen daraus konkrete Werte und somit einen *konkreten Wertebereich* darstellen.

Abstrakte Interpretation

Konkrete Interpretation lässt sich in der Praxis nicht anwenden, da die Wertemengen schnell anwachsen. Bereits 1965 stellte Peter Naur fest, dass es ausreichend sein kann, abstrakte Werte bei der Programmanalyse zu verwenden [341]. Basierend auf diesem Ergebnis entwickelten Cousot und Cousot 1977 die Methode der *abstrakten Interpretation* [118]. Abstrakte Interpretation bedient sich zweier zentraler Konzepte: dem sog. *abstrakten Wertebereich*, der eine Approximation eines konkreten Wertebereichs ist, und sog. *abstrakter Funktionen*, die dazu dienen, konkrete Werte in abstrakte Werte zu übersetzen. Abstrakte Interpretation hat zum Ziel, eine approximative Lösung des Programmanalyseproblems zu liefern. Dabei wird das Verhalten eines Programms für abstrakte Wertebereiche analysiert. Dies geschieht, indem die Operationen auf den konkreten Wertebereichen aus der konkreten Interpretation durch geeignete Funktionen ersetzt werden.

Abstrakte Wertebereiche lassen sich in *relationale* und *nichtrelationale abstrakte Wertebereiche* einteilen. Beispiele für nichtrelationale Wertebereiche sind *Vorzeichen-Wertebereiche*, *Intervall-Wertebereiche* oder *Kongruenz-Wertebereiche*. Der Vorzeichen-Wertebereich besitzt drei Elemente $\{pos, neg, zero\}$ zur Unterscheidung positiver und negativer Zahlen, sowie der Null. Intervall-Wertebereiche sind expressiver, da der Vorzeichen-Wertebereich durch $\{[-\infty,0),[0,0],(0,\infty]\}$ repräsentiert werden kann.

Beispiel 7.3.3. Betrachtet wird wiederum das Programmsegment aus Beispiel 7.3.2 mit dem Kontroll-Datenflussgraphen aus Abb. 7.21. Ein möglicher abstrakter Wertebereich ist z. B. die Menge der Intervalle $\{[a,b] \mid a \leq b \wedge a,b \in \mathbb{Z}\}$. Die in der konkreten Interpretation verwendeten Operationen Addition und Vereinigung müssen für die abstrakte Interpretation ersetzt werden durch Addition und Vereinigung von Intervallen. Die abstrakte Interpretation ist in Abb. 7.22 an dem Beispiel durchgeführt.

Im Folgenden seien *min* bzw. *max* der minimale bzw. maximale Wert, der sich im Wertebereich INT darstellen lässt. In Zustand v_1, zu Beginn der ersten Iteration, ist die Variable `i` noch nicht initialisiert und kann somit einen beliebigen Wert im Intervall $[min, max]$ annehmen. Nach der Initialisierung (Zustand v_2) besitzt `i` den

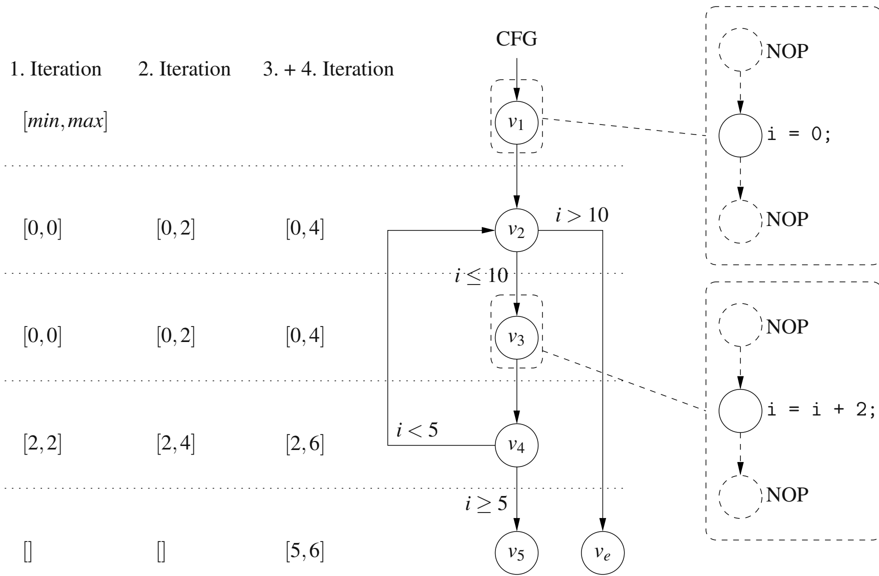


Abb. 7.22. Abstrakte Interpretation mit Intervallen

konkreten Wert 0, was als Intervall $[0,0]$ geschrieben wird. Bis auf die Intervallschreibweise der Wertemengen verläuft die erste Iteration der abstrakten Interpretation äquivalent zur konkreten Interpretation aus Beispiel 7.3.2. Man beachte aber, dass Zuweisung und Addition sich jeweils auf Unter- und Obergrenze des Intervalls auswirkt. Nach Beendigung der ersten Iteration wird das Intervall $[2,2]$ von Zustand v_4 an Zustand v_2 propagiert. Zusammen mit dem Intervall $[0,0]$ aus der Initialisierung von `i` ergibt sich eine Überapproximation des Wertebereichs durch das Intervall $[0,2]$. Man sieht also, dass die Mengenvereinigung in der Intervallarithmetik durch die Minimums- bzw. Maximumsoperation über die Bereichsgrenzen ersetzt wurde. Bei Erreichen von Zustand v_4 in der zweiten Iteration wird die Wertemenge mit dem Intervall $[2,4]$ approximiert.

Zu Beginn der dritten Iteration ergibt sich somit die Wertemenge für v_2 von $[\min\{0,2\},\max\{0,4\}] = [0,4]$. Die Wertemenge für v_4 wird in dieser Iteration mit $[2,6]$ approximiert. In diesem Fall ist die Obergrenze erstmals größer als die Schranke 5, die zur Steuerung der `while`-Schleife verwendet wird. Aus diesem Grund muss die Wertemenge geteilt werden, um eine Fallunterscheidung zu erreichen. Die approximierte Wertemenge $[2,4]$ wird zurück an v_2 propagiert, während das verbleibende Intervall $[5,6]$ an Zustand v_5 übertragen wird. Nach dieser Aufteilung der Wertemenge ist das an v_2 propagierte Intervall aber das selbe, das auch am Ende von Iteration zwei propagiert wurde. Eine vierte Iteration offenbart somit den Fixpunkt in der abstrakten Interpretation.

Da es sich bei der Intervallbildung um eine Überapproximation handelt, kann man auch für jede konkrete Programmausführung schließen, dass die Zusicherung `assert(x <= 10)` immer gültig ist.

Kongruenz-Wertebereiche repräsentieren den Wert $\text{value}(x)$ einer Variablen x als $(\text{value}(x) \bmod k)$ für ein gegebenes k. Für $k = 2$ erhält man den sog. *Paritätswertebereich* $\{odd, even\}$. Man beachte, dass der Intervall-Wertebereich zwar deutlich mehr Elemente enthalten kann, aber dennoch nicht expressiver ist als der Paritätswertebereich, da die geraden und ungeraden Zahlen nicht durch eine endliche Anzahl an Intervallen repräsentiert werden können. Das folgende Beispiel zu Kongruenz-Wertebereichen stammt aus [140].

Beispiel 7.3.4. Betrachtet wird der Ausdruck $1/(x-y)$. Kann unter Verwendung der abstrakten Interpretation mit Kongruenz-Wertebereichen für ein gegebenes, aber beliebiges k gezeigt werden, dass $(x \bmod k) \neq (y \bmod k)$ ist, so kann daraus geschlossen werden, dass eine Division durch Null in dieser Anweisung nicht auftreten kann.

Nicht relationale abstrakte Wertebereiche lassen sich leicht handhaben. Allerdings können mit ihnen bereits Zusicherungen der Form $x \leq y$ nicht mehr überprüft werden. *Relationale abstrakte Wertebereiche* lösen dieses Problem. Eine einfache Form relationaler abstrakter Wertebereiche sind die sog. *differenzenbeschränkten Matrizen* (engl. *difference bound matrices, DBMs*). Diese bestehen aus Konjunktionen von Ungleichungen der Form $x - y \leq c$. Obwohl DBM-Wertebereiche expressiver als Intervall-Wertebereiche sind besteht dennoch eine Einschränkung in der Form, dass Beschränkungen der Art $-x-y \leq c$ nicht ausgedrückt werden können. Oktagon-Wertebereiche beseitigen diese Einschränkung, indem sie Beschränkungen der Form $ax + by \leq c$ erlauben, wobei $a, b \in \{-1, 0, 1\}$ sind. Eine Erweiterung nehmen schließlich Oktaeder-Wertebereiche durch Verallgemeinerung auf mehr als zwei Variablen vor.

Eine der ersten relationalen abstrakten Wertebereiche sind Polyeder-Wertebereiche. Diese werden durch die Konjunktion von Ungleichungen der Form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq c$ beschrieben, wobei $a_1, \ldots, a_n, c \in \mathbb{Z}$ sind. Die Manipulation von Polyeder-Wertebereichen ist allerdings nicht trivial und bedarf der Berechnung sog. *konvexer Hüllen*. Die Komplexität dieser Berechnungen ist exponentiell in der Anzahl der Variablen.

Die Expressivität eines Wertebereichs entscheidet darüber, welche funktionalen Eigenschaften mit abstrakter Interpretation gezeigt werden können. Einige von den oben genannten abstrakten Wertebereichen lassen sich aber nicht hinsichtlich ihrer Expressivität vergleichen. Hier sei nochmals das Beispiel des Paritätswertebereichs und des Intervall-Wertebereichs genannt. Die Vorzeichen-, Intervall-, DBM-, Oktagon-, Oktaeder- und Polyeder-Wertebereiche bilden eine Hierarchie. Obwohl Polyeder-Wertebereiche die Expressivsten in dieser Hierarchie sind, reichen diese nicht aus, um Eigenschaften über Ungleichheiten zu zeigen.

Beispiel 7.3.5. Gegeben ist der Ausdruck $1/(2 \cdot x + 1 - y)$ [140]. Um mittels abstrakter Interpretation zu zeigen, dass keine Division durch Null in dieser Anweisung auftreten kann, muss ein abstrakter Wertebereich gewählt werden, in dem $2 \cdot x + 1 \neq y$

gezeigt werden kann. Dies ist nicht mit den oben beschriebenen relationalen abstrakten Wertebereichen möglich. Der abstrakte Wertebereich der *linearen Kongruenzen* verbindet Polyeder-Wertebereiche mit Kongruenz-Wertebereichen. Diese enthalten auch Gleichungen der Form $a_1x_1 + \cdots + a_nx_n = c \bmod k$ für ein gegebenes k. Damit kann gezeigt werden, dass $(2 \cdot x + 1) \bmod k$ ungleich $y \bmod k$ ist, was auch die Frage nach einer möglichen Division durch Null beantwortet.

7.3.2 SAT-basierte Modellprüfung von C-Programmen

Für die SAT-basierte Modellprüfung (siehe Abschnitt 5.3.2) wird, neben der zu prüfenden Eigenschaft, ein Modell des Programms benötigt. Dieses muss ein endlicher Automat sein, der aus Zuständen und Zustandsübergängen besteht. In einem Programm fasst ein Zustand den Wert des Programmzählers, die Werte der Programmvariablen und den Zustand des Speichers zusammen. Zustandsübergänge beschreiben mögliche Änderungen bei der Programmausführung von einem Zustand zum nächsten.

Da der Kontrollfluss lediglich am Ende eines Grundblocks verzweigen und am Anfang eines Grundblocks eintreten kann, kann es sinnvoll sein, anstatt jede Anweisung einzeln zu codieren, eine Codierung auf Grundblöcken vorzunehmen. Ein solches Verfahren ist in [242] beschrieben.

Beispiel 7.3.6. Gegeben ist das folgende C-Programm [242]:

```
 1     void func1() {
 2       int x = 3;
 3       int y = x - 3;
 4       while (x <= 4) {
 5         y++;
 6         x = func2(x);
 7       }
 8       y = func2(y);
 9     }
10
11     int func2(int s) {
12       int t = s + 2;
13       if (t > 6)
14         t -= 3;
15       else
16         t--;
17       return t;
18     }
```

Der resultierende Kontroll-Datenflussgraph ist in Abb. 7.23 zu sehen. Man erkennt, dass jeder Zustand v_i im Kontrollflussgraphen einem Grundblock im Programm entspricht. Außerdem ist gezeigt, wie Parameter im Falle nichtrekursiver Funktionsaufrufe übergeben werden. Dabei werden Argumente in der Variable `l` übergeben

und mittels der Variablen `r` wird unterschieden, an welche Variable das Ergebnis zu übertragen ist.

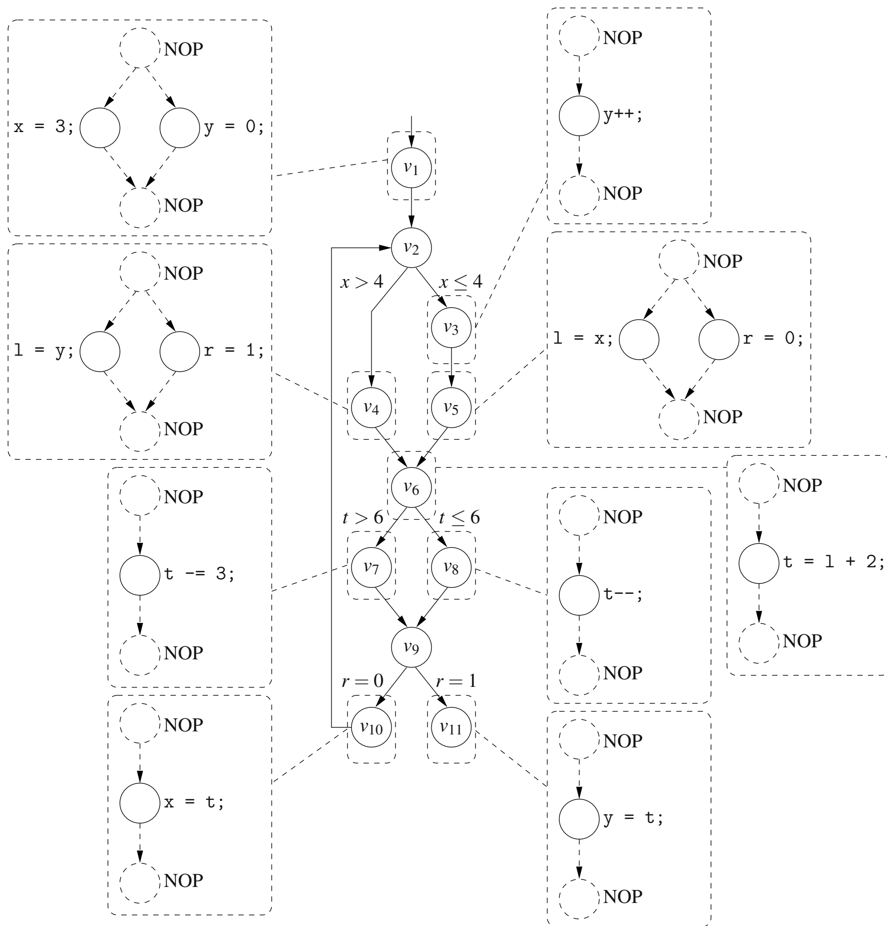


Abb. 7.23. Kontroll-Datenflussgraph für das Programm aus Beispiel 7.3.6

Jedem Grundblock i (Zustand im Kontrollflussgraphen) wird nun eine einzelne binäre Variable $b_i \in \mathbb{B}$ zugeordnet, die anzeigt, ob der Kontrollfluss gerade in diesem Grundblock liegt. Die Belegung der Variablen b_i kann direkt aus dem Programmzähler abgeleitet werden. Man beachte, dass $b_i = \mathsf{T}$ impliziert, dass $\forall_{j \neq i} b_j = \mathsf{F}$ ist.

Nach der Konstruktion des Kontroll-Datenflussgraphen werden zunächst die Datenflussgraphen zu jedem Basisblock einzeln in Booleschen Formeln codiert. Dabei wird für jede Zuweisung `var = expr` eine kombinatorische Schaltung für `expr` er-

stellt, welche sich direkt als Boolesche Funktionen darstellen lässt. Eine Addition wird beispielsweise durch einen Ripple-Carry-Addierer ersetzt. Der Ausdruck V_{ik} bezeichne dann die Zuweisung an die Variable $\mathtt{var}_i$ im Grundblock k. Die Codierung aller Grundblöcke resultiert darin, dass alle Variablen durch endliche Bitbreiten repräsentiert werden. Diese Bitbreiten ergeben sich aus der Mikroarchitektur des verwendeten Prozessors.

Zum Aufstellen der Zustandsübergangsrelation R wird nun für jede Variable $\mathtt{var}_i$ der Folgezustand $\mathtt{var}_i'$ berechnet. Unter der Annahme, dass es n Grundblöcke gibt und die Variable $\mathtt{var}_i$ in den Grundblöcken $1,\ldots,m$ Werte zugewiesen bekommt ($m \leq n$) und in den Blöcken $m+1,\ldots,n$ nicht geschrieben wird, kann dies durch folgenden Ausdruck erfolgen:

$$\mathtt{var}_i' = \left(\bigvee_{j=1}^{m} b_j \wedge V_{ij}\right) \vee \left(\bigvee_{j=m+1}^{n} b_j \wedge \mathtt{var}_i\right)$$

Für die SAT-basierte Modellprüfung muss die Zustandsübergangsrelation abgerollt werden. Wird die Übergangsrelation k mal abgerollt, so ist die resultierende Boolesche Funktion um den Faktor k größer. Für große Programme verbietet sich ein solches Vorgehen, wie das folgende Beispiel aus [140] zeigt.

Beispiel 7.3.7. Abbildung 7.24a) zeigt den Kontrollflussgraphen eines Programms. Jeder Knoten entspricht einem Basisblock und die Kanten zeigen den möglichen Kontrollfluss an, wobei die Bedingungen nicht dargestellt sind. In Abb. 7.24b) ist der 6-fach abgerollte Kontrollflussgraph zu sehen. Man beachte, dass v_1 lediglich in Schritt 0 zu erreichen ist. Weiterhin ist v_2 nur zu den Zeitschritten $1,3,5$ erreichbar.

Zustandsraumreduktion durch Schleifenabwicklung

In Beispiel 7.3.7 wurde gezeigt, dass k-faches Abrollen des Kontrollflussgraphen in einem Zustandsraum resultiert, der viele unerreichbare Zustände enthält. Eine wichtige Beobachtung in diesem Zusammenhang ist, dass v_1, v_4 und v_5 auf jedem Pfad maximal einmal durchlaufen werden. Dennoch sind in Abb. 7.24b) drei erreichbare Instanzen von v_4 und v_5 zu sehen. Eine solche Redundanz kann verhindert werden.

Die grundlegende Idee ist, dass nicht die gesamte Übergangsrelation abgerollt, sondern Schleifen separat abgewickelt werden. Syntaktisch bedeutet dies, dass der Schleifenrumpf repliziert wird und der ursprüngliche Kontrollfluss durch geeignete Wächterfunktionen erhalten bleibt.

Beispiel 7.3.8. Gegeben ist die folgende Schleife:

```
1     while(x)
2       body();
```

Eine zweifache Schleifenabwicklung sieht dann wie folgt aus:

```
1     if(x) {
2       body();
3       if(x)
4         body();
```

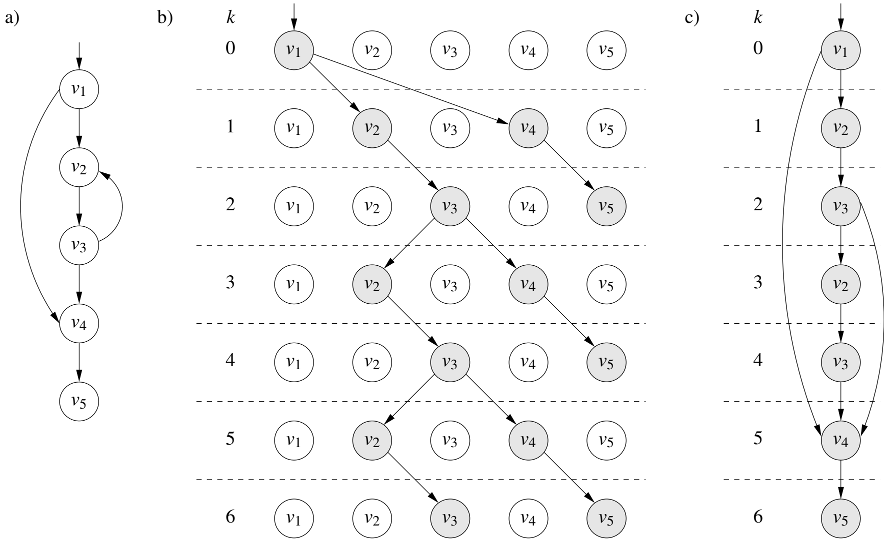


Abb. 7.24. a) Kontrollflussgraph, b) für $k = 6$ abgerollter Kontrollflussgraph und c) Kontrollflussgraph mit abgewickelter Schleife [140]

```
5        else
6          assert(false);
7      }
```

Für das Programm aus Beispiel 7.3.7 kann der Effekt einer zweifachen Schleifenabwicklung in Abb. 7.24c) gesehen werden. Der abgerollte Kontrollflussgraph ist deutlich kompakter als der in Abb. 7.24b) dargestellte Kontrollflussgraph, der sich durch Abrollen der Übergangsrelation ergibt. Der Nachteil ist allerdings, dass durch die Schleifenabwicklung mehr Evaluierungsschritte benötigt werden können, um einen gesuchten Zustand zu erreichen. In Abb. 7.24b) erkennt man nach dem Abrollen von einem Zeitschritt, dass der Zustand v_4 erreichbar ist, während in Abb. 7.24c) hierzu fünf Zeitschritte benötigt werden.

7.3.3 Modellprüfung durch Abstraktionsverfeinerung

Die Unterscheidung von statischer Programmanalyse und Modellprüfung ist historisch entstanden. Während die statische Programmanalyse von Beginn an die Überprüfung einfacher Tatsachen in Programmen zum Ziel hatte, wurden Modellprüfungsverfahren zur Überprüfung temporallogischer Aussagen auf endlichen Automaten entwickelt. Aufgrund der praktischeren Zielsetzung entwickelten sich die Methoden zur statischen Programmanalyse schnell in Richtung Abstraktionstechniken. Dahingegen blieb die Modellprüfung jahrelang eine Technik, die exakte Ergebnisse ohne Überapproximationen liefert. Mittlerweile unterstützen Werkzeuge zur

statischen Programmanalyse auch komplexe Formeln als Spezifikation zu prüfender funktionaler Eigenschaften und Verfahren zur Modellprüfung von Programmen integrieren Abstraktionstechniken. Somit verschwindet die Grenze zwischen statischer Programmanalyse und Modellprüfung zunehmend.

Für die SAT-basierte Modellprüfung von Programmen muss das Programm zunächst in eine Boolesche Formel übersetzt werden. Dies kann wie im vorherigen Abschnitt 7.3.2 vorgenommen werden, wobei der Programmzähler, die globalen Variablen und die Speicherbelegung den Zustand des Programms bilden. Da der Zustandsraum eines Programms durch die Verwendung von komplexen Datentypen sehr groß sein kann, muss man typischerweise bei der Erstellung des Programmmodells davon abstrahieren. Die heutzutage wichtigste Technik dafür ist die sog. *Prädikatenabstraktion*. Die Wahl geeigneter Prädikate ist bei der Prädikatenabstraktion die zentrale Herausforderung. Ein Verfahren, welches auf *Abstraktionsverfeinerung* beruht, wurde von Clarke et al. [100] vorgeschlagen und ist unter dem Namen *CEGAR* (engl. *CounterExample-Guided Abstraction Refinement*) bekannt geworden. Die prinzipielle Idee des Verfahrens ist in Abb. 7.25 zu sehen.

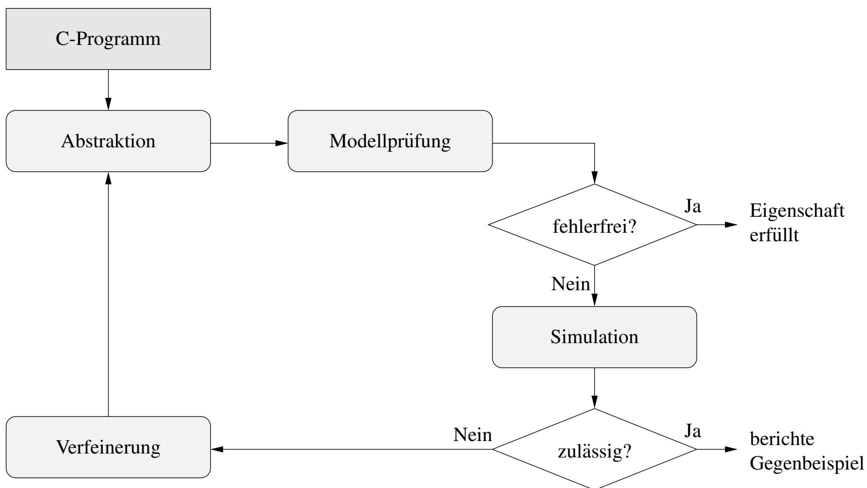


Abb. 7.25. Eine durch Gegenbeispiele gesteuerte Abstraktionsverfeinerung

Die Abstraktionsverfeinerung wird dabei durch Gegenbeispiele, die ein Modellprüfungsverfahren liefert, gesteuert. Dabei wird in einem ersten Schritt eine Prädikatenabstraktion des C-Programms bestimmt. Anschließend wird das Modell dahingehend überprüft, ob es die zu prüfende Eigenschaft erfüllt. Ist dies der Fall, ist die Modellprüfung erfolgreich abgeschlossen, d. h. das C-Programm erfüllt die geforderte Eigenschaft. Liefert die Modellprüfung jedoch ein negatives Ergebnis, muss zunächst geprüft werden, ob das gefundene Gegenbeispiel überhaupt einem zulässigen Berechnungspfad im C-Programm entspricht. Dies ist notwendig, da durch die

Überapproximation falschnegative Ergebnisse entstehen können. Die Überprüfung kann beispielsweise durch die Simulation des Gegenbeispiels mit dem Programm erfolgen. Ist das gefundene Gegenbeispiel zulässig, kann es mit dem C-Programm reproduziert werden. Somit wurde ein echtes Gegenbeispiel gefunden und das C-Programm erfüllt die geforderte Eigenschaft nicht. Ist das Gegenbeispiel hingegen nicht zulässig, muss verhindert werden, dass die Modellprüfung in einer weiteren Iteration den gleichen Fehler im abstrakten Modell findet, der ja im konkreten Programm nicht vorliegt. Dies erfolgt dadurch, dass die Überapproximation verfeinert wird.

Die Abstraktionsverfeinerung im CEGAR-Ansatz läuft also iterativ in vier Schritten ab: Abstraktion, Modellprüfung, Simulation und Verfeinerung. Diese Schritte werden im Folgenden näher diskutiert (siehe auch [140]):

Abstraktion

C-Programme bestehen aus Sequenzen von Anweisungen $\langle i_1, i_2, \ldots \rangle$. Durch die Ausführung einer Instruktion i wird der momentane Programmzustand in einen Folgezustand überführt. Dies wird durch die Relation R_i beschrieben, die für Programmzustände Folgezustände, die aus der Ausführung von i resultieren, bestimmt. Die Vereinigung aller Relationen R_i ist die Übergangsrelation R des Programms, welche die Grundlage für die Modellprüfung bildet.

Da für reale Programme die Übergangsrelation R zu komplex werden würde, muss zunächst eine Abstraktion $\hat{R}$ bestimmt werden. Dies kann mittels Prädikatenabstraktion erfolgen, wobei Prädikate über Programmvariablen gebildet werden. Dabei bipartitioniert ein Prädikat p die Zustandsmenge des Programms in Zustände, in denen $p = \mathsf{T}$ ist, und Zustände, in denen $p = \mathsf{F}$ gilt. Durch Verwendung mehrerer Prädikate werden die Partitionen weiter unterteilt. Jede Partition bildet dann einen *abstrakten Zustand*. Werden also n Prädikate gebildet, so wird der Zustandsraum in 2^n abstrakte Zustände partitioniert. Jeder abstrakte Zustand entspricht dann einer der 2^n möglichen Variablenbelegungen. Dieses Vorgehen entspricht dem der funktionalen Äquivalenzklassenbildung zur funktionsorientierten Testfallgenerierung (siehe Abschnitt 7.2.1).

Seien $\hat{s}_0$ und $\hat{s}_1$ abstrakte Zustände. Es gilt dann: Es gibt einen Zustandsübergang in $\hat{R}$, wenn es konkrete Zustände s_0 und s_1 in den zugehörigen Partitionen gibt, die durch einen Zustandsübergang miteinander verbunden sind, d. h.

$$(\hat{s}_0, \hat{s}_1) \in \hat{R} \Rightarrow \exists s_0 \in \hat{s}_0, s_1 \in \hat{s}_1 : (s_0, s_1) \in R.$$

Dies wird als *existentielle Abstraktion* bezeichnet, die korrekte Abstraktionen für Erreichbarkeitsanalysen liefert [101]. Das resultierende abstrakte Programm, welches sich aus $\hat{R}$ ergibt, wird durch ein sog. *Boolesches Programm* repräsentiert [27]. Dieses Boolesche Programm besitzt die selben Kontrollanweisungen wie das originale C-Programm, allerdings sind alle verwendeten Variablen nun binär.

Beispiel 7.3.9. Betrachtet wird ein Programm ähnlich wie in Beispiel 7.3.2. In Zeile 4 ist das Inkrement um 2 durch ein Inkrement um 1 ersetzt. Das neue Programm sieht

somit wie folgt aus:

```
1     int i = 0;
2     do {
3       assert(i <= 10);
4       i++;
5     } while (i < 5);
```

Der zugehörige Kontroll-Datenflussgraph ist in Abb. 7.26 zu sehen.

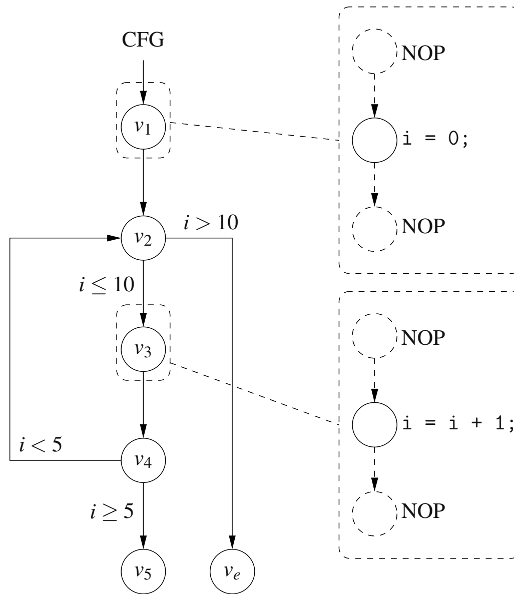


Abb. 7.26. Kontroll-Datenflussgraph für das Programm aus Beispiel 7.3.9

Prädikatenabstraktion für diesen Programm lässt sich durch das Prädikat $i = 0$ erreichen. Der Wert des Prädikates wird in der binären Variablen $b_1 \in \mathbb{B}$ repräsentiert. Abb. 7.27a) zeigt den Kontroll-Datenflussgraphen für das resultierende Boolesche Programm. Zunächst wird im Datenflussgraphen im Zustand v_1 die Zuweisung $b_1 := \mathsf{T}$ vorgenommen, was dem Prädikatenwert entsprechend der Zuweisung `i = 0` im C-Programm entspricht. Die Bedingungen der Verzweigung im Zustand v_2 müssen nun auch ausschließlich mit der Variablen b_1 erfolgen. Da der Übergang in den Fehlerzustand v_e nur eintreten kann, wenn $i > 10$ ist, muss in diesem Fall b_1 zwingend gleich F sein. Der Fall $i \leq 10$ schließt auch $i = 0$ ein, was zur Folge hat, dass die Bedingung zur Konstanten T wird, also überhaupt nicht von b_1 abhängt. Die Prädikatenabstraktion für die Konditionale in der Verzweigung in Zustand v_4 werden auf die gleiche Art bestimmt.

Die Berechnung des Inkrements im Zustand v_3 muss nun ebenfalls nur mit b_1 erfolgen. Besitzt die Variable i bei Erreichen der Inkrement-Anweisung den Wert 0

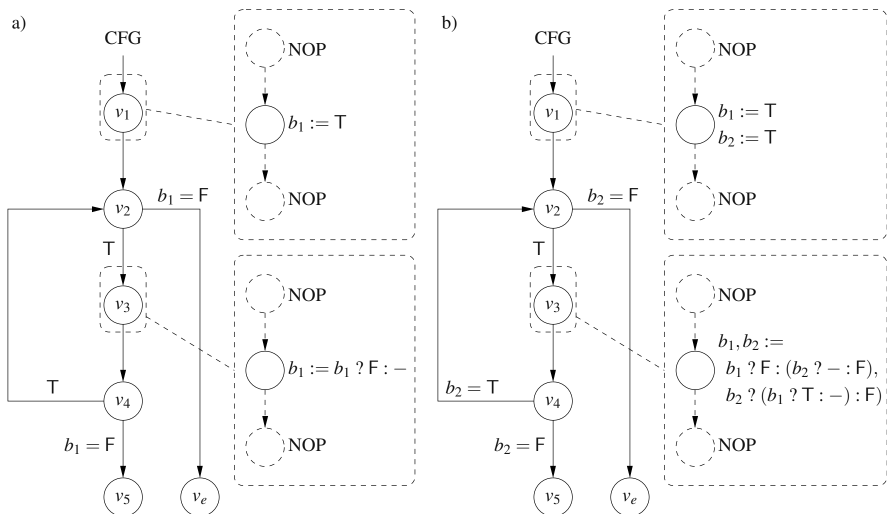


Abb. 7.27. Kontroll-Datenflussgraphen für die Prädikatenabstraktion a) $i = 0$ und b) $i = 0$ und $i < 5$

($b_1 = \mathsf{T}$), so wird sie anschließend den Wert $i = 1$ tragen. Durch die Prädikatenabstraktion wird daraus $b_1 := \mathsf{F}$. Besitzt hingegen die Variable i einen beliebigen von null verschiedenen Wert, d. h. $b_1 = \mathsf{F}$, so kann keine Aussage über den neuen Wert von b_1 getroffen werden: Variable i könnte den Wert -1 haben, dann ist $b_1 := \mathsf{T}$, in allen anderen Fällen ist $b_1 := \mathsf{F}$. Dies wird durch das Symbol „–“ dargestellt. Diese konditionale Zuweisung wird mittels der $c\ ?\ a : b$-Operation ausgedrückt, wobei c das Konditional und a das Ergebnis für den Fall $c = \mathsf{T}$ und b das Ergebnis für den Fall $c = \mathsf{F}$ ist.

Man beachte, dass in dem resultierenden Booleschen Programm alle Programmzustände erreichbar geblieben sind, die auch im C-Programm erreichbar waren. Außerdem handelt es sich bei dem Booleschen Programm um eine Überapproximation des C-Programms, da beispielsweise im Booleschen Programm unendlich viele Pfade zum Zustand v_5 führen, während es im C-Programm lediglich einen Pfad gibt. Durch Wahl weiterer Prädikate kann das Boolesche Programm verfeinert werden.

Es wird nun, zusätzlich zum Prädikat $i = 0$, noch das Prädikat $i < 5$ betrachtet (repräsentiert durch b_2). Der Kontroll-Datenflussgraph des resultierenden Booleschen Programms ist in Abb. 7.27b) dargestellt. Durch die parallele Zuweisung an die Variablen b_1 und b_2 im Zustand v_3 sind von einem Programmzustand $\neg(i = 0) \wedge (i < 5)$ Programmzustände $\neg(i = 0) \wedge \neg(i < 5)$ und $(i = 0) \wedge (i < 5)$ erreichbar. Allerdings ist von einem Programmzustand $\neg(i = 0) \wedge \neg(i < 5)$ kein Programmzustand $\neg(i = 0) \wedge (i < 5)$ erreichbar, da dies gegen die Definition des Inkrements verstoßen würde.

Bei Verwendung von n Prädikaten ergeben sich 2^n abstrakte Zustände und es werden $(2^n)^2$ Schritte benötigt, um die Abstraktion zu berechnen.

Modellprüfung

Die Modellprüfungsphase ist oftmals der Flaschenhals in der auf Prädikatenabstraktion basierenden Verifikation. Der Grund hierfür ist, dass SAT-basierte Verfahren zur Modellprüfung im Allgemeinen nicht geeignet sind, um Fixpunkte zu detektieren (siehe auch Abschnitt 5.3.2). Deshalb müssen BDD-basierte Verfahren zum Einsatz kommen, die aufgrund des exponentiellen Speicherbedarfs nicht skalieren und für realistische Systeme im Allgemeinen nicht anwendbar sind. Eine Alternative stellen sog. *QBF-Solver* (quantifizierte Boolesche Funktionen) dar. Diese haben allerdings ähnliche Probleme wie BDD-basierte Verfahren.

Ohne diese Problematik weiter zu vertiefen, wird anhand eines Beispiels [140] illustriert, wie die Erreichbarkeitsanalyse durchgeführt werden kann.

Beispiel 7.3.10. Betrachtet wird der Kontroll-Datenflussgraph aus Abb. 7.27b). Nach dem ersten Zustandsübergang von v_1 nach v_2 sind beide Variablen b_1 und b_2 mit dem Wert $\top$ belegt. Die symbolische Repräsentation des Programmzustands lautet somit $b_1 \wedge b_2$. Nach Erreichen des Zustands v_4 ist der abstrakte Zustand gegeben durch $\neg b_1 \wedge b_2$. Durch das Zurückpropagieren zu Zustand v_2 ergibt sich nun für v_2 der Zustand $(b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2)$. Durch Anwendung der Absorptionsregel ergibt dies den abstrakten Zustand b_2. Bei einer weiteren Iteration wird über die Zustände v_2 und v_3 somit der Zustand $\top$ in v_4 erreicht. Davon werden aber lediglich $(b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2)$ an v_2 und $(b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge \neg b_2)$ an v_5 propagiert. Dies ist offensichtlich ein Fixpunkt in der Erreichbarkeitsanalyse.

Interessant ist in diesem Zusammenhang, dass das Erreichbarkeitsproblem von Booleschen Programmen entscheidbar ist, und zwar, obwohl der Programmstack potentiell unendlich groß ist. Die Erklärung ist, dass für einen gegeben Zustand der resultierende Folgezustand aus dem obersten Element auf dem Stack und der Belegung der globalen Variablen folgt, was als endliche Menge darstellbar ist.

Simulation

Findet die Erreichbarkeitsanalyse einen Pfad vom abstrakten Anfangszustand zu einem abstrakten Zustand, der einen Fehlerfall darstellt, so muss überprüft werden, ob es sich bei diesem Pfad um ein zulässiges Gegenbeispiel handelt. Dies bedeutet, es muss überprüft werden, ob eine konkrete Testfalleingabe für das C-Programm existiert, so dass vom konkreten Anfangszustand ein Fehlerzustand erreichbar ist. Da eine konkrete Testfalleingabe gesucht wird, spricht man auch von der Simulation des Programms.

Beispiel 7.3.11. In Abb. 7.27a) ist der Fehlerzustand v_e über den Pfad $\langle v_1, v_2, v_3, v_4, v_2, v_e \rangle$ erreichbar. Die verwendete Abstraktion ist in diesem Fall durch das einzelne Prädikat $i = 0$ gegeben. Dieses Gegenbeispiel ist allerdings nicht zulässig, da der Wert der Variable `i` in der Simulation beim zweiten Erreichen von v_2 den Wert 1

trägt und somit kein Übergang in den Zustand v_e möglich ist. Für einen solchen Übergang müsste die Bedingung $i > 10$ erfüllt sein.

Die Erkennung unzulässiger Gegenbeispiele verlangt nach einer Verfeinerung des abstrakten Modells, so dass dieses unzulässige Gegenbeispiel in einem folgenden Verifikationsschritt nicht wieder auftreten kann.

Verfeinerung

Ein Grund für die Entstehung unzulässiger Gegenbeispiele besteht darin, dass die gewählte Prädikatenmenge nicht aussagekräftig genug gewählt wurde. Dieses Problem kann durch Hinzunahme weiterer Prädikate gelöst werden. Das folgende Beispiel stammt wiederum aus [140].

Beispiel 7.3.12. Das unzulässige Gegenbeispiel in Beispiel 7.3.11 führte zu der Ausführungssequenz $\langle \texttt{i} = 0, [i \leq 10], \texttt{i}++, [i < 5], [i > 10] \rangle$. Hierfür können die sog. *schwächste Vorbedingung* und die *stärkste Nachbedingung* bestimmt werden. Die schwächste Vorbedingung ist die schwächste aller möglichen Bedingungen, die vor Beginn der Ausführung der Sequenz gelten müssen, so dass $(i < 5) \wedge (i > 10)$ nach Ausführung der Sequenz gilt. In diesem Fall ist dies $(i < 5) \wedge (i > 10)$. Die stärkste Nachbedingung ist die stärkste aller möglichen Bedingungen, die nach Ausführung der Sequenz gelten muss, so dass vor Ausführung der Sequenz die Bedingung $(i = 0) \wedge (i \leq 10)$ gilt. Dies ist $(i = 1) \wedge (i \geq 5)$. Sowohl die schwächste Vorbedingung als auch die stärkste Nachbedingung sind inkonsistent und somit ein Beweis, dass das abstrakte Gegenbeispiel unzulässig ist. Für die Verfeinerung ist es nun beispielsweise ausreichend, die neuen Prädikate $i = 1$ und $i < 5$ mit aufzunehmen, um dieses Gegenbeispiel von zukünftigen Verifikationsphasen auszuschließen.

Wie Abb. 7.27b) zeigt, ist die Abstraktionsverfeinerung mit dem Prädikat $i < 5$ hinreichend, um das Nichterreichen von v_e im resultierenden Booleschen Programm zu zeigen. In der Tat wäre es sogar ausreichend gewesen, nur $i < 5$ als Prädikat von Beginn an zu verwenden, was zu einer korrekten Abstraktion geführt hätte.

Man sieht also, dass die Auswahl geeigneter Prädikate durchaus nicht leicht ist. Aus diesem Grund werden heutzutage im Wesentlichen Heuristiken verwendet, um möglichst kleine Prädikatenmengen zu generieren, die ein Gegenbeispiel erklären.

7.4 Zeitanalyse

Für die nichtfunktionale Eigenschaftsprüfung von Zeitaspekten der Software wird, wie im Fall der Hardware-Verifikation, davon ausgegangen, dass das Problem zweistufig ist, d. h., dass zunächst das Zeitverhalten der Software analysiert und die Ergebnisse dieser Zeitanalyse mit den nichtfunktionalen Anforderungen der Spezifikation verglichen werden. Auf Modulebene liegen diese Anforderungen typischerweise als sog. engl. *Deadlines* für einzelne Software-Prozesse vor. Dabei spezifiziert eine Deadline die maximale Zeitspanne von Aktivierung eines Prozesses bis zu dessen

Abarbeitung (Beendigung). Eine Zeitanalyse auf Modulebene hat somit die Aufgabe, für alle Prozesse die maximalen Zeitspannen von deren Aktivierung bis deren Beendigung, unter Berücksichtigung von möglichen Unterbrechungen durch andere Prozesse, zu bestimmen. Da es sich hierbei um die Überprüfung von Echtzeitanforderungen handelt, spricht man auch von *Echtzeitanalyse*.

Zur Durchführung der Echtzeitanalyse ist es aber notwendig, neben den Unterbrechungszeiten durch andere Prozesse, die Ausführungszeit jedes Prozesses auf dem Prozessor zu kennen für den Fall, dass der Prozess nicht unterbrochen wird. Diese Zeiten werden als engl. *Core Executiuon Times* (*CET*) bezeichnet. Man unterscheidet zwischen einer kürzesten (engl. *Best Case Execution Time, BCET*) und einer längsten Ausführungszeit (engl. *Worst Case Execution Time, WCET*). Sowohl BCETs als auch WCETs von Prozessen hängen vom Kontrollfluss des Prozesses, der Mikroarchitektur und dem Anfangszustand des verwendeten Prozessors sowie der Eingabe ab.

Im Folgenden werden zunächst Verfahren zur BCET- und WCET-Analyse vorgestellt. Anschließend erfolgt eine Einführung in Echtzeitanalysemethoden auf Modulebene.

7.4.1 BCET- und WCET-Analyse

Für die Zeitanalyse auf höheren Abstraktionsebenen ist es notwendig, die Ausführungszeiten aller Prozesse zu kennen. Genauer formuliert, müssen die kürzesten (engl. *Best Case Execution Times, BCET*) und die längsten Ausführungszeiten (engl. *Worst Case Execution Time, WCET*) jedes Prozesses auf dem gegebenen Prozessor unter der Annahme bekannt sein, dass der Prozess nicht unterbrochen oder blockiert wird. Diese Zeiten hängen von verschiedenen Faktoren ab: Zum einen vom Kontrollfluss des Prozesses sowie den Eingabedaten, zum anderen ist die Mikroarchitektur und der Anfangszustand des Prozessors (inklusive Cache) entscheidend.

Abbildung 7.28 zeigt eine Verteilung der Ausführungszeit eines Prozesses. Die kürzeste Ausführungszeit ist die BCET, während die längste Ausführungszeit die WCET ist. Verfahren zur Bestimmung von BCET und WCET eines Prozesses können simulativ oder formal sein. Stand der Technik im industriellen Umfeld sind simulative Methoden, zu denen auch messbasierte Verfahren zählen. Die Simulation verwendet eine Reihe von Stimuli, um BCET und WCET eines Prozesse zu bestimmen. Das Ergebnis ist aber im Allgemeinen eine *kleinste beobachtete Ausführungszeit*, die eine Überapproximation der BCET ist, und eine *größte beobachtete Ausführungszeit*, die eine Unterapproximation der WCET ist. Aufgrund der Unvollständigkeit simulativer Methoden sind diese im Allgemeinen nicht für das Verifikationsziel des Beweises geeignet, wie in Abb. 7.28 zu sehen ist.

Für die Anwendung formaler Zeitanalysemethoden muss aufgrund der hohen Rechenzeit eine Abstraktion des Systems betrachtet werden. Aus diesem Grund erhält man bei Einsatz formaler Methoden *untere* und *obere Zeitschranken*. Diese müssen einerseits sicher sein, d. h. sie dürfen zu keinen falschpositiven Ergebnissen der Eigenschaftsprüfung führen, wie im Fall der simulativen Analyse, andererseits müssen

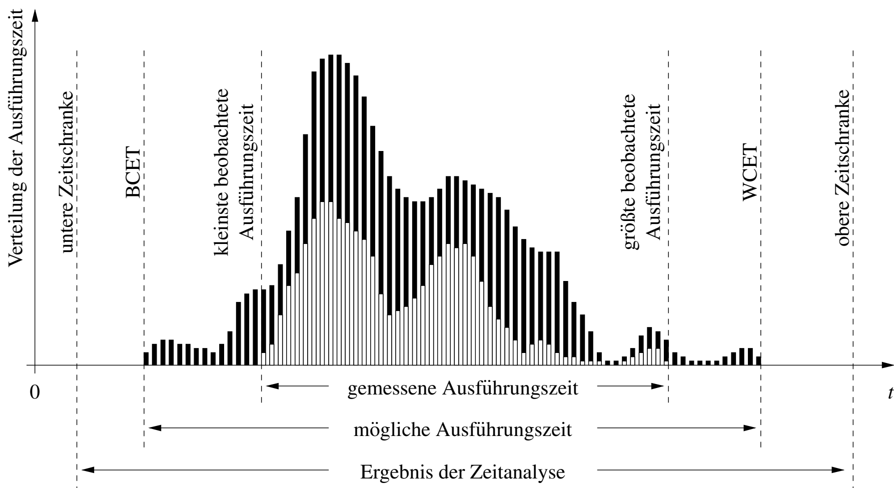


Abb. 7.28. Verteilungsfunktion der Ausführungszeiten eines Prozesses [464]

diese Schranken möglichst genau BCET und WCET widerspiegeln, um falschnegative Ergebnisse bei der Verifikation des Zeitverhaltens zu vermeiden. Im Folgenden wird die formale BCET/WCET-Analyse vorgestellt.

Da BCET und WCET eines Prozesses von dessen Kontrollfluss und den Eingabedaten sowie der Mikroarchitektur und -zustand abhängen, kann die formale BCET/WCET-Analyse in zwei Teilprobleme aufgeteilt werden:

1. *Programmpfadanalyse*: Zum einen muss bestimmt werden, welcher Ausführungspfad, und somit welche Eingabedaten, zu einer schnellsten bzw. langsamsten Ausführung eines Prozesses führen. Der Ausführungspfad ist dann eine Sequenz an Instruktionen, die durchlaufen werden.
2. *Modellierung der Zielarchitektur*: Architekturmerkmale, die Einfluss auf das Zeitverhalten der Pfadausführung haben, müssen berücksichtigt werden. Zu diesen Merkmalen zählen Pipelines, Caches, spekulative Instruktionsausführung etc. Weiterhin hat der Zustand, in dem sich der Prozessor befindet, einen entscheidenden Einfluss auf die Ausführungszeit. Der Zustand muss betrachtet werden, da mehrere Prozesse auf ein und dem selben Prozessor ausgeführt werden können.

Diese beiden Probleme sind voneinander abhängig, d. h. es gibt nicht zwangsläufig einen Ausführungspfad für den Prozess, der für alle möglichen Mikroarchitekturen und Zuständen die schnellste bzw. langsamste Ausführungszeit liefert.

Programmpfadanalyse

Zunächst wird die WCET-Analyse unabhängig von der Architekturmodellierung vorgestellt. Aufgabe der Programmpfadanalyse ist es, zu bestimmen, welche Ausfüh-

rungsreihenfolge der Instruktionen zu einem bestmöglichen bzw. schlechtestmöglichen Zeitverhalten führt. Für die Analyse wird angenommen, dass der Prozess den Prozessor exklusiv belegt, der Prozessor lediglich eine skalare Funktionseinheit besitzt, keine Interrupts unterstützt und kein Betriebssystem auf dem Prozessor ausgeführt wird. Weiterhin wird angenommen, dass das Programm keine rekursiven Funktionsaufrufe beinhaltet, keine dynamische Speicherallokation durchführt und die Schleifen im Programm beschränkt sind bzw. beschränkt werden. Grundlage für die Programmpfadanalyse bildet der Kontrollflussgraph $G_C = (V_C, E_C)$ eines Prozesses. Die Knoten in dem Kontrollflussgraphen stellen Berechnungen (Grundblöcke) dar, während Kanten Kontrollflussabhängigkeiten darstellen. Auf Basis des Kontrollflussgraphen eines Prozesses lässt sich die engl. *Worst Case Execution Time* (WCET) formal definieren:

Definition 7.4.1 (WCET). *Ein Prozess besteht aus n Grundblöcken, wobei jeder Grundblock b_i eine Ausführungszeit δ_i hat und maximal x_i mal ausgeführt wird. Dann ist die WCET*

$$\delta_{\max} := \sum_{i:=1}^{n} \delta_i \cdot x_i \tag{7.6}$$

Somit besteht die Aufgabe der Programmpfadanalyse darin, herauszufinden, wie oft jeder Grundblock auf einem Pfad, der zu einer maximal langen Ausführungszeit gehört, ausgeführt wird. Die Bestimmung der BCET kann analog formuliert werden: Wie oft wird jeder Grundblock auf einem Pfad, der zu einer minimalen Ausführungszeit gehört, ausgeführt. Aufgrund dieser Analogie wird im Folgenden lediglich das WCET-Problem weiter betrachtet. Die zentrale Herausforderung bei der Programmpfadanalyse besteht allerdings darin, dass die Anzahl der möglichen Ausführungspfade exponentiell anwächst.

Ohne weitere Beschränkungen würde Gleichung (7.6) beliebig anwachsen, d. h. $\delta_{\max} \to \infty$. Aus diesem Grund wird die Anzahl der Ausführungen x_i eines Basisblocks B_i beschränkt. Dabei werden zwei Arten von Beschränkungen unterschieden:

1. *Strukturelle Beschränkungen* ergeben sich aus der Struktur des Kontrollflussgraphen.
2. *Funktionale Beschränkungen* ergeben sich aus der Spezifikation des Prozesses.

Strukturelle Beschränkungen werden auch als *Flussbeschränkungen* bezeichnet. Sie besagen, dass jedes Mal, wenn der Kontrollfluss einen Grundblock erreicht, der Kontrollfluss diesen Grundblock auch wieder verlassen muss.

Sei $d : E_C \to \mathbb{Z}_{\geq 0}$ eine Funktion, die jeder Kante $e \in E_C$ im Kontrollflussgraph G_C einen Wert zuweist, der anzeigt, wie oft der Kontrollfluss während einer Ausführung über diese Kante gegangen ist. Sei weiterhin $\text{in}(v) := \{e \mid e = (\tilde{v}, v) \in E_C\}$ die Menge eingehender Kanten in den Kontrollflussknoten $v \in V_C$ und $\text{out}(v) := \{e \mid e = (v, \tilde{v}) \in E_C\}$ die Menge ausgehender Kanten. Dann sieht die strukturelle Beschränkung für Knoten $v_i \in V_C$ der den Grundblock b_i repräsentiert wie folgt aus:

$$\sum_{e_j \in \text{in}(v_i)} d(e_j) = \sum_{e_k \in \text{out}(v_i)} d(e_k) = x_i \tag{7.7}$$

Beispiel 7.4.1. Gegeben ist das Programm in Abb. 7.29a) mit dem in Abb. 7.29b) dargestellten Kontrollflussgraphen. Die Flussgleichungen ergeben sich zu:

$$
\begin{aligned}
x_1 &= d_1 = d_2 \\
x_2 &= d_2 + d_8 = d_3 + d_9 \\
x_3 &= d_3 = d_4 + d_5 \\
x_4 &= d_4 = d_6 \\
x_5 &= d_5 = d_7 \\
x_6 &= d_6 + d_7 = d_8 \\
x_7 &= d_9 = d_{10}
\end{aligned}
$$

a) C-Programm

```
 1  s = k;
 2  while (k<10) {
 3      if (ok)
 4          j++;
 5      else {
 6          j=0;
 7          ok=true;
 8      }
 9      k++;
10  }
11  r=j;
```

b) CFG:

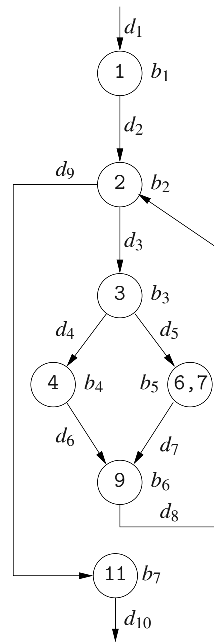


Abb. 7.29. a) Programm und b) Kontrollflussgraph

Obwohl der Lösungsraum durch die Verwendung struktureller Beschränkungen eingeschränkt ist, können immer noch unendliche große Ausführungszeiten entstehen. Durch die Verwendung funktionaler Beschränkungen wird dies verhindert. Funktionale Beschränkungen ergeben sich aus der Spezifikation des Systems oder aus einer Programmanalyse. Funktionale Beschränkungen begrenzen somit die Anzahl an Schleifendurchläufen.

Beispiele für das Programm aus Abb. 7.29 sind:

- Die `while`-Schleife wird maximal zehnmal durchlaufen: $x_3 \leq 10 \cdot x_1$.
- Der Grundblock b_5 wird maximal einmal durchlaufen: $x_5 \leq 1$.

Durch funktionale Beschränkungen werden unzulässige Ausführungspfade aus der Betrachtung bei der Ermittlung der WCET herausgenommen.

Mit Hilfe von strukturellen und funktionalen Beschränkungen lässt sich die WCET-Analyse als ganzzahliges lineares Programm (engl. *Integer Linear Program, ILP*) formulieren:

$$\delta_{\max} := \max \left\{ \sum_{i:=1}^{n} \delta_i \cdot x_i \mid \mathcal{B}_{\text{struct}} \wedge \mathcal{B}_{\text{func}} \right\} \tag{7.8}$$

Dabei bezeichnet $\mathcal{B}_{\text{func}}$ die funktionalen Beschränkungen. Die strukturellen Beschränkungen $\mathcal{B}_{\text{struct}}$ lauten wie folgt:

$$(d_1 = 1) \wedge \bigwedge_{i:=1}^{n} \left(\sum_{e_j \in \text{in}(v_i)} d(e_j) = \sum_{e_k \in \text{out}(v_i)} d(e_k) = x_i \right)$$

Modellierung der Zielarchitektur

Der verwendete Prozessor, aber auch der Zustand in dem sich ein Prozessor bei der Prozessausführung befindet, hat großen Einfluss auf die Ausführungszeit eines Grundblocks und somit des Prozesses. Um diese Einflüsse berücksichtigen zu können, ist es notwendig, eine Modellierung der Zielarchitektur vorzunehmen, und auf Basis dieser Modelle die Abschätzung für eine gegebene Mikroarchitektur durchzuführen. Besondere Schwierigkeiten werden dabei durch dynamische Effekte der Fließbandverarbeitung und der Caches, sowie durch Optimierungen der Compiler verursacht.

Um die Ausführungszeit eines einzelnen Grundblocks oder einer Sequenz von Grundblöcken zu schätzen, gibt es prinzipiell zwei Möglichkeiten:

1. *ITA*: (engl. *Instruction Timing Addition*) ist eine Methode ähnlich der statischen Zeitanalyse von kombinatorischen Schaltungen (siehe Abschnitt 6.5). Dabei wird angenommen, dass jede Instruktion eine konstante Ausführungszeit besitzt. Durch Summation aller Instruktionen in einem Grundblock erhält man somit die Ausführungszeit für den gesamten Grundblock.
2. *PSS*: (engl. *Path Segment Simulation*) basiert auf der zyklenakkuraten Simulation eines Basisblocks, wobei die zugrundeliegende Mikroarchitektur sehr genau modelliert werden kann.

Um den Einfluss von Compiler-Optimierungen möglichst klein zu halten, erfolgt die Zeitschätzung auf compilierten Programmen.

Die wesentlichen Architekturmerkmale bei der Modellierung der Zielarchitektur sind [154]:

- *Datenabhängige Ausführungszeiten der Instruktionen*: Datenabhängige Ausführungszeiten treten überwiegend in CISC-Architekturen (engl. *Complex Instruction Set Computer*) auf. Aber auch Prozessoren, die manche Instruktionen als Software-Funktionen realisieren, sind von diesem Aspekt betroffen. Da die Ausführungszeit einer Instruktion in diesem Fall von den Operanden abhängt, ist auch die Ausführungszeit des Grundblocks von den Eingabedaten abhängig. Während die ITA-Methode unterschiedliche Ausführungszeiten einer Instruktion unterstützen kann, ist dies in der PSS-Methode aufwendiger.
- *Fließbandverarbeitung*: Die ITA-Methode kann Zeiteffekte durch die verschränkte Ausführung von Instruktionen nicht berücksichtigen. Im Gegensatz dazu unterstützen die meisten PSS-Methoden die Modellierung von Pipelines (Fließbandverarbeitung). Problematisch ist allerdings, wenn die Pipeline-Tiefe so groß ist, dass selbst Grundblöcke überlappend ausgeführt werden können. Dies ist typischerweise bei nahezu allen aktuellen Prozessoren der Fall. Aus diesem Grund sollte die PSS-Methode nicht lediglich auf Grundblöcke, sondern auf längere Pfadsegmente angewendet werden. Dabei gilt: Je länger das Pfadsegment, desto akkurater ist das Ergebnis.
- *Superskalare Architekturen*: Bei superskalaren Mikroarchitekturen können mehrere Grundblöcke parallel berechnet und Instruktionen dynamisch geplant werden. Die Analyse basierend auf der ITA-Methode kann hier nicht eingesetzt werden. Die Genauigkeit beim Einsatz der PSS-Methode hängt wiederum von der Pfadlänge ab.
- *Instruktionscaches*: Bei Verwendung eines Instruktionscaches werden die Instruktionen eines Grundblocks b_i auf l_i Cachezeilen abgebildet. Jede Cachezeile führt dann abhängig von der Cachearchitektur und der Verdrängungsstrategie entweder zu einem Cache-Hit oder einem Cache-Miss: Als Erweiterung des ILP aus Gleichung (7.8) kann dies wie folgt modelliert werden:

  $$\forall i : \forall 1 \leq j \leq l_i : x_i = x_{i,j} = x_{i,j}^{hit} + x_{i,j}^{miss} \tag{7.9}$$

  Sei $\delta_{i,j}^{hit}$ die Ausführungszeit einer Instruktion im Falle, dass ein Cache-Hit vorliegt, und $\delta_{i,j}^{miss}$ die Ausführungszeit, falls ein Cache-Miss auftritt. Dann lässt sich die Kostenfunktion des ILP wie folgt ändern:

  $$\delta_{\max} := \max \left\{ \sum_{i:=1}^{n} \sum_{j:=1}^{l_i} \left(\delta_{i,j}^{hit} \cdot x_{i,j}^{hit} + \delta_{i,j}^{miss} \cdot x_{i,j}^{miss} \right) \right\} \tag{7.10}$$

  Aufgrund einer Konfliktanalyse können Beschränkungen für das Cache-Modell aufgestellt werden. In Gegenwart von Instruktionscaches macht lediglich der Einsatz der PSS-Methode Sinn. Bei der Verwendung der ITA-Methode müssten Cache-Hits und Cache-Misses anderweitig ermittelt werden.
- *Datencaches*: Bei der Verwendung von Datencaches hat die Zugriffsreihenfolge auf Daten einen großen Einfluss auf die Ausführungszeit einzelner Instruktionen und somit des Grundblocks. Wie im Fall des Instruktionscaches ist die Anwendung der PSS-Methode sinnvoll.

Somit ist die ITA-Methode lediglich für sehr einfache Mikroarchitekturen anwendbar, während die PSS-Methode für komplexe Architekturen anwendbar ist, solange keine datenabhängigen Ausführungszeiten zu berücksichtigen sind. Typische Prozessoren kombinieren heutzutage viele dieser Architekturmerkmale, was die WCET-Analyse für moderne Prozessoren erschwert.

7.4.2 Echtzeitanalyse für Einprozessorsysteme

Auf Modulebene werden mehrere Software-Prozesse auf einem einzelnen Prozessor ausgeführt. Um eventuelle Ressourcenkonflikte aufzulösen, muss eine Ablaufplanung der Prozesse vorgenommen werden. Diese Ablaufplanung kann statisch oder dynamisch erfolgen, wobei letzteres Vorgehen heutzutage überwiegt. Dies gilt insbesondere für Echtzeitsystemen. Dies sind Systeme, bei denen die Antwortzeiten durch sog. engl. *Deadlines* vorgegeben und garantiert werden müssen. Die vorgegebenen Deadlines stellen dabei nichtfunktionale Anforderungen an die Implementierung dar. Die Zeitanalyse solcher Echtzeitsysteme muss also in der Lage sein, zu bewerten, ob die vorgegebenen Deadlines unter allen Bedingungen eingehalten werden. Im Folgenden werden Probleme und Analyseverfahren als Zusammenfassung aus [426] vorgestellt.

Betrachtet wird im Folgenden das Modell eines Problemgraphen $G(V,E)$. Die Knoten $v \in V$ modellieren Software-Prozesse, denen früheste Startzeitpunkte (*Releasezeiten*, *Ankunftszeiten*) und späteste Endzeitpunkte (*Deadlines*) zugewiesen wurden. Kanten $e \in E$ entsprechen Datenabhängigkeiten. Falls nicht anders bemerkt gelte, dass allen Prozessen die Ankunftszeit 0 und die Deadline ∞ zugewiesen wurde. Ferner gilt die Ressourcenbeschränkung mit der vereinfachten Eigenschaft, dass jeder Knoten $v \in V$ auf dem einen verfügbaren Prozessor abgearbeitet werden kann und die Ausführungszeit δ_i eines Knotens $v_i \in V$ als bekannt gilt, z. B. durch Annahme der WCET.

Im Allgemeinen benötigt der Algorithmus zur Ablaufplanung auf einem Prozessor selbst Rechenzeit, um festzulegen, welcher Prozess als nächstes den Prozessor belegen soll. Diese Zeitspanne bezeichnet man im Allgemeinen als *Dispatchlatenz*.

Definition 7.4.2 (Dispatchlatenz). *Die* Dispatchlatenz Λ_D *bezeichnet die maximale Zeitspanne zwischen dem Stoppen eines Prozesses $v_i \in V$ und dem Starten des nächsten Prozesses $v_j \in V$ auf dem Prozessor.*

Offensichtlich sollte die Dispatchlatenz so klein wie möglich sein. Im Folgenden gelte die Annahme, dass die Dispatchlatenz null sei. Für die Latenz eines Ablaufplans gelte die gleiche Definition wie für statische Ablaufplanungsverfahren (siehe Definitionen 6.5.3 auf Seite 349 bzw. Gleichung (6.25) auf Seite 349).

Folgende Kriterien bei der Bewertung von Algorithmen zur dynamischen Ablaufplanung zeigen sich als nützlich:

Definition 7.4.3 (Ressourcenauslastung). *Gegeben sei ein Problemgraph $G(V,E)$ und ein Ablaufplan auf einem Prozessor der Latenz Λ. Sei δ_i die Ausführungszeit von Knoten $v_i \in V$. Dann bezeichnet*

$$U := \frac{\sum_{i:=1}^{|V|} \delta_i}{\Lambda} \cdot 100 \tag{7.11}$$

die Ressourcenauslastung *des Prozessors in Prozent.*

Offensichtlich ist man bestrebt, die Ressourcenauslastung möglichst bei 100% zu halten. Im Falle präemptiver Ablaufplanung kann die Ausführung eines Prozesses unterbrochen werden, um zu einem späteren Zeitpunkt wieder aufgenommen zu werden. Für solche Verfahren ist es deshalb interessant, wie lange es dauert, bis ein Prozess endgültig abgearbeitet ist.

Definition 7.4.4 (Abarbeitungszeit). *Gegeben sei ein Problemgraph $G(V,E)$. Sei δ_i die Ausführungszeit von Knoten $v_i \in V$, $\tau_\mathrm{b}(v_i)$ der Zeitpunkt, an dem v_i zum ersten Mal den Prozessor belegt, und $\tau_\mathrm{e}(v_i)$ der Zeitpunkt, an dem v_i vollständig abgearbeitet ist (Endzeitpunkt). Dann beträgt die* Abarbeitungszeit $\delta_\mathrm{A}(v_i)$ *von $v_i \in V$:*

$$\delta_\mathrm{A}(v_i) := \tau_\mathrm{e}(v_i) - \tau_\mathrm{b}(v_i) \tag{7.12}$$

Im Weiteren definiert man die sog. *Wartezeit* (engl. *waiting time*) eines Prozesses wie folgt:

Definition 7.4.5 (Wartezeit). *Gegeben sei ein Problemgraph $G(V,E)$. Sei δ_i die Ausführungszeit von Knoten $v_i \in V$, $\tau_\mathrm{r}(v_i)$ die Releasezeit von Knoten v_i und $\tau_\mathrm{e}(v_i)$ der Zeitpunkt, an dem v_i vollständig abgearbeitet ist (Endzeitpunkt). Dann bezeichnet $\delta_\mathrm{W}(v_i)$ mit*

$$\delta_\mathrm{W}(v_i) := \tau_\mathrm{e}(v_i) - \tau_\mathrm{r}(v_i) - \delta_i \tag{7.13}$$

die Wartezeit *von Prozess i.*

Die Wartezeit eines Prozesses ist damit die Dauer der Zeitspanne, die er ab dem Zeitpunkt seiner Ankunft (Releasezeit) den Prozessor nicht belegt hat, also sozusagen auf seine (weitere) Abarbeitung wartet. Ein ähnliches Maß ist die sog. *Flusszeit*, die Summe von Ausführungszeit und Wartezeit:

Definition 7.4.6 (Flusszeit/Antwortzeit). *Gegeben sei ein Problemgraph $G(V,E)$. Sei δ_i die Ausführungszeit von Knoten $v_i \in V$, $\tau_\mathrm{r}(v_i)$ die Releasezeit von Knoten v_i und $\tau_\mathrm{e}(v_i)$ der Zeitpunkt, an dem v_i vollständig abgearbeitet ist (Endzeitpunkt). Dann bezeichnet $\delta_\mathrm{F}(v_i)$ mit*

$$\delta_\mathrm{F}(v_i) := \tau_\mathrm{e}(v_i) - \tau_\mathrm{r}(v_i) \tag{7.14}$$

die Flusszeit *von Prozess i. In der Literatur wird die Flusszeit auch häufig als* Antwortzeit *(engl.* response time*) bezeichnet.*

Im Zusammenhang mit Echtzeitsystemen spielt die Einhaltung von Deadlines zur Beurteilung von Ablaufplanungsverfahren eine große Rolle, da bei diesen Systemen die Nichteinhaltung katastrophale Folgen haben kann. Hier sind deshalb auch folgende Definitionen von Eigenschaften eines Ablaufplans von Interesse:

Definition 7.4.7 (Lateness, Tardiness). *Gegeben sei ein Problemgraph $G(V,E)$. Sei $\tau_{\mathrm{d}}(v_i)$ die Deadline (spätester Endzeitpunkt) von Knoten $v_i \in V$ und $\tau_{\mathrm{e}}(v_i)$ der Zeitpunkt, an dem v_i vollständig abgearbeitet ist (Endzeitpunkt). Dann bezeichnet $\delta_{\mathrm{L}}(v_i)$ mit*

$$\delta_{\mathrm{L}}(v_i) := \tau_{\mathrm{e}}(v_i) - \tau_{\mathrm{d}}(v_i) \tag{7.15}$$

die sog. engl. Lateness *von Prozess i und $\delta_{\mathrm{T}}(v_i)$ mit*

$$\delta_{\mathrm{T}}(v_i) := \max\{\tau_{\mathrm{e}}(v_i) - \tau_{\mathrm{d}}(v_i), 0\} \tag{7.16}$$

die sog. engl. Tardiness *von Prozess i.*

Aperiodische, dynamische Ablaufplanung

Ein Verfahren, das die Minimierung der maximalen Lateness der Prozesse zum Ziel hat, kann im Falle nichtpräemptiver Ablaufplanung von Prozessen ohne Datenabhängigkeiten auf einem Prozessor mittels der Regel von Jackson angegeben werden [243]:

Theorem 7.4.1 (Jackson-Regel [243]). *Gegeben sei ein Problemgraph $G\ (V,\{\ \})$. Für die Ankunftszeiten gelte $\forall i = 1,\ldots,|V| : \tau_{\mathrm{r}}(v_i) = 0$. Ferner sei für jeden Prozess eine Deadline $\tau_{\mathrm{d}}(v_i)$ gegeben. Dann ist der Algorithmus, der die Prozesse in der Reihenfolge nicht kleiner werdender Deadlines plant, ein exaktes Verfahren zur Minimierung der maximalen Lateness.*

Ein Algorithmus, der gemäß der Jackson-Regel einen Ablaufplan bestimmt, heißt auch *Earliest-Due-Date*-Algorithmus (EDD).

Man kann zeigen, dass bei von null verschiedenen Ankunftszeiten der EDD-Algorithmus nicht mehr exakt ist. Allerdings führt hier ebenfalls die Erlaubnis der Präemption zu einer einfachen exakten Erweiterung: Zu jedem Zeitpunkt wird unter allen Prozessen, deren Ankunftszeit kleiner oder gleich der aktuellen Zeit ist, derjenige Prozess geplant, dessen Deadline am kleinsten ist. Diese Strategie von Horn [225] ist auch bekannt als *Earliest-Deadline-First*-Algorithmus (EDF). Bei verbotener Präemption ist das Problem der Minimierung der maximalen Lateness im Falle ungleicher Ankunftszeiten bis auf wenige Spezialfälle $\mathcal{NP}$-schwer.

Berücksichtigung von Datenabhängigkeiten

Der EDF-Algorithmus kennt keine Datenabhängigkeiten. Im nichtpräemptiven Fall bei gleichen Ankunftszeiten gibt es jedoch folgenden Algorithmus von Lawler [292], der als *Latest-Deadline-First*-Algorithmus (LDF) bekannt ist. Der Algorithmus baut einen Ablaufplan, der die Datenabhängigkeiten erfüllt, wie folgt auf: Unter allen Prozessen ohne ungeplante Nachfolger selektiert der LDF-Algorithmus denjenigen Prozess zuerst, dessen Deadline am größten ist. Dieser Schritt wird so lange iteriert, bis alle Prozesse selektiert wurden. Einen Ablaufplan, der die Datenabhängigkeiten erfüllt, erhält man dann durch Ausführung der Prozesse in umgekehrter Reihenfolge der Selektion, d. h. beginnend mit dem zuletzt selektierten Prozess und endend mit dem zuerst selektierten Prozess.

Beispiel 7.4.2. Betrachtet wird ein Problemgraph mit sechs Prozessen $v_1, v_2, v_3, v_4, v_5, v_6$ und den in Abb. 7.30a) dargestellten Datenabhängigkeiten. Für die Ausführungszeiten gilt $\delta_1 = \delta_2 = \delta_3 = \delta_4 = \delta_5 = \delta_6 := 1$ und die Deadlines $\tau_\mathrm{d}(v_1) := 2, \tau_\mathrm{d}(v_2) := 5, \tau_\mathrm{d}(v_3) := 4, \tau_\mathrm{d}(v_4) := 3, \tau_\mathrm{d}(v_5) := 5, \tau_\mathrm{d}(v_6) := 6$. Abbildung 7.30b) zeigt einen mit der LDF-Strategie gewonnenen Ablaufplan. Unter den drei Prozessen v_4, v_5 und v_6 ohne ungeplante Nachfolger wird v_6 zuerst selektiert, da dessen Deadline am größten ist. Dann wird unter den drei Prozessen v_3, v_4 und v_5 mit gleichem Argument v_5 selektiert, usw. Zuletzt wird v_1 selektiert. Geplant wird dann in umgekehrter Reihenfolge der Selektion. Im Beispiel erfüllen alle Prozesse ihre Deadlines.

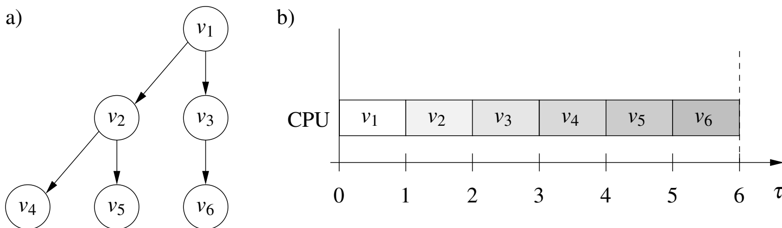


Abb. 7.30. Ablaufplanung mit LDF-Strategie

Theorem 7.4.2 (LDF-Regel [292]). *Gegeben sei ein Problemgraph $G(V,E)$. Für die Ankunftszeiten gelte $\tau_\mathrm{r}(v_i) = 0\ \forall i = 1, \cdots, |V|$. Ferner sei für jeden Prozess eine Deadline $\tau_\mathrm{d}(v_i)$ gegeben. Dann ist der Algorithmus, der die Prozesse nach der LDF-Regel plant, ein exaktes Verfahren zur Minimierung der maximalen Lateness.*

Bei ungleichen Ankunftszeiten kann das Problem der Minimierung der maximalen Lateness nur dann exakt polynomiell gelöst werden, wenn man Präemption gestattet. Chetto et al. [90] haben hierzu eine Erweiterung des EDF-Algorithmus vorgestellt. Da EDF keine Datenabhängigkeiten kennt, werden die Ankunftszeiten $\tau_\mathrm{r}(v_i)$ und die Deadlines $\tau_\mathrm{d}(v_i)$ im ersten Schritt in neue Ankunftszeiten $\tau_\mathrm{r}^*(v_i)$ und neue Deadlines $\tau_\mathrm{d}^*(v_i)$ transformiert und im nächsten Schritt die EDF-Strategie angewendet. Die folgende Transformation der Ankunftszeiten und Deadlines bewirkt dabei, dass kein Prozess vor Beginn all seiner Vorgänger starten und keine Nachfolger unterbrechen kann. Diese Modifikation heißt *EDF**-Algorithmus [90]. Man kann zeigen, dass ein Ablaufplan für das ursprüngliche Problem mit Datenabhängigkeiten genau dann existiert, wenn EDF* einen Ablaufplan für das transformierte Problem ohne Datenabhängigkeiten findet.

Die Transformation lautet wie folgt:

$$\tau_\mathrm{r}^*(v_j) := \max\{\tau_\mathrm{r}(v_j), \max_{(v_i,v_j)\in E}\{\tau_\mathrm{r}^*(v_i) + \delta_i\}\} \tag{7.17}$$

Offensichtlich kann Prozess v_j erst nach seiner Ankunftszeit und nicht früher als zum frühestmöglichen Endzeitpunkt aller seiner direkten Vorgänger starten.

$$\tau_{\mathrm{d}}^{*}(v_i) := \min\{\tau_{\mathrm{d}}(v_i), \min_{(v_i,v_j)\in E}\{\tau_{\mathrm{d}}^{*}(v_j) - \delta_j\}\} \tag{7.18}$$

Weiterhin muss ein Prozess v_i vor seiner Deadline beendet sein, darf aber auch nicht später als zum spätestmöglichen Startzeitpunkt aller seiner direkten Nachfolger enden.

Beispiel 7.4.3. Betrachtet wird die Datenabhängigkeit (v_i, v_j) in Abb. 7.31a) mit Ausführungszeiten $\delta_i := 5$ und $\delta_j := 7$. Die Ankunftszeiten betragen $\tau_{\mathrm{r}}(v_i) := 2$, $\tau_{\mathrm{r}}(v_j) := 0$, die Deadlines $\tau_{\mathrm{d}}(v_i) := 15$, $\tau_{\mathrm{d}}(v_j) := 14$. Für die transformierte Ankunftszeit von v_j erhält man $\tau_{\mathrm{r}}^{*}(v_j) := \max\{0, 2+5\} = 7$. Für die transformierte Deadline von v_i erhält man $\tau_{\mathrm{d}}^{*}(v_i) := \min\{15, 14-7\} = 7$. Plant man anschließend die Prozesse nach der EDF-Regel, so erhält man den Ablaufplan in Abb. 7.31b). Man vergewissere sich, dass eine Ablaufplanung nach EDF ohne Transformation der Ankunftszeiten und Deadlines die Datenabhängigkeiten nicht respektieren würde.

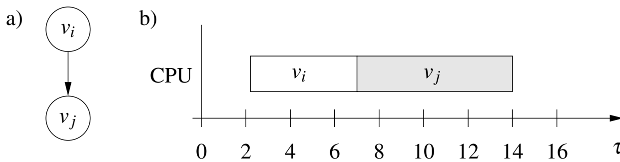


Abb. 7.31. Ablaufplanung mit EDF*-Strategie: a) Problemgraph und b) Ablaufplan mit EDF-Strategie bei Verwendung der transformierten Zeiten

Periodische, dynamische Ablaufplanung

Wichtige Ergebnisse zur Theorie dieser Klasse von iterativen dynamischen Ablaufplanungsproblemen stammen von Liu und Layland [307], die im Folgenden zusammengefasst werden. Deren Untersuchungen beziehen sich auf ein Modell von periodischen Prozessen, die sich durch einen iterativen Problemgraphen $G(V, \{\,\}, -)$, d. h. durch einen Problemgraphen ohne Datenabhängigkeiten und folglich ohne Indexverschiebungen darstellen lassen. Den Prozessen $v_i \in V$ sind bekannte Ausführungszeiten δ_i zugewiesen. Ferner wird angenommen, dass Präemption erlaubt sei. Im Gegensatz zu iterativen Ablaufplanungsproblemen mit Datenabhängigkeiten haben hier einzelne Prozesse meist unterschiedliche Perioden $P(v_i)$. Betrachtet wird damit folgendes Ablaufplanungsmodell:

Definition 7.4.8 (Iteratives, dynamisches Ablaufplanungsmodell). *Ein* iteratives, dynamisches Ablaufplanungsmodell *besteht aus*

- *einem iterativen Problemgraphen $G(V, \{\,\}, -)$,*
- *Ausführungszeiten δ_i für Knoten $v_i \in V$,*
- *Perioden $P(v_i)$ für Knoten $v_i \in V$,*

- periodischen Releasezeiten $\tau_\mathrm{r}(v_i,n)$ *mit*

$$\forall n \geq 0 : \tau_\mathrm{r}(v_i,n) := \tau_\mathrm{r}^*(v_i) + n \cdot P(v_i)\,.$$

  Dies bedeutet, dass die Releasezeiten neuer Iterationen eines Knotens v_i immer im Abstand des Iterationsintervalls (der Periode) $P(v_i)$ auftreten. Die Releasezeiten $\tau_\mathrm{r}^*(v_i)$ werden auch als Phasen *bezeichnet. Ferner gibt es*
- periodische Deadlines $\tau_\mathrm{d}(v_i,n)$ *mit*

$$\forall n \geq 0 : \tau_\mathrm{d}(v_i,n) := \tau_\mathrm{r}(v_i,n) + \tau_\mathrm{d}^*(v_i)\,.$$

  Dies bedeutet, dass sich die Deadlines ebenfalls im Abstand der Periode $P(v_i)$ eines Prozesses wiederholen. Die periodische Deadline $\tau_\mathrm{d}^*(v_i)$ ist hier keine absolute Zeitbeschränkung, sondern als Zeitspanne relativ zur Releasezeit einer Iteration definiert.

Liu und Layland beschäftigten sich zunächst nur mit dem Spezialfall $\forall v_i \in V : t_\mathrm{d}^*(v_i) = P(v_i)$, d. h. die Deadline einer Iteration eines Prozesses ist immer gleich der Dauer der Periode einer Iteration. Zur Erläuterung der Begriffe soll folgendes Beispiel dienen.

Beispiel 7.4.4. Gegeben ist ein Problemgraph mit zwei Knoten v_1 und v_2, den Ausführungszeiten $\delta_1 := 1$, $\delta_2 := 2{,}5$ sowie den Perioden $P(v_1) := 2$ und $P(v_2) := 5$. Für die Deadlines gilt $\tau_\mathrm{d}^*(v_1) = P(v_1)$, $\tau_\mathrm{d}^*(v_2) = P(v_2)$, für die Ankunftszeiten der nullten Iteration gilt $\tau_\mathrm{r}^*(v_1) = \tau_\mathrm{r}^*(v_2) := 0$. Abbildung 7.32 zeigt zwei periodische Ablaufpläne. Der Ablaufplan in Abb. 7.32a) erfüllt beide Deadlines und wiederholt sich periodisch nach zwei Iterationen von Knoten v_2 bzw. fünf Iterationen von Knoten v_1. Der Ablaufplan in Abb. 7.32b) erfüllt die Deadline der nullten Iteration von Knoten v_2 nicht ($\tau_\mathrm{d}(v_2,0) = \tau_\mathrm{r}(v_2,0) + \tau_\mathrm{d}^*(v_2) = 0 + 5 < 5{,}5$).

Liu und Layland betrachteten anschließend Algorithmen zur iterativen Ablaufplanung mit Echtzeitbedingungen und statischen Prioritäten. Um hinreichende Kriterien angeben zu können, wann ein Algorithmus zur Ablaufplanung für beliebige Instanzen von Ablaufplanungsproblemen alle Deadlines erfüllen wird, muss man zeigen, dass der Algorithmus auch für den schlimmsten Fall angenommener Releasezeiten einen in diesem Sinn gültigen Ablaufplan findet. So zeigten Liu und Layland, dass dieser Fall immer durch die Phasen $\forall v_i \in V : \tau_\mathrm{r}^*(v_i) = 0$ gegeben ist, d. h. wenn alle Prozesse gleichzeitig ankommen. Anschaulich beginnt in diesem Fall der Wettlauf der Zeit mit den Deadlines aller Prozesse gleichzeitig, was offensichtlich nicht ungünstiger sein könnte. Dann zeigten sie, dass ein Algorithmus zur Ablaufplanung mit statischen Prioritäten eine Probleminstanz unter Erfüllung aller Deadlines immer dann planen kann, wenn für die Phasen $\forall v_i \in V : \tau_\mathrm{r}^*(v_i) = 0$ alle Deadlines für die nullte Iteration erfüllt werden.

Um diese Ergebnisse anschaulicher zu machen, wird nun ein bekannter Algorithmus zur iterativen Ablaufplanung mit statischen Prioritäten vorgestellt.

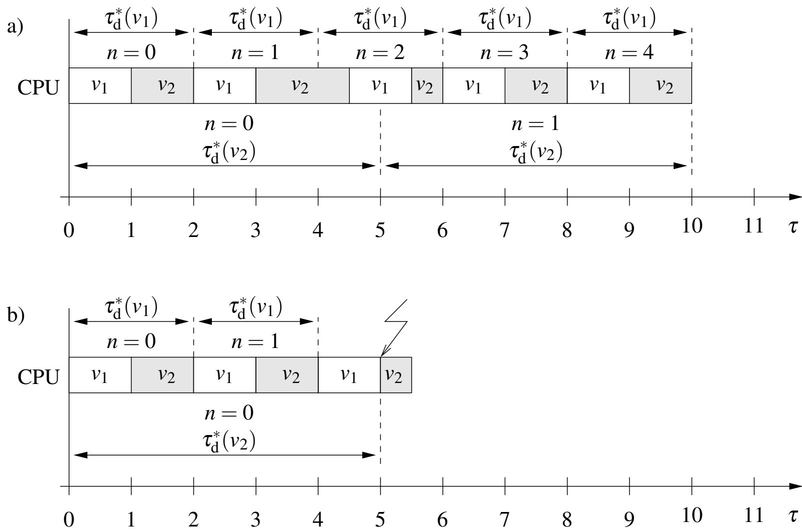


Abb. 7.32. Iterative, dynamische Ablaufpläne: a) mit der EDF-Strategie und b) mit dem RMS-Algorithmus

Ratenmonotone Ablaufplanung

Unter dem Begriff *ratenmonotone Ablaufplanung* (engl. *rate-monotonic scheduling, RMSs*) haben Liu und Layland ein präemptives Ablaufplanungsverfahren beschrieben, das den Prozessen statische Prioritäten nach folgendem Prinzip zuweist: v_i habe größere Priorität als v_j, falls $P(v_i) < P(v_j)$. Also wählt man die Priorität statisch beispielsweise proportional zur gegebenen Rate eines Prozesses (Kehrwert der Periode). Prozesse mit kleinerer Periode werden also höher priorisiert als Prozesse mit größerer Periode. Dies gilt also unabhängig von den Ausführungszeiten der Prozesse. Bei Ankunft eines Prozesses höherer Priorität wird der laufende Prozess unterbrochen.

Beispiel 7.4.5. Abbildung 7.32b) zeigt einen mit der RMS-Strategie gewonnenen Ablaufplan für das in Beispiel 7.4.4 vorgestellte Problem.

Liu und Layland zeigten dann, dass der RMS-Algorithmus unter allen Algorithmen mit statischen Prioritäten (für den von ihnen angenommen Fall $\tau_\mathrm{d}^*(v_i) = P(v_i)$) immer einen gültigen Ablaufplan bestimmt, falls ein solcher existiert.

Beispiel 7.4.6. Aus diesem Ergebnis lässt sich schließen, dass es keinen Algorithmus mit festen Prioritäten gibt, der für das Problem aus Beispiel 7.4.4 alle Deadlines erfüllt, da der RMS-Algorithmus keinen solchen Ablaufplan bestimmt hat.

Nun lässt sich allerdings ein einfaches, hinreichendes Kriterium angeben, wann der RMS-Algorithmus immer einen Ablaufplan findet, der alle Deadlines erfüllt:

Theorem 7.4.3 (Liu und Layland). *Gegeben sei ein Ablaufplanungsproblem nach Definition 7.4.8 mit der Eigenschaft $\forall v_i \in V : \tau_{\mathrm{d}}^*(v_i) = P(v_i)$. Die Menge von Prozessen kann unter Erfüllung aller Deadlines mit dem RMS-Algorithmus geplant werden, falls*

$$\sum_{i:=1}^{|V|} \frac{\delta_i}{P(v_i)} \leq |V| \cdot (2^{1/|V|} - 1) \tag{7.19}$$

gilt.

Für $|V| = 1$ erhält man als rechte Seite von Ungleichung (7.19) den Wert 1. Damit kann das RMS-Verfahren ein Problem mit einem Prozess unter der Erfüllung dessen Deadline immer planen, wenn (offensichtlich) die Ausführungszeit kleiner gleich der Deadline (hier auch Periode) ist. Interessanter ist der Fall für $|V| \to \infty$. Dann erhält man für die rechte Seite der Ungleichung (7.19) den Wert $\ln(2)$. Damit lässt sich die Aussage treffen, dass für alle Problemgrößen $|V|$ immer ein gültiger Ablaufplan gefunden wird, wenn $\sum_{i:=1}^{|V|} \delta_i/P(v_i) \leq \ln(2)$ gilt, d. h. wenn der Prozessor weniger als $1/\ln 2 \cdot 100\,\% \approx 69,3\,\%$ ausgelastet ist. Man beachte, dass das Kriterium in Theorem 7.4.3 natürlich nur eine hinreichende Bedingung darstellt, so dass es auch Probleminstanzen geben kann, für die der RMS-Algorithmus auch bei höherer Prozessorauslastung einen Ablaufplan finden kann, der alle Deadlines einhält.

Aus Ungleichung (7.19) lässt sich übrigens auch eine notwendige Bedingung für die Einhaltbarkeit aller Deadlines erkennen: $\sum_{i:=1}^{|V|} \delta_i/P(v_i) \leq 1$. Dies entspricht dem Grenzfall einer Prozessorauslastung von 100%. Falls diese Bedingung nicht erfüllt ist, kann offensichtlich auch kein Algorithmus mit dynamischen Prioritäten einen gültigen Ablaufplan finden.

Damit stellt sich aber die Frage, ob es denn einen Algorithmus gibt, der auch für den extremen Fall $\sum_{i:=1}^{|V|} \delta_i/P(v_i) = 1$ einen gültigen Ablaufplan finden kann? Die Antwort heißt ja, und ein solcher Algorithmus ist der EDF-Algorithmus.

Theorem 7.4.4 (Liu und Layland). *Gegeben sei ein Ablaufplanungsproblem nach Definition 7.4.8 mit der Eigenschaft $\forall v_i \in V : \tau_{\mathrm{d}}^*(v_i) = P(v_i)$. Die Menge von Prozesse kann unter Erfüllung aller Deadlines mit dem EDF-Algorithmus geplant werden, falls*

$$\sum_{i:=1}^{|V|} \frac{\delta_i}{P(v_i)} \leq 1 \tag{7.20}$$

gilt.

Der EDF-Algorithmus ist prioritätsgesteuert mit dynamischer Priorität, wobei die höchste Priorität demjenigen Prozess zugewiesen wird, dessen Deadline am kleinsten ist. Präemption erfolgt immer dann, wenn es einen noch nicht fertig abgearbeiteten Prozess gibt, der eine höhere Priorität besitzt.

Beispiel 7.4.7. Abbildung 7.32a) zeigt einen mit der EDF-Strategie ermittelten Ablaufplan, der alle Deadlines für das Problem aus Beispiel 7.4.4 erfüllt. Dies gilt bei einer Prozessorauslastung von 100% ($\sum_{i:=1}^{2} \frac{\delta_i}{P(v_i)} = 1/2 + 2,5/5 = 1$). In Abb. 7.32b)

ist ein Ablaufplan nach der RMS-Strategie dargestellt. EDF und RMS werden in der Praxis nahezu ausschließlich in der hier dargestellten präemptiven Version eingesetzt.

Antwortzeitanalyse

Neben den vorgestellten einfachen auf Ressourcenauslastung beruhenden Tests in Gleichung (7.19) für RMS-Ablaufplanung und in Gleichung (7.20) für EDF, wurden Tests entwickelt, die detailliertere Informationen über das Zeitverhalten und Prioritäten von Prozessen zur Bestimmung der Planbarkeit periodischer Prozesse ausnutzen. Für RMS betrachten Joseph und Pandya in [249] die Antwortzeit eines Prozesses $v_i \in V$ im ungünstigsten Fall gleicher Ankunftszeiten aller Prozesse. Die Antworts- bzw. Flusszeit $\delta_{\mathrm{F}}(v_i)$ ist dann durch die Summe der eigenen Ausführungszeit δ_i und der maximalen sog. *Interferenz* I_i gegeben. Der Interferenzterm I_i bestimmt dabei die Summe der Verzögerungszeiten, denen v_i unterliegt, wenn dieser durch höherpriore Prozesse unterbrochen wird. Sei $hp(v_i) \subseteq V$ die Menge höherpriorer Prozesse von v_i und die Deadline $t_{\mathrm{d}}^*(v_i)$ gleich der Periode $P(v_i)$ für alle $v_i \in V$. Dann gilt offensichtlich:

$$\delta_{\mathrm{F}}(v_i) := \delta_i + \sum_{v_j \in hp(v_i)} \delta_j \left\lceil \frac{\delta_{\mathrm{F}}(v_i)}{P(v_j)} \right\rceil \leq t_{\mathrm{d}}^*(v_i) = P(v_i) \qquad (7.21)$$

Gibt es für diese rekursive Gleichung eine Lösung im Intervall $[0, \ldots, t_{\mathrm{d}}^*(v_i)]$, so erfüllt der Prozess v_i unter RMS-Ablaufplanung seine Deadline $t_{\mathrm{d}}^*(v_i)$.

Zeitgetriebene, dynamische Ablaufplanung

Dynamische, prioritätsbasierte Ablaufplanung stellt eine effiziente Methode zur Planung von Prozessen unter vielen möglichen Umständen dar. Dies hat allerdings häufig zur Folge, dass das System bei dessen Ausführung ein starkes dynamisches Verhalten zeigt, was die Analyse erschwert. Im Gegensatz dazu weist eine zeitgetriebene, dynamische Ablaufplanung jedem Prozess einen mehr oder weniger feststehenden Zeitschlitz für dessen Abarbeitung zu. Damit kann ein Prozess einen Prozessor für ein genau festgelegtes Zeitbudget belegen. Wenn dieses Zeitbudget aufgebraucht ist, wird dem nächsten Prozess der Prozessor zugeteilt. Im Folgenden werden zwei prominente Vertreter zeitgetriebener, dynamischer Ablaufplanung vorgestellt: *Round-Robin-* und *TDMA-Ablaufplanung*.

Round-Robin-Ablaufplanung (RR)

Bei der sog. *Round-Robin-Ablaufplanung* (RR) gibt es ein festes Zeitintervall Q, nach dessen Ablauf spätestens ein Kontextwechsel eingeleitet wird (z. B. alle $Q =$ 100 Zeitschritte), falls nicht ein geplanter Prozess vor Ablauf des Zeitquantums beendet wurde. Die laufbereiten Prozesse werden in einer zirkulären Warteschlange verwaltet und *reihum* für die maximale Zeitdauer eines Zeitquantums geplant.

Beispiel 7.4.8. Betrachtet wird ein Problemgraph mit drei Prozessen v_1, v_2 und v_3 ohne Datenabhängigkeiten mit den Ausführungszeiten $\delta_1 := 24, \delta_2 := 3, \delta_3 := 30$ und den Ankunftszeiten $\tau_r(v_1) = \tau_r(v_2) = \tau_r(v_3) := 0$. Abbildung 7.33 zeigt den mit der RR-Strategie gewonnenen Ablaufplan für ein Zeitquantum von $Q := 4$ Zeitschritten. Man erkennt, dass zum Zeitpunkt $\tau = 4$ der Knoten v_1 *suspendiert* wird, da das Zeitquantum abgelaufen ist. Erst zum Zeitpunkt $\tau = 11$ wird v_1 weiter berechnet. Die mittlere Wartezeit beträgt in diesem Beispiel $\overline{\delta_W} = (47 - 24) + (7 - 3) + (57 - 30)/3 = 18$ Zeiteinheiten.

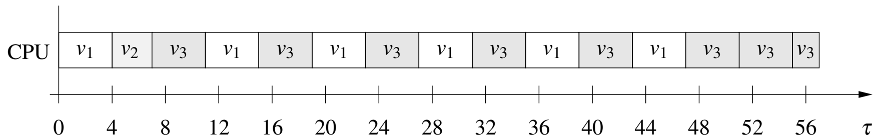


Abb. 7.33. Ablaufplanung mit RR-Strategie

Round-Robin-Ablaufplanung wird häufig in Mehrbenutzerbetriebssystemen angewendet. Als Nachteil muss man häufig lange Wartezeiten in Kauf nehmen. Bei einer Anzahl von N planbaren Prozessen und einem Zeitquantum Q kann man jedoch die Schranke $(N-1)Q$ als längstes Zeitintervall, bis ein Prozess spätestens wieder weiter abgearbeitet wird, angeben. Falls Q gegen unendlich strebt, erhält man den bekannten *FCFS-Algorithmus* (engl. *First Come, First Served*). Falls Q gegen null strebt, verhält sich das System (unter Vernachlässigung von Kontextwechselzeiten) wie ein System mit N Prozessoren, die allerdings jeweils nur die $1/N$-fache Rechenleistung besitzen.

TDMA

Neben der Round-Robin-Ablaufplanung gibt es eine weitere populäre zeitgetriebene Ablaufplanungsstrategie, die als *TDMA* (engl. *Time Division Multiple Access*) bekannt ist. Hier werden jedem Prozess periodisch auf einem Prozessor Zeitslots zugewiesen. Im Gegensatz zur Round-Robin-Strategie erfolgt dies unabhängig davon, ob der Prozess gerade laufbereit ist oder nicht. Damit hängt das Zeitverhalten eines Prozesses nicht mehr vom Zeitverhalten anderer Prozesse ab, was die Zeitanalyse stark vereinfacht. Zum Beispiel lässt sich die Antwortzeit eines Prozesses v_i mit Periode $P(v_i)$, zugewiesener Slotbreite $S(v_i)$, Ausführungszeit δ_i und Zeitquantum Q wie folgt berechnen:

$$\delta_F(v_i) = \delta_i + (Q - S(v_i)) \left\lceil \frac{\delta_i}{S(v_i)} \right\rceil \tag{7.22}$$

Der eigenen Ausführungszeit δ_i zuzuschlagen sind maximal $\left\lceil \frac{\delta_i}{S(v_i)} \right\rceil$ Zeitscheiben, in denen v_i die Ressource $(Q - S(v_i))$ Zeitschritte nicht belegt. TDMA ist zwar im

Bereich der Ablaufplanung auf Prozessoren wenig bekannt, wird hingegen oft zur Planung von Kommunikationen auf Bussen genutzt.

7.5 Literaturhinweise

Die formale Äquivalenzprüfung von Assemblerprogrammen auf Basis von symbolischer Simulation ist in [122] beschrieben. Spezielle Eigenschaften zu Assemblerprogrammen für DSPs und VLIW-Prozessoren und deren Behandlung in der Äquivalenzprüfung sind in [123] bzw. [159] aufgeführt. Der Einsatz von Schnittpunkten zur strukturellen Äquivalenzprüfung von Assemblerprogrammen ist in [160] beschrieben. Eine Erweiterung der strukturellen Äquivalenzprüfung zur Äquivalenzprüfung von C-Programmen mit Hardware-Schaltungen findet sich in [161]. Weitere Ansätze zum Vergleich von C-Programmen mit Hardware sind in [270], [264] und [229] präsentiert. Die Äquivalenzprüfung von C-Programmen auf Basis von symbolischer Simulation ist in [315] beschrieben. In [394] ist die Äquivalenzprüfung von Schleifenprogrammen beschrieben. Weiterführende Ansätze sind in [395] und [396] vorgestellt.

Die Verfahren zur funktions- und strukturorientierten Testfallgenerierung werden ausführlich in [305] diskutiert. Bei den simulativen Verifikationsmethoden ist eine 100%-ige Zweigüberdeckung heutzutage das Minimalkriterium in der Software-Entwicklung. Der Standard DO-178B für die Software-Entwicklung in sicherheitskritischen Bereichen fordert beispielsweise einen *modifizierten Bedingungs-/Entscheidungsüberdeckungstest* [382]. Darin werden Testfälle gefordert, die zeigen, dass jede atomare Bedingung die Evaluierung einer Entscheidung unabhängig von anderen atomaren Bedingungen beeinflussen kann. Eine vergleichende Studie zwischen Zweig-, Pfad- und strukturiertem Pfadüberdeckungstest findet sich in [227, 228]. Ein modifizierter strukturierter Pfadüberdeckungstest ist in [305] beschrieben. In diesem wird die Komplexität reduziert, indem die Überdeckung von ausführbaren Teilpfaden in Schleifen gefordert wird, nicht aber die Überdeckung sämtlicher Kombinationen.

Die datenflussorientierte Testfallgenerierung spielt in der Praxis heutzutage noch nicht die Rolle wie die kontrollflussorientierte Testfallgenerierung. Ein Vergleich der *all defs-*, *all c-uses-* und *all p-uses-*Überdeckungstests findet sich in [190]. Darüber hinausgehende Techniken sind beispielsweise der *required k-tuples-Überdeckungstest* [349, 109, 350] und der *Datenkontextüberdeckungstest* [290, 291, 109]. Letzterer verlangt, dass alle existierenden Möglichkeiten den in einem gegebenen Grundblock verwendeten Variablen Werte zuzuweisen, berücksichtigt werden. Dabei kann auch die unterschiedliche Reihenfolge der Wertzuweisung berücksichtigt werden.

Eine gute Übersicht über Methoden und Werkzeuge zur formalen funktionalen Eigenschaftsprüfung von Programmen findet sich in [140]. Darin wird eine Unterscheidung in Verfahren zur statischen Programmanalyse, zur Modellprüfung und zur SAT-basierten Modellprüfung vorgenommen. Die abstrakte Interpretation zur statischen Programmanalyse wurde erstmal 1977 vorgestellt [118]. Die notwendigen mathematischen Zusammenhänge zwischen konkretem und abstraktem Wertebereich,

unter denen der mittels abstrakter Interpretation berechnete abstrakte Fixpunkt eine korrekte Approximation des konkreten Fixpunktes ist, werden in [119] beschrieben. An gleicher Stelle ist beschrieben, wie abstrakte Wertebereiche nach Bedarf konstruiert werden können. Bei den abstrakten Wertebereichen wird dabei zwischen relationalen und nichtrelationalen abstrakten Wertebereichen unterschieden. Die relationalen DBM-Wertebereiche wurden zunächst zur Analyse zeitbehafteter Petri-Netze verwendet [471]. Eine höhere Expressivität besitzen Oktagon- [327] und Oktaeder-Wertebereiche [96]. Polyeder-Wertebereiche wurden erstmals zur Verifikation nummerischer Eigenschaften von Programmen und des Zeitverhaltens eingebetteter Software verwendet [120, 213]. Ein weiteres großes Anwendungsgebiet statischer Programmanalyse ist die Analyse von Programmen, die Zeiger verwenden. Die sog. *Alias-Analyse* untersucht, ob zwei Zeiger die selbe Speicherstelle adressieren. Die Ermittlung der Speicherstelle, auf die ein Zeiger zugreift, wird als engl. *point to analysis* bezeichnet [220]. Eine Generalisierung auf dynamisch erzeugte Datenstrukturen erfolgt in [376, 248] und wird als engl. *shape analysis* bezeichnet.

Die Zusammenhänge und Unterschiede zwischen statischer Programmanalyse und Modellprüfung sind in [410, 389, 46] diskutiert. Werkzeuge zur Modellprüfung von Software können generell in *explizite* und *symbolische* Modellprüfer unterschieden werden. Der wohl bekannteste Vertreter expliziter Modellprüfer ist *SPIN* [222]. Erste Versionen des *Java Pathfinder* haben auf SPIN aufgesetzt [456]. Neben SPIN und Java Pathfinder gibt es mit *CMC* [339], *ZING* [15] und *VeriSoft* [195] drei weitere wichtige Vertreter expliziter Modellprüfer. Letzterer ist zustandslos, d. h. besuchte Zustände werden nicht gespeichert, und begegnet damit der Zustandsexplosion. Allerdings ist dieses Verfahren unvollständig für Systeme, die Zyklen enthalten. Für die Terminierung muss zusätzlich die Suchtiefe beschränkt werden.

Modellprüfungsverfahren, die auf Abstraktionsverfeinerung beruhen, verwenden zur Erstellung der abstrakten Modelle heutzutage Prädikatenabstraktion [205]. Ein iteratives Verfahren zur Prädikatenabstraktion ist unter den Namen CEGAR (engl. *counterexample-guided abstraction refinement*) bekannt geworden [23, 100]. Die Entscheidungsprozeduren, die zur Bestimmung des abstrakten Modells benötigt werden, basieren entweder auf Theoremlösern [24, 131] oder SAT-Solvern [103, 114]. Ein erstes Modellprüfungsverfahren auf Basis von Prädikatenabstraktion mit dem Namen *SLAM* wurde von der Firma Microsoft entwickelt [25]. SLAM besteht aus mehreren Werkzeugen wie C2BP [26] zur Prädikatenabstraktion, dem symbolischen Modellprüfer BEBOP [27] sowie dem Simulations- und Verfeinerungswerkzeug NEWTON [28]. Während SLAM Theoremlöser zur Prädikatenabstraktion einsetzen, verwendet *SATABS* einen SAT-Solver [103]. Für die eigentliche Verifikation setzt SATABS den QBF-Solver-basierten Modellprüfer BOPPO [115] ein.

Eine der ersten Implementierungen eines SAT-basierten Modellprüfers für C-Programme heißt *CBMC*, welches unterschiedliche Prozessor- und Speicherarchitekturen unterstützt [104, 102]. Der Ansatz zur Zustandsraumreduktion durch Schleifenabwicklung geht auf Currie et al. zurück [123]. Die grundblockbasierte SAT-basierte Modellprüfung in [242] unterscheidet sich von BMC durch die Abstraktion vom Programmzähler, wodurch es u. a. leichter wird, Funktionsaufrufe zu modellieren. Eine Erweiterung von CBMC auf SMT-Solver unter Verwendung von ganzzahliger

linearer Arithmetik ist in [17, 18] beschrieben. Der einzige SAT-basierte Modellprüfungsansatz der ein Abrollen der Zustandsübergangsrelation vornimmt ist *F-Soft* [241].

WCET-Analyse im Allgemeinen ist nicht entscheidbar. Sowohl Kligerman and Stoyenko [263] als auch Puschner and Koza [368] haben die Bedingungen untersucht, unter denen das Problem entscheidbar wird. Zu diesen Bedingungen zählen beschränkte Schleifen, Abwesenheit von rekursiven Funktionsaufrufen und Abwesenheit von dynamischen Funktionsaufrufen. Eine ILP-basierte WCET-Analyse ist in [302] vorgestellt. Eine Erweiterung um die Berücksichtigung eines Instruktionscaches ist in [303] beschrieben. In [428] zeigen Theiling et al., dass es möglich ist, die Programmpfadanalyse von der Analyse der Zielarchitektur zu separieren, indem Ergebnisse der Zielarchitekturanalyse durch *abstrakte Interpretation* in dem ILP zur Bestimmung des längsten Programmpfads verwendet werden.

Dynamische Ablaufplanungsverfahren und Echtzeitbedingungen sind ausführlich in den Büchern [262], [308] und [79] präsentiert. Detaillierte Informationen über EDF enthält das EDF-Buch [408]. In [468, 78] werden auch interessante Vergleiche zwischen ratenmonotoner Ablaufplanung (RMS) und EDF beschrieben. Erweiterungen der ratenmonotonen Ablaufplanung unter dem Namen *deadlinemonotone Ablaufplanung* sind von Leung und Whitehead [301] sowie Lehoczky und Sha [300] betrachtet worden, wobei die Deadlines von Prozessen nicht notwendigerweise gleich ihrer Perioden sind. Joseph und Pandya entwickelten einen ersten Ansatz zur Antwortzeitanalyse bei ratenmonotoner Ablaufplanung [249]. Diese Antwortzeitanalyse erlaubt die Betrachtung beliebiger Prioritätszuweisungen. Er gilt daher insbesondere auch im Falle der deadlinemonotonen Ablaufplanung, wenn die Deadlines kleiner gleich den Perioden der Prozesse sind. Für den Fall, dass Deadlines größer als die Perioden eines Prozesses sein dürfen, besteht das Problem, dass Instanzen einer folgenden Iteration ankommen können, bevor die Ausführung von Instanzen der aktuellen Iteration beendet ist. Im Ansatz von Lehoczky [299] werden dazu Fenster von mehreren aufeinander folgenden Iterationen zusammen analysiert und daraus eine kombinierte Antwortzeit berechnet. Von Audsley [21] und Tindell [439] stammen schließlich Erweiterungen der Antwortzeitanalyse, die auch das Phänomen nicht exakt periodisch ankommender Prozesse, sondern sog. engl. *release jitter* zulassen sowie das Auftreten von periodischen sporadischen Prozessen, sog. engl. *sporadic bursts*. Für EDF-Ablaufplanung wird eine Antwortzeitanalyse in [406, 407] beschrieben.

Bei den zeitgetriebenen, dynamischen Ablaufplanungsverfahren sind die Round-Robin- und TDMA-Ablaufplanung die prominentesten Vertreter. Kopetz gibt in seinem Buch [266] eine gute Übersicht über TDMA-Ablaufplanung und der kommerziellen Anwendung im sog. engl. *Time-Triggered Protocol* (*TTP*) [443]. Andere industrielle Anwendungen spiegeln sich wieder im sog. *FlexRay*-Busstandard [165]. Im Mittel führt Round-Robin-Ablaufplanung zu besseren Ergebnissen, da bei TDMA ungenutzte Zeitslots für andere Prozesse nicht zur Verfügung stehen. Man kann zeigen, dass Round-Robin-Ablaufplanung aber im schlechtesten Fall das Verhalten von TDMA besitzt. Daher kann man die TDMA-Analyse auch zur Analyse von Round-Robin-Ablaufplanung einsetzen.

8

Systemverifikation

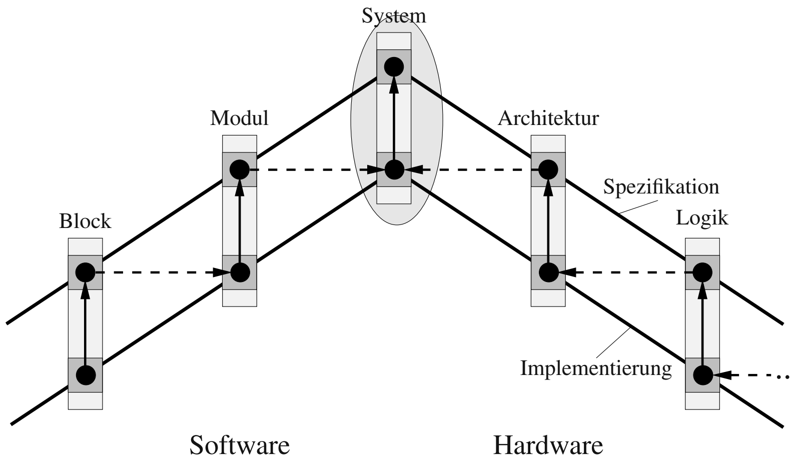


Abb. 8.1. Verifikation auf der Systemebene

Der technologische Fortschritt ermöglicht es, Systeme mit einer immer größeren Komplexität zu bauen. Um dabei mit den Entwurfsmethoden Schritt zu halten, ist auch ein Schritt auf eine höhere Abstraktionsebene notwendig, die es erlaubt, Systeme unabhängig von ihrer späteren Aufteilung in Hardware- und Software-Komponenten zu betrachtet. Diese Abstraktionsebene wird als Systemebene (engl. *Electronic System Level*) bezeichnet.

Auf Systemebene wird das Systemverhalten oftmals als eine Menge kommunizierender Prozesse beschrieben. Insbesondere, wenn ausführbare Verhaltensmodelle zum Einsatz kommen, werden diese in zunehmenden Maße mit der Systembeschreibungssprache SystemC beschrieben.

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6_8, © Springer-Verlag Berlin Heidelberg 2010

Andererseits sind die Strukturmodelle auf Systemebene eine Netzliste aus Komponenten mit der Granularität von Prozessoren, Hardware-Beschleunigern, Speichern und Bussen. Auch diese Strukturmodelle werden zunehmend mit der Systembeschreibungssprache SystemC modelliert. Von besonderem Interesse ist dabei die Interaktion der einzelnen Komponenten. Diese wird typischerweise als Transaktionen modelliert. Die resultierenden Modelle werden als *Transaktionsebenenmodelle* (engl. *Transaction Level Models, TLMs*) bezeichnet.

Transaktionsebenenmodelle spielen bei der Verifikation des Zeitverhaltens bereits heutzutage eine zentrale Rolle. Der Grund hierfür ist, dass TLMs aufgrund der hohen Abstraktionsebene eine schnelle Simulation ermöglichen. Dabei liefern sie bereits gute Abschätzungen für nichtfunktionale Eigenschaften. Neben der Verbesserung simulativer Verfahren zur Verifikation des Zeitverhaltens, gab es in den letzten zehn Jahren enorme Fortschritte im Bereich der formalen Verifikation des Zeitverhaltens auf der Systemebene.

Im Folgenden werden Ansätze zur Eigenschaftsprüfung auf der Systemebene behandelt. Dabei werden zunächst formale Modellprüfungsverfahren für SystemC-Verhaltensmodelle präsentiert. Anschließend werden formale und simulative Verfahren zur funktionalen Eigenschaftsprüfung von SystemC-Transaktionsebenenmodellen vorgestellt. Zum Abschluss werden simulative und formale Methoden zur Zeitanalyse auf Systemebene diskutiert.

8.1 Funktionale Eigenschaftsprüfung von SystemC-Modellen

Im Folgenden werden Methoden zur funktionalen Eigenschaftsprüfung von SystemC-Modellen vorgestellt.

8.1.1 Symbolische CTL-Modellprüfung von SysteMoC-Modellen

Dieser Abschnitt ist der symbolischen CTL-Modellprüfung von SysteMoC-Modellen (siehe Abschnitt 2.3.2) gewidmet. Die prinzipielle Vorgehensweise ist in [191] beschrieben. Diese basiert auf Arbeiten über symbolische Techniken für FunState-Modelle, wie sie in [414] zusammengefasst sind. Im Folgenden wird zunächst der Zustandsraum eines SysteMoC-Modells genauer definiert und dieser anschließend mit Hilfe von Intervall-Entscheidungsdiagrammen symbolisch repräsentiert. Die symbolische Modellprüfung wird anschließend diskutiert.

Repräsentation des Zustandsraums

Für die später vorgestellte symbolische CTL-Modellprüfung wird eine symbolische Darstellung von Zuständen und Zustandsübergängen von SysteMoC-Modellen benötigt. Aufgrund der Komplexität von SysteMoC-Modellen wird bei der Repräsentation der Zustandsräume eine Abstraktion auf das zugrundeliegende FunState-Modell betrachtet. Dabei werden Datenwerte nicht berücksichtigt. Der (abstrakte)

Zustand eines SysteMoC-Modells ist dann durch die momentanen Zustände der endlichen Automaten der Aktoren im SysteMoC-Modell und die Füllstände der Kanäle gegeben.

Reguläre Zustandsautomaten

Der abstrakte Zustandsraum eines SysteMoC-Modells ist aufgrund der unbeschränkten FIFO-Kanäle potentiell unendlich groß. Eine endliche Repräsentation des Zustandsraums für SysteMoC-Modelle kann allerdings über sog. *Reguläre Zustandsautomaten* (engl. *Regular State Machines, RSMs*) [434] erreicht werden.

RSMs sind dafür geeignet, reguläre Zustandsänderungen von nebenläufigen Kontroll- und/oder Datenflussmodellen (unter anderem Petri-Netze, FunState- und SysteMoC-Modellen) zu beschreiben. Dabei können RSMs bestimmte Klassen von unendlichen Zustandssystemen modellieren und erweitern die Klasse der endlichen Automaten. RSMs werden hier lediglich in einer eingeschränkten Form vorgestellt und über ihre graphischen Repräsentationen als *statische* und *dynamische Zustandsübergangsdiagramme* eingeführt. Erweiterungen sind in [434] zu finden.

Definition 8.1.1 (Statisches Zustandsdiagramm einer RSM). *Ein* statisches Zustandsübergangsdiagramm *ist ein gerichteter Graph* $G = (V, E, D, P, v_0, I_0)$ *mit*

- *einer Menge von Knoten* V,
- *einer Menge gerichteter Kanten* $E \subseteq V \times V$,
- *einer Funktion* $D : E \to \mathbb{Z}^m$, *die jeder Kante* $e = (v_i, v_j) \in E$ *einen* Distanzvektor *ganzer Zahlen* $d(e) = d(v_i, v_j) \in \mathbb{Z}^m$ *der Dimension* m *zuordnet,*
- *einer* Prädikatenfunktion $P : E \times \mathbb{Z}^m \to \mathbb{B}$, *die jeder Kante* $e \in E$ *ein Prädikat* $P(e, I)$ *zuordnet und*
- *einem Knoten* $v_0 \in V$ *und einem Vektor* $I_0 \in \mathbb{Z}^m$, *die den Anfangszustand bilden.*

Das statische Zustandsübergangsdiagramm ist eine abkürzende Schreibweise für das möglicherweise unendliche, dynamische Zustandsübergangsdiagramm einer RSM. Das dynamische Zustandsübergangsdiagramm ergibt sich, indem die Zustände des statischen Zustandsübergangsdiagramms an jeden gültigen Indexpunkt, der sich aus den Distanzvektoren unter Berücksichtigung der Prädikate ergibt, kopiert werden, und anschließend die Zustandsübergänge entsprechend zwischen den Zuständen und Indexpunkten eingetragen werden.

Definition 8.1.2 (Dynamisches Zustandsübergangsdiagramm einer RSM). *Das* dynamische Zustandsübergangsdiagramm $G_\mathrm{d} = (X, T, x_0)$ *eines gegebenen statischen Zustandsübergangsdiagramms* $G = (V, A, D, P, v_0, I_0)$ *ist ein unendlicher, gerichteter Graph, der wie folgt definiert ist:*

- *Die Knoten* $x \in X$ *repräsentieren* (dynamische) Zustände *des dynamischen Zustandsübergangsdiagramms. Es gilt* $X = V \times \mathbb{Z}^m$ *mit der* Indexmenge *des dynamischen Zustandsübergangsdiagramms* $\mathbb{Z}^m$. *Der Knoten* $x = (v, I) \in X$ *bezeichnet den Zustand für einen Knoten* $v \in V$ *eines statischen Zustandsübergangsdiagramms und einen Indexpunkt* $I \in \mathbb{Z}^m$.

- *Der Zustand* $x_0 = (v_0, I_0)$ *ist der Anfangszustand des dynamischen Zustandsübergangsdiagramms.*
- *Die Kanten T stellen* (dynamische) Transitionen *des dynamischen Zustandsübergangsdiagramms dar. Es gibt eine Kante* $t = (x_1, x_2) \in T$ *von Zustand* $x_1 = (v_1, I_1) \in X$ *zu Zustand* $x_2 = (v_2, I_2) \in X$*, genau dann, wenn* $e = (v_1, v_2) \in A$, $I_2 - I_1 = d(v_1, v_2)$ *und* $P(e, I_1) = \mathsf{T}$ *gilt.*

Damit lassen sich RSMs mit einer für Transitionssysteme üblichen Semantik definieren.

Definition 8.1.3 (Semantik einer RSM). *Das Verhalten eines* regulären Zustandsautomaten *mit einem dynamischen Zustandsübergangsdiagramm* $G_\mathrm{d} = (X, T, x_0)$ *ist wie folgt definiert:*

- *Anfangs ist der reguläre Zustandsautomat im Zustand* x_0.
- *Eine Transition* $x_1 \xrightarrow{t} x_2$ *verändert den Zustand einer RSM von* $x_1 \in X$ *zu* $x_2 \in X$*, wobei* $t = (x_1, x_2) \in T$ *nichtdeterministisch aus allen Transitionen t mit Anfangszustand* x_1 *gewählt wird.*

Mit einem *Pfad* durch ein dynamisches Zustandsübergangsdiagramm ist eine konkrete Folge von dynamischen Zuständen und Transitionen gemeint, die eindeutig von einem Indexpunkt in dem dynamischen Zustandsübergangsdiagramm zu einem anderen Indexpunkt führt.

Beispiel 8.1.1. Abbildung 8.2a) zeigt ein statisches Zustandsübergangsdiagramm. Das entsprechende dynamische Zustandsübergangsdiagramm ist in Abb. 8.2b) dargestellt. An den Kanten des statischen Zustandsübergangsdiagramms ist der jeweilige Distanzvektor annotiert und, falls vorhanden, ein Prädikat vorangestellt (abgetrennt mit /).

Repräsentation von SysteMoC-Modellen als RSMs

Um den abstrakten Zustandsraum eines SysteMoC-Modells (N, M, I, O) mit Netzgraph (A, C, E) (siehe Definition 2.3.1 auf Seite 68) als RSM zu repräsentieren, wird im Folgenden gezeigt, wie ein statisches Zustandsübergangsdiagramm nach Definition 8.1.1 für ein gegebenes SysteMoC-Modell aufgebaut wird. Der abstrakte Zustandsraum eines SysteMoC-Modells setzt sich aus

1. den möglichen Zuständen der endlichen Automaten aller SysteMoC-Aktoren und
2. den möglichen FIFO-Füllständen zusammen.

In einem ersten Schritt wird deshalb der Produktautomat M_p mit Zustandsmenge Q_p und Zustandsübergangsrelation R_p aller endlichen Automaten der Aktoren im SysteMoC-Modell gebildet. Basierend auf dem Produktautomaten kann anschließend das statische Zustandsübergangsdiagramm $G = (V, E, D, P, v_0, I_0)$ wie folgt gebildet werden:

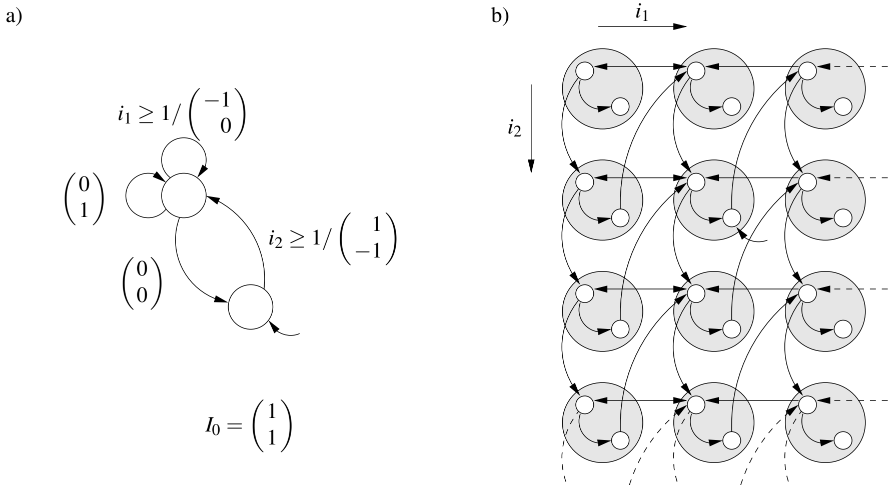


Abb. 8.2. a) statisches und b) dynamisches Zustandsübergangsdiagramm [414]

- Die Menge der Knoten V ist die Menge der Zustände Q_p.
- Die Kanten E sind die Zustandsübergangsrelationen in R_p, also $\forall e_i = (q,q') \in E : \exists (q,q') \in R_p$.
- Die Dimension m der Distanzvektoren aus $\mathbb{Z}^m$ entspricht der Anzahl der Kanäle in dem SysteMoC-Modell, also $m = |C|$.
- Die Funktion D kann aus den Attributen der Zustandsübergänge $(q,q') \in R_p$ extrahiert werden, indem aus den Konsumptions- und Produktionsraten des Zustandsübergangs für die Kante $e_i = (q,q') \in E$ der Distanzvektor für die Kante bestimmt wird. Dabei werden Produktionsraten zu positiven Elementen des Vektors und Konsumptionsraten zu negativen Elementen. Nicht beteiligte Kanäle erhalten darin den Wert null.
- Die Prädikate $P(e,I)$ werden so gewählt, dass kein Indexpunkt mit negativen Elementen und kein Indexpunkt, der in einem Element die Kanalbeschränkung verletzt, erreicht werden kann.
- Anfangszustand v_0 ist der aus allen Anfangszuständen des endlichen Automaten der Aktoren zusammengesetzte Zustand.
- Der Anfangsindexpunkt I_0 wird aus der Anfangsmarkierung der Kanäle bestimmt.

Beispiel 8.1.2. Abbildung 8.3 zeigt ein SysteMoC-Modell und das daraus konstruierte statische Zustandsübergangsdiagramm. In dem SysteMoC-Modell wird durch jede Transition eine Marke von Kanal c_1 nach Kanal c_2 verschoben, der rechte Aktor kann dabei je nach Ergebnis von $f_{\texttt{check}}$ auch seinen Zustand wechseln. Die Kanäle c_1 und c_2 haben jeweils die Kapazität von acht Marken und beinhalten jeweils eine Marke im Anfangszustand. Da der linke Aktor lediglich einen Zustand besitzt,

enthält der Produktautomat lediglich zwei Zustände. In den Prädikaten an den Zustandsübergängen bezeichnen $c_1\#$ bzw. $c_2\#$ die im Kanal c_1 bzw. Kanal c_2 verfügbare Anzahl an Marken.

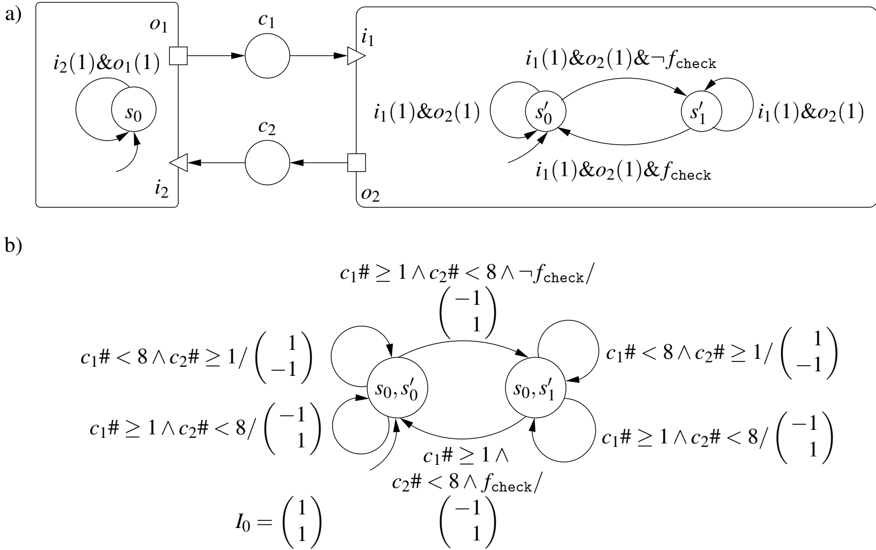


Abb. 8.3. a) SysteMoC-Modell und b) zugehöriges statisches Zustandsübergangsdiagramm [193]

Symbolische Repräsentation des Zustandsraums

Es ist denkbar auf der Repräsentation von regulären Zustandsautomaten eine explizite CTL-Modellprüfung zu implementieren (siehe Abschnitt 5.2). Effizienter sind allerdings symbolische Verfahren, also Verfahren, die auf einer impliziten Repräsentation des Zustandsraums basieren (siehe auch Abschnitt 5.3). Während es zunächst scheint, dass ROBDDs eine mögliche symbolische Repräsentation des Zustandsraums für SysteMoC-Modelle darstellen, kommen diese bei einer genauen Betrachtung nicht mehr in Frage. Die potentiell unbeschränkten FIFO-Kanäle im SysteMoC-Modell, die einen potentiell unendlichen Zustandsraum aufspannen, lassen sich durch BDDs nicht repräsentieren.

Aus diesem Grund werden im Folgenden sog. *Intervall-Entscheidungsdiagramme* (engl. *Interval Decision Diagram, IDD*) zur symbolischen Repräsentation des Zustandsraums für SysteMoC-Modelle betrachtet. Neben IDDs wird eine zweite Klasse von Intervalldiagrammen benötigt, die sog. *Intervall-Abbildungsdiagramme* (engl. *Interval Mapping Diagram, IMD*), um Zustandsübergänge symbolisch zu repräsentieren. IDDs und IMDs bilden zusammen die Grundlage für effiziente symbolische Verfahren wie Modellprüfung auf einer Vielzahl von Berechnungsmodellen,

z. B. Prozessnetzwerke [415], Petri-Netze [416] und FunState-Modelle [414] oder für symbolische Ablaufplanung von FunState- [418, 414] und SysteMoC-Modellen [192].

IDDs und IMDs werden im Folgenden mit der Beschränkung auf ganzzahlige Intervalle eingeführt (siehe auch [414, 193]).

Intervall-Entscheidungsdiagramme (IDDs)

Intervall-Entscheidungsdiagramme (IDDs) erlauben die Darstellung von Funktionen mit n Variablen und einem diskreten, endlichen Definitionsbereich. Weitere notwendige Eigenschaften der Funktionen werden weiter unten diskutiert. Sei $f(x_1, x_2, ..., x_n)$ eine Funktion mit $f : A_1 \times A_2 \times ... \times A_n \rightarrow B$, mit $A_i \subseteq \mathbb{Z}$ für die Definitionsbereiche aller x_i und $B \subset \mathbb{Z}$, wobei die Mächtigkeit von B endlich ist. Variablen in A_i seien Intervalle auf $\mathbb{Z}$, dargestellt mit $[a_l, a_u]$. Dabei bezeichnet a_l die untere Schranke und a_u die obere Schranke des *geschlossenen Intervalls*. Das geschlossene Intervall enthält sowohl a_l als auch a_u. Intervalle, bei denen eine oder beide Grenzen unendlich bzw. minus unendlich sind, werden als *halboffene* oder *offene Intervalle* bezeichnet. Das Zeichen ∞ wird für eine unendliche obere Schranke, das Zeichen $-\infty$ für eine unendliche untere Schranke eingesetzt. Offene Enden eines Intervalls werden durch (bzw.) gekennzeichnet, z. B. $(-\infty, a_u)$. In diesem Fall enthält das Intervall weder $-\infty$ noch a_u.

Definition 8.1.4 (Intervallüberdeckung). *Die Menge $\Delta(A_i) = \{\Delta_1, \Delta_2, \ldots, \Delta_{|\Delta(A_i)|}\}$ aus $|\Delta(A_i)|$ Teilintervallen Δ_j nennt man* Intervallüberdeckung *von A_i, genau dann, wenn jedes Δ_j ein Teilintervall von A_i ist, d. h. $\Delta_j \subseteq A_i$, und $\Delta(A_i)$ vollständig ist, d. h. $A_i = \bigcup_{\Delta_j \in \Delta(A_i)} \Delta_j$ erfüllt ist.*

Eine Intervallüberdeckung eines Definitionsbereichs deckt diesen also exakt und lückenlos ab, erlaubt aber Überschneidungen der Intervalle. Eine *disjunkte Intervallüberdeckung* verbietet genau diese Überschneidungen. Mit anderen Worten: Jedes Element aus A_i muss in genau einem Teilintervall der Intervallpartition $\Delta(A_i)$ enthalten sein.

Definition 8.1.5 (Disjunkte Intervallüberdeckung). *Gegeben sei eine Intervallüberdeckung $\Delta(A_i)$. $\Delta(A_i)$ heißt* disjunkt *oder auch* Intervallpartition, *genau dann, wenn $\forall 1 \leq s, t \leq |\Delta(A_i)|, s \neq t : \Delta_s \cap \Delta_t = \emptyset$ gilt.*

Gegeben sei eine Funktion f und ein Intervall $\Delta_j \in \Delta(A_i)$. Der Kofaktor bezüglich x_i sei bezeichnet durch $f|_{x_i:=b}(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$. Falls für alle Belegungen von x_i aus dem Intervall Δ_j gilt, dass $\forall b, c \in \Delta_j : f|_{x_i:=b} = f|_{x_i:=c}$ ist, so sagt man, dass f unabhängig von $x_i \in \Delta_j$ ist. Der zugehörige Kofaktor wird mit $f|_{x_i:\in\Delta_j}$ bezeichnet. Das Intervall Δ_j wird dann als *Unabhängigkeitsintervall* (engl. *Independence Interval, II*) von f in Bezug auf x_i bezeichnet. Sind alle Teilintervalle einer Intervallpartition $\Delta(A_i)$ Unabhängigkeitsintervalle, heißt $\Delta(A_i)$ *Unabhängigkeitsintervallpartition* (engl. *Independence Interval Partition, IIP*).

Intervalle heißen schließlich *benachbart*, wenn sie zu einem einzelnen Intervall zusammengefasst werden können. Dabei dürfen die Intervalle auch überlappen.

Definition 8.1.6 (Reduzierte Intervallpartition). *Gegeben sei ein IIP* $\Delta(A_i) = \{\Delta_1, \Delta_2, \ldots, \Delta_{|\Delta(A_i)|}\}$. $\Delta(A_i)$ *heißt genau dann* minimal*, wenn sie keine benachbarten Teilintervalle enthält, die zu einem Unabhängigkeitsintervall zusammengefasst werden können.* $\Delta(A_i)$ *heißt* geordnet*, genau dann, wenn alle oberen Schranken aller Teilintervalle in Bezug auf ihren Index der Größe nach aufsteigend geordnet sind. Ein IIP, das minimal und geordnet ist, heißt* reduziert.

Ein reduziertes IIP einer Funktion ist eindeutig [414]. Das folgende Beispiel illustriert die obigen Definitionen. Es stammt aus [414]

Beispiel 8.1.3. Gegeben ist die folgende Funktion

$$f(x_1, x_2, x_3) := \begin{cases} \mathsf{F} \text{ falls } x_1^{[0,3]} \wedge x_2^{[0,5]} \vee x_1^{[4,5]} \vee x_1^{[6,\infty)} \wedge x_3^{[0,7]} \\ \mathsf{T} \text{ sonst} \end{cases}$$

Der Ausdruck $x^{[a_l,a_u]}$ ist ein *Literal* einer Variablen x in Bezug auf das Intervall $[a_l, a_u]$ und wird durch folgende Funktion definiert (siehe auch Anhang B.1):

$$x^{[a_l,a_u]} = \begin{cases} \mathsf{F} \text{ wenn } x \notin [a_l, a_u] \\ \mathsf{T} \text{ wenn } x \in [a_l, a_u] \end{cases}$$

Der Definitionsbereich für x_1, x_2, x_3 ist $A_1 = A_2 = A_3 = [0, \infty)$. Die Wertemenge der Funktion f ist $B = \{\mathsf{F}, \mathsf{T}\} = \mathbb{B}$.

Der Kofaktor $f|_{x_1 :\in [4,5]}(x_1, x_2, x_3)$ ist gleich F. Die Intervalle $[0,7]$ und $[4,5]$ sind Unabhängigkeitsintervalle der Funktion f bezüglich der Variablen x_3. Dies gilt jedoch nicht für das Intervall $[6,9]$. Die Menge $\Delta(A_1) = \{\Delta_1, \Delta_2, \Delta_3\}$ mit $\Delta_1 = [0,3]$, $\Delta_2 = [4,5]$ und $\Delta_3 = [6,\infty)$ ist eine Intervallpartition. Für f ist $\Delta(A_1)$ eine reduzierte IIP.

Darstellung von IDDs

IDDs können, wie andere Entscheidungsdiagramme, graphisch dargestellt werden (siehe Anhang B.1). Sie dienen der Repräsentation von mehrwertigen Funktionen, welche sich mit Hilfe von *Intervallpartitionen* mit endlich vielen Unabhängigkeitsintervallen zerlegen lassen.

Definition 8.1.7 (IDD). *Ein* Intervall-Entscheidungsdiagramm *(*IDD*) für eine Funktion* $f : A_1, \ldots, A_n \to \mathbb{B}$ *ist ein gerichteter azyklischer Graph* $G = (V, E)$ *mit Knotenmenge* V *und Kantenmenge* E *sowie den folgenden Eigenschaften:*

- *G ist ein Baum,*
- *V ist eine Partition von Terminalknoten V_T und Nichtterminalknoten V_N,*
- *eine Funktion* $\mathrm{index} : V_\mathrm{N} \to \{1, \ldots, n\}$ *weist jedem Nichtterminalknoten $v \in V_\mathrm{N}$ eine Variablen $x_{\mathrm{index}(v)}$ zu,*
- *alle Intervallpartitionen $\Delta(A_{\mathrm{index}(v)}) = \{\Delta_1, \Delta_2, \ldots, \Delta_{|\Delta(A_{\mathrm{index}(v)})|}\}$ sind IIP,*
- *für jeden Nichtterminalknoten $v \in V_\mathrm{N}$ weist eine Funktion* $\mathrm{child} : V_\mathrm{N} \times \{1, \ldots, |\Delta(A_{\mathrm{index}(v)})|\} \to V$ *dem Knoten v seine Nachfolger zu, d. h.* $(v, \mathrm{child}(v,k)) \in E$, *mit* $1 \leq k \leq |\Delta(A_{\mathrm{index}(v)})|$,

- *eine Funktion* value : $V_T \to \mathbb{B}$ *weist jedem Terminalknoten einen Wert aus der Zielmenge* $\mathbb{B}$ *zu.*

Bei Entscheidungsdiagrammen werden Funktionen als äquivalent zu ihrem zugehörigen Knoten betrachtet. Eine Funktion f_v mit zugehörigem Knoten v und Index $\text{index}(v)$ kann auch als $(|\Delta(A_{\text{index}(v)})|+1)$-Tupel dargestellt werden:

$$f_v = (x_{\text{index}(v)}, (\Delta_1, F_1), \ldots, (\Delta_{|\Delta(A_{\text{index}(v)})|}, F_{|\Delta(A_{\text{index}(v)})|})) \tag{8.1}$$

Dabei besteht jedes Paar (Δ_k, F_k) aus einem Intervall Δ_k der Intervallpartition $\Delta(A_i)$ und einer Funktion F_k beziehungsweise dem dieser zugehörigem Knoten $\text{child}(v,k)$.

Die Funktion f_v mit zugehörigem Knoten $v \in V$ wird rekursiv definiert:

- Ist v ein Terminalknoten, so gilt $f_v := \text{value}(v)$.
- Ist v ein Nichtterminalknoten mit Index $\text{index}(v)$, dann ist der Wert von f_v gegeben durch die Boole-Shannon Zerlegung

$$f_v := \bigvee_{\Delta_j \in \Delta(A_{\text{index}(v)})} x_{\text{index}(v)}^{\Delta_j} \wedge f_{\text{child}(v,j)}$$

Die Argumente einer Funktion beschreiben also einen Pfad durch das zugehörige IDD. Der Pfad beginnt bei dem Quellknoten und endet in einem Terminalknoten, der dem Funktionswert entspricht. Die durch das IDD darzustellende Funktion f ist dem Quellknoten des IDD zugeordnet.

Beispiel 8.1.4. Das IDD für die Funktion aus Beispiel 8.1.3 ist in Abb. 8.4 zu sehen.

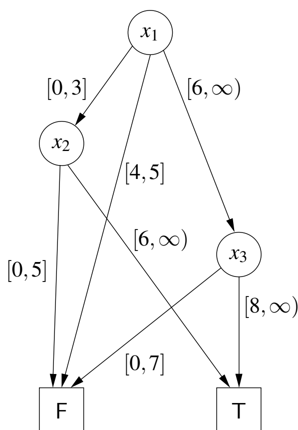


Abb. 8.4. Intervall-Entscheidungsdiagramm für die Funktion f aus Beispiel 8.1.3 [414]

Um eine kanonische Darstellung von Funktionen durch IDDs zu erreichen, wird eine Variablenordnung benötigt. Ein IDD heißt *geordnet* (engl. *Ordered Interval Decision Diagram, OIDD*), genau dann, wenn auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten. Ein OIDD heißt *reduziert* (engl. *Reduced Ordered Interval Decision Diagram, ROIDD*), genau dann, wenn

1. jeder Nichtterminalknoten $v \in V_{\mathrm{N}}$ mindestens zwei voneinander verschiedene Nachfolger besitzt,
2. er nicht zwei unterschiedliche Knoten v und v' enthält, die Quellknoten von isomorphen Teilgraphen sind, und
3. die IIPs $\Delta(A_{\mathrm{index}(v)})$ aller Nichtterminalknoten $v \in V_{\mathrm{N}}$ reduziert sind.

Zwei Teilgraphen eines IDD mit Quellknoten $v, v' \in V$ und $v \neq v'$ heißen *isomorph*, wenn die durch sie repräsentierten Funktionen äquivalent sind, also $f_v \equiv f_{v'}$ gilt. Isomorphe Teilgraphen können, ähnlich zu ROBDDs, in der Darstellung zusammengefasst werden. Um ein OIDD zu reduzieren, werden folgende Reduktionsregeln angewendet:

1. *Elimination*: Wenn alle ausgehenden Kanten eines Nichtterminalknotens $v \in V_{\mathrm{N}}$ in dem gleichen Knoten v' enden, wird Knoten v entfernt und alle vorher in v eingehenden Kanten zeigen auf v'.
2. *Verschmelzung*: Wenn unterschiedliche Nichtterminalknoten mit gleichem Index und gleicher Intervallpartition existieren, und diese jeweils die gleichen Nachfolger haben (oder wenn unterschiedliche Terminalknoten $v \in V_{\mathrm{T}}$ mit gleichem Wert existieren), werden all diese bis auf einen Knoten entfernt und deren eingehende Kanten zeigen auf den verbleibenden Knoten.
3. *Normalisierung*: Wenn eine IIP eines Nichtterminalknotens $v \in V_{\mathrm{N}}$ nicht minimal und geordnet ist, muss diese reduziert werden. Dafür werden die Teilintervalle und ihre zugehörigen Nachfolger geordnet. Kanten aus v mit benachbarten Intervallen, die im gleichen Nachfolger enden, werden durch eine Kante ersetzt, die alle diese benachbarten Intervalle zusammenfasst.

Die Reduktionsregeln 1. und 2. sind in Abb. 8.5 dargestellt. Für alle x_i gilt der Definitionsbereich $A_i = [0, \infty)$. Strehl und Thiele beweisen in [415], dass ROIDDs eine kanonische Darstellung für mehrwertige Funktionen sind, welche sich mit Hilfe von Intervallpartitionen mit endlich vielen Unabhängigkeitsintervallen zerlegen lassen.

Intervall-Abbildungsdiagramme

Für die Manipulation von IDDs werden sog. *Intervall-Abbildungsdiagramme* (engl. *Interval Mapping Diagrams, IMDs*) verwendet. Bevor diese formal eingeführt werden, wird eine einfache Intervallarithmetik als Grundlage präsentiert. Mit dieser können beispielsweise zwei nichtleere Intervalle addiert oder subtrahiert werden. Die Argumente und das Ergebnis dieser Operationen sind jeweils Intervalle aus $\mathbb{Z}$. Die Definitionen zeigen der Übersicht halber nur geschlossene Intervalle.

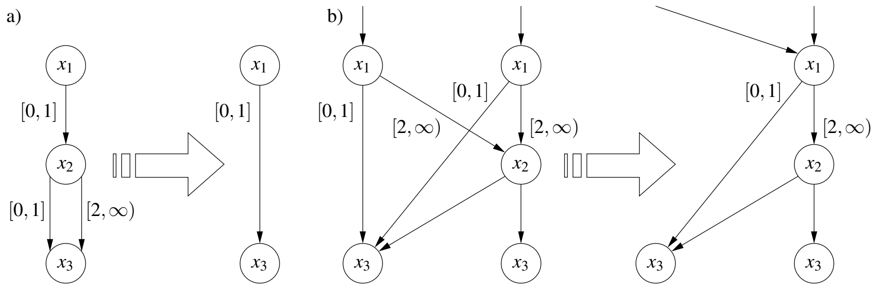


Abb. 8.5. Reduktionsregeln für Intervall-Entscheidungsdiagramm [193]

Definition 8.1.8 (Intervalladdition). *Die* Addition *zweier Intervalle* $[a,b]$ *und* $[c,d]$ *ist definiert als* $[a,b]+[c,d]=[a+c,b+d]$.

Das aus der Addition resultierende Intervall enthält also alle Zahlen, die durch Addition von einer beliebigen Zahl aus dem ersten Intervall zu einer beliebigen Zahlen aus dem zweiten Intervall resultieren können.

Definition 8.1.9 (Intervallsubtraktion). *Die* Subtraktion *zweier Intervalle* $[a,b]$ *und* $[c,d]$ *ist definiert als* $[a,b]-[c,d]=[a,b]+[-d,-c]$.

Das Ergebnis der Subtraktion zweier Intervalle ist ein Intervall, das alle Zahlen enthält, die durch Subtraktion einer beliebigen Zahl aus dem Intervall $[c,d]$ von einer beliebigen Zahl aus dem Intervall $[a,b]$ resultieren können.

IMDs werden durch sog. *Abbildungsgraphen* graphisch repräsentiert.

Definition 8.1.10 (Abbildungsgraph). *Ein* Abbildungsgraph *ist ein azyklischer gerichteter Graph* $G=(V,E)$ *mit ausgezeichnetem Quellknoten. Die Menge der Knoten* V *ist in zwei Partitionen eingeteilt, die* Nichtterminalknoten V_{N} *und die* Terminalknoten V_{T}. *Jeder* Nichtterminalknoten $v \in V_{\mathrm{N}}$ *hat einen Index* $\mathrm{index}(v)$, *eine Menge von* Intervall-Abbildungsfunktionen $\mathrm{func}(v)=\{f_1,f_2,\ldots,f_n\}$ *und* $|\mathrm{func}(v)|$ *Nachfolger, bezeichnet als* $\mathrm{child}(v,k)\in V$ *mit* $1\le k\le|\mathrm{func}(v)|$. *Die Abbildungsfunktionen* $f_k(v)\in\mathrm{func}(v)$ *sind mit den entsprechenden Kanten* $(v,\mathrm{child}(v,k))\in E$ *assoziiert. Es gibt genau einen Terminalknoten* $v\in V_{\mathrm{T}}$ *mit dem Wert* $\mathrm{value}(v)=\mathsf{T}$.

Eine *Intervall-Abbildungsfunktion* $f:\mathbb{I}\to\mathbb{I}$ bildet Intervalle auf Intervalle ab, wobei $\mathbb{I}$ die Menge aller Intervalle aus $\mathbb{Z}$ beschreibt. Eine Intervall-Abbildungsfunktion $f(\Delta)=\Delta$, die ein Intervall auf sich selbst abbildet, wird als *neutral* bezeichnet. IMDs können verwendet werden, um IDDs zu manipulieren. Dabei müssen beide Diagramme die gleiche Variablenordnung verwenden. Für die Übergangsbeschreibung von SysteMoC-Modellen wird eine eingeschränkte Klasse von IMDs verwendet. Diese werden als engl. *Predicate Action Diagrams* (*PADs*) bezeichnet.

Definition 8.1.11 (PAD). *Ein* Predicate Action Diagram *(*PAD*) ist ein IMD, das nur zwei Arten von Intervall-Abbildungsfunktionen verwendet:*

$$f_+(\Delta) := \begin{cases} \Delta \cap \Delta_P + \Delta_A & \textit{falls } \Delta \cap \Delta_P \neq \emptyset \\ [] & \textit{sonst} \end{cases}$$

und

$$f_{:=}(\Delta) := \begin{cases} \Delta_A & \textit{falls } \Delta \cap \Delta_P \neq \emptyset \\ [] & \textit{sonst} \end{cases}$$

f_+ *wird* Versetzungs- *und* $f_{:=}$ Zuweisungsfunktion *genannt. Jeder Abbildungsfunktion sind zwei wählbare Intervalle zugeordnet, das* Prädikatenintervall Δ_P *und das* Aktionsintervall Δ_A.

Im Folgenden werden PADs als IMDs bezeichnet. Die Versetzungs- und Zuweisungsfunktion schneiden erst das Argumentintervall Δ mit ihrem Prädikatenintervall Δ_P. Ist die Schnittmenge leer, ist das Resultat der Funktionen das leere Intervall. Ansonsten liefert die Versetzungsfunktion ein Intervall, das aus der Addition der Schnittmenge mit dem Aktionsintervall Δ_A der Funktion resultiert. Die Zuweisungsfunktion ergibt bei einer nichtleeren Schnittmenge immer ihr Aktionsintervall. Die Abbildungsfunktionen eines IMD werden also ausschließlich durch zwei Intervalle als Parameter vollständig beschrieben.

Für die Versetzungsfunktion f_+ wird im Folgenden die Notation $\Delta_P/+\Delta_A$ und für die Zuweisungsfunktion $f_{:=}$ die Notation $\Delta_P/:=\Delta_A$ verwendet. Ein Versetzen von Intervallen in negative Richtung ist mit der Versetzungsfunktion auch möglich. Soll um ein Intervall $\Delta_A = [a,b]$ in negative Richtung versetzt werden (Intervallsubtraktion), wird das durch Addition von $[-b,-a]$ erreicht. Dies wird auch als $\Delta_P/-\Delta_A$ geschrieben. Für eine Variable x_i ist die Abbildungsfunktion $\Delta_P/+[0,0]$ mit $\Delta_A = A_i$ neutral. Eine allgemeine neutrale Zuweisungsfunktion existiert nicht. Abbildung 8.6b) zeigt ein Beispiel für ein IMD mit drei Variablen.

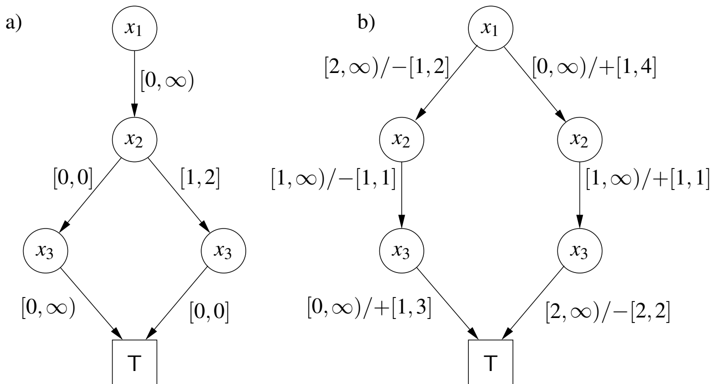


Abb. 8.6. a) Intervall-Entscheidungsdiagramm ohne Kanten zum Terminalknoten mit Wert F und b) Intervall-Abbildungsdiagramm [414]

Im Gegensatz zu IDDs ist bei IMDs im Allgemeinen keine kanonische Darstellung möglich. Dennoch können häufig, abhängig von der Struktur der repräsentierten

Funktion, Reduktionsregeln für IMDs angeben werden. Die später für die Modellprüfung vorgestellten symbolischen Bildberechnungen profitieren von kompakteren IMDs und können mit weniger Aufwand berechnet werden. Ein IMD wird mit folgenden Regeln reduziert:

1. Wenn ein Nichtterminalknoten $v \in V_N$ nur eine ausgehende Kante mit der neutralen Abbildungsfunktion in einen Knoten v' besitzt, kann v entfernt werden, und alle vorher in v eingehende Kanten zeigen auf v'.
2. Wenn unterschiedliche Nichtterminalknoten mit gleichem Index und gleicher Menge von Abbildungsfunktionen existieren, und diese jeweils die gleichen Nachfolger haben, werden all diese Knoten bis auf einen entfernt, und deren eingehende Kanten zeigen auf den übrigen Knoten.
3. Wenn ein Nichtterminalknoten $v \in V_N$ Kanten mit gleicher Abbildungsfunktion hat, werden diese durch eine Kante mit dieser Abbildungsfunktion zu einem neuen Knoten w mit Index gleich dem höchsten Index der Nachfolger von v ersetzt. Haben alle Nachfolger von v den gleichen Index, bekommt der neue Knoten w all deren ausgehende Kanten, und alle Nachfolger von v werden bis auf den neu erstellten Knoten w entfernt. Haben die Nachfolger von v verschiedene Indices, bekommt der neue Knoten w alle ausgehenden Kanten der Nachfolger mit gleichem Index, die entfernt werden. Eine Kante mit neutraler Abbildungsfunktion wird von dem Knoten w zu jedem übrigen Nachfolgerknoten eingefügt.

Anhand eines Beispiels aus [191] wird die symbolische Repräsentation von SysteMoC-Modellen vorgestellt.

Beispiel 8.1.5. Gegeben ist das SysteMoC-Modell aus Abb. 2.13 auf Seite 70. Die symbolische Repräsentation des Anfangszustands als IDD ist in Abb. 8.7a) zu sehen. Im Anfangszustand des Modells sind alle Kanäle bis auf c_6 leer. Kanal c_6 enthält anfangs eine Marke. In der Darstellung wurden zur Übersichtlichkeit alle Kanten zu dem Terminalknoten mit Wert F entfernt. Der abstrakte Zustand eines Modells, wie in Abschnitt 8.1.1 beschrieben, lässt sich wie folgt als IDD codieren:

- Für jeden FIFO-Kanal c_i wird eine Variable $c_i\#$ mit Definitionsbereich $[0, n_i]$ benötigt, wobei n_i der Tiefe des FIFO-Kanals c_i, entspricht. In diesem Beispiel wird davon ausgegangen, dass die FIFO-Kanäle eine beschränkte Größe von 8 besitzen. Somit ist der Definitionsbereich $\forall 1 \leq i \leq 6 : c_i\# \in [0, 8]$.
- Für jeden Aktor mit mehr als einem Zustand wird eine Variable mit dem Definitionsbereich $[0, s_i - 1]$ benötigt, wobei s_i gleich der Anzahl der Zustände des Aktors ist. In dem Beispiel aus Abb. 2.13 besitzt lediglich der Aktor `SqrLoop` mehr als einen Zustand, weshalb $q_1 \in [0, 1]$ gilt, wobei $q_1 = 0$ für den Zustand `start` und $q_1 = 1$ für den Zustand `loop` steht.

Abbildung 8.7b) zeigt die Erreichbarkeitsmenge des SysteMoC-Modells Abbildung 8.7c) zeigt ein IMD, welches die drei Transitionen des Aktors `SqrLoop` darstellt. Darin entspricht jeder der drei Pfade von dem Quellknoten zu dem Terminalknoten einer der Transitionen. Im Unterschied zu IDDs sind an den Kanten zwei Intervalle und eine Funktion annotiert, die der Änderung einer Variablen durch die Transition entsprechen.

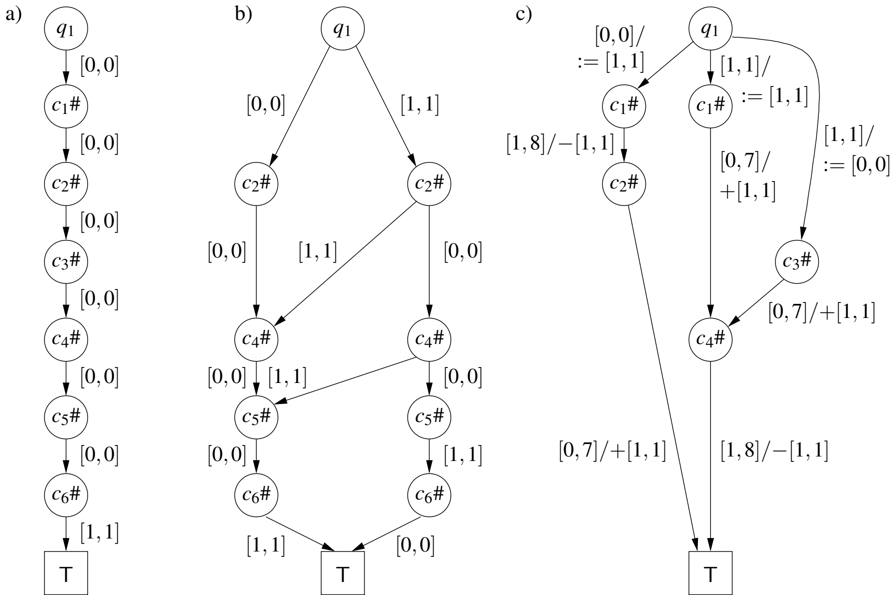


Abb. 8.7. a) Anfangszustand und b) Erreichbarkeitsmenge des SysteMoC-Modells aus Abb. 2.13 auf Seite 70 als IDD sowie c) die Zustandsübergangsrelation des Aktors `SqrLoop` als IMD [191]

Modellprüfung von SysteMoC-Modellen

Abbildung 8.8 zeigt einen Überblick der symbolischen CTL-Modellprüfung von SysteMoC-Modellen. Zunächst wird aus der Spezifikation eines digitalen Systems das SysteMoC-Verhaltensmodell mittels eines Parsers in eine RSM übersetzt, welche als IDD/IMD symbolisch repräsentiert wird. Die funktionalen Anforderungen an das System liegen in Form von CTL-Formeln vor, welche ebenfalls aus der Spezifikation extrahiert werden. Mittels eines symbolischen Modellprüfers wird überprüft, ob das SysteMoC-Modell die geforderten funktionalen Eigenschaften besitzt oder nicht. Der Modellprüfer bestätigt entweder die Einhaltung der gegebenen Formeln oder liefert ein Gegenbeispiel.

Die Anforderungen an funktionale Eigenschaften werden in CTL formuliert. Deren atomare Präpositionen sind Werte der Variablen der Intervalldiagramme und beziehen sich damit auf die Füllstände von FIFO-Kanälen oder die Zustände der endlichen Automaten in den Aktoren. Für das Beispiel aus Abb. 2.13 auf Seite 70 lässt sich etwa die Anforderung „Immer wenn `Src` eine Marke produziert, soll `Snk` auch eine Marke konsumieren können“ formulieren. Ist diese erfüllt, ist sichergestellt, dass alle Marken, die in das System eingebracht werden, auch wieder entnommen werden können. In CTL kann dies als AG $(c_1\# > 0 \Rightarrow \text{AF}\, c_3\# > 0)$ formuliert wer-

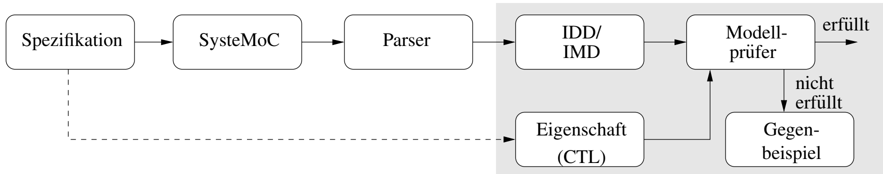


Abb. 8.8. Überblick der symbolischen CTL-Modellprüfung von SysteMoC-Modellen

den. Die Überprüfung solcher Formeln erfolgt mit Hilfe der bekannten Fixpunktalgorithmen aus Abschnitt 5.3.

Zentral für die Fixpunktberechnungen ist die Funktion COMPUTE_EX, welche auf der Bild- bzw. Urbildberechnung beruht. Bei der Bildberechnung mit Intervalldiagrammen wird ein IMD (charakteristische Funktion der Transitionsrelationen) mit einem IDD S (charakteristische Funktion einer Menge von Zuständen) ähnlich der APPLY-Operation bei ROBDDs verwoben. Das Ergebnis ist ein weiteres IDD S', das die Menge aller aus S in einem Schritt erreichbaren Zustände enthält. Die Bildberechnung kann durch eine rekursive Funktion MAP_FORWARD(v, w) erfolgen. Dabei ist v Knoten eines IDD und w Knoten eines IMD. Beide Intervalldiagramme müssen die gleiche Variablenordnung haben. In [414] gibt Strehl für die Funktion MAP_FORWARD(v, w) einen Algorithmus an.

Die Argumente sind dabei die Quellknoten der Intervalldiagramme, für die das Bild berechnet werden soll. Er steigt rekursiv bis zu den jeweiligen Terminalknoten in den Intervalldiagrammen ab und berechnet dann das resultierende IDD S' von unten nach oben. Das übergebene IDD S sollte reduziert sein, da dann der Algorithmus auch ein reduziertes IDD als Resultat liefert. Bei Aufruf mit Nichtterminalknoten werden drei Fälle unterschieden:

1. $\text{index}(v) < \text{index}(w)$: Ein IMD Knoten w mit gleichem Index wie der des IDD Knotens v fehlt auf diesem Pfad im IMD. Ein obsoleter IMD Knoten mit neutraler Abbildungsfunktion wird angenommen und die Intervalle der ausgehenden Kante von v bleiben unverändert. Der rekursive Aufruf wird mit allen Kanten von v, die nicht in dem Terminalknoten mit Wert F enden, fortgesetzt.
2. $\text{index}(v) > \text{index}(w)$: Auf dem Pfad fehlt ein IDD Knoten v mit gleichem Index wie der des IMD Knotens w. Wieder wird ein obsoleter Knoten angenommen, diesmal im IDD, und die Abbildungsfunktionen der Kanten von w werden auf den Definitionsbereich der Variablen mit $\text{index}(w)$ angewandt. Der rekursive Aufruf erfolgt nur, wenn das resultierende Intervall nicht leer ist.
3. $\text{index}(v) = \text{index}(w)$: Die Indices der Knoten v und w sind gleich und die Abbildungsfunktionen der Kanten des IMD Knotens w werden auf die Intervalle der Kanten des IDD Knotens v angewendet. Wie in Fall 2. erfolgt auch hier der rekursive Aufruf nur für nichtleere Resultatintervalle.

Während für die Bildberechnung die Funktion MAP_FORWARD verwendet wird, ist für die Berechnung des Urbildes eine analoge MAP_BACKWARD-Funktion

nötig. Allerdings reicht es aus, die MAP_FORWARD-Funktion zu benutzen, und immer die inverse Abbildungsfunktion zu berechnen. Dieses Vorgehen ist allerdings nicht für alle Arten von Abbildungsfunktionen möglich, da nicht immer eine inverse Abbildungsfunktion existiert. Es wurde mit den PADs aus Definition 8.1.11 aber bereits eine eingeschränkte Klasse von IMDs definiert, für deren Abbildungsfunktionen immer inverse Funktionen existieren.

Für PADs sind zwei Arten von Abbildungsfunktionen definiert, die Versetzungsfunktion f_+ und die Zuweisungsfunktion $f_{:=}$. Für diese gilt:

Definition 8.1.12 (Inverse Versetzungs- und Zuweisungsfunktion). *Die* inverse Versetzungsfunktion *ist definiert als*

$$f_+^{-1}(\Delta) := (\Delta - \Delta_A) \cap \Delta_P$$

und die inverse Zuweisungsfunktion *lautet*

$$f_{:=}^{-1}(\Delta) := \begin{cases} \Delta_p & \text{falls } \Delta \cap \Delta_A \neq \emptyset \\ [\,] & \text{sonst} \end{cases}$$

Damit ist für PADs die Berechnung des Bildes und des Urbildes mit einem Algorithmus möglich, der sich nur durch die Berechnung der Resultatintervalle unterscheidet.

Durch die abstrakte Darstellung von SysteMoC-Modellen lassen sich nur Eigenschaften bezüglich Füllständen oder Zuständen von endlichen Automaten in CTL formulieren. Außerdem kann der Modellprüfer Eigenschaften als nicht erfüllt erkennen, die das zugrundeliegende Modell tatsächlich erfüllt (falschnegatives Ergebnis). Um spezifischere (etwa datenabhängige) Eigenschaften zu prüfen, muss die symbolische Repräsentation detailliert werden. Abstraktionsverfeinerung könnte hier eingesetzt werden, um spezifischere Informationen zu prüfen.

8.1.2 Modellprüfung von SystemC-Modellen

Der oben beschriebene IDD-basierte Ansatz zur Modellprüfung von SysteMoC-Modellen prüft funktionale Eigenschaften direkt auf dem Verhaltensmodell, welches unabhängig von der Hardware/Software-Partitionierung ist. Allerdings liegen bereits oft SystemC-Module vor, die eine Hardware- oder eine Software-Implementierung eines oder mehrerer Aktoren darstellen. In [271] ist ein spezialisierter Ansatz zur Modellprüfung von SystemC-Modellen vorgestellt, der die Partitionierung in Hardware und Software zur Konstruktion des Modells zur Modellprüfung berücksichtigt. Dieser wird im Folgenden vorgestellt.

Attributierte temporale Strukturen

Grundlage für die Modellprüfung von SystemC-Modellen sind sog. *attributierte temporale Strukturen* (engl. *Labeled Temporal Structure, LTS*) oder auch *attributierte Kripke Strukturen* (engl. *labeled Kripke structures*) [89].

Definition 8.1.13 (Attributierte temporale Struktur, LTS). *Eine* attributierte temporale Struktur, LTS *ist ein 4-Tupel* $M = (S, R, L_S, L_R)$*, wobei*

- S *die Menge der Zustände,*
- $R \subseteq S \times S$ *die Zustandsübergangsrelation,*
- $L_S : S \rightarrow 2^{V_S}$ *die Zustandsmarkierungsfunktion und* V_S *die Menge aussagenlogischer Variablen (atomarer Formeln) sowie*
- $L_R : R \rightarrow \left(2^{V_R} \backslash \{\emptyset\}\right)$ *die Übergangsmarkierungsfunktion und* V_R *eine Menge an Ereignissen ist.*

In einer LTS sind sowohl Zustände als auch Zustandsübergänge markiert. Zustandsübergänge werden als $s \xrightarrow{E} s'$ geschrieben, wobei $(s, s') \in R$ und $E \subseteq L_R(s, s')$ ist. Besteht E lediglich aus einem Element, d. h. $E = \{e\}$, so wird anstelle von $s \xrightarrow{\{e\}} s'$ auch einfach $s \xrightarrow{e} s'$ geschrieben.

Ein *Pfad* $\tilde{s} = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ in einer LTS ist eine alternierende, unendliche Sequenz von Zuständen und Ereignissen, wobei

$$\forall i \geq 0 : s_i \in S \wedge e_i \in L_R(s_i, s_{i+1}) \wedge s_i \xrightarrow{e_i} s_{i+1}$$

Die attributierte temporale Struktur M besitzt eine Menge $S_0 \subseteq S$ an Anfangszuständen. Die Menge an Pfaden in M, die in einem Anfangszustand $s \in S_0$ beginnen und kein Präfix eines anderen Pfades darstellen, wird als *Sprache* $\mathcal{L}(M)$ von M bezeichnet.

Definition 8.1.14 (Abstraktion). *Seien* $M = (S, R, L_S, L_R)$ *und* $\hat{M} = (\hat{S}, \hat{R}, \hat{L}_S, \hat{L}_R)$ *zwei LTS.* $\hat{M}$ *ist eine* Abstraktion *von* M*, geschrieben als* $\hat{M} \sqsubseteq M$*, genau dann, wenn*

- $\hat{V}_S \subseteq V_S$ *gilt, also die Menge der atomaren Formeln in* $\hat{M}$ *eine Teilmenge der atomaren Formeln in* M *ist,*
- $\hat{V}_R = V_R$ *ist, also die Menge von Ereignissen in* $\hat{M}$ *und* M *gleich sind, und*
- *für jeden Pfad* $\tilde{s} = \langle s_0, e_0, \ldots \rangle$ *der Sprache* $\mathcal{L}(M)$ *ein abstrakter Pfad* $\tilde{s}' = \langle s'_0, e'_0, \ldots \rangle$ *in der Sprache* $\mathcal{L}(\hat{M})$ *existiert, d. h.*

$$\forall \tilde{s} \in \mathcal{L}(M) : \exists \tilde{s}' \in \mathcal{L}(\hat{M}) : \forall i \geq 0 : (e_i = e'_i) \wedge \left(\hat{L}_S(s'_i) = L_S(s_i) \cap \hat{V}_S\right)$$

Mit anderen Worten: $\hat{M}$ ist eine Abstraktion von M, wenn die „aussagenlogische“ Sprache von $\hat{M}$ die „aussagenlogische“ Sprache von M enthält, indem M auf die atomaren Formeln $\hat{V}_S$ aus $\hat{M}$ beschränkt wird.

Man beachte, dass die hier vorgestellte Abstraktion auch zur Definition einer *Äquivalenzrelation* $\equiv$ von zwei LTS M und M' verwendet werden kann:

$$M \equiv M' \Leftrightarrow M \sqsubseteq M' \wedge M' \sqsubseteq M$$

Für zwei attributierte temporale Strukturen kann eine *parallele Komposition* durchgeführt werden.

Definition 8.1.15 (Parallele Komposition). *Seien $M_1 = (S_1, R_1, L_{S,1}, L_{R,1})$ und $M_2 = (S_2, R_2, L_{S,2}, L_{R,2})$ zwei attributierte temporale Strukturen, mit $S = S_1 = S_2$ und $V_S = V_{S,1} = V_{S,2}$ und $L_S = L_{S,1} = L_{S,2}$, d. h. beide LTS haben den selben Zustandsraum. Im Folgenden bedeute $s \xrightarrow{E}_i s'$, dass M_i einen Zustandsübergang von s nach s' durchführen kann. Die* parallele Komposition *von M_1 und M_2 ist gegeben durch:*

$$M_1 \parallel M_2 := (S, R_{\parallel}, L_S, L_{R_{\parallel}})$$

wobei $R_{\parallel}$ und $L_{R_{\parallel}}$ so definiert sind, dass $s \xrightarrow{E} s'$ gilt, genau dann, wenn $E \neq \emptyset$ ist und eine der folgenden Bedingungen erfüllt ist:

- *$E \subseteq V_{S,1} \setminus V_{S,2}$ und $s \xrightarrow{E}_1 s'$,*
- *$E \subseteq V_{S,2} \setminus V_{S,1}$ und $s \xrightarrow{E}_2 s'$ oder*
- *$E \subseteq V_{S,1} \cap V_{S,2}$ und $s \xrightarrow{E}_1 s'$ und $s \xrightarrow{E}_2 s'$.*

Die Anfangszustände $S_{0_{\parallel}}$ der parallelen Komposition $M_1 \parallel M_2$ ergeben sich als Schnittmenge der Anfangszustände von M_1 und M_2, d. h. $S_{0_{\parallel}} = S_{0,1} \cap S_{0,2}$.

LTS-Repräsentation von SystemC-Modellen

SystemC-Modelle (siehe Abschnitt 2.3.1) bestehen aus einem oder mehreren Prozessen. SystemC-Prozesse können entweder Threads oder Methoden sein. Da Methoden aber eine Spezialform von Threads sind, ist im Folgenden eine Betrachtung von Threads ausreichend.

Syntaktisch sind SystemC-Prozesse C++-Methoden einer SystemC-Modulklasse. Zu deren Aktivierung verfügen SystemC-Prozesse über eine Liste an Ereignissen, die sog. *Sensitivitätsliste*. Sobald ein Ereignis aus der Liste eintritt, wird der Prozess aktiviert und solange ausgeführt, bis er terminiert oder er aufgrund einer `wait`-Anweisung blockiert wird. Ereignisse können dabei explizit durch die `notify`-Anweisung oder implizit durch Änderung eines Signals erzeugt werden.

Beispiel 8.1.6. Das folgende SystemC-Programm aus [271] verdeutlicht diese Konzepte :

```
 1    class SUV1: public sc_module {
 2      public:
 3        sc_in<bool>  data_in;
 4        sc_in<bool>  clock;
 5        sc_out<bool> data_out;
 6
 7      private:
 8        void thread1();
 9        void thread2();
10
11      public:
```

```
12        SC_HAS_PROCESS(SUV1);
13
14        SUV1(sc_module_name name) : sc_module(name) {
15          SC_THREAD(thread1);
16          sensitive << data_in;
17
18          SC_THREAD(thread2);
19          sensitive << clock;
20        }
21    };
```

Das Modul SUV1 besitzt zwei Eingangsports, einen Ausgangsport und zwei Threads. Alle Ports haben den Typ `bool`. Der Thread `thread1` ist sensitiv auf das Signal `data_in` und repräsentiert eine kombinatorische Schaltung. Der Thread `thread2` ist sensitiv auf das Taktsignal `clock` und modelliert eine getaktet Schaltung.

In SystemC werden bei Threads drei Prozesszustände unterschieden: *bereit*, *laufend* und *blockiert*. Ein laufender Thread ist in der Lage, Ereignisse zu erzeugen. Die Erzeugung eines Ereignisses versetzt später einen blockierten Thread, der auf dieses Ereignis sensitiv ist, in den Zustand bereit. Ein Thread, der auf ein Ereignis sensitiv und bereits im Zustand *laufend* ist, ignoriert das Ereignis. Ein einzelnes Ereignis kann viele Threads aktivieren. Bei Erreichen einer `wait`-Anweisung, wird ein laufender Thread blockiert. In diesem Fall wählt der SystemC-Simulator einen der bereiten Threads zur Ausführung aus. Die Reihenfolge in der bereite Threads zur Ausführung gebracht werden, ist nicht spezifiziert und somit nichtdeterministisch. Man beachte, dass Thread nicht unterbrochen werden können, sondern nur von sich aus bei Erreichen einer `wait`-Anweisung blockieren. Die Erzeugung von Ereignissen kann mit der Anweisung `notify_delayed` verzögert werden. In diesem Fall wird das Ereignis erst erzeugt, sobald alle Threads im Zustand *blockiert* sind.

Abbildung von SystemC-Modellen auf LTSs

Jeder Thread im SystemC-Modell wird als eine attributierte temporale Struktur M_i repräsentiert. Existieren n Threads, so kann das Verhalten des SystemC-Modells als parallele Komposition von n LTSs dargestellt werden, d. h. $M := M_1 \parallel M_2 \parallel \cdots \parallel M_n$. Der globale Zustandsraum des SystemC-Modells $S = S_1 = \cdots = S_n$ lässt sich über die Programmzähler pc_i der Threads, die Belegung der Variablen pv_i der Threads und die aktuellen Prozesszustände der Threads $ps_i \in \{bereit, laufend, blockiert\}$ definieren. Mit anderen Worten: Ein globaler Zustand $s = (PV, PC, PS) \in S$ ist ein Tripel (PV, PC, PS) bestehend aus dem Vektor $PV = (pv_1, \ldots, pv_n)^\top$ der Variablenbelegungen, dem Vektor $PC = (pc_1, \ldots, pc_n)^\top$ der Programmzähler und dem Vektor $PS = (ps_1, \ldots, ps_n)^\top$ der Prozesszustände. Für einen gegeben globalen Zustand $s \in S$ und einen Thread i kann die Projektion der Werte pv_i, pc_i und ps_i über den .-Operator erfolgen, d. h. $s.pv_i$, $s.pc_i$ und $s.ps_i$.

Die Initialisierung in SystemC erfolgt, indem alle Prozesse nacheinander gestartet werden. Dabei wird jedoch keine Reihenfolge der Aktivierung vorgegeben. Die Menge der Anfangszustände $S_0 \subseteq S$ von M lässt sich daher wie folgt definieren:

$$S_0 := \{s \in S \mid \forall 1 \leq i \leq n : (s.pc_i = 0) \wedge (s.ps_i \neq blockiert) \wedge (s.ps_i = laufend \Rightarrow \forall j \neq i : s.ps_j \neq laufend)\} \quad (8.2)$$

Somit sind für alle Prozesse die Programmzähler auf den Anfang gesetzt, sowie alle Prozesse bis auf einen im Prozesszustand *bereit*. Der verbleibende Prozess befindet sich im Prozesszustand *laufend*.

Zur Definition der Zustandsübergangsrelation müssen zunächst die Mengen $V_{R,i}$ der Ereignisse der insgesamt n Thread, auf die ein Thread i sensitiv ist, definiert werden. Zunächst werden n globale Ereignis $\omega_1, \ldots, \omega_n$ definiert, welche die Auswahl des Threads i durch den SystemC-Simulator modellieren. Diese sind in allen LTS M_i sichtbar, d. h. $\Omega := \{\omega_1, \ldots, \omega_n\} \subseteq V_{R,i}$. Neben diesen globalen Ereignissen kann es in jedem Thread auch bis zu einem lokalen Ereignis geben. Deshalb werden n lokale Ereignisse $\tau_1, \ldots, \tau_n$ definiert, wobei jedes τ_i genau einem Thread i und somit der LTS M_i zugeordnet ist, d. h.

$$\forall 1 \leq i \leq n : \tau_i \in V_{R,i} \Rightarrow \forall j \neq i : \tau_i \notin V_{R,j}$$

Falls der Thread aus dem Kontext klar ist, wird im Folgenden $s \xrightarrow{\tau}_i s'$ anstelle von $s \xrightarrow{\tau_i}_i s'$ geschrieben.

Die Zustandsübergänge der einzelnen LTS werden durch eine Fallunterscheidung in laufende und nichtlaufende Threads eingeteilt. Sei s der globale Zustand und i derjenige Thread, der sich im Prozesszustand *laufend* befindet, d. h. $s.ps_i = laufend$. Weiterhin sei $\mathcal{P}(s.pc_i)$ die momentane Instruktion, auf die der Programmzähler in Thread i zeigt und sei $\mathcal{A}(\mathcal{P}(s.pc_i))$ diejenige Menge an Threads, die durch die Instruktion $\mathcal{P}(s.pc_i)$ aktiviert wurde.

Zunächst werden die Zustandsübergänge eines Threads i, der sich im Prozesszustand *laufend* befindet, definiert:

- Falls die aktuelle Instruktion eine `notify`-Anweisung ist, so vollführt der Thread i eine τ-Transition und ändert den globalen Zustand s entsprechend:

$$\mathcal{P}(s.pc_i) == \texttt{notify();} \Rightarrow s \xrightarrow{\tau}_i s'$$

mit

$$s'.PV := s.PV$$
$$s'.pc_j := \begin{cases} s.pc_j + 1 & \text{falls} \quad j = i \\ s.pc_j & \text{sonst} \end{cases}$$
$$s'.ps_j := \begin{cases} bereit & \text{falls} \quad j \in \mathcal{A}(\mathcal{P}(s.pc_i)) \\ s.ps_j & \text{sonst} \end{cases}$$

- Falls die aktuelle Instruktion eine Zuweisung des Ausdrucks `a` mit Wert a an die Variable `x` ist, so vollführt der Thread i eine τ-Transition und ändert den globalen Zustand s entsprechend:

$$\mathcal{P}(s.pc_i) == \texttt{x} = \texttt{a;} \Rightarrow s \xrightarrow{\tau}_i s'$$

mit

$$
\begin{aligned}
s'.x &:= a \\
s'.y &:= s.y \text{ für } y \in s.PV \wedge y \neq x \\
s'.pc_j &:= \begin{cases} s.pc_j + 1 & \text{falls } \; j = i \\ s.pc_j & \text{sonst} \end{cases} \\
s'.PS &:= s.PS
\end{aligned}
$$

- Falls die aktuelle Instruktion eine konditionale Sprunganweisung mit Sprungbedingung g mit Wahrheitswert g und Sprungziel `t` mit Wert t ist, so vollführt der Thread i eine τ-Transition und ändert den globalen Zustand s entsprechend:

$$\mathcal{P}(s.pc_i) == \texttt{if}(\texttt{g})\ \texttt{goto t}; \Rightarrow s \xrightarrow{\tau}_i s'$$

mit

$$
\begin{aligned}
s'.PV &:= s.PV \\
s'.pc_j &:= \begin{cases} t & \text{falls } \; (j = i) \wedge g \\ s.pc_j + 1 & \text{falls } \; (j = i) \wedge \neg g \\ s.pc_j & \text{sonst} \end{cases} \\
s'.PS &:= s.PS
\end{aligned}
$$

- Falls die aktuelle Instruktion eine `wait`-Anweisung ist, so vollführt der Thread i eine ω-Transition. Dabei wird ein Ereignis $\omega \in \Omega \backslash \{\omega_i\}$ zur Synchronisation mit anderen Thread generiert. Dieser Übergang wird den Thread i in den Prozesszustand *blockiert* versetzen. Entsprechend ändert sich der globale Zustand s:

$$\mathcal{P}(s.pc_i) == \texttt{wait}(); \Rightarrow s \xrightarrow{\Omega \backslash \{\omega_i\}}_i s' \tag{8.3}$$

mit

$$
\begin{aligned}
s'.PV &:= s.PV \\
s'.PC &:= s.PC \\
s'.ps_j &:= \begin{cases} \mathit{blockiert} & \text{falls } \; j = i \\ s.ps_j & \text{sonst} \end{cases}
\end{aligned}
$$

Nichtlaufende Prozesse können lediglich auf ein globales ω-Ereignis reagieren. Dieses dient, wie oben erläutert, zur Synchronisation nach einer `wait`-Anweisung. Während Gleichung (8.3) die nichtdeterministische Auswahl eines Ereignisses $\omega_j \in \Omega \backslash \{\omega_i\}$ durch den SystemC-Simulator modelliert, muss dennoch darauf geachtet werden, dass nur Threads im Prozesszustand *bereit* aktiviert werden dürfen:

$$(s.ps_i = \mathit{bereit}) \Rightarrow s \xrightarrow{\omega_i}_i s'$$

mit

$$s'.PV := s.PV$$
$$s'.PC := s.PC$$
$$s'.ps_j := \begin{cases} laufend & \text{falls} \quad j = i \\ s.ps_j & \text{sonst} \end{cases}$$

Die Prozesszustandsübergänge für eine LTS eines einzelnen Threads i sind in Abb. 8.9 zu sehen.

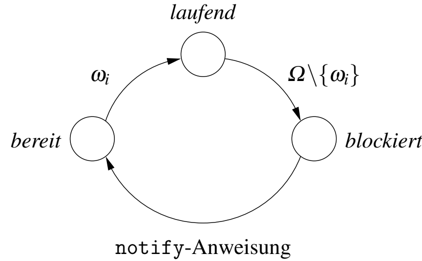


Abb. 8.9. Prozesszustandsübergänge für Thread i [271]

Modellprüfung

Das in [271] beschriebene Verfahren basiert auf LTL-Modellprüfung. Hierzu ist es aber notwendig, die geforderten Eigenschaften so zu formulieren, dass sie auf den attributierten temporalen Strukturen interpretiert werden können. Hierzu kann beispielsweise *SE-LTL* (engl. *State/Event-LTL*) verwendet werden [259, 89].

Definition 8.1.16 (Syntax von SE-LTL). *V_S sei die Menge der aussagenlogischen Variablen (atomaren Formeln) einer Zustandsmarkierungsfunktion. V_R sei die Menge der aussagenlogischen Variablen (atomaren Formeln) einer Übergangsmarkierungsfunktion. Eine* SE-LTL-Formel *wird rekursiv wie folgt definiert:*

- *Atomare Formeln $\varphi \in (V_S \cup V_R)$ sind SE-LTL-Formeln.*
- *Seien φ und ψ SE-LTL-Formeln, so sind auch $\neg\varphi$ und $\varphi \vee \psi$ SE-LTL-Formeln.*
- *Seien φ und ψ SE-LTL-Formeln, so sind auch* $\mathrm{X}\ \varphi$ *und* $\varphi\ \mathrm{U}\ \psi$ *LTL-Formeln.*

Im Folgenden ist $\tilde{s} = \langle s_0, e_0, s_1, e_1, \ldots, s_i, e_i, \ldots \rangle$ ein Pfad. Das Suffix $\tilde{s}^i$ ist definiert zu $\tilde{s}^i := \langle s_i, e_i, \ldots \rangle$.

Definition 8.1.17 (Semantik von SE-LTL-Formeln). *Sei φ eine SE-LTL-Formel, so bedeutet $M, \tilde{s} \models \varphi$, dass φ entlang des Pfades $\tilde{s} = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ der LTS M mit Anfangszustand s_0 gilt. Seien φ und ψ zwei SE-LTL-Formeln, dann kann $\models$ wie folgt definiert werden:*

$$
\begin{array}{ll}
M,\tilde{s} \models \top & \Leftrightarrow \textit{gilt immer} \\
M,\tilde{s} \models p_s & \Leftrightarrow p_s \in L_S(s_0),\ \textit{falls}\ p_s \in V_S \\
M,\tilde{s} \models p_e & \Leftrightarrow p_e \in L_R(e_0),\ \textit{falls}\ p_e \in V_R \\
M,\tilde{s} \models \neg\varphi & \Leftrightarrow M,\tilde{s} \not\models \varphi \\
M,\tilde{s} \models \varphi \vee \psi & \Leftrightarrow M,\tilde{s} \models \varphi\ \textit{oder}\ M,\tilde{s} \models \psi \\
M,\tilde{s} \models \mathrm{X}\,\varphi & \Leftrightarrow M,\tilde{s}^1 \models \varphi \\
M,\tilde{s} \models \varphi\,\mathrm{U}\,\psi & \Leftrightarrow \exists k \geq 0 : M,\tilde{s}^k \models \psi \wedge \forall 0 \leq j \leq k : M,\tilde{s}^j \models \varphi
\end{array}
$$

Weiterhin sei $M \models \varphi$ definiert zu:

$$M \models \varphi \Leftrightarrow \forall \tilde{s} \in \mathcal{L}(M) : \tilde{s} \models \varphi$$

Beispiel 8.1.7. Gegeben ist die LTS M aus Abb. 8.10. Diese hat zwei Zustände s_0 und s_1, wobei s_0 der Anfangszustand ist. Die Menge atomarer Zustandsaussagen ist $V_S = \{p,q,r\}$. Die Menge an Ereignissen ist $V_R = \{a,b,c,d\}$.

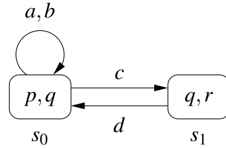


Abb. 8.10. Attributierte temporale Struktur [89]

Für diese LTS gilt: $M \models \mathrm{G}\,(c \Rightarrow \mathrm{F}\,r)$ und $M \not\models \mathrm{G}\,(b \Rightarrow \mathrm{F}\,r)$.

SE-LTL-Modellprüfung

Eine Möglichkeit die SE-LTL-Modellprüfung mit Standard-LTL-Modellprüfung entsprechend zu Abschnitt 5.2.2 durchzuführen ist, Ereignisse als Änderungen in zusätzlichen Zustandsvariablen zu codieren. Dies führt im Allgemeinen zu einer signifikanten Vergrößerung des Zustandsraums und ist nicht praktikabel. Aus diesem Grund ist in [89] ein Verfahren vorgestellt, was ohne die Veränderung der Größe der LTS auskommt. Hierzu wird die LTS als Büchi-Automat (siehe Definition 5.2.1 auf Seite 186) formuliert.

Definition 8.1.18 (Temporale Struktur als Büchi-Automat). *Sei $M = (S,R,L_S,L_R)$ eine LTS mit Anfangszuständen $S_0 \subseteq S$ und atomaren Zustandsaussagen V_S sowie Ereignismenge V_R. Die Zustandsmarkierungsfunktion $L_S : S \rightarrow 2^{V_S}$ gibt an, für welchen Zustand welche atomaren Aussagen $v \in V_S$ gelten. Dies ist gleichbedeutend mit: „Die Aussage $\hat{L}_S(s)$ gilt in s“, wobei*

$$\hat{L}_S(s) := \bigwedge_{v \in L_S(s)} v \wedge \bigwedge_{v \in (V_S \setminus L_S(s))} \neg v$$

Die Übergangsmarkierungsfunktion $L_R : R \rightarrow 2^{V_R}$ gibt an, für welchen Zustandsübergang welches Ereignis $v \in V_R$ gilt. Dies ist gleichbedeutend mit: „Das Ereignis $\hat{L}_R(s,s')$ gilt beim Zustandsübergang (s,s')“, wobei

$$\hat{L}_R(s,s') := \bigwedge_{v \in L_R(s,s')} v \wedge \bigwedge_{v \in (V_R \setminus L_R(s,s'))} \neg v$$

Daneben kann jede SE-LTL-Formel φ über V_S und V_R auch als LTL-Formel φ_{LTL} über $V_S \cup V_R$ interpretiert werden. Dabei sind φ und φ_{LTL} syntaktisch identisch, unterscheiden sich aber semantisch.

Da jede LTS mit Definition 8.1.18 als Büchi-Automat interpretiert werden kann, kann, wie im Fall der LTL-Modellprüfung, der Produktautomat aus einer LTS und einem Büchi-Automaten gebildet werden. Sei $B = (S_B, R_B, L_B, A_B)$ ein Büchi-Automat mit Anfangszuständen $S_{B,0}$ und $M = (S, R, L_S, L_R)$ eine attributierte temporale Struktur mit Anfangszuständen S_0. Der Büchi-Automat verwendet die atomaren Aussagen $V_B = V_S \cup V_R$. Der Produktautomat $M_p := M \otimes B = (S_p, R_p, L_p, A_p)$ ist definiert durch:

- $S_p = \{(s, s_B) \in S \times S_B \mid \hat{L}_S(s) \Rightarrow \exists V_R : L_B(s_B)\}$ (dies beschreibt die Formel $L_B(s_B)$, in der alle Variablen $v \in V_R$ existentiell quantifiziert wurden),
- $((s, s_B), (s', s'_B)) \in R_p$, genau dann, wenn $\exists v \in V_R : s \xrightarrow{v} s' \wedge (s_B, s'_B) \in R_B \wedge (\hat{L}_S(s) \wedge \hat{L}_R(s,s')) \Rightarrow L_B(s_B)$ und
- $(s, s_B) \in A_p$, genau dann, wenn $s_B \in A_B$.

Die Anfangszustände ergebenen sich aus den Paaren der Anfangszustände beider Automaten.

Da SE-LTL-Formeln syntaktisch identisch mit LTL-Formeln sind, lassen sich SE-LTL-Formeln als Büchi-Automaten modellieren. Somit kann nun die SE-LTL-Modellprüfung, wie die LTL-Modellprüfung, als Test auf eine leere Sprache des Produktautomaten umformuliert werden.

Theorem 8.1.1. *Für eine LTS M und eine SE-LTL-Formel φ gilt:*

$$M \models \varphi \Leftrightarrow \mathcal{L}(M \otimes B_{\neg \varphi_{LTL}}) = \emptyset$$

Die Sprache des Produktautomaten ist definiert, wie in Abschnitt 5.2.2 beschrieben.

Abstraktion

Neben der oben beschriebenen Abbildung von SystemC-Modellen auf parallele Kompositionen von LTS lassen sich weitere Abstraktionen durchführen, welche die Verifikationszeit verkürzen können.

Hardware/Software-Partitionierung

Durch Hardware/Software-Partitionierung kann die Modellkomplexität deutlich reduziert werden. Hierzu werden die SystemC-Prozesse in *kombinatorische*, *getaktete* und *uneingeschränkte Threads* klassifiziert und bei der Generierung der LTS gesondert behandelt.

Kombinatorische Threads besitzen die Eigenschaft, dass diese

- Sensitiv auf alle Eingänge sind,
- nicht mit dem Taktsignal verbunden sind,
- keine `wait`-Anweisung enthalten und
- keine Schleifen ohne konstante Grenzen enthalten.

Kombinatorische Threads werden in eine Formel f übersetzt und aus dem Modell entfernt. Jedes Mal, wenn Ausgangsvariablen des kombinatorischen Threads gelesen werden, werden diese mit der Formel f beschränkt.

Getaktete Threads besitzen die Eigenschaft, dass diese

- Sensitiv auf eine positive oder negative Taktflanke sind,
- keine `wait`-Anweisung mit Argumenten enthalten und
- jede Schleife ohne konstante Grenze mindestens eine `wait`-Anweisung enthält.

Getaktete Threads können in endliche Automaten übersetzt werden. Hierfür wird für den Start, das Ende und jede `wait`-Anweisung ein Zustand generiert. Die Zustandsübergänge werden entsprechend aus den Pfaden im Kontroll-Datenflussgraphen des Prozesses erzeugt. Jede τ-Transition der LTS, die diesen Prozess repräsentiert, entspricht einem Zustandsübergang in dem endlichen Automaten.

Während kombinatorische und getaktete Threads als ungetaktete und getaktete Hardware-Komponenten angesehen werden können, modellieren *uneingeschränkte Threads* Software-Prozesse. Uneingeschränkte Threads unterliegen erwartungsgemäß keinen Einschränkungen und lassen somit bei der Abbildung auf LTS auch keine Optimierungen zu.

SAT-basierte Prädikatenabstraktion

SystemC unterstützt unterschiedliche Datentypen. Um einen Zugang zum Hardware-Entwurf zu erleichtern, werden auch Bitvektoren unterstützt. Die Operationen, die auf Bitvektoren arbeiten, vergrößern allerdings den Zustandsraum. Eine mögliche Abstraktion kann in diesem Fall durch den Einsatz von arithmetischen Operationen erreicht werden.

Erfolgt die Prädikatenabstraktion durch SAT-Solver (siehe Abschnitt 7.3.3), so kann die Abstraktion nach folgender Formeln realisiert werden:

$$\hat{s} \xrightarrow{A} \hat{s}' \Leftrightarrow \exists s, s' \in S : s \xrightarrow{A} s' \wedge \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}'$$

Dabei ist α die Funktion zur Zustandsabstraktion und S der globale Zustand der parallelen Komposition der LTS. Die Formel sagt aus, dass ein abstrakter Zustandsübergang $(\hat{s}, \hat{s}')$ unter den Ereignissen $a \in A$ existiert, sofern im konkreten Modell ein Zustandsübergang (s, s') unter den selben Ereignissen existiert und der Start- und Endzustand Abstraktionen der konkreten Start- und Endzustände sind.

Thread-modulare Abstraktion

Die Repräsentation von SystemC-Modellen durch LTS erfolgt, indem für jeden SystemC-Prozess eine LTS generiert wird und diese anschließend parallel komponiert werden. Es kann nun eine Abstraktion des gesamten Modells dadurch erreicht

werden, dass zunächst die einzelnen LTS abstrahiert und anschließend parallel komponiert werden.

Formal bedeutet dies:

Theorem 8.1.2. *Sei $M_{||}$ die parallele Komposition von n LTS, d. h. $M_{||} = M_1 \parallel \cdots \parallel M_n$. Sei weiterhin $\hat{M}_i$ die Abstraktion von M_i und $\hat{M}_{||}$ die Abstraktion von $M_{||}$. Dann gilt:*

$$\hat{M}_{||} \equiv \hat{M}_1 \parallel \cdots \parallel \hat{M}_n$$

Mit anderen Worten: Die Abstraktion der parallelen Komposition der konkreten LTS M_1 bis M_n ist äquivalent zur parallelen Komposition der abstrakten LTS $\hat{M}_1$ bis $\hat{M}_n$.

Dies bedeutet aber auch, dass die Projektion $\tilde{s}|M_i$ eines Pfades $\tilde{s} = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ in der parallelen Komposition $M_{||}$ der LTS M_1 bis M_n auf eine einzelne LTS M_i eine Abstraktion von $M_{||}$ ist, d. h.

$$(\tilde{s}|M_i) \sqsubseteq M_{||}$$

Dabei gilt die Projektion $\tilde{s}|M_i$ einer Teilsequenz $\hat{\tilde{s}}$ von $\tilde{s}$, die man durch Entfernen der Paare (a_j, s_{j+1}) erhält, für alle $a_j \notin V_{r,i}$.

Die beschriebenen Abstraktionen können zu *falschnegativen Ergebnissen* führen, weshalb für die Verifikation von SystemC-Modellen eine *Abstraktionsverfeinerung* eingesetzt wird [271]. Die Abstraktionsverfeinerung wird dabei durch die Gegenbeispiele gesteuert, d. h., ergibt die Modellprüfung einer funktionalen Eigenschaft ein Negativergebnis, wird ein Gegenbeispiel generiert. Lässt sich dieses Gegenbeispiel im ursprünglichen SystemC-Modell jedoch nicht simulieren, so muss das Modell geeignet verfeinert werden.

8.1.3 Formale Modellprüfung von Transaktionsebenenmodellen

Transaktionsebenenmodelle (engl. *Transaction Level Model, TLM*) werden zunehmend als Strukturmodell auf Systemebene eingesetzt. Dabei handelt es sich um eine Netzliste von Prozessoren, Hardware-Beschleunigern, Bussen und Speichern. Die Umsetzung erfolgt dabei häufig in SystemC [352].

Transaktionen

In TLMs werden Transaktionen als Abstraktion der Kommunikation zwischen nebenläufigen Prozessen verwendet. Transaktionen können entweder atomar oder unterbrechbar sein. Atomare Transaktionen werden in SystemC-TLM als *blockierend*, unterbrechbare als *nichtblockierend* bezeichnet. Basierend auf blockierenden und nichtblockierenden Transaktionen definiert SystemC-TLM unterschiedliche Codierungsrichtlinien, die als *schwach zeitbehaftet* (engl. *Loosely Timed, LT*), *approximiert zeitbehaftet* (engl. *Approximately Timed, AT*) und *zyklenakkurat* (engl. *Cycle Accurate, CA*) bezeichnet werden.

In einem LT-TLM werden ausschließlich blockierende Transaktionen (die SystemC-Methode `b_transport`) verwendet, die durch einen Start- und einen Endzeitpunkt charakterisiert sind. Diese beiden Zeitpunkte können aber durchaus identisch

sein, d. h. die Transaktion hat keine Zeit benötigt. Ein AT-TLM kann mehrere Transaktionsphasen enthalten und erlaubt somit eine detailliertere Modellierung, etwa von Ressourcenbeschränkungen. Dies wird durch nichtblockierende Transaktionen (die SystemC-Methode `nb_transport`) erreicht, für die SystemC-TLM vier charakteristische Zeitpunkte definiert: *begin_request*, *end_request*, *begin_response* und *end_response*. Die Definition eigener Zeitpunkte für Protokolle mit anderen Phasen ist aber auch möglich. CA-TLMs sind vergleichbar mit Hardware-Beschreibungen auf der Registertransferebene und verfeinern AT-TLM mittels eines Taktsignals. AT- und CA-TLMs stellen also Verfeinerungen von LT-TLMs dar und werden aufgrund der höheren Genauigkeit vor allem bei der Verifikation des Zeitverhaltens eingesetzt. Für die Verifikation funktionaler Eigenschaften werden im Folgenden lediglich LT-TLMs betrachtet, die mit blockierenden Transaktionen auskommen, was die Notation vereinfacht. Außerdem soll die Kommunikation zeitfrei erfolgen.

Schwach zeitbehaftete Transaktionsebenenmodelle bestehen aus nebenläufigen Prozessen, die über blockierende Transaktionen miteinander kommunizieren. Die Prozesse können dabei an den Transaktionen blockieren, z. B. aufgrund nicht vorhandener Daten. Zentral für die Verifikation von TLMs ist die korrekte Zusammenarbeit (Kommunikation) dieser Prozesse. Um funktionale Eigenschaften eines TLM als Anforderungen zu formulieren, müssen also die zeitlichen Zusammenhänge der Transaktionen spezifiziert werden. Da es in LT-TLMs im Allgemeinen kein Taktsignal gibt, müssen diese zeitlichen Zusammenhänge relativ zueinander spezifiziert werden.

Beispiel 8.1.8. Abbildung 8.11 zeigt mögliche relative zeitliche Zusammenhänge zwischen drei blockierenden Transaktionen t_1, t_2 und t_3. *start* bezeichnet dabei den Startzeitpunkt einer Transaktion, *end* den Endzeitpunkt. Man sieht, dass Transaktion t_3 nach Transaktion t_2 und Transaktion t_1 startet, da $t_3.start > t_2.start > t_1.start$. Obwohl Transaktion t_2 nach Transaktion t_1 startet, wird diese früher beendet. Für die Formulierung von funktionalen Eigenschaften ist es manchmal notwendig, den Abstand zweier Ereignisse zu bestimmen. Ohne Taktsignal kann dies nur relativ zueinander erfolgen. Beispielsweise ist der Abstand zwischen den Ereignissen $t_1.start$ und $t_1.end$ vier, da drei Ereignisse zwischen diesen liegen.

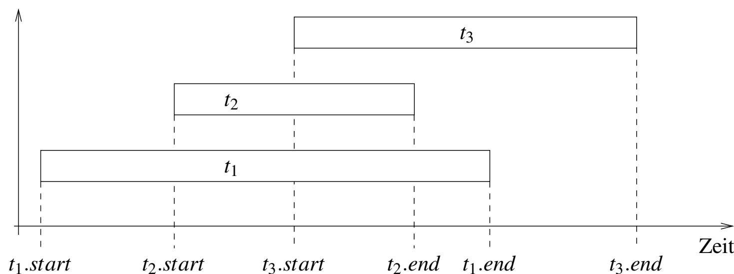


Abb. 8.11. Drei blockierende Transaktionen in einem LT-TLM [143]

Möglichen Relationen zwischen zwei Transaktionen in einem LT-TLM sind in Abb. 8.12 zu sehen. Diese können lediglich anhand der Position eines Start- oder Endeereignisses einer Transaktion bezüglich anderer Ereignisse definiert werden. Abbildung 8.12a) zeigt, dass Transaktion t_1 und Transaktion t_2 gleichzeitig stattfinden, da $t_1.start = t_2.start$ und $t_1.end = t_2.end$. In Abb. 8.12b) ist der Fall zu sehen, dass die Transaktion t_2 nach t_1 startet, beide aber gleichzeitig beendet werden. Abbildung 8.12c) zeigt den Fall, dass die Transaktionen t_1 und t_2 gleichzeitig starten, t_1 aber früher beendet wird, d. h. $t_1.start = t_2.start \wedge t_1.end < t_2.end$. Schließlich zeigt Abb. 8.12d), dass Transaktion t_1 und t_2 direkt aufeinander folgen, d. h. $t_1.end = t_2.start$.

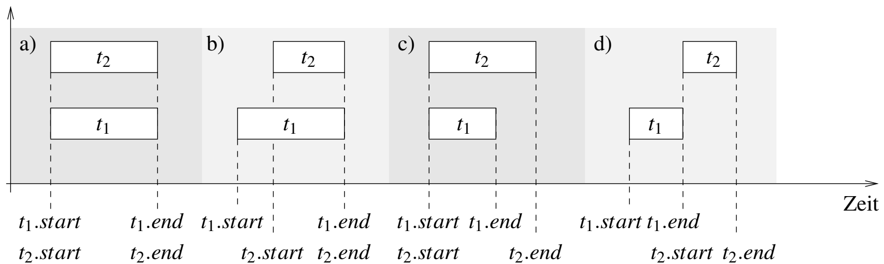


Abb. 8.12. Beziehungen zwischen Transaktionen in LT-TLMs [143]

Formalisierung von LT-TLMs

Die Formalisierung eines SystemC-TLMs erfolgt ähnlich dem in Abschnitt 8.1.2 beschriebenen Ansatz für SystemC-Modelle. Da die Synchronisation jetzt nicht mehr ausschließlich über Ereignisse, sondern über blockierende Transaktionen erfolgt, wird die Kommunikation ebenfalls formal modelliert. Um den Zustandsraum eines TLM effizient zu modellieren, wird in [345, 347] ein Modell kommunizierender endlicher Automaten vorgestellt.

Definition 8.1.19 (Transaktionsebenenmodell). *Ein* Transaktionsebenenmodell *ist ein Tripel* (TM_I, TM_T, T)*, wobei*

- TM_I *die Menge an sog.* Initiatormodulen*,*
- TM_T *die Menge an sog.* Targetmodulen*, und*
- $T \subseteq M_I \times M_T$ *die Menge an* Transaktionen

SystemC-Module werden in einem TLM als Initiator- oder Targetmodul klassifiziert. Dabei kann ein Modul tm auch beides sein, d. h. $tm \in TM_I \wedge tm \in TM_T$. Ein Initiatormodul enthält einen SystemC-Prozess, wobei die Diskussion auf SystemC-Threads reduziert werden kann. Targetmodule implementieren die Transaktionen

`b_transport`, weshalb Transaktionen mit Targetmodulen assoziiert werden. Andererseits können Transaktionen lediglich von Initiatormodulen aufgerufen werden. Welches Initiatormodul welche Transaktion auf welchem Targetmodul aufrufen kann, ist als Paar in der Transaktion gespeichert. Targetmodule, die keine Initiatormodule sind, enthalten keinen SystemC-Thread.

Beispiel 8.1.9. Ein Beispiel für ein SystemC-TLM ist in Abb. 8.13 dargestellt. Es besteht aus sechs Modulen (zwei CPUs, zwei Speicher, ein Bus und ein Arbiter). Die Menge der Initiatormodule TM_I besteht aus den beiden CPUs und dem Bus. Die Menge der Targetmodule TM_T besteht aus den beiden Speichern, dem Bus und dem Arbiter. Da die beiden Speichermodule und der Arbiter keine Initiatormodule sind, besitzen sie auch keinen SystemC-Thread. Module mit SystemC-Thread sind eingefärbt.

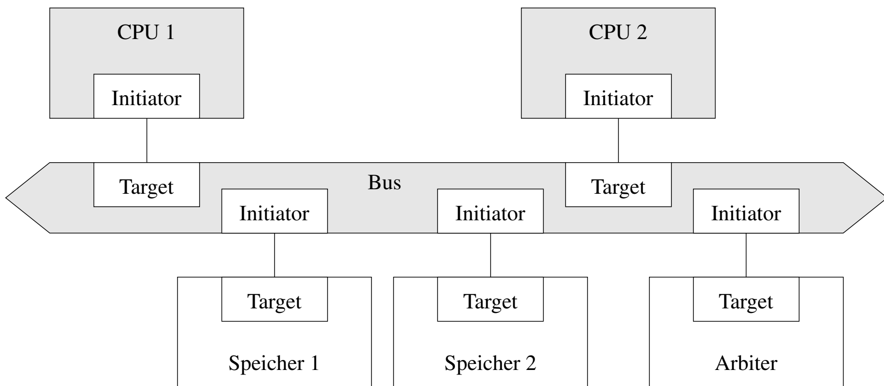


Abb. 8.13. SystemC-TLM [346]

Zur Konstruktion des formalen Modells wird jedes Modul tm_i einzeln betrachtet und in einem kommunizierenden endlichen Automaten übersetzt. Analog zu dem Ansatz in Abschnitt 8.1.2 muss der Zustandsraum modelliert werden, indem die Variablenbelegungen, die Programmzähler der Prozesse und die Prozesszustände erfasst werden. Die Programmzähler der Prozesse sowie die Variablenbelegungen lassen sich als endliche Automaten $M_{PC,i}$ und $M_{PV,i}$ modellieren (siehe Abschnitt 2.2.2). Enthält ein Modul keinen SystemC-Thread, wird kein endlicher Automat für den Programmzähler benötigt. Weiterhin wird für jede Transaktion $t \in \{t' = (tm, tm') \in T \mid tm = tm_i\}$ ein Initiator-Automat $M_{I,t}$ und für jede Transaktion $t \in \{t' = (tm, tm') \in T \mid tm' = tm_i\}$ ein Target-Automat $M_{T,t}$ gebildet. Dies ist in Abb. 8.14 zu sehen.

Weiterhin zeigt Abb. 8.14 das Zusammenspiel der kommunizierenden endlichen Automaten. Der Automat $M_{PC,1}$, der einen SystemC-Thread modelliert, kann somit Änderungen an der Variablenbelegung vornehmen, indem er mit dem Automaten

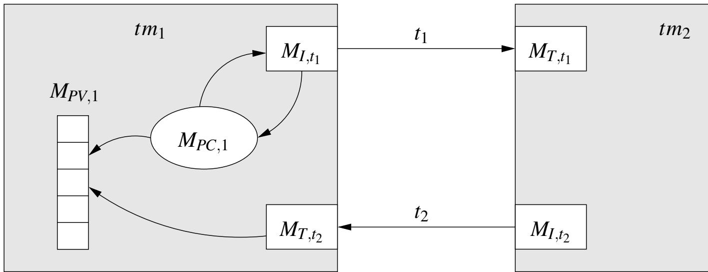


Abb. 8.14. Formales Modell eines SystemC-TLM [346]

$M_{PV,1}$ kommuniziert. Auch kann der SystemC-Thread Transaktionen vom Typ t_1 initiieren, indem er ein Signal an den Automaten M_{I,t_1} schickt. Das Ergebnis der Transaktion teilt M_{I,t_1} ebenfalls über ein Signal dem Automaten $M_{PC,1}$ mit. Die Target-Automat M_{T,t_2} reagiert auf Transaktionen vom Typ t_2. Diese können Änderungen an der Variablenbelegung zur Folge haben.

Definition 8.1.20 (Kommunizierende endliche Automaten). *Gegeben seien zwei endliche Automaten* $M_1 = (I_1, O_1, S_1, s_{0,1}, f_1, g_1)$ *und* $M_2 = (I_2, O_2, S_2, s_{0,2}, f_2, g_2)$ *nach Definition 2.2.13 auf Seite 47 mit Anfangszustand* $s_{0,1}$ *bzw.* $s_{0,2}$*. Die beiden Automaten können miteinander* kommunizieren*, wenn* $O_1 \subseteq I_2$ *und* $O_2 \subseteq I_1$ *ist.*

Sei beispielsweise M_1 im Zustand $s_{1,1}$ und M_2 im Zustand $s_{1,2}$. Führe M_1 den Zustandsübergang $s_{1,1} \xrightarrow{/e} s_{2,1}$ durch, so würde M_1 in dem Automaten M_2 einen Zustandsübergang aktivieren, wenn ein Zustandsübergang mit Eingangssymbol e im Zustand $s_{1,2}$ existiert, d. h. $s_{1,2} \xrightarrow{e} s_{2,2}$. Im Folgenden werden die einzelnen endlichen Automaten zur formalen Modellierung von TLMs genauer beschrieben.

Prozess- und Variablen-Automat

Prozess-Automaten modellieren das Verhalten der SystemC-Prozesse, dies ist vergleichbar mit dem Programmzähler in dem in Abschnitt 8.1.2 beschriebenen Ansatz. Hier wird kurz auf die Besonderheiten bei der Modellierung der Kommunikation eingegangen.

Beispiel 8.1.10. Gegeben ist ein Initiatormodul mit Port `init_socket` vom Typ `tlm_initiator_socket`. Der Prozess ist wie folgt definiert:

```
1    void process() {
2      while(1) {
3        init_socket.b_transport(data);
4        d++;
5      }
6    }
```

Die durch `b_transport` ausgelöste Transaktion überträgt dabei ein Datum `data` vom Typ `tlm_generic_payload`. Das Datenfeld von `data` enthält den Wert der Variablen `d` vom Typ `sc_uint<2>`, einen Bitvektor der Breite zwei. Somit wird der Wert, der in `d` gespeichert ist, durch die Transaktion übertragen. Der resultierende Prozess-Automat M_{PC} ist in Abb. 8.15a) zu sehen. Im Anfangszustand s_0 wartet der Prozess-Automat auf das Startsignal *run*, welches vom SystemC-Simulator erzeugt wird. Der Simulator wird später ebenfalls formal modelliert. Hat der Prozess-Automat das *run*-Signal empfangen, beginnt die Transaktion durch Erzeugung des Signals *transport_start*. Erst wenn die Kommunikation beendet wurde (*transport_end*), wird die eigentliche Berechnung des Moduls ausgeführt, also das Inkrement der Variable `d` berechnet. Da hierdurch die Variablenbelegung geändert wird, erfolgt die Änderung in dem Variablen-Automaten M_{PV}, welcher durch das Signal *calculate* aktiviert wird. Abbildung 8.15b) zeigt den Variablen-Automaten.

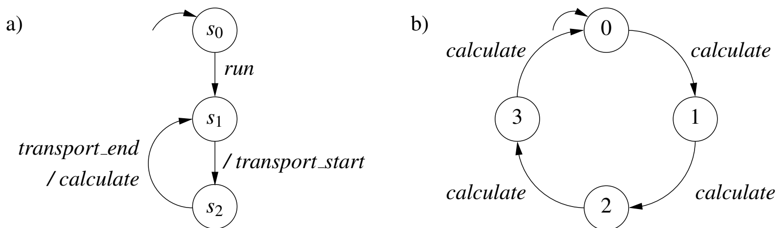


Abb. 8.15. a) Prozess-Automat und b) Variablen-Automat [347]

Initiator- und Target-Automat

Die Modellierung einer Transaktion erfolgt als Paar von Initiator- und Target-Automat. Der Initiator-Automat reagiert auf das Signal *transport_start* des Prozess-Automaten und initiiert die Kommunikation mittels des Signals *request*. Der Initiator-Automat wiederholt die Anfrage mit dem selben Signal solange, bis der Target-Automat mit dem Signal *response* das Ende der Transaktion signalisiert. Der Initiator-Automat ist in Abb. 8.16a) dargestellt.

Der korrespondierende Target-Automat ist in Abb. 8.16b) dargestellt. Der Target-Automat verbleibt im Anfangszustand s_0 solange, bis das Signal *request* eine Transaktion initiiert. Ob die Transaktion direkt durchgeführt werden kann, hängt vom Zustand des Targetmoduls ab, dies wird über die Variable f signalisiert. Kann die Transaktion momentan nicht durchgeführt werden ($\neg f$), so wird dies dem SystemC-Simulator mittels des *wait*-Signals mitgeteilt. Als Argument wird dem SystemC-Simulator ein Ereignis e übergeben, welches die Blockierung aufheben kann. Dies veranlasst den SystemC-Simulator, den SystemC-Prozess des Initiatormodul, der die Transaktion initiiert hat, zu blockieren. Die Transaktion kann erst fortgeführt werden, wenn der SystemC-Simulator das Signal *run* erzeugt. Ist die Bedingung zur

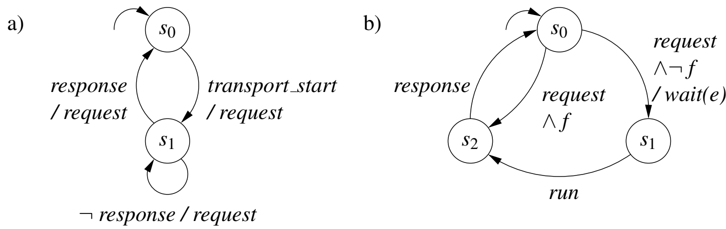


Abb. 8.16. a) Initiator-Automat und b) Target-Automat [347]

Abarbeitung der Transaktion direkt gegeben (f), wird die Transaktion durchgeführt. Die Beendigung der Transaktion wird durch das Signal *response* angezeigt.

Man beachte, dass durch den Zustand s_1 das Blockieren eines SystemC-Prozesses modelliert wird. Dieser Mechanismus kann auch für `wait`-Anweisungen im SystemC-Prozess angewendet werden.

Modellierung des SystemC-Simulators

Schließlich muss noch der SystemC-Simulator selbst modelliert werden. Der im Folgenden vorgestellte Ansatz unterscheidet sich von dem in Abschnitt 8.1.2 dadurch, dass dynamische Sensitivitätslisten unterstützt werden und Zeitbetrachtungen mit aufgenommen werden können.

Der SystemC-Simulator wird ebenfalls als endlicher Automat modelliert und realisiert die Ablaufplanung nach dem SystemC-Standard [236]. Ein SystemC-Simulator für n SystemC-Prozesse $p_1, \ldots, p_n$ ist ein endlicher Automat, wobei die Zustände mit Trippeln (σ, π, ϕ) markiert sind. Dabei ist

- $\sigma \in \{\perp, p_1, \ldots, p_n\}$ die Prozessauswahl , die eventuell leer ist ($\perp$),
- $\pi := (\pi_1, \ldots, \pi_n)$ der Zustandsvektor aller n Prozesse, mit dem Prozesszustand $\pi_i \in \{bereit, laufend, blockiert\}$ für jeden SystemC-Prozess p_i, und
- $\phi \in \{evaluate, update, delta_notify, timed_notify\}$ zeigt die Ablaufplanungsphase an.

Die Initialisierung des Simulators erfolgt als

$$s_0 = (\perp, bereit, \ldots, bereit, evaluate)$$

Somit startet der Simulator in der Evaluierungsphase, in der alle Prozesse *bereit* sind.

Der SystemC-Simulator wird im nächsten Schritt einen Prozess p_i auswählen und ausführen, d. h. die Prozessauswahl wird auf $\sigma = p_i$ aktualisiert und p_i geht in den Prozesszustand $\pi_i = laufend$ über. Prozess p_i wird ausgeführt, bis er die nächste `wait`-Anweisung erreicht. Dabei geht er in den Prozess-Zustand $\pi_i = blockiert$ über und die Prozessauswahl wird auf $\sigma = \perp$ gesetzt.

Der Simulator wählt so lange Prozesse aus und führt diese aus, bis kein Prozess mehr im Prozesszustand *bereit* ist, d. h.

$$\sigma = \perp \wedge \forall 1 \leq i \leq n : \pi_i = blockiert$$

Daraufhin schaltet der SystemC-Simulator in die Ablaufplanungsphase $\phi = update$. In dieser Phase werden die SystemC-Kanäle aktualisiert, dies bedeutet, dass die Variablen f in Abb. 8.16b) neu bewertet werden. Anschließend wechselt der SystemC-Simulator in die Ablaufplanungsphase $\phi = delta_notify$. Dies hat zur Folge, dass alle Ereignisse, die während der Evaluierungsphase für den nächsten Delta-Zyklus erzeugt wurden, nun sichtbar werden. Damit werden alle Prozesse p_i mit $\pi_i = blockiert$, die auf ein solches Ereignis warten, auf $\pi_i = bereit$ gesetzt werden. Schließlich erfolgt der Wechsel des SystemC-Simulators in die Ablaufplanungsphase $\phi = evaluate$. Wird allerdings kein Prozess laufbereit, so wechselt der SystemC-Simulator zunächst in den Zustand *timed_notify* und die Simulationszeit wird auf den nächste Zeitpunkt gesetzt, zu dem ein Ereignis auftritt.

Beispiel 8.1.11. Abbildung 8.17 zeigt die Prozesszustandsübergänge für den Prozess aus Beispiel 8.1.10. Der Prozesszustand π_i wird von *bereit* auf *laufend* gesetzt, sobald der Prozess p_i ausgewählt wurde ($\sigma = p_i$) und Signal *run* erzeugt wird. Beim Erreichen einer `wait`-Anweisung blockiert der Prozess an dem zugehörigen Ereignis e. Erst das Auftreten von e kann den Prozess wieder in den Zustand *bereit* versetzen.

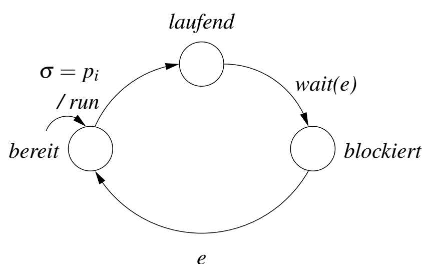


Abb. 8.17. Prozesszustandsübergänge für den Prozess p_i aus Beispiel 8.1.10 [347]

Modellierung von Kanälen

Die Berücksichtigung von Kanälen verlangt, dass der in SystemC verwendete *request/update*-Mechanismus umgesetzt wird. Dies erfolgt wiederum in einem endlichen Automaten (siehe Abb. 8.18a)) und zwar für jeden Kanal einzeln. Der Automat startet im Zustand s_0. Wird eine Kanalaktualisierung angezeigt, d. h. ein Prozess hat eine Transaktion auf dem Kanal durchgeführt, geht der jeweilige Automat in den Zustand s_1 über. Wechselt der SystemC-Simulator in die Ablaufplanungsphase $\phi = update$, so wechseln alle Automaten von geänderten Kanälen in den Zustand s_2, in welchem die eigentliche Aktualisierung stattfindet. Diese Aktualisierung ist kanalspezifisch und deshalb nicht in Abb. 8.18a) dargestellt. Die Beendigung der Aktualisierung erfolgt durch Übergänge nach s_0 und senden von *done*-Signalen.

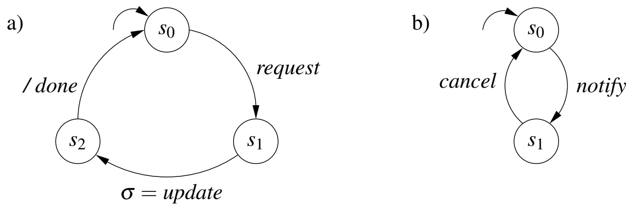


Abb. 8.18. a) Modellierung der Kanalaktualisierung und b) Modellierung von Ereignissen [347]

Modellierung von Ereignissen

Jedes SystemC-Ereignis wird ebenfalls als endlicher Automat modelliert. Dieser ist in Abb. 8.18b) dargestellt. Bei der Erzeugung eines Ereignisses durch eine `notify`-Anweisung wird das Ereignis *notify* erzeugt, was den Zustandsübergang nach s_1 verursacht. Dieser Zustand zeigt an, dass ein Ereignis später sichtbar werden soll. Durch Aufruf einer *cancel*-Anweisung kann dieser Zustand wieder verlassen werden, bevor der Simulator das Ereignis berücksichtigt.

Dieses formale Modell kommunizierender Automaten erfasst das Verhalten beliebiger SystemC-TLMs. Für die Modellprüfung kann die Übersetzung eines Modells eines SystemC-TLMs in eine temporale Struktur erfolgen [347]. Alternativ ist eine Übersetzung auf attributierte temporale Strukturen denkbar.

8.1.4 Zusicherungsbasierte Eigenschaftsprüfung für Transaktionsebenenmodelle

Auf der Register-Transfer-Ebene werden Zusicherungen als Sequenzen Boolescher Ausdrücke über beobachtete Signalwerte beschrieben. Die Auswertung einer Sequenz erfolgt dann mittels eines Monitors synchron zum SUV (engl. *System Under Verification*). Dabei wird der Wechsel eines Taktsignals oder eines anderweitig geeigneten Signals verwendet, um diejenigen Zeitpunkte zu bestimmen, an denen die Booleschen Ausdrücke auszuwerten sind. Auf Transaktionsebene werden Modelle abstrakter modelliert und Ereignisse weitestgehend vermieden, um Kontextwechsel zum Simulator zu vermeiden. Nur an denjenigen Stellen, wo es zu Konflikten im Daten- oder Kontrollfluss kommen kann, werden Ereignisse zur Synchronisation eingesetzt. Dies hat aber auch zu Folge, dass in der Regel kein Taktsignal verfügbar ist, um die Zeitpunkte zur Evaluierung zu bestimmen. Aus diesem Grund ist es notwendig, geeignete Ereignisse im TLM zu generieren.

In [361] ist ein Ansatz beschrieben der auf dem sog. engl. *Observer Pattern* basiert. Jeder Monitor verwendet einen *Observer*, um diejenigen Kanäle und Signale im SUV zu beobachten, die er zur Überprüfung der Zusicherung benötigt. Dabei kann ein Observer-Objekt von mehreren Monitoren verwendet werden. Weiterhin kann ein Kanal oder ein Signal von mehreren Observern beobachtet werden. Damit die Observer über relevante Ereignisse (hierzu gehören auch Änderungen am Status eines

Kanals) informiert werden, ist es notwendig, die verwendeten Kommunikationsmethoden (*nb_transport*, *b_transport* etc.) zu überladen. Dies wird an dem Beispiel aus [361] für einen einfachen FIFO-Kanal illustriert.

Beispiel 8.1.12. Es wird lediglich die Leseoperation `read` betrachtet. Der FIFO-Kanal ist vom Typ `fifo`. Ohne die genaue Implementierung des FIFO-Kanals zu kennen, kann die Leseoperation beobachtbar gemacht werden, indem von `fifo` ein neuer Typ `observable_fifo` abgeleitet wird.

```
1   class observable_fifo : public fifo, observer {
2     public:
3       observable_fifo(sc_module_name name) : fifo(name) {};
4       void read(char& c) {
5         fifo::read(c);
6         obs_notify(&c);
7       }
8   };
```

In Zeile 3 ist der Konstruktor des beobachtbaren FIFO-Kanals implementiert, indem lediglich der Default-Konstruktor der ursprünglichen FIFO-Klasse aufgerufen wird. In Zeile 4 wird die Methode `read` überladen. Diese ruft die originale `read`-Methode auf und benachrichtigt alle registrierten Observer in Zeile 6.

Zur Generierung der Monitore auf Transaktionsebene kann dann das in Abschnitt 6.4.1 beschriebene Verfahren angewandt werden, wobei komplexe Monitore aus elementaren Monitoren zusammengesetzt werden. Dabei ist die Verwendung eines Taktsignals nicht mehr nötig. Dies wird ebenfalls anhand eines Beispiels [361] illustriert.

Beispiel 8.1.13. Es soll ein elementarer Monitor zur Überprüfung der PSL-Formel always *expr* gebaut werden. Eine Implementierung in C++ kann dann wie folgt aussehen:

```
1   class monitor_always : public monitor {
2     protected:
3       bool reset_n, start, expr, checking, valid,
4         start_always, start_t, valid_t;
5
6     public:
7       monitor_always(const char* name) : monitor(name) {
8         valid_t = true;
9       }
10      void update() {
11        start_always = start || start_t;
12        if ((!reset_n) || (!start_always))
13          valid_t = true;
14        else
```

```
15          if (expr)
16            valid_t = true;
17          else
18            valid_t = false;
19        if (!reset_n)
20          start_t = false;
21        else
22          if (start)
23            start_t = true;
24        valid = valid_t;
25        checking = start_always;
26      }
27  };
```

Die Variablen `reset_n`, `start` und `expr` sind die Eingangssignale des Monitors und können von außerhalb des Monitors geschrieben werden. Die Methode `update` wird nach jeder Änderung an einem dieser drei Signale aufgerufen. Die Ausgabe des Monitors sind die Signale `checking` und `valid` (wie bereits in Abschnitt 6.4.1 eingeführt).

Die Variable `start_t` wird verwendet, um ein einmal gegebenes Startsignal bis zum nächsten Zurücksetzen zu speichern. Somit zeigt die in Zeile 11 berechnete Variable `start_always` an, ob überhaupt eine Überprüfung bei dem aktuellen Aufruf von `update()` durchgeführt wird. Falls dies nicht der Fall ist oder gerade ein aktives Rücksetzsignal (`reset`) anliegt, wird die Variable `valid_t` auf den Default-Wert `true` gesetzt (Zeile 12 und 13). Andernfalls wird in den Zeilen 15 bis 18 überprüft, ob `expr` erfüllt ist. Die Ausgabe wird erst in den Zeilen 24 und 25 erzeugt.

Man sieht, dass die Monitore als C++-Klassen implementiert werden können. Da diese über den Observer gesteuert werden, finden keine Kontextwechsel durch Monitore statt, was ansonsten einen enormen Overhead an Simulationszeit während der Verifikation zur Folge hätte.

Die Konstruktion komplexer Monitore für PSL-Zusicherungen erfolgt mit dem in [361] beschriebenen Ansatz durch Verschaltung elementarer Monitore (siehe auch Abschnitt 6.4.1). Dies wird anhand eines Beispiels aus [361] illustriert.

Beispiel 8.1.14. Abbildung 8.19a) zeigt ein Transaktionsebenenmodell eines Producer/Consumer-Systems. Der Produzent (engl. *producer*) erzeugt Nachrichten, welche aus einzelnen Zeichen bestehen, und schreibt diese in einen FIFO-Kanal. Eine Nachricht beginnt stets mit dem Symbol „%“ und endet mit dem Zeichen „$“. Der Konsument (engl. *consumer*) liest die Nachrichten als Sequenz von Zeichen aus dem FIFO-Kanal. Es soll nun gezeigt werden, dass die Übertragung einer neuen Nachricht erst beginnt, nachdem die vorherige Nachricht vollständig geschrieben wurde.

Mit Hilfe des in Beispiel 8.1.12 eingeführten Monitors, kann dies als PSL-Zusicherung formuliert werden.

$$\text{assert always } ((char ==' \%') \text{ -> next}((\text{not}(char ==' \%')) \text{ until! } (char ==' \$')))$$

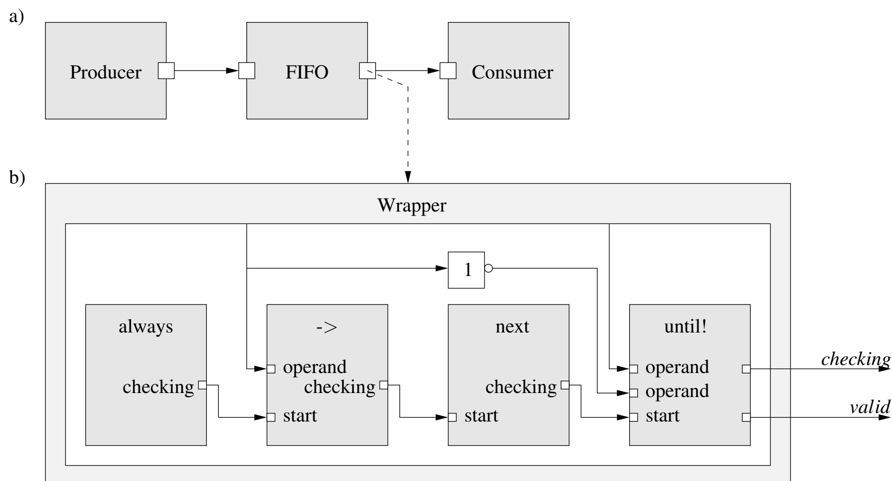


Abb. 8.19. PSL-Monitor für ein TLM eines Producer/Consumer-System [361]

Die Methode `read` des FIFO-Kanals mit beobachtbarer Leseoperation kann mittels einer C++-Klasse das gelesene Zeichen *char* zurückliefern. Da PSL neben VHDL und SystemVerilog ebenfalls SystemC unterstützt, kann dieses Zeichen direkt zum Vergleich gegen das Start- und Endezeichen einer Nachricht innerhalb einer PSL-Zusicherung verwendet werden. Die Umsetzung der PSL-Zusicherung erfolgt durch Verschaltung elementarer Monitore [361]:

```
 1   class monitor_msg_seq : public monitor {
 2     public:
 3       monitor_msg_seq(const char* name) : monitor(name) {
 4         mnt_always = new monitor_always("always");
 5         mnt_imply = new monitor_imply("->");
 6         mnt_next = new monitor_next("next", WEAK, 1);
 7         mnt_until = new monitor_until("until!", STRONG);
 8         ...
 9       };
10       void update() {
11         expr_0 = (char == '%');
12         expr_1 = !(char == '%');
13         expr_2 = (char == '$');
14         mnt_until->update();
15         valid = valid_i;
16         checking = checking_i;
17         ...
18       }
19   };
```

Im Konstruktor `monitor_msg_seq` werden die elementaren Monitore instantiiert. Hierbei wird auch der Always-Monitor aus Beispiel 8.1.13 verwendet. Der Until- und Next-Monitor erhalten zusätzliche Argumente, die anzeigen, ob es sich um einen schwachen oder einen starken Operator handelt. Der Next-Operator bekommt zusätzlich die Anzahl an Zeitschritten mitgeteilt, nach denen die zugehörige Teilformel ausgewertet werden soll. Die Zeitschritte ergeben sich aus dem Ereignis, welches in der `read`-Methode des FIFO erzeugt wird. In der Methode `update` werden die auszuwertenden Ausdrücke gesetzt. Anschließen wird die Auswertung des am weitesten rechts stehenden Until-Monitors gestartet (Zeile 14), der wiederum die angeschlossenen Monitore startet. Die Ausgabe des Until-Monitors (`valid_i` und `checking_i`) wird mit den Ausgängen `valid` und `checking` des zusammengesetzten Monitors verbunden.

Ein weiteres Beispiel [361] soll die zusicherungsbasierte Eigenschaftsprüfung von Transaktionsebenenmodellen verdeutlichen.

Beispiel 8.1.15. Abbildung 8.20 zeigt ein System bestehend aus einer CPU, einem DMA-Controller (engl. *direct memory access*), zwei Speicher-Modulen und einem Bus. In dem DMA-Controller gibt es vier Register, die über bestimmte Adressen angesprochen werden können. Für Kopieroperationen zwischen den beiden Speichermodulen wird der DMA-Controller von der CPU wie folgt programmiert:

1. Zunächst schreibt die CPU die Quelladresse in das zugehörige DMA-Register `source`.
2. In einem zweiten Schritt wird die Zieladresse von der CPU in das DMA-Register `sink` geschrieben.
3. Anschließend speichert die CPU die Länge des zu kopierenden Blocks in das DMA-Register `length`.
4. Schließlich initiiert die CPU den Transfer durch setzten des DMA-Kontrollregisters `control`.

Alle Kopieroperationen werden von der CPU initiiert und mittels `b_transport`-Aufrufen realisiert. Das eigentliche Kopieren der Daten führt der DMA-Controller dabei selbständig durch alternierende Lese- und Schreiboperationen aus. Die Beendigung des Kopiervorgangs signalisiert der DMA-Controller der CPU mittels eines Interrupt-Signals *dma_irq*. Dieses Interrupt-Signal ist im TLM als ein Signal implementiert. Das Interrupt-Signal wird von der CPU gelöscht, indem das Kontrollregister im DMA-Kontroller zurückgesetzt wird.

Im Folgenden werden vier PSL-Zusicherungen für dieses Modell vorgestellt. Dabei wird einfachheitshalber davon ausgegangen, dass Kopieroperationen stets vom Speicher 1 (`mem1`) in den Speicher 2 (`mem2`) erfolgen. Die erste Zusicherung besagt, dass nach der Übertragung der Quelladresse als nächste Transaktion die Übertragung der Zieladresse erfolgt. Hierzu muss der Initiator-Socket `cpu_init_socket` der CPU überwacht werden. Dies erfolgt durch Überladen der `b_transport`-Implementierung und die Verwendung eines Wrapper-Moduls. Die PSL-Zusicherung kann dann wie folgt formuliert werden:

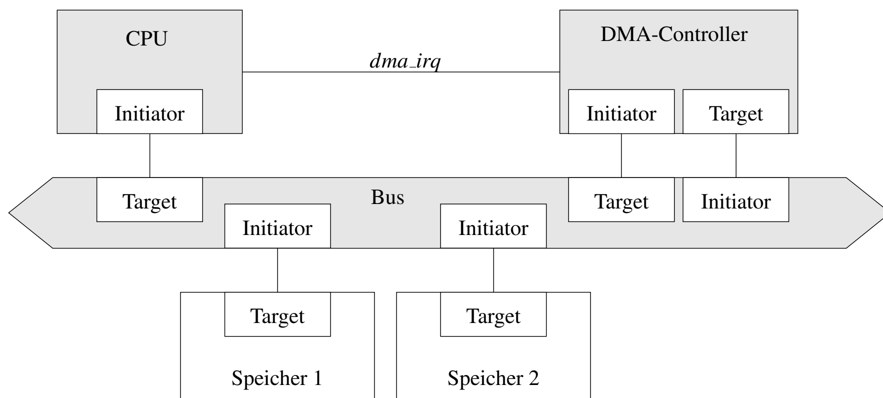


Abb. 8.20. SystemC-TLM für DMA-Speicherzugriffe

assert always
((`cpu_init_socket.isWritten`()&&(`writtenAddr` == `source`)) ->
(next (next_event(`cpu_init_socket.isWritten`())
(`writtenAddr` == `sink`))));

Die zweite Zusicherung fordert, dass jedes Mal, nachdem das Kontrollregister des DMA-Controller geschrieben wurde (nicht zurückgesetzt wurde), das Interrupt-Signal *dma_irq* gesetzt werden muss, bevor das Kontrollregister wieder geschrieben wird.

assert always
((`cpu_init_socket.isWritten`()&&(`writtenAddr` == `control`)&&
(`writtenData`! =`reset`)) ->
(next (*dma_irq* before
(`cpu_init_socket.isWritten`()&&(`writtenAddr` == `control`)))));

Schließlich soll gezeigt werden, dass die Lese- und Schreiboperationen zwischen den beiden Speichern alternieren:

assert always (next_event(`mem1.isRead`())
(next (`mem2.isWritten`() before `mem1.isRead`())))

Die vierte Zusicherung schließlich fordert, dass die verwendete decodierte Leseadresse (`decodedAddr`) identisch ist mit der von der CPU geschriebenen Leseadresse (`readAddr`).

assert always
((`cpu_init_socket.isWritten`()&&(`writtenAddr` == `source`)) ->
(next_event!(`mem1.isRead`()) (`readAddr` == `decodedAddr`)));

8.2 Zeitanalyse auf Systemebene

Eingebettete Systeme werden zunehmend als heterogene Multiprozessorsysteme entworfen. In einem solchen System sind die Komponenten oftmals hoch optimiert, um die Anforderungen zu erfüllen, die sich aus der Umgebung und der zu erbringenden Funktionalität ergeben. Dies gilt sowohl für die Prozessoren und Hardware-Beschleuniger als auch für die Kommunikationsbusse und Speicher. Hieraus ergibt sich, dass der Entwurf und die Verifikation eingebetteter Systeme zunehmend schwieriger wird. Dabei beschränkt sich die Verifikation nicht nur auf funktionale Eigenschaften, wie Gefahrlosigkeit und Lebendigkeit, sondern sie muss auch die Korrektheit des Zeitverhaltens garantieren.

Um die Komplexität der Zeitanalyse auf Systemebene zu verstehen, muss man die Entscheidungen im Systementwurf kennen, da diese Einfluss auf das Zeitverhalten der Implementierung haben. Typische Entwurfsentscheidungen auf Systemebene sind [426]: Die Partitionierung in Hardware- und Software-Komponenten, die Allokation der Prozessoren, die Bindung von Prozessen an die Komponenten, die Abbildung von Daten in Speicher, das Routing von Speicherzugriffen sowie die Ablaufplanung von Prozessen und Transaktionen auf gemeinsam genutzten Ressourcen. Alle diese Entwurfsentscheidungen haben Einfluss auf das Zeitverhalten der Implementierung und müssen daher in einer Zeitanalyse berücksichtigt werden. Dies ist im *Y-Diagramm* in Abb. 8.21 dargestellt.

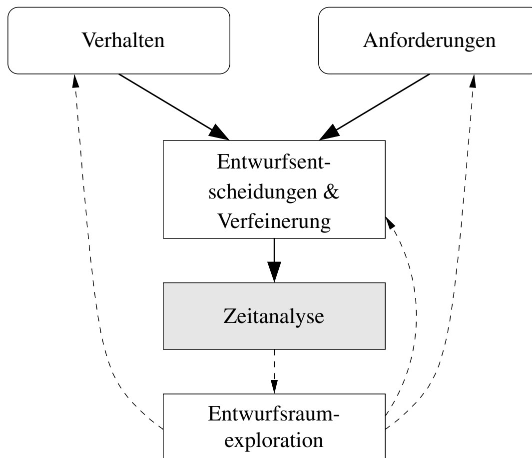


Abb. 8.21. Y-Diagramm

Ausgehend von dem Verhaltensmodell und den Anforderungen der Spezifikation werden die oben beschriebenen Entwurfsentscheidungen getroffen und das Verhaltensmodell verfeinert. Die resultierende Implementierung ist die Eingabe für die Zeitanalyse, deren Ergebnisse im Entwurfsprozess zur Steuerung der Optimie-

rung verwendet werden. Man beachte, dass das Y-Diagramm Teilaspekte der X-Diagramme für den Entwurf und für die Verifikation zusammenfasst. Weiterführende Erläuterungen zum Entwurfsprozess und zur Entwurfsraumexploration finden sich in [426] und werden hier nicht näher erläutert.

Das Verifikationsziel bei der Prüfung von Zeiteigenschaften kann entweder der Beweis oder die Falsifikation der Eigenschaften sein. Bei der Falsifikation werden solange neue Ergebnisse mit der Zeitanalyse ermittelt und mit den Anforderungen verglichen, bis ein Gegenbeispiel gefunden ist bzw. bis der Entwickler soviel Vertrauen in die Implementierung hat, dass keine weiteren Bewertungen vorgenommen werden. Die Zeitanalyse spielt hierbei unterschiedliche Szenarien durch, um möglichst schnell ein Gegenbeispiel zu generieren. Ist das Verifikationsziel der Beweis, so liefert die Zeitanalyse sog. *Zeitschranken* zurück (siehe auch Abb. 7.28 auf Seite 433). Zeitschranken sind minimale und maximale Werte für Zeiteigenschaften, wie Latenz, Durchsatz etc. Man spricht in diesem Zusammenhang auch von *unteren* und *oberen Zeitschranken*. Die ermittelten Zeitschranken werden mit den Zeitanforderungen verglichen, und nur, wenn die Zeitschranken keine Anforderung verletzen, ist der Beweis erbracht, dass das System die Zeitanforderungen erfüllt.

Aus dem *Y-Diagramm* lassen sich Kriterien für Verfahren zur Zeitanalyse auf Systemebene ableiten [435]:

- *Korrektheit:* Die Ergebnisse der Zeitanalyse müssen korrekt sein. Handelt es sich bei dem Verifikationsziel um einen Beweis, so muss das Ergebnis der Zeitanalyse für alle erreichbaren Systemzustände und alle möglichen Eingaben gelten. So darf beispielsweise die obere Schranke für eine Latenz die tatsächlich maximale Antwortzeit eines Systems nicht unterschreiten. Handelt es sich bei dem Verifikationsziel um eine Falsifikation, so muss das erbrachte Gegenbeispiel, eine zulässige Eingabe für die Implementierung darstellen.
- *Genauigkeit:* Um *falschnegative* und *falschpositive Ergebnisse* zu verhindern, müssen die Ergebnisse der Zeitanalyse genau genug sein. Dies bedeutet, dass die berechneten unteren und ober Schranken möglichst genau den tatsächlich minimal und maximal erreichbaren Zeiteigenschaften entsprechen bzw. das Gegenbeispiel genau genug ist, so dass es auch in einem gefertigten System potentiell gilt. Beispielsweise kann eine obere Schranke an die Latenz beliebig hoch gesetzt werden, ohne das die tatsächliche maximale Antwortzeit des Systems unterschritten wird (Korrektheit). Allerdings hilft eine solche willkürlich hohe Schranke nicht, das System angemessen auszulegen.
- *Anwendbarkeit:* Damit eine Zeitanalyse für eine Vielzahl an Systemen anwendbar ist, muss sie möglichst alle oben genannten Entwurfsentscheidungen repräsentieren können. Darüber hinaus muss eine Zeitanalyse eine möglichst große Klasse an Berechnungsmodellen für die Modellierung des Verhaltens unterstützen.
- *Geschwindigkeit:* Schließlich sollten die Ergebnisse der Zeitanalyse möglichst schnell vorliegen, um während der Entwurfsraumexploration eine Vielzahl an Lösungen bewerten zu können.

Wie im Fall der bisher vorgestellten Methoden zur funktionalen Verifikation, ist es im Allgemeinen auch im Fall der Zeitanalyse nicht möglich, alle oben genannten Kriterien zu erfüllen. Bei der Zeitanalyse können die verwendeten Verfahren wiederum in simulative und formale Methoden eingeteilt werden. Wie bei der funktionalen Verifikation gilt auch hier, dass simulative Methoden unvollständig sind und sich somit im Allgemeinen nur für das Verifikationsziel der Falsifikation eignen. Formale Methoden zur Zeitanalyse können wiederum für das Verifikationsziel eines Beweises verwendet werden. Auch sonst stehen die oben genannten Kriterien im Konflikt zueinander, weshalb eine Zeitanalysemethode nicht alle Kriterien in vollem Umfang erfüllen kann. Unabhängig davon, ob simulative oder formale Methoden eingesetzt werden, wird im Allgemeinen eine höhere Genauigkeit mit einer längeren Laufzeit einhergehen. Andererseits bedeutet eine höhere Anwendbarkeit auch in der Regel eine Einschränkung in der Korrektheit der Methode.

Im Folgenden werden nun simulative und formale Methoden zur Zeitanalyse auf Systemebene vorgestellt. Zunächst wird ein simulatives Verfahren vorgestellt, welches eine Vielzahl an Entwurfsentscheidungen berücksichtigen kann und somit für die Zeitbewertung moderner eingebetteter Systeme auf Systemebene geeignet ist. Der Nachteil ist allerdings, dass dieses Verfahren, wie alle anderen simulativen Ansätze, unvollständig ist. Anschließend werden zwei formale Methoden zur Zeitanalyse auf Systemebene vorgestellt. Das erste der beiden Verfahren ist eine kompositionale Zeitanalyse, die Ereignisströme zur Kopplung bestehender Planbarkeitsanalysen für Einprozessorsystemen verwendet. Das zweite Verfahren ist eine modulare Zeitanalyse basierend auf Ankunfts- und Servicekennlinien.

8.2.1 Simulative Zeitbewertung

Ein Ansatz zur simulativen Zeitanalyse der direkt dem Y-Diagramm in Abb. 8.21 entspricht, ist an der Universität Erlangen-Nürnberg entwickelt und in das Entwurfssystem SystemCoDesigner zur Entwurfsraumexploration und Systemsynthese integriert worden [215, 216, 254]. Für die Spezifikation erwartet das Entwurfssystem ein ausführbares SysteMoC-Verhaltensmodell (siehe Abschnitt 2.3.2). Entwurfsbeschränkungen werden in Form einer Plattform, modelliert als Architekturgraph, und Bindungsbeschränkungen angegeben. Die Entwurfsraumexploration mit SystemCoDesigner erfolgt vollautomatisch, wobei mehrere Zielgrößen gleichzeitig optimiert werden können. Für die Abschätzung des Zeitverhaltens während der Exploration werden aus dem SysteMoC-Modell und den Entwurfsentscheidungen automatisch zeitbehaftete SystemC-Modelle generiert. Um dabei die Explorationszeiten gering zu halten, erfolgt die Erstellung des zeitbehafteten Modells per Konfiguration und bedarf keiner Neukompilation des Modells.

Virtual Processing Components (VPC)

Der in SystemCoDesigner verwendete Ansatz zur Abschätzung des Zeitverhaltens ist eine simulative Zeitanalyse, basierend auf einer SystemC-Bibliothek mit dem Namen *VPC* (engl. *Virtual Processing Components*). Das Verhaltensmodell ist in SysteMoC programmiert. Es handelt sich dabei um ein rein funktionales Modell, ohne

jegliche Informationen über nichtfunktionale Eigenschaften wie Zeitverhalten. Da SysteMoC-Modelle ausführbare Verhaltensmodelle sind, können sie entsprechend zur Simulation des Verhaltens verwendet werden. Die allozierten Komponenten der Plattform, z. B. Prozessoren, Hardware-Beschleuniger, Speicher, Busse, werden als *virtuelle Komponenten* (engl. *Virtual Processing Components, VPCs*) modelliert. Jede VPC repräsentiert somit eine Komponente der Plattform. Technisch sind VPCs als SystemC-Module in einer SystemC-Bibliothek, der sog. *VPC-Bibliothek* realisiert. VPCs sind in der Simulation dafür zuständig, die Ausführung von Aktoren entsprechend ihrer geschätzten Ausführungszeiten zu verzögern, d. h. VPCs übernehmen die Zeitsimulation.

Zur Kopplung der Anwendung mit der Plattform, müssen SysteMoC-Aktoren entsprechend der Entwurfsentscheidungen an VPCs gebunden werden. Dies erfolgt in zwei Schritten:

1. Jede VPC erhält die auf tieferen Abstraktionsebenen geschätzten Ausführungszeiten derjenigen Aktoren, die an die entsprechende Ressource gebunden sind.
2. Funktionen in SysteMoC-Aktoren werden am Ende der Funktion mit einem Funktionsaufruf ergänzt, der die weitere Ausführung des Aktors an derjenigen VPC blockiert, an die der Aktor gebunden wurde.

Das Ergebnis dieser beiden Schritte ist ein kombiniertes Simulationsmodell, welches sowohl das funktionale Verhalten als auch das Zeitverhalten simuliert. Bei den Ausführungszeiten handelt es sich um die geschätzten Zeiten für die Ausführung der Kernfunktionalität (engl. *core execution times*), also derjenigen Zeit, die zur Berechnung des Aktors ohne Unterbrechung durch externe Ereignisse benötigt wird.

Die Ausführungssemantik eines einzelnen SysteMoC-Aktors, mit endlichen Automaten M, in einer kombinierten Verhaltens- und Zeit-Simulation ist damit wie folgt:

1. *Initialisierung:* Der aktuelle Zustand des endlichen Automaten M wird auf den Anfangszustand gesetzt.
2. *Prädikatenevaluierung:* Alle Bedingungen von aus dem aktuellen Zustand ausgehenden Zustandsübergängen des endlichen Automaten M werden evaluiert.
3. *Funktionsausführung:* Ein Zustandsübergang aus dem aktuellen Zustand, dessen Prädikat erfüllt ist, wird nichtdeterministisch ausgewählt, und die annotierte Funktion berechnet. Ist kein Prädikat erfüllt, wartet der Aktor auf eine Änderung der Füllstände der angeschlossenen Kanäle und führt dann seine Ausführung in Schritt 2. fort.
4. *Zeitsimulation:* Durch Aufruf der VPC, an die der Aktor gebunden ist, blockiert der Aktor und die VPC übernimmt die Zeitsimulation.
5. *Zustandsänderung:* Entsprechend dem Prädikat des ausgewählten Zustandsübergangs werden Daten konsumiert und produziert sowie der Folgezustand gesetzt. Die Ausführung wird mit Schritt 2. fortgesetzt.

Bei der beschriebenen Ausführungssemantik sieht man, dass Funktionen atomar also ohne Unterbrechung berechnet werden. Hierdurch werden Seiteneffekte vermieden, die evtl. den in Variablen gespeicherten Zustand ändern könnten.

Beispiel 8.2.1. Die Funktionsweise einer VPC sowie die Kopplung mit einem SysteMoC-Aktor ist in Abb. 8.22 dargestellt. Auf der linken Seite in Abb. 8.22 sieht man die Aktivierungen des SysteMoC-Aktors a_0. Die Laufzeit der Simulation steigt dabei nach unten an. Man beachte, dass in Abb. 8.22 zwei Zeitskalen verwendet werden: Zum einen die Laufzeit der Simulation, zum anderen die simulierte Zeit. Es werden nacheinander die Funktionen f_0,f_1, f_0, f_3 und f_2 aktiviert. Die Berechnung der Funktionen benötigt jeweils den grau schraffierten Bereich an Simulationszeit. Man sieht, dass nach jeder Berechnung einer Funktion die VPC DSP aufgerufen wird und währenddessen die Ausführung des Aktors a_0 blockiert ist.

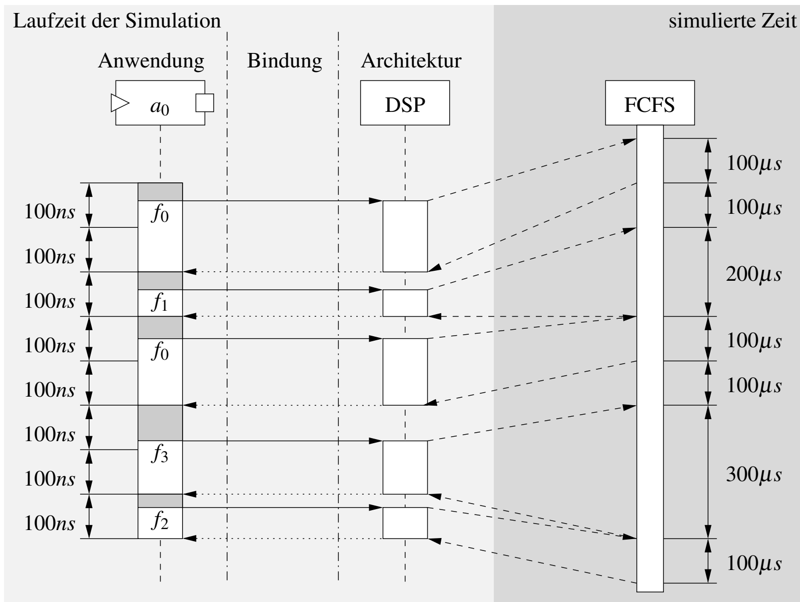


Abb. 8.22. Kombinierte Verhaltens- und Zeitsimulation

Die VPC gibt den Aufruf an einen assoziierten Scheduler mit FCFS-Algorithmus (engl. *First Come, First Served*) weiter. Der Scheduler kennt die geschätzten Ausführungszeiten des Aktors und simuliert die entsprechende Zeit durch eine `wait`-Anweisung in SystemC. Die simulierte Zeit wächst in Abb. 8.22 ebenfalls nach unten an. Erst danach gibt der Scheduler über die VPC die Kontrolle an den Aktor a_0 zurück. Hierdurch kann der Aktor seinen neuen Zustand einnehmen und die Daten von den Eingangsports konsumieren und auf den Ausgangsports produzieren. Anschließend kann der Aktor gegebenenfalls die nächste Funktion ausführen.

Man beachte, dass eine VPC so konfiguriert werden kann, dass jede Funktion eine andere Ausführungszeit erhält. Es ist sogar möglich, dass diese Ausführungszeit abhängig von dem Berechnungspfad der Funktion dynamisch gesetzt wird. Weiterhin

können die Verzögungszeiten von Wächterfunktionen (engl. *guards*), die als Bedingungen an Zustandsübergängen annotiert sind, ebenfalls mit Hilfe von VPCs simuliert werden. Um die oben beschriebene Ausführungssemantik aber nicht unnötig kompliziert zu machen, wird hier auf eine weitere Diskussion dieses Aspektes verzichtet.

Modellierung dynamischer Ablaufplanungsstrategien

Aufgrund der Nebenläufigkeit von Aktoren in einem SysteMoC-Modell kann es zu Ressourcenkonflikten kommen, sobald mehr als ein Aktor an die selbe Komponente gebunden ist. Um diese Ressourcenkonflikte aufzulösen, wird jede VPC mit einer Ablaufplanungsstrategie konfiguriert. Diese Strategien können präemptiv oder nichtpräemptiv sein. Die Ablaufplanungsstrategien liegen in der Bibliothek als C++-Klassen mit einheitlichem Interface vor. Als Ablaufplanungsstrategien können u. a. RR, FCFS oder prioritätsbasierte Verfahren zum Einsatz kommen.

Zur Realisierung präemptiver Ablaufplanungsstrategien definiert die VPC-Bibliothek ein Interface zum Scheduler. Der Scheduler implementiert die Ablaufplanungsstrategie und wird, wie Abb. 8.22 zeigt, von der assoziierten VPC aufgerufen. Die Schnittstelle zwischen VPC und Scheduler für eine präemptive Ablaufplanungsstrategie ist in Abb. 8.23 dargestellt. Die Ablaufplanungsstrategie ist eine präemptive prioritätsbasierte Ablaufplanung (PPrio). Man sieht, dass die Verzögerung des Aktors a_1 durch die Unterbrechung durch den Aktor a_0 vergrößert wird.

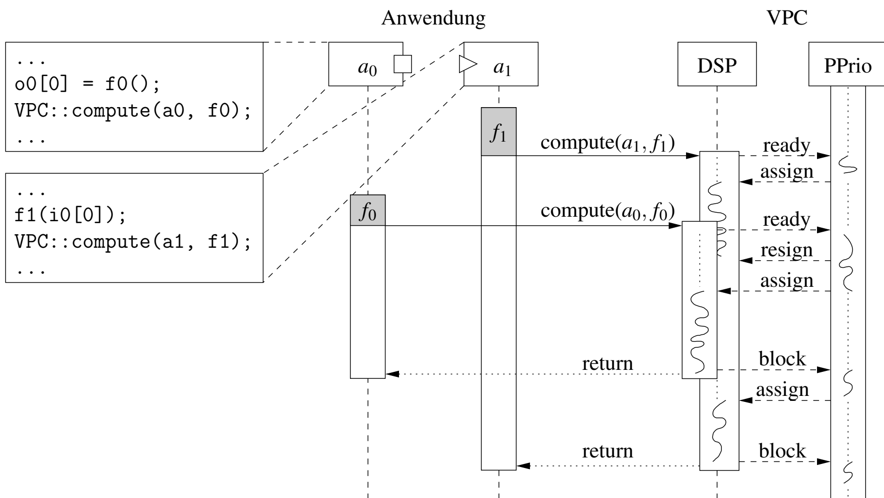


Abb. 8.23. Präemptive Ablaufplanung in der VPC-Bibliothek

In Abb. 8.23 wird zunächst Aktor a_1 gestartet, der die Funktion f_1 berechnet. Durch den Funktionsaufruf `VPC::compute(a1, f1)` erfolgt die Bindung des Aktors an die Komponente DSP. Die Angabe des Aktors (`a1`) und der ausgeführten

Funktion (`f1`) ermöglicht die Ermittlung der richtigen Ausführungszeit für die Kernfunktionalität der Funktion f_1 auf dem digitalen Signalprozessor entsprechend der Konfiguration. Die VPC-Bibliothek kümmert sich darum, dass entsprechend der Bindung des Aktors der Aufruf an die richtige VPC weitergeleitet wird. Die VPC DSP ihrerseits benachrichtigt den Scheduler mittels des ready-Signals. Der Scheduler PPrio teilt daraufhin die Ressource dem Aktor a_1 zu (Signal assign). Die VPC DSP simuliert nun mittels einer SystemC-`wait`-Anweisung die Ausführungszeit der Funktion f_1.

Die Simulation der Ausführungszeit wird allerdings durch die Aktivierung des Aktors a_0 unterbrochen. Dieser führt die Funktion f_0 aus und initiiert die Zeitsimulation mittels des Funktionsaufrufs `VPC::compute(a0, f0)`. Die Komponente DSP informiert den Scheduler PPrio mittels des ready-Signals, dass der Aktor a_0 nun ebenfalls zu planen ist. In diesem Beispiel wird davon ausgegangen, dass der Aktor a_0 eine höhere Priorität als der Aktor a_1 besitzt, weshalb es zur Unterbrechung der Zeitsimulation für Aktor a_1 kommt (Signal resign). Der Scheduler speichert dabei die noch verbleibende zu simulierende Ausführungszeit des Aktors a_1. Der Prozessor wird anschließend dem Aktor a_0 zugeteilt und die VPC DSP übergibt den Ausführungskontext mittels `wait`-Anweisung.

Nachdem die Ausführungszeit des Aktors a_0 vollständig simuliert ist, benachrichtigt die Komponente DSP den Scheduler PPrio mittels des block-Signals. Zur selben Zeit kehrt auch der Funktionsaufruf `VPC::compute(a0, f0)` zum Aktor a_0 zurück. Dieser nimmt nun die Aktualisierung des Systemzustands vor, d. h. er setzt den Folgezustand in dem endlichen Automaten des Aktors und produziert das Datum auf dem Kanal. Der Scheduler wiederum teilt dem Aktor a_1 die Komponente DSP zur weiteren Zeitsimulation zu (Signal assign). Dabei wird der VPC die verbleibende zu simulierende Ausführungszeit mitgeteilt. Ist auch diese abgelaufen, wird der Scheduler benachrichtigt (Signal block) und der Funktionsaufruf `VPC::compute(a1, f1)` kehrt zum Aktor a_1 zurück. Dieser nimmt daraufhin in seinem endlichen Automaten den Folgezustand ein und konsumiert das verarbeitete Datum vom Kanal.

Neben prioritätsbasierten Ablaufplanungsstrategien können ebenfalls zeitgetriebene Verfahren zum Einsatz kommen.

Beispiel 8.2.2. Abbildung 8.24a) zeigt ein SysteMoC-Modell bestehend aus zwei Aktoren und einem Kanal. Das SysteMoC-Modell ist auf eine Architektur bestehend aus einem DSP, einem Bus und einem Speicher abgebildet. In Abb. 8.24b) sind potentielle Ressourcenkonflikte für die beiden Aktoren auf dem DSP zu sehen. Diese Konflikte werden durch einen RR-Scheduler aufgelöst. Darin bedeutet W, der Aktor ist ausführungsbereit, die Ressource ist aber anderweitig belegt, und R bedeutet, die Ressource ist momentan dem Aktor zugeordnet. Weiterhin kann man erkennen, dass ein zusätzlicher Overhead für den Scheduler simuliert werden kann, der anfällt, wenn einem neuen Aktor eine Ressource zugeteilt wird.

Modellierung der Kommunikationslatenzen

Kommunikation zwischen SysteMoC-Aktoren ist auf dedizierte Punkt-zu-Punkt Kanäle mit FIFO-Semantik beschränkt (siehe Abschnitt 2.3.2). Diese SysteMoC-Kanäle

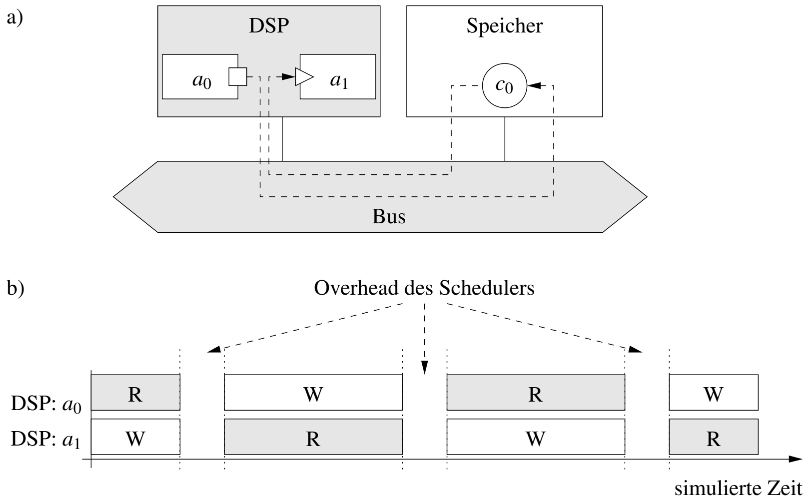


Abb. 8.24. a) Allokation und Bindung eines SysteMoC-Modells und b) Ressourcenkonflikt auf dem DSP

übernehmen dabei drei Aufgaben: 1) den Datentransport sowie 2) die Datenspeicherung und 3) die Synchronisation zwischen Aktoren. Die Implementierung eines SysteMoC-Kanals muss somit diese drei Aspekte unterstützen. Dies resultiert in einer Abbildung des Datenpuffers auf einen physikalischen Speicher sowie der Definition von Transaktionen für Speicherzugriffe und der Synchronisation. In heutigen Mehrprozessorsystemen werden die Transaktionen typischerweise über mehrere Ressourcen geroutet.

Beispiel 8.2.3. Eine generische Implementierung eines SysteMoC-Kanals noch obigem Schema ist in Abb. 8.25 dargestellt. Abbildung 8.25a) zeigt ein SysteMoC-Modell bestehend aus zwei Aktoren und einem Kanal. Die beiden Aktoren werden derart verfeinert, dass für jeden Lese- und Schreibport eine Schnittstelle bestehend aus drei Sockets entsteht. Ein Initiator-Socket ist dann für die Implementierung der Lese- bzw. Schreibtransaktionen zuständig. Der andere Initiator-Socket übernimmt die Benachrichtigung des adjazenten Aktors über erfolgte Lese- bzw. Schreibzugriffe. Der Target-Socket nimmt diese Benachrichtigungen vom anderen Aktor entgegen.

Die oben skizzierte Implementierung eines SysteMoC-Kanals gibt die Modellierung in der Zeitsimulation vor. Prinzipiell erfolgt die Zeitsimulation für die Kommunikation identisch zur Zeitsimulation von Berechnungen auf VPCs. Allerdings können bei der Kommunikation mehrere Ressourcen beteiligt sein, so dass ein `compute`-Aufruf für die Kommunikation von mehreren VPCs abgearbeitet werden muss.

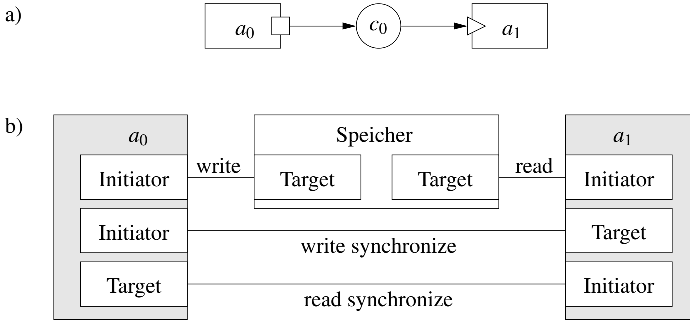


Abb. 8.25. Implementierung eines SysteMoC-Kanals in einem Mehrprozessorsystem

Beispiel 8.2.4. Für das SysteMoC-Modell aus Beispiel 8.2.2 mit der Implementierung des SysteMoC-Kanals aus Abb. 8.25 ergibt sich dann der Ablaufplan in Abb. 8.26. In diesem Beispiel, blockiert der DSP beim Schreiben der Daten von Aktor a_0. Der Bus und der Speicher werden sukzessive für andere Zugriffe gesperrt. Da dies in diesem Fall keine Zeit verbraucht, ist die Blockierung des Aktors a_0 in Abb. 8.26 nicht zu sehen. Nachdem die Route vom DSP zum Speicher aufgebaut ist, findet die write-Transaktion statt. Ist diese erfolgreich abgeschlossen, findet die Synchronisation mit dem Aktor a_0 statt. Da die Aktoren a_0 und a_1 auf die selbe Ressource gebunden sind, treten keine externen Ereignisse auf. Anschließend folgt die read-Transaktion des Aktors a_1. Dafür wird wiederum zunächst die Route aufgebaut und somit der DSP, der Bus und der Speicher blockiert. Nach dem Lesen der Daten kann Aktor a_1 seine Berechnung durchführen.

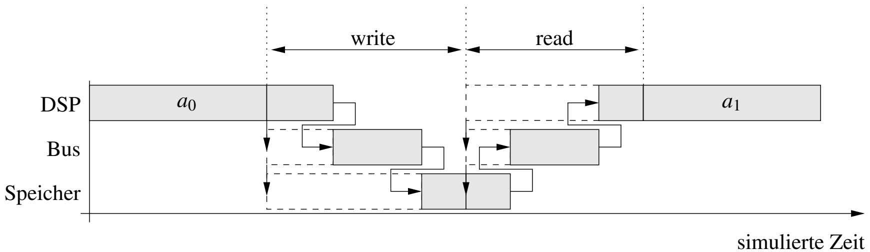


Abb. 8.26. Ablauf der Kommunikation zwischen zwei SysteMoC-Aktoren

Zeitanalyse mit der VPC-Bibliothek

Zur Zeitanalyse mittels der VPC-Bibliothek liegt das Verhaltensmodell als SysteMoC-Modell vor. Die Architektur- und Bindungsinformationen werden in einer

Konfigurationsdatei angegeben. Während der Elaborationsphase des SystemC-Modells konfiguriert sich die VPC-Bibliothek mittels der gegebenen Konfigurationsdatei, die ebenfalls die geschätzten Ausführungszeiten der Kernfunktionalität der Aktoren enthält. In dieser Elaborationsphase, legt die VPC-Bibliothek für jede allozierte Komponente eine VPC an und instantiiert die assoziierten Scheduler. Dabei werden auch die Informationen für die Ablaufplanung in einer Aktordatenstruktur zusammengestellt. Die Aktordatenstruktur enthält neben einer eindeutigen Identifikationsnummer, die Ausführungszeiten, gegebenenfalls Prioritäten, zugewiesene Zeitschlitze, Zeitbudgets etc. Diese Informationen werden während der Simulation zur Zeitbewertung verwendet.

Die kombinierte Verhaltens- und Zeitsimulation läuft dann wie oben beschrieben ab. Das Ergebnis ist ein Ausführungstrace der Simulation. Der Trace enthält alle Start- und Endzeitpunkte von Aktorausführungen. Diese können anschließend analysiert werden, um somit die erzielte Latenz bzw. den Durchsatz des Systems zu bewerten.

8.2.2 Kompositionale Zeitanalyse über Ereignisströme

Simulative Verfahren zur Zeitanalyse können aufgrund ihrer Unvollständigkeit im Allgemeinen nicht garantieren, dass sie den schlechtesten oder den besten Fall bei der Bewertung des Zeitverhaltens simulieren. Eine erschöpfende Simulation, die dies sicherstellen könnte, ist aufgrund der hohen Laufzeit meist nicht durchführbar. Formale Methoden zur Zeitanalyse hingegen sind vollständig und können für das Verifikationsziel des Beweises eingesetzt werden, da sie garantieren, dass es keinen besseren oder schlechteren Fall als durch die Analyse identifizierte Fälle im Zeitverhalten gibt.

Im Folgenden wird eine *kompositionale Zeitanalyse* vorgestellt. Diese verwendet Ergebnisse aus der Echtzeitanalyse für Einprozessorsysteme (siehe Abschnitt 7.4.2) wieder. Hierfür wird die Aktivierung von Prozessen auf einem Prozessor in Form eines sog. *Ereignisstroms* beschrieben, der das zeitliche Verhältnis zwischen zwei aufeinander folgender Aktivierungen eines Prozesses beschreibt. Da viele der Echtzeitanalysemethoden für Einprozessorsysteme in der Lage sind, viele unterschiedliche Ereignisströme zu analysieren, kann für jeden Prozess die minimale und maximale Antwortzeit bestimmt werden. Die minimale und maximale Antwortzeit wiederum kann als Ereignisstrom dargestellt und zur Aktivierung weiterer Prozesse (auch auf anderen Ressourcen) verwendet werden. Somit ergeben sich nach und nach für alle Prozesse im System die Aktivierungen und Antwortzeiten, die schließlich zur Bestimmung von Ende-zu-Ende-Latenzen herangezogen werden.

Ereignisströme

Die Kopplung der verschiedenen Methoden zur Echtzeitanalyse erfolgt über sog. *Ereignisströme*. Diese beschreiben in welchem zeitlichen Verhältnis Ereignisse zueinander stehen. Vier wichtige Klassen von Ereignisströmen unter Vernachlässigung der Offsets zwischen Ereignisströmen werden unterschieden:

1. *periodisch auftretende Ereignisse*: Die einfachste Form eines Ereignisstroms ist das periodische Auftreten eines Ereignisses mit einer Periode P. Unter Vernachlässigung des Offsets genügt der Parameter P, um den gesamten Ereignisstrom zu beschreiben.
2. *periodisch auftretende Ereignisse mit Jitter*: In der Implementierung von komplexen Systemen kann es immer wieder vorkommen, dass z. B. Ressourcen kurzzeitig nicht verfügbar sind. Hierdurch kann es zu leichten Verzögerungen bei der Erzeugung periodischer Ereignisse kommen. Diese Verzögerungen werden als *Jitter J* bezeichnet. Der Parameter J gibt an, um wie viel früher bzw. später nach der eigentlichen Periode ein Ereignis auftreten kann. Der Ereignisstrom kann dann durch die Parameter (P,J) vollständig beschrieben werden.
3. *periodisch auftretende Ereignisse mit Bursts*: Wird der Jitter größer als eine Periode P, so können innerhalb einer Periode mehr als zwei Ereignisse auftreten. Dies wird als *Burst* bezeichnet. In diesem Fall kommt der minimale zeitliche Abstand (d) zweier Ereignisse ins Spiel. Der Parameter d wird auch als minimale Ankunftszeit bezeichnet. Ein Ereignisstrom, der aus periodisch auftretenden Ereignissen mit Bursts besteht, kann mit den Parametern (P,J,d) vollständig beschrieben werden. Alternativ kann der Ereignisstrom auch mit den Parametern (P,d,b) beschrieben werden, wobei P die Periode, d die minimale Ankunftszeit und b die maximale Anzahl an Ereignissen in einer Periode P darstellen.
4. *sporadisch auftretende Ereignisse*: Bei sporadischen Ereignissen wird lediglich die minimale Ankunftszeit d zweier aufeinander folgender Ereignisse beschränkt.

Diese Klassen werden auch als *Ereignismodelle* bezeichnet. Die Ereignismodelle mit Ausnahme der Klasse sporadischer Ereignisse sind in Abb. 8.27 dargestellt. Darin bezeichnet j_0 den zeitlichen Offset des ersten Auftreten eines Ereignisses und j_{i-1} den Jitter des i-ten Ereignis.

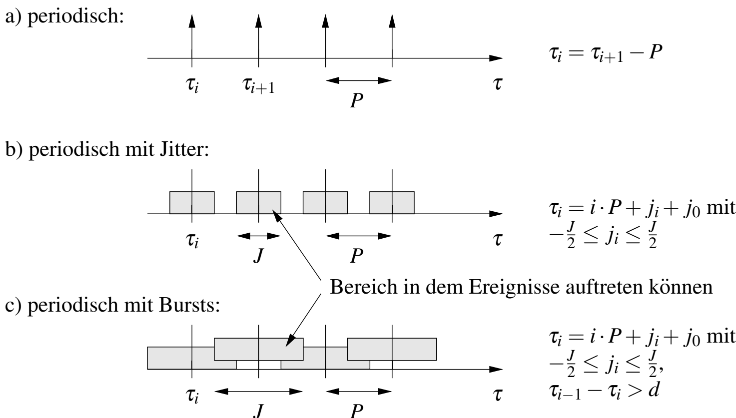


Abb. 8.27. Wichtige Ereignisströme [435]

Bei der Kopplung der Analysemethoden muss allerdings berücksichtigt werden, dass der errechnete *Ausgabeereignisstrom* eines Prozesses kompatibel zum erwarteten *Eingabeereignisstrom* der Zeitanalysemethode zur Aktivierung eines weiteren Prozesses auf der angeschlossenen Komponenten ist. Mit anderen Worten: Die lokale Echtzeitanalyse legt fest, welches Ereignismodell für die Aktivierung der zu analysierenden Prozesse erwartet wird und gibt das Analysemodell wiederum in Form eines Ereignismodells zurück. Dies ist in Abb. 8.28 dargestellt. Der Teil der Anwendung, der auf die gegebene Komponente abgebildet ist, besteht in diesem Fall aus drei Prozessen. Die Komponente selbst hat Einfluss auf die besten (engl. *Best Case Execution Time, BCET)* und schlechtesten Ausführungszeiten (engl. *Worst Case Execution Time, WCET*) der einzelnen Prozesse sowie die Strategie, mit der Ressourcenkonflikte bei der Ausführung von Prozessen gelöst werden (engl. *Ressource Sharing STrategy, RSST*). Die Echtzeitanalyse beschränkt die Eingabeschnittstelle der Komponente indem es ein Ereignismodell für die Eingabeereignisströme definiert. Mit der Anwendung, den Eingabeereignisströmen, den BCETs und WCETs sowie der RSST kann die Echtzeitanalyse die Ausgabeereignisströme bestimmen, welche über ein Ausgabe-Interface mit anderen Komponenten über eine entsprechende Eingabeschnittstelle verbunden werden.

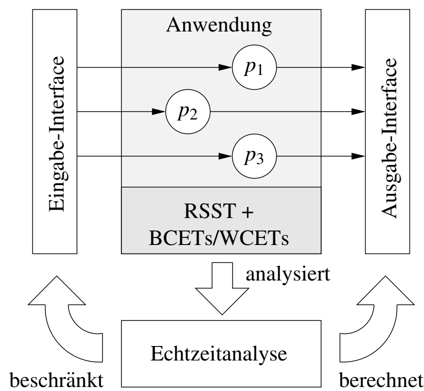


Abb. 8.28. Lokale komponentenbasierte Echtzeitanalyse zur kompositionalen Zeitanalyse

Ereignisstromkopplung

Zentral für die kompositionale Zeitanalyse ist somit die Kopplung von Ereignisströmen. Dies ist in Abb. 8.29a) zu sehen. Die Frage ist somit, ob der Ausgabeereignisstrom des Prozesses p_1 kompatibel zum erwarteten Eingabeereignisstrom von p_2 ist, also die Ereignismodelle kompatibel sind.

Um diese Frage zu beantworten, betrachten Richter und Ernst in [378] die vier oben eingeführten Ereignismodelle sowie die Möglichkeiten, die Parameter dieser

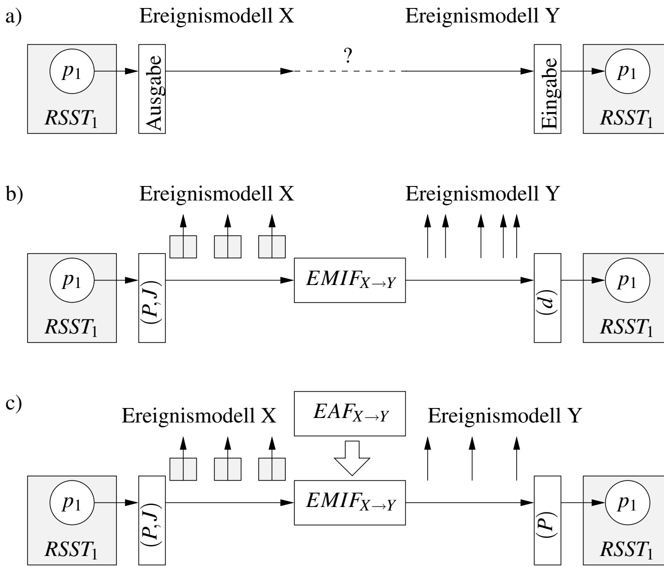


Abb. 8.29. a) Ereignisstromkopplung mit b) Ereignismodellschnittstelle und c) Ereignisadaptierung

Ereignismodelle ineinander umzurechnen. Dabei identifizieren sie drei Klassen von möglichen Umrechnungen:

1. *verlustfreie Umwandlung*: Bei dieser Klasse von Transformationen kann ein Ereignismodell X in ein anderes Ereignismodell Y transformiert werden, ohne dass Informationen, die in X gespeichert waren, verloren gehen. Ein Beispiel für eine verlustfreie Umrechnung ist von dem Ereignismodell „periodische Ereignisse" in das Ereignismodell „periodische Ereignisse mit Jitter", da bei der Umrechnung der Jitter einfach als null angenommen werden kann.
2. *verlustbehaftete Umwandlung ohne Adaptierung*: Jedes periodische Ereignismodell kann in ein sporadisches Ereignismodell umgerechnet werden. Der Grund hierfür liegt darin, dass die minimale Ankunftsrate aus den periodischen Ereignisströmen berechnet werden kann. Allerdings kann man nach der Transformation aus dem sporadischen Ereignismodell nicht mehr darauf schließen, dass es sich auch um ein periodisches Ereignismodell handelt, während solche Rückschlüsse bei den verlustfreien Umrechnungen noch möglich sind.
3. *verlustbehaftete Umwandlung mit Adaptierung*: Schließlich gibt es noch Umrechnungen, die eine Anpassung des Ereignisstrom erfordern. Ein Beispiel hierfür ist die Umwandlung „periodischer Ereignisse mit Jitter" in das Ereignismodell „periodische Ereignisse". Ist der Jitter bekannt, kann durch vorübergehende Speicherung von Ereignissen wieder ein periodischer Ereignisstrom erzeugt werden. Dabei können die Verzögerung eines Ereignisses und die benötigte Puf-

fergröße aus den Parametern des Ereignismodells bestimmt werden. Diese Umwandlung ist ebenfalls verlustbehaftet.

Die Umwandlung der Ereignisströme kann mittels sog. *Ereignismodellschnittstellen* (*EMIF*) erfolgen. Abbildung 8.29b) zeigt wie ein Ausgabeereignisstrom (P, J) mit Periode P und Jitter J mittels eines *EMIF* in einen sporadischen Eingabeereignisstrom (d) mit minimaler Ankunftszeit d umgewandelt wird. Eine Umwandlung mit Ereignisadaptierung ist in Abb. 8.29c) dargestellt. Zusätzlich zu dem EMIF wird eine sog. *Ereignisadaptierungsfunktion* (*EAF*) benötigt. Die möglichen Umwandlungen von Ereignisströmen sind noch einmal in Abb. 8.30 dargestellt.

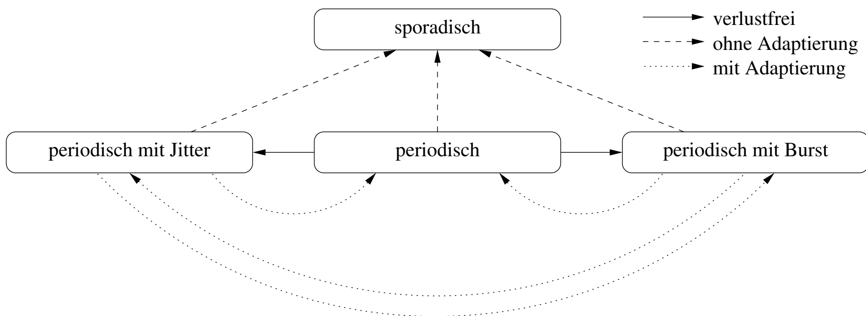


Abb. 8.30. Umwandlung von Ereignisströmen [378]

Die Bestimmung der Parameter für Transformationen ohne Adaptierung kann mittels der Vorschriften in Tabelle 8.1 erfolgen. Darin bezeichnet $X = (P_X)$ einen Ereignisstrom mit periodischen Ereignissen mit Periode P_X, $X = (P_X, J_X)$ einen periodischen Ereignisstrom mit Periode P_X und Jitter J_X, und $X = (P_X, d_X, b_X)$ einen periodischen Ereignisstrom mit Periode P_X, minimaler Ankunftszeit d_X und maximal b_X Ereignisse in einer Periode P. Der Ereignisstrom $X = (d_X)$ besteht nur aus sporadischen Ereignissen mit minimaler Ankunftszeit d_X.

Tabelle 8.1. EMIF für die Umwandlung von Ereignisströmen [378]

$EMIF_{X \to Y}$	$Y = (P_Y)$	$Y = (P_Y, J_Y)$	$Y = (P_Y, d_Y, b_Y)$	$Y = (d_Y)$
$X = (P_X)$	$P_Y := P_X$	$P_Y := P_X, J_Y := 0$	$P_Y := P_X, d_Y := P_X, b_Y := 1$	$d_Y := P_X$
$X = (P_X, J_X)$	–	$P_Y := P_X, J_Y := J_X$	–	$d_Y := P_X - J_X$
$X = (P_X, d_X, b_X)$	–	–	$P_Y := P_X, d_Y := d_X, b_Y := b_X$	$d_Y := d_X$
$X = (d_X)$	–	–	–	$d_Y := d_X$

Für die Umwandlung mit Adaptierung sind weitere Schritte notwendig. Beispielsweise lässt sich ein periodischer Ereignisstrom mit Jitter nicht durch einen rein

periodischen Ereignisstrom darstellen. Allerdings kann durch Pufferung und periodische Wiedergabe von Ereignissen eine rein periodischer Ereignisstrom aus einem periodischen Ereignisstrom mit Jitter konstruiert werden. Hierzu ist die Ermittlung der Ereignisadaptierungsfunktion (EAF) notwendig (siehe Abb. 8.29c)). Die Ermittlung der EAF erfolgt auf Basis des Ereignismodells für den Ausgabeereignisstrom und dem Ereignismodell für den Eingabeereignisstrom.

Für das Beispiel aus Abb. 8.29c) ($X = (P_X, J_X)$ und $Y = (P_Y)$) kann ein Puffer der Größe $n_{EAF} := 1$ verwendet werden und die Periode P_Y auf P_X gesetzt werden. Die Verzögerungszeit eines Ereignisses beträgt dann $d^+_{EAF} = P_X$ Zeiteinheiten. Etwas komplizierter ist der Fall für die Umwandlung eines periodischen Ereignisstroms mit Bursts ($X = (P_X, d_X, b_X)$) in einen rein periodischen Ereignisstrom ($Y = (P_Y)$). Mit dem Wissen, dass maximal b_X Ereignisse in einer Zeitspanne von P_X Zeiteinheiten auftreten können, kann P_Y zu $P_Y := \frac{P_X}{b_X}$ bestimmt werden. Die maximale Verzögerung eines Ereignisses beträgt:

$$d^+_{EAF} = P_X - (b_X - 1) \cdot d_X \tag{8.4}$$

Dies wird anhand eines Beispiels aus [378] verdeutlicht:

Beispiel 8.2.5. Gegeben ist das Ereignismodell eines periodischen Ausgabeereignisstroms mit Periode P_X und minimaler Ankunftszeit d_X. Die maximale Anzahl an Ereignissen in einer Periode P_X ist $b_X := 5$. Hieraus ergibt sich die Periode P_Y zu $P_Y = \frac{P_X}{5}$. Abbildung 8.31 zeigt die maximale Verzögerung eines Ereignisses als grauen Kasten. Diese ergibt sich für das letzte Ereignis eines Bursts.

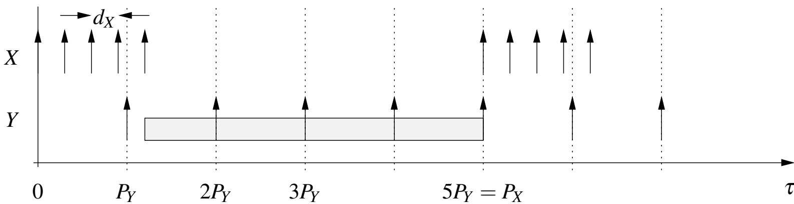


Abb. 8.31. Maximale Verzögerungszeit für ein Ereignis

Da die Ereignisse periodisch aus dem Puffer entnommen werden, lässt sich die maximale Anzahl n^+_{EAF} an Ereignissen in dem Puffer bestimmen zu:

$$\begin{aligned} n^+_{EAF} &= \left\lceil \frac{d^+_{EAF}}{P_Y} \right\rceil = \left\lceil \frac{P_X - (b_X - 1) \cdot d_X}{\frac{P_X}{b_X}} \right\rceil \\ &= b_X - \left\lceil b_x \cdot (b_x - 1) \cdot \frac{d_X}{P_X} \right\rceil \end{aligned} \tag{8.5}$$

Für die Umwandlung von Ereignisströmen mit Adaptierung aus Abb. 8.30 sind die Puffergrößen n_{EAF}, die Perioden P_{EAF} und die maximalen Verzögerungszeiten

d^{+}_{EAF} für die jeweiligen Ereignisadaptierungsfunktionen sowie die Ereignismodellschnittstelle in Tabelle 8.2 angegeben. Man erkennt, dass die Ereignisadaptierungsfunktion lediglich von dem Ausgabeereignisstrom X abhängt, was an der Verwendung periodischer Adpatierungsfunktionen liegt.

Tabelle 8.2. Ereignisadaptierungsfunktionen und Ereignismodellschnittstellen für die Umwandlung von Ereignisströmen mit Adaptierung nach Abb. 8.30 [378]

$X \rightarrow Y$	$EMIF_{X \rightarrow Y}$	P_{EAF}	n_{EAF}	d^{+}_{EAF}
$(P_X, J_X) \rightarrow (P_Y)$	$P_Y := P_{EAF}$	P_X	1	P_X
$(P_X, J_X) \rightarrow (P_Y, d_Y, b_Y)$	$P_Y := P_{EAF}, d_Y := P_Y, b_Y := 1$	P_X	1	P_X
$(P_Y, d_Y, b_Y) \rightarrow (P_Y)$	$P_Y := P_{EAF}$	$\frac{P_X}{b_X}$	Glg. (8.5)	Glg. (8.4)
$(P_Y, d_Y, b_Y) \rightarrow (P_Y, J_Y)$	$P_Y := P_{EAF}, J_Y := 0$	$\frac{P_X}{b_X}$	Glg. (8.5)	Glg. (8.4)

Analysemethode

Auf Basis der Kopplung von Ereignisströmen lassen sich nun Echtzeitanalyseansätze miteinander kombinieren. Dies wird anhand eines Beispiels aus [380] veranschaulicht.

Beispiel 8.2.6. Gegeben ist das System in Abb. 8.32. Das System besteht aus zwei Komponenten CPU1 und CPU2, welche die Analysedomänen festlegen. Jeder der beiden Prozessoren führt je zwei Prozesse aus, wobei die Aktivierungen der Prozesse p_1 und p_2 durch die Umgebung erfolgen. Der Prozess p_1 wird dabei mit einer Periode $P_{p_1,in} := 40ms$ aktiviert. Prozess p_2 wird mit einer Periode von $P_{p_2,in} := 20ms$ aktiviert, wobei die Aktivierung einem Jitter $J_{p_2,in} := 5ms$ unterliegt. Die Prozesse auf CPU1 werden mittels einer ratenmonotonen Ablaufplanung eingeplant. CPU2 verwendet ein Round-Robin-Ablaufplanungsverfahren, mit den Slotbreiten $S(p_3) := 5ms$ und $S(p_4) := 3ms$. Die Ausführungszeiten (BCET und WCET) der Prozesse sind gegeben zu: $\delta(p_1) := [15ms, 17ms]$, $\delta(p_2) := [8ms, 11ms]$, $\delta(p_3) := [10ms, 11ms]$ und $\delta(p_4) := [3ms, 5ms]$. Im Folgenden soll die Ende-zu-Ende-Latenz von der Aktivierung des Prozesses p_1 bis zur Beendigung des Prozesses p_3 bzw. von der Aktivierung des Prozesses p_2 bis zur Beendigung des Prozesses p_4 bestimmt werden.

Die Echtzeitanalysemethoden für CPU1 und CPU2 setzen voraus, dass CPU1 lediglich rein periodische Eingabeereignisströme erhält. Als Ausgabe werden periodische Ereignisströme mit Jitter erzeugt. CPU2 benötigt sporadische Ereignismodelle und erzeugt als Analyseergebnisse ebenfalls sporadische Ereignismodelle. Für die Analyse müssen zunächst die Ereignisströme gekoppelt werden. Hierzu wird am Eingang von CPU1 zum Prozess p_2 eine Ereignisadaptierungsfunktion (EAF) benötigt, da die Umgebung ein periodisches Ereignismodell mit Jitter zur Verfügung stellt, die Analysemethode jedoch ein rein periodisches Ereignismodell voraussetzt. Zur Kopplung der Ereignisströme zwischen den Prozessen p_1 und p_3 bzw. p_2 und p_4 können

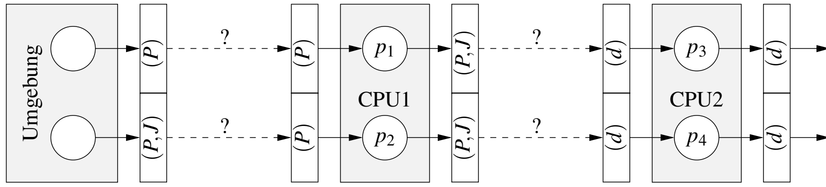


Abb. 8.32. Kopplung der Echtzeitanalysemethoden [380]

Ereignismodellschnittstellen (EMIFs) nach Tabelle 8.1 zum Einsatz kommen. Das Ergebnis ist in Abb. 8.33 zu sehen. Man sieht, dass die EAFs und EMIFs als weitere Komponenten im Analysemodell auftreten.

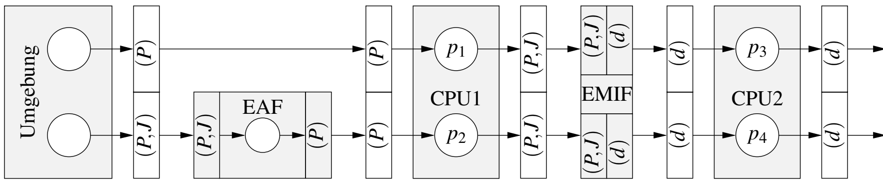


Abb. 8.33. Kopplung der Ereignisströme [380]

Für die EAF ergibt sich nach Tabelle 8.2, dass ein Puffer der Größe $n_{EAF} = 1$ verwendet und mit der Periode $P_{p_2,in}$ betrieben werden kann. Nun kann die CPU1 analysiert werden. Durch die Antwortzeitanalyse (siehe Gleichung (7.21) auf Seite 446) ergibt sich, dass die Antwortzeiten $\tau_{\mathrm{F}}(p_1)$ und $\tau_{\mathrm{F}}(p_2)$ der Prozesse p_1 bzw. p_2 wie folgt gegeben sind:

$$\tau_{\mathrm{F}}(p_1) = [31ms, 39ms]$$
$$\tau_{\mathrm{F}}(p_2) = [8ms, 11ms]$$

Diese Zeiten lassen sich wie folgt erklären. Aufgrund der geringeren Periode besitzt Prozess p_2 in der ratenmonotonen Ablaufplanung die höhere Priorität. Für die Zeitanalyse wird der Fall betrachtet, dass beide Prozesse die selbe Phase besitzen, also gleichzeitig aktiviert werden. Aufgrund der höheren Priorität wird p_2 zuerst ausgeführt. Die Antwortzeit ergibt sich gemäß der Ausführungszeit des Prozesses. Nach Beendigung der Ausführung von p_2 kann Prozess p_1 gestartet werden. Dieser wird allerdings durch die nächste Aktivierung von p_2 unterbrochen. Aus diesen Antwortzeiten lassen sich die Ausgabeereignisströme ableiten:

$$
\begin{aligned}
P_{p_1,out} &= 40ms \\
J_{p_1,out} &= \tau_F^+(p_1) - \tau_F^-(p_1) = 8ms \\
P_{p_2,out} &= 20ms \\
J_{p_2,out} &= \tau_F^+(p_2) - \tau_F^-(p_2) = 3ms
\end{aligned}
$$

Dabei bezeichnen τ_F^+ und τ_F^- die obere bzw. untere Schranke einer Antwortzeit. Diese Ausgabeereignisströme müssen nun an die erwarteten Eingabeereignisströme von CPU2 angepasst werden. Da in diesem Fall keine Adaptierung der Ereignisströme erfolgen muss, können die Umwandlungen aus Tabelle 8.1 direkt verwendet werden. Die Eingabeereignisströme ergeben sich dann zu:

$$
\begin{aligned}
d_{p_3,in} &= P_{p_1,out} - J_{p_1,out} = 32ms \\
d_{p_4,in} &= P_{p_2,out} - J_{p_2,out} = 17ms
\end{aligned}
$$

Die Echtzeitanalyse auf CPU2 führt zu folgenden Antwortzeiten [380] (siehe auch Gleichung (7.22) auf Seite 447):

$$
\begin{aligned}
\tau_F(p_3) &= [13ms, 20ms] \\
\tau_F(p_4) &= [3ms, 15ms]
\end{aligned}
$$

Der jeweils beste Fall ergibt sich, wenn der betreffende Prozess zuerst geplant wird und dieser die BCET zur Ausführung benötigt. Der schlechteste Fall ergibt sich, wenn jeweils der andere Prozess zuerst geplant wird und beide Prozesse die WCET zur Ausführung benötigen.

Abschließend können nun die gesuchten Ende-zu-Ende-Latenzen Λ bestimmt werden, indem entsprechend der Ereignisstromkopplung die Ergebnisse zusammengefasst werden. Es ergibt sich somit:

$$
\begin{aligned}
\Lambda(p_1 \rightarrow p_3) &= \tau_F(p_1) + \tau_F(p_3) = [44ms, 59ms] \\
\Lambda(p_2 \rightarrow p_4) &= \tau_F(p_2) + \tau_F(p_4) = [11ms, 26ms]
\end{aligned}
$$

Die kompositionale Zeitanalyse lässt sich somit durch folgende Schritte beschreiben:

1. Wähle eine Komponente als Analysedomäne, für die alle Eingabeereignisströme bekannt sind.
2. Wähle eine Echtzeitanalysemethode für diese Analysedomäne.
3. Falls notwendig, passe Eingabeereignisströme mittels EMIFs bzw. EAFs an.
4. Bestimme die Antwortzeiten der Prozesse in der Analysedomäne sowie die Ausgabeereignisströme.
5. Wiederhole Schritte 1 bis 5 bis alle Prozesse analysiert sind.

Behandlung zyklischer Abhängigkeiten

Bisher wurde lediglich die Analyse von Prozessketten beschrieben, wobei die Analyse keine zyklischen Abhängigkeiten enthielt. Im Allgemeinen kann die Analyse allerdings zyklische Abhängigkeiten enthalten, wie das folgende Beispiel aus [380] zeigt.

Beispiel 8.2.7. Abbildung 8.34 zeigt wiederum ein System mit zwei Prozessoren mit je zwei Prozessen. Beide Prozessoren verwenden eine präemptive prioritätsbasierte Ablaufplanung. Auf CPU1 hat Prozess p_1 die höhere Priorität, auf CPU2 hat Prozess p_4 die höhere Priorität.

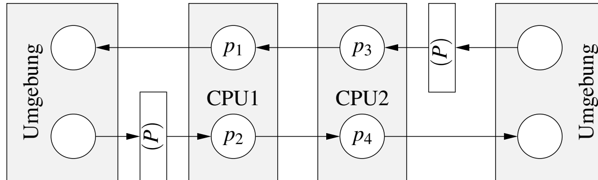


Abb. 8.34. Zyklische Abhängigkeiten in der Zeitanalyse [380]

Die zyklische Abhängigkeit ergibt sich folgendermaßen: Prozess p_2 aktiviert Prozess p_4, der wiederum Prozess p_3 in der Ausführung unterbrechen kann. Die Ausführung von Prozess p_3 aktiviert den Prozess p_1, der seinerseits Prozess p_2 unterbrechen kann.

Man beachte bei diesem Beispiel, dass die Anwendung selbst keine zyklischen Abhängigkeiten enthält. Diese ergeben sich erst durch die Ressourcenwiederverwendung, die in der Zeitanalyse berücksichtigt werden muss.

Die Zeitanalyse von Systemen mit zyklischen Abhängigkeiten erfolgt als Fixpunktberechnung, indem eine Iteration über die Berechnung von Ereignisströmen solange durchgeführt wird, bis sich kein Ereignisstrom im System mehr ändert.

8.2.3 Modulare Zeitanalyse mit RTC

Im Folgenden wird eine weitere formale Methode, die als *modulare Zeitanalyse* bezeichnet wird, vorgestellt. Diese basiert auf dem sog. *Echtzeitkalkül* (engl. *Real Time Calculus, RTC*), einer Erweiterung des *Netzwerkkalküls*. Zunächst wird die modulare Zeitanalyse für Netzwerkprozessoren vorgestellt. Anschließend wird die Anwendung der modularen Zeitanalyse für zyklische markierte Graphen präsentiert.

Das Echtzeitkalkül

Das Netzwerkkalkül ist eine formale Methode, um Zeiteigenschaften in Netzwerken zu analysieren. Dabei werden die Beschränkungen, die beim Transport der Pakete durch beschränkte Kapazitäten der Verbindungen und andere Kommunikationsströme auftreten, berücksichtigt. Die Analyse basiert dabei auf Faltungsopertation in Max-Plus-Algebra. Das Echtzeitkalkül erweitert das Netzwerkkalkül im Wesentlichen um Aspekte der dynamischen Ablaufplanung in Echtzeitsystemen. Mit dem Echtzeitkalkül lassen sich somit Echtzeitsysteme, die verteilt z. B. als Mehrprozessorsystem implementiert wurden, analysieren.

Beispiel 8.2.8. Gegeben ist der Netzwerkprozessor in Abb. 8.35, bestehend aus zwei Prozessorkernen (DSP, ARM), zwei Hardware-Beschleunigern (CheckSum, Cipher) und einem Bus. Die Prozessbindung und die Kommunikation zwischen Prozessen ist ebenfalls in Abb. 8.35 gegeben. Die Anwendung beschreibt die Schritte zur Entschlüsselung eines Datenstroms als Kette aus elf Aktoren.

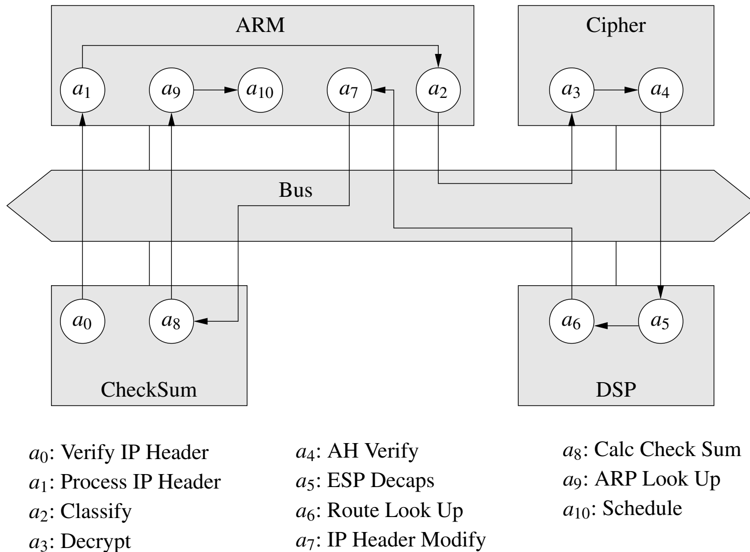


Abb. 8.35. Paketverarbeitung in einem Netzwerkprozessor [429]

Für die Zeitanalyse muss bekannt sein, wie viele Datenpakete jeder Prozess zu verarbeiten hat. Dabei wird von der tatsächlichen Anzahl abstrahiert und, mit Hilfe sog. *Ankunftskennlinien*, die minimale und maximale Anzahl an Aktivierungen eines Prozesses pro Zeitintervall Δ angeben, beschrieben.

Definition 8.2.1 (Ankunftskennlinie). *Eine* untere *bzw.* obere Ankunftskennlinie $\overline{\alpha}^l$ *und* $\overline{\alpha}^u$ *bilden ein positives Zeitintervall* $\Delta \in \mathbb{T}$ *auf die minimale bzw. maximale Anzahl an Aktivierungen eines Prozesses im Zeitintervall* Δ *ab, d. h.* $\overline{\alpha}^{l,u} : \mathbb{T} \rightarrow \mathbb{R}_{\geq 0}$.

In Definition 8.2.1 beschreibt $\mathbb{T}$ alle möglichen Zeitpunkte. Im Folgenden wird angenommen, dass $\mathbb{T} := \mathbb{R}_{\geq 0}$ ist. Die Schranken der Aktivierungen werden als kontinuierliche Größe dargestellt. Der Fall, dass die Aktivierungen nur als diskrete Größen auftreten, ist darin enthalten.

Die Ankunftskennlinien können beispielsweise aus Simulationstraces gewonnen werden, indem ein Fenster der Größe Δ über dem Trace verschoben wird, und die minimale und maximale Anzahl an Aktivierungen eines Prozesses in diesem Zeitfenster auf dem Trace bestimmt wird. Manchmal lassen sich die Ankunftskennlinien auch

aus der Spezifikation konstruieren. Treten beispielsweise Aktivierungen mit einem bekannten Muster auf, so kann dieses Muster in die minimale und maximale Ankunftskennlinie transformiert werden. Ein Beispiel für solche Spezifikationen sind die im vorherigen Abschnitt eingeführten Ereignisströme. Dabei wird jedes Ereignis als Aktivierung eines Prozesses interpretiert. Für die in Abb. 8.27 auf Seite 500 eingeführten Ereignisströme sind die Ankunftskennlinien in Abb. 8.36 dargestellt. Die unteren Ankunftskennlinien sind gestrichelt gezeichnet.

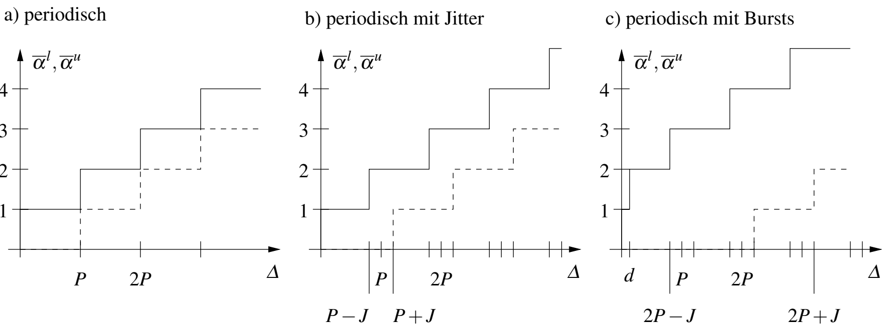


Abb. 8.36. Ankunftskennlinien für die Ereignisströme aus Abb. 8.27 [435]

Neben der Modellierung der Prozesse müssen ebenfalls die Ressourcen in der Architektur modelliert werden. Hierzu zählen die Prozessoren und die Hardware-Beschleuniger, aber auch die Busse. Die Möglichkeiten Berechnungen oder Kommunikationen durchzuführen, werden im Echtzeitkalkül mittels sog. *Servicekennlinien* beschrieben.

Definition 8.2.2 (Servicekennlinie). *Eine* untere *bzw.* obere Servicekennlinie β^l *und* β^u *bilden ein positives Zeitintervall* $\Delta \in \mathbb{T}$ *auf die minimal bzw. maximal verfügbare Rechenzeit einer Ressourcen in jedem Zeitintervall der Größe* Δ *ab. Es gilt* $\beta^l(0) = \beta^u(0) := 0$ *und*

$$\forall \tau > 0, \Delta > 0 : \beta^l(\Delta) \leq c(\tau + \Delta) - c(\tau) \leq \beta^u(\Delta),$$

wobei $c(\tau)$ *die bis zum Zeitpunkt* τ *kumulative verfügbare Rechenzeit darstellt.*

Dies bedeutet, dass jede Servicefunktion, die innerhalb der unteren β^l und oberen Servicekennlinie β^u liegt, durch β^l und β^u charakterisiert werden kann. Abbildung 8.37 zeigt die Servicekennlinien einer TDMA-Ablaufplanung. Die untere Servicekennlinie ist dabei gestrichelt gezeichnet.

Durch Ausführung eines Prozesses wird die Servicekennlinie verändert, da nach Ausführung des Prozesses weniger Rechenkapazität zur Verfügung steht als vorher. Um die Änderung allerdings bestimmen zu können, müssen die Ankunftskennlinien modifiziert werden. Dies ist notwendig, da die Ankunftskennlinien $\overline{\alpha}^l$ und $\overline{\alpha}^u$ die

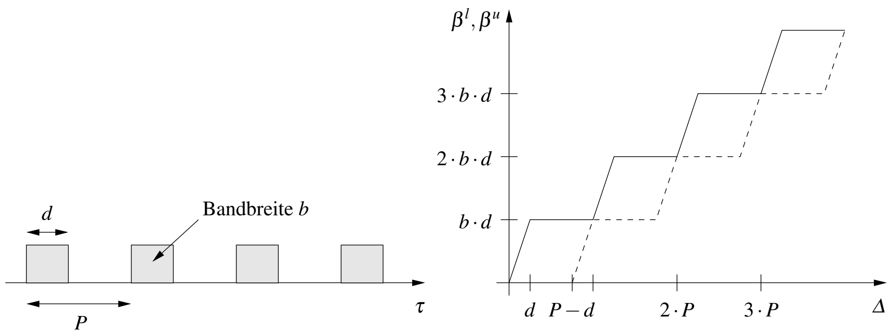


Abb. 8.37. Servicekennlinie einer TDMA-Ablaufplanung [435]

minimalen und maximalen Aktivierungen eines Prozesses angeben. Diese müssen zunächst in eine Rechenlast umgerechnet werden.

Sei w die Rechenlast pro Aktivierung eines Prozesses, so können die Schranken an die Rechenlast, verursacht durch diesen Prozess, berechnet werden. Diese Schranken werden wiederum als Ankunftskennlinien beschrieben. Sei $[\overline{\alpha}^l, \overline{\alpha}^u]$ ein Intervall, das die Anzahl der Aktivierungen eines Prozesses beschränkt. Dann kann die Rechenlast durch diesen Prozess mit Ausführungszeit w wie folgt beschränkt werden:

$$[\alpha^l, \alpha^u] := [w \cdot \overline{\alpha}^l, w \cdot \overline{\alpha}^u] \tag{8.6}$$

Mit den Ankunftskennlinien α^l und α^u für einen Prozess sowie den Servicekennlinien β^l und β^u einer Ressource lassen sich die Schranke $\beta^{l'}$ und $\beta^{u'}$ für die verbleibende Rechenzeit der Ressource bestimmen, wenn der Prozess auf der Ressource ausgeführt wird [430]:

$$\beta^{l'}(\Delta) := \sup_{0 \le \lambda \le \Delta} \{\beta^l(\lambda) - \alpha^u(\Delta)\} \tag{8.7}$$

$$\beta^{u'}(\Delta) := \sup_{0 \le \lambda \le \Delta} \{\beta^u(\lambda) - \alpha^l(\Delta)\} \tag{8.8}$$

Das Supremum (auch obere Schranke) bezeichnet das kleinste Element, das größer oder größer gleich allen Elementen der gegeben Menge ist.

Weiterhin können die Ankunftskennlinie $\alpha^{l'}$ und $\alpha^{u'}$ für die Prozessausführung auf einer Ressource mit Servicekennlinien β^l und β^u bei Aktivierung des Prozesses mit Ankunftskennlinien α^l und α^u bestimmt werden [430]:

$$\alpha^{l'}(\Delta) := \inf_{0 \le \lambda \le \Delta} \{\alpha^l(\lambda) + \beta^l(\Delta - \lambda)\} \tag{8.9}$$

$$\alpha^{u'}(\Delta) := \inf_{0 \le \lambda \le \Delta} \{\sup_{\nu \ge 0}\{\alpha^u(\lambda + \nu) - \beta^l(\nu)\} + \beta^u(\Delta - \lambda), \beta^u(\Delta)\} \tag{8.10}$$

Das Infimum (auch untere Grenze) ist das größte Element, dass kleiner oder kleiner gleich allen Elementen der gegeben Menge ist.

Um aus den Ankunftskennlinien für die Prozessausführung wieder in Ankunftskennlinien für die Aktivierung eines datenabhängigen Prozesses umzuwandeln, kann für jedes Intervall $[\alpha^l, \alpha^u]$ ein Intervall der folgenden Form berechnet werden:

$$[\overline{\alpha}^{l'}, \overline{\alpha}^{u'}] := \left[\left\lceil \frac{\alpha^{l'}}{w} \right\rceil, \left\lceil \frac{\alpha^{u'}}{w} \right\rceil \right] \tag{8.11}$$

In Abb. 8.38 ist die Transformation der Ankunfts- und Servicekennlinien graphisch als Blockdiagramm dargestellt. Durch Verschalten dieser Blöcke mit weiteren Blöcken können, z. B. durch die Ankunftskennlinien, weitere Prozesse aktiviert bzw. die verbleibende Rechenleistung einer Ressource für andere Prozesse verwendet werden. Durch die Verschaltung entsteht ein sog. *Zeitbewertungsnetzwerk*, welches zur Bestimmung der erreichbaren Latenzen verwendet werden kann.

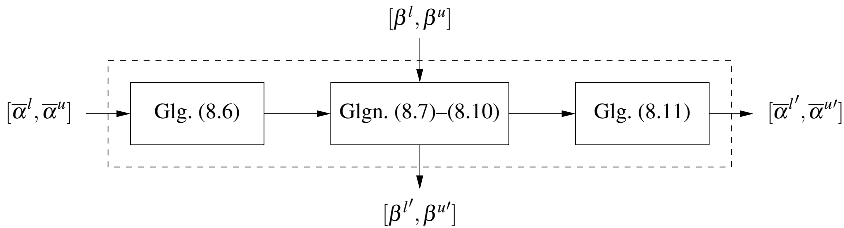


Abb. 8.38. Transformation der Ankunfts- und Servicekennlinien [429]

Beispiel 8.2.9. Zur Analyse des Zeitverhaltens zweier Prozesse auf einem einzelnen Prozessor, die mittels statischer Prioritäten präemptiv und dynamisch geplant werden, kann das in Abb. 8.39 dargestellte Zeitbewertungsnetzwerk verwendet werden. Die verfügbare Rechenzeit des Prozessors im Idle-Zustand ist durch die Servicekennlinien β^l und β^u beschränkt. Die Ankunftskennlinien $\overline{\alpha}_0^l$ und $\overline{\alpha}_0^u$ sowie $\overline{\alpha}_1^l$ und $\overline{\alpha}_1^u$ beschreiben die Aktivierungen der beiden Prozesse. Prozess 0 hat die höhere Priorität. Man sieht, dass die verbleibende Rechenzeit, beschränkt durch $\beta^{l'}$ und $\beta^{u'}$, dem Prozess 1 zur Verfügung gestellt wird.

Durch die Verschaltung der einzelnen Blöcke kann das Zeitbewertungsnetzwerk Stück für Stück aus Teilsystemen (Modulen) zusammengesetzt werden, weshalb diese Analysemethode als *modulare Zeitanalyse* bezeichnet wird.

Bestimmung der maximalen Latenzen und des maximalen Backlogs

Mit einem Zeitbewertungsnetzwerk ist es nun möglich, die Latenz, aber auch den Backlog in Netzwerkprozessoren zu beschränken. Bei der Latenz handelt es sich um die maximale Ende-Zu-Ende-Latenz eines Datenpakets, dass durch den Netzwerkprozessor verarbeitet wird. Der Backlog gibt den maximalen Speicherbedarf eines

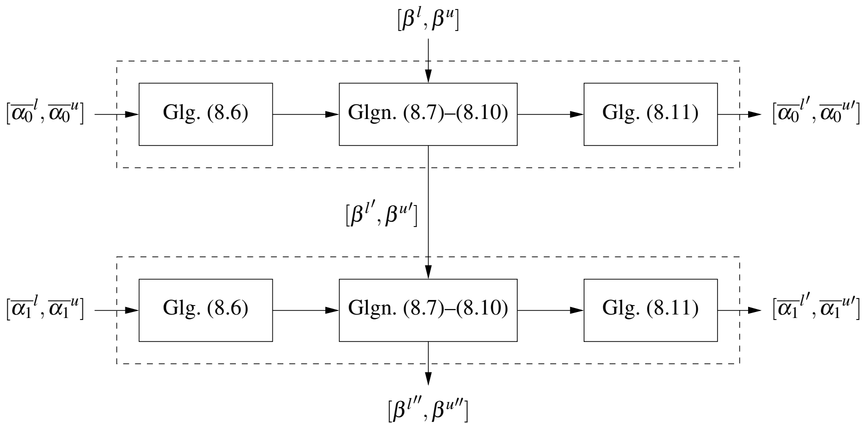


Abb. 8.39. Präemptive Ablaufplanung mit statischen Prioritäten [429]

Datenstroms im Netzwerkprozessor an. Mit Ergebnissen des Netzwerkkalküls lassen sich die Latenz Λ und der Backlog Ω wie folgt beschränken:

$$\Lambda := \sup_{\lambda \geq 0}\{\inf\{\tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau)\}\} \tag{8.12}$$

$$\Omega := \sup_{\lambda \geq 0}\{\alpha^u(\lambda) - \beta^l(\lambda)\} \tag{8.13}$$

Somit ist die maximale Latenz durch den maximalen horizontalen Abstand, und der maximale Backlog durch den maximalen vertikalen Abstand zwischen α^u und β^l beschränkt.

Zur Bestimmung der Werte durch Gleichung (8.12) und (8.13) muss zunächst geklärt werden, welches α^u und welches β^l zu verwenden ist. Für die Ankunftskennlinie ist der Fall klar: α^u ist die obere Ankunftskennlinie für den ersten Prozess in der Verarbeitungskette, da diese Aktivierung durch die Umgebung begrenzt ist, d. h. durch die Anzahl der zu verarbeitenden Pakete. Bei der Rechenzeitbeschränkung handelt es sich um die kumulierte Rechenzeitbeschränkung entlang der Kette von Prozessen, die ein Datenpaket bearbeiten. Seien $\beta_0^l, \beta_1^l, \ldots, \beta_{m-1}^l$ die unteren Servicekennlinien der Ressourcen entlang des Verarbeitungspfads, dann lässt sich die kumulierte Servicekennlinie β^l durch sukzessive Faltung berechnen:

$$\begin{aligned} \overline{\beta}_0^l &:= \beta_0^l \\ \overline{\beta}_{i+1}^l &:= \inf_{0 \leq \lambda \leq \Delta}\{\overline{\beta}_i^l + \beta_{i+1}^l(\Delta - \lambda)\} \; \forall i \geq 1 \\ \beta^l &:= \overline{\beta}_m^l \end{aligned} \tag{8.14}$$

Beispiel 8.2.10. Das Zeitbewertungsnetzwerk für die Entschlüsselung eines Datenstroms auf dem Netzwerkprozessor in Abb. 8.35 auf Seite 509 ist in Abb. 8.40 zu

sehen. Die runden Knoten bezeichnen dabei Quellen bzw. Senken von Ankunfts- und Servicekennlinien, d. h. sie modellieren die Umgebung des Systems. So wird beispielsweise die Ankunftsrate von Paketen beschränkt. Die Rechtecke bezeichnen die Transformation der Ankunfts- und Servicekennlinien entsprechend Abb. 8.38. Man beachte, dass im Echtzeitkalkül Kommunikation genauso wie Berechnungen auf Prozessoren behandelt wird.

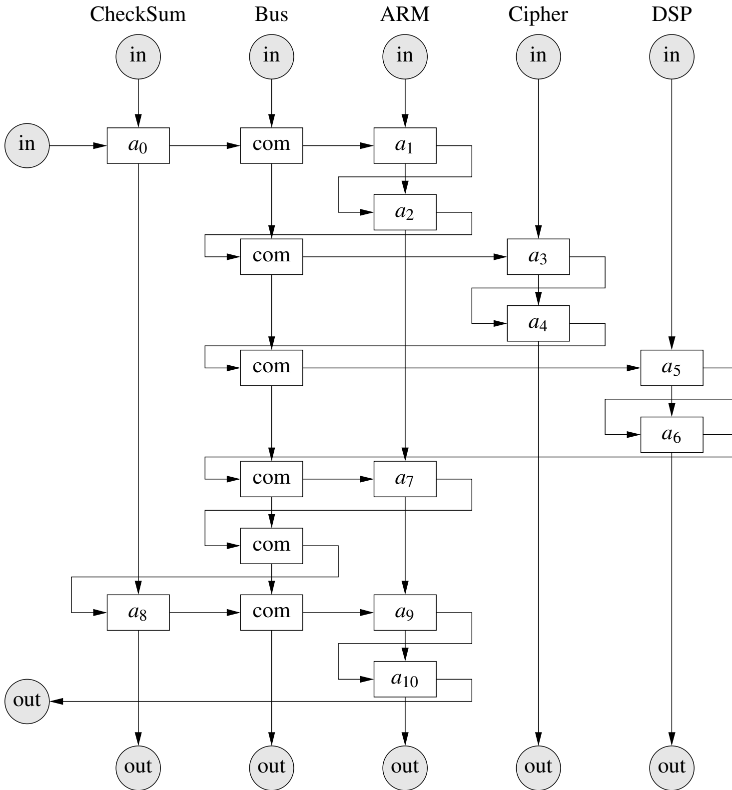


Abb. 8.40. Zeitbewertungsnetzwerk für den Netzwerkprozessor aus Abb. 8.35 [429]

Modulare Zeitanalyse zyklischer markierter Graphen

In Abschnitt 6.5.2 wurde eine formale Methode zur Zeitanalyse von latenzinsensitiven Systemen vorgestellt. Das zentrale Ergebnis war, dass sich solche Systeme als markierte Graphen modellieren lassen (siehe Abb. 6.82 auf Seite 356). Da Implementierungen mit beschränktem Speicher auskommen müssen, wurden FIFO-Kanäle mit beschränkter Kapazität durch zusätzliche rückwärts gerichtete Kanten

mit Anfangsmarkierung modelliert. Dadurch wurden Ressourcenbeschränkungen für die Kanäle berücksichtigt. Ressourcenbeschränkungen für die Aktoren wurden durch Selbstschleifen mit einer einzelnen Anfangsmarke modelliert, wodurch ein Aktor stets erst nach Beendigung einer Berechnung die nächste Berechnung starten kann. Aktoren sind in diesem Modell dediziert gebunden, d. h. jeder Aktor entspricht einer Hardware-Komponente. In diesem Abschnitt wird beschrieben, wie die modulare Zeitanalyse für Systeme, bei denen mehrere Aktoren auf die selben Hardware-Komponente gebunden sein können, angewendet werden kann.

Modellierung markierter Graphen

Gegeben sei ein markierter Graph $G = (A, C, M_0)$ mit Aktoren $a \in A$, Kanälen $c \in C \subseteq A \times A$ und der Anfangsmarkierung $M_0 : C \to \mathbb{R}_{\geq 0}$. Man beachte, dass die Markierung eines Kanals kontinuierlich sein kann. Dies schließt selbstverständlich den Fall ein, dass Marken diskret sein können. Ein Aktor verarbeitet einen Strom aus Marken, der als *Ereignisstrom* modelliert wird.

Die Anzahl zu verarbeitender Marken, die in einem Kanal (v_i, v_j) gespeichert wurden, wird als sog. *Ankunftsfunktion* $r_{ij}(\tau) : \mathbb{T} \to \mathbb{R}_{\geq 0}$ modelliert. Diese weist jedem Zeitpunkt $\tau \in \mathbb{T}$ die Anzahl an Marken zu, die im Zeitintervall $[0, \tau)$ an dem Kanal (v_i, v_j) angekommen sind. Dies entspricht der Anzahl produzierter informativer Daten eines Aktors in einem latenzinsensitiven System (siehe Gleichung (6.26)). Die Anfangsmarkierung des Kanals ergibt sich zu $r_{ij}(0) := M_0(v_i, v_j)$.

Für einen Aktor $a_i \in A$ lassen sich die Ankunftsfunktionen der Eingangskanäle zu einem Vektor r_i^{in} zusammenfassen: $r_i^{in} := (r_{ji} \mid (v_j, v_i) \in C)$. Analog lassen sich die Ankunftsfunktionen der Ausgangskanäle in einem Vektor r_i^{out} zusammenfassen: $r_i^{out} := (r_{ik} \mid (v_i, v_k) \in C)$. Man sagt, für zwei Vektoren r_i^{in} und $\tilde{r}_i^{in}$ von Ankunftsfunktionen, dass $r_i^{in} \geq \tilde{r}_i^{in}$ ist, falls $\forall (a_j, a_i) \in A, \tau \in \mathbb{T}$ gilt, dass $r_{ij}^{in}(\tau) \geq \tilde{r}_{ij}^{in}(\tau)$ ist.

Das zeitliche Verhalten eines Aktors a_i lässt sich als Abbildung der Ankunftsfunktionen der Eingangskanäle r_i^{in} auf die Ankunftsfunktionen der Ausgangskanäle r_i^{out} beschreiben:

$$r_i^{out} : \pi_i \circ r_i^{in} \tag{8.15}$$

π_i wird als *Transferfunktion* des Aktors a_i bezeichnet. Da Aktoren in markierten Graphen deterministisches Verhalten besitzen, beschreibt π_i eine deterministische Abbildung. Weiterhin besitzen markierte Graphen monotones Verhalten, d. h., falls $r_i^{in} \geq \tilde{r}_i^{in}$ gilt, dass $\pi_i \circ r_i^{in} \geq \pi_i \circ \tilde{r}_i^{in}$.

Ohne Ressourcenbeschränkungen bei selbstplanender Ausführung der Aktoren gilt, dass die Anzahl der produzierten Marken eines Aktors a_i auf jedem Ausgangskanal gleich dem Minimum der auf seinen Eingangskanälen verarbeitbaren Marken ist (siehe auch Gleichung (6.28) auf Seite 354):

$$\forall (v_i, v_k) \in C : r_{ik}^{out}(\tau) := \min_{(v_j, v_i) \in C} \{ r_{ji}^{in}(\tau) \} \tag{8.16}$$

Bei der selbstplanenden Ausführung von Aktoren feuern diese, sobald genügend Marken auf den Eingangskanälen verfügbar sind.

Die Ankunftsfunktionen der Ausgangskanäle eines Aktors a_i stellen für andere Aktoren oder a_i selbst Ankunftsfunktionen für Eingangskanäle dar. Aufgrund der FIFO-Semantik der Kanäle in markierten Graphen müssen die Ankunftsfunktionen um die Anfangsmarkierung verschoben werden. Dies kann durch die Sprungfunktion $\rho_{ji}(\tau)$ für einen Kanal (v_j, v_i) dargestellt werden:

$$\rho_{ji}(\tau) := \begin{cases} 0 & \text{falls } \tau \leq 0 \\ M_0(v_j, v_i) & \text{falls } \tau > 0 \end{cases}$$

Damit kann der Zusammenhang zwischen der Ankunftsfunktion $r_{ji}^{out}(\tau)$ auf einem Ausgangskanal eines Aktors a_j und der Ankunftsfunktion $r_{ji}^{in}(\tau)$ für den selben Kanal beim lesenden Aktor a_i hergestellt werden:

$$r_{ji}^{in}(\tau) := r_{ji}^{out}(\tau) + \rho_{ji}(\tau) \tag{8.17}$$

Gleichungen (8.16) und (8.17) lassen sich zu einem Gleichungssystem zusammenfassen (siehe auch Gleichung (6.30) auf Seite 354):

$$r := \Pi \circ r \tag{8.18}$$

Dabei enthält r die Ankunftsfunktionen an den Ausgabekanälen $(v_j, v_i) \in C$, d. h. $r := (r_{ji}^{out} \mid (v_j, v_i) \in C)$, und Π enthält die Abbildungsfunktionen aller Aktoren. Diese Gleichung besitzt einen eindeutigen Fixpunkt, setzt man δ-Kausalität voraus, d. h. dass Marken frühestens einen kurzen Zeitschritt $\delta > 0$, nachdem sie produziert worden sind, einen folgenden Aktor aktivieren können. Liegen keine Ressourcenbeschränkungen vor, kann der maximale Durchsatz des markierten Graphen auf Basis von Max-Plus-Algebra bestimmt werden (siehe Abschnitt 6.5.2). Der Fixpunkt charakterisiert den Eigenwert der Matrix Π und damit den Durchsatz. Beschränkte Kanalkapazitäten können durch Rückkanten modelliert werden.

Modellierung von Ressourcenbeschränkungen

Um zu modellieren, dass die Ausführung eines Aktors a_i Ressourcenbeschränkungen unterliegt, muss die Abbildungsfunktion π_i angepasst werden [432]. Hierzu wird angenommen, dass der Aktor a_i auf einer Ressource ausgeführt wird, die durch eine sog. *Servicefunktion* $c(\tau)$ beschrieben wird. Dabei beschreibt $c(\tau) \in \mathbb{R}_{\geq 0}$ die Anzahl an Aktorfeuerungen, welche die Ressource in dem Zeitintervall $[0, \tau)$ ausführen kann, wobei $c(0) := 0$ ist.

Die Verwendung von Servicefunktionen erlaubt eine Modellierung beliebiger Ressourcenbeschränkungen und Ablaufplanungsstrategien. Das Kommunikationsverhalten eines Aktors a_i mit einem Aktor k bei selbstplanender Ausführung kann unter Berücksichtigung von Ressourcenbeschränkungen mit der folgenden Transferfunktion beschrieben werden:

$$r_{ik}^{out}(\tau) := \inf_{0 \leq \lambda \leq \tau} \{ \min_{(v_j, v_i) \in C} \{ r_{ji}^{in}(\lambda) \} + c(\tau) - c(\lambda) \} \tag{8.19}$$

Gleichung (8.19) kann dann wie folgt interpretiert werden [432]: In einem Intervall $[\lambda,\tau)$ können $c(\tau)-c(\lambda)$ Aktorfeuerungen ausgeführt werden, sofern genügend Marken am Eingang verfügbar sind, d. h. $r_{ik}^{out}(\tau) \leq r_{ik}^{out}(\lambda) + c(\tau) - c(\lambda)$. Weiterhin muss zu jedem Zeitpunkt gelten, dass $r_{ik}^{out}(\lambda) \leq \min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda)\}$ ist, da ein Aktor höchstens so viele Marken produzieren kann, wie auf einem Eingangskanal verfügbar waren. Ohne Ressourcenbeschränkung würde bei selbstplanender Ausführung gelten, dass beide Seiten der Ungleichung gleich sind. Zusammen gilt $r_{ik}^{out}(\tau) \leq \min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda)\} + c(\tau) - c(\lambda)$.

Nun gibt es einen Zeitpunkt λ^* wo zum letzten Mal galt, dass $r_{ik}^{out}(\lambda^*) = \min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda^*)\}$ ist. Dieser Zeitpunkt kann entweder $\lambda^* = 0$, $\lambda^* = \tau$ oder ein Zeitpunkt dazwischen sein. Nach diesem Zeitpunkt λ^* ist die Ressource permanent mit der Ausführung des Aktors a_i ausgelastet. Es gilt: $r_{ik}^{out}(\tau) = r_{ik}^{out}(\lambda^*) + c(\tau) - c(\lambda^*) = \min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda^*)\} + c(\tau) - c(\lambda^*)$. Genau dieses λ^* wird mit der Infimum-Operation bestimmt, die in der Transferfunktion in Gleichung (8.19) angegeben ist.

Beispiel 8.2.11. Gegeben ist der markierte Graph in Abb. 8.41 mit drei Aktoren. Jeder Aktor a_i wird auf einer Ressource mit der Servicefunktion $c_i(\tau)$ ausgeführt. Die Transferfunktionen für den markierten Graph ergeben sich unter Verwendung von Gleichung (8.19) und (8.17) dann zu:

$$
\begin{aligned}
r_{0,1}^{out}(\tau) &:= \inf_{0\leq\lambda\leq\tau}\{r_{1,0}^{in}(\lambda) + c_0(\tau) - c_0(\lambda)\} \\
&= \inf_{0\leq\lambda\leq\tau}\{r_{1,0}^{out}(\lambda) + \rho_{1,0}(\lambda) + c_0(\tau) - c_0(\lambda)\} \\
r_{1,2}^{out}(\tau) &:= \inf_{0\leq\lambda\leq\tau}\{\min\{r_{0,1}^{in}(\lambda), r_{2,1}^{in}(\lambda)\} + c_1(\tau) - c_1(\lambda)\} \\
&= \inf_{0\leq\lambda\leq\tau}\{\min\{r_{0,1}^{out}(\lambda), r_{2,1}^{out}(\lambda) + \rho_{2,1}(\lambda)\} + c_1(\tau) - c_1(\lambda)\} \\
r_{2,1}^{out}(\tau) &:= \inf_{0\leq\lambda\leq\tau}\{r_{1,2}^{in}(\lambda) + c_2(\tau) - c_2(\lambda)\} \\
&= \inf_{0\leq\lambda\leq\tau}\{r_{1,2}^{out}(\lambda) + c_2(\tau) - c_2(\lambda)\} \\
r_{1,0}^{out}(\tau) &:= \inf_{0\leq\lambda\leq\tau}\{\min\{r_{0,1}^{in}(\lambda), r_{2,1}^{in}(\lambda)\} + c_1(\tau) - c_1(\lambda)\} \\
&= \inf_{0\leq\lambda\leq\tau}\{\min\{r_{0,1}^{out}(\lambda), r_{2,1}^{out}(\lambda) + \rho_{2,1}(\lambda)\} + c_1(\tau) - c_1(\lambda)\}
\end{aligned}
$$

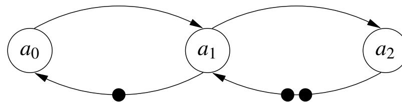


Abb. 8.41. Markierter Graph [432]

Die verbleibende Servicefunktion $c'(\tau)$ nachdem ein Aktor a_i ausgeführt wurde, ergibt sich zu:

$$c'(\tau) := c(\tau) - \min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\tau)\} \tag{8.20}$$

Abstraktion der Servicefunktion

Um die Zeitanalyse effizient durchführen zu können, müssen sowohl die Servicefunktionen als auch die Ankunftsfunktionen abstrahiert werden. Zunächst werden die Servicefunktionen durch untere und obere Schranken begrenzt. Diese Schranken werden als *Servicekennlinien* (siehe Definition 8.2.2 auf Seite 510) bezeichnet. Dabei wird von dem Zeitbereich in den Zeitintervallbereich gewechselt.

Unter Verwendung von Servicekennlinien β_i^l und β_i^u für den Aktor a_i kann Gleichung (8.19) wie folgt angenähert werden:

$$r_{ik}^{out}(\tau) \le \inf_{0\le\lambda\le\tau}\{\min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda)\} + \beta_i^u(\tau-\lambda)\}$$
$$r_{ik}^{out}(\tau) \ge \inf_{0\le\lambda\le\tau}\{\min_{(v_j,v_i)\in C}\{r_{ji}^{in}(\lambda)\} + \beta_i^l(\tau-\lambda)\}$$

Unter Verwendung des Operators $a \oplus b := \min\{a,b\}$ und des Operators $a(\tau) \otimes b(\tau) := \inf_{0\le\lambda\le\tau}\{a(\lambda) - b(\tau-\lambda)\}$ erhält man:

$$\left(\bigoplus_{(v_j,v_i)\in C} r_{ji}^{in}(\tau)\right) \otimes \beta_i^l(\tau) \le r_{ik}^{out}(\tau) \le \left(\bigoplus_{(v_j,v_i)\in C} r_{ji}^{in}(\tau)\right) \otimes \beta_i^u(\tau) \tag{8.21}$$

In [432] beweisen Thiele und Stoimenov: Wenn die Servicefunktion $c(\tau)$ durch die Servicekennlinien ersetzt wird, erhält man untere bzw. obere Schranken für die Ankunftsfunktion.

$$r_{ik}^{out}(\tau) \ge \beta_i^l(\tau) \oplus \left(\bigoplus_{(v_j,v_i)\in C} (r_{ji}^{out}(\tau) \otimes (\rho_{ji}(\tau) + \beta_i^l(\tau)))\right)$$
$$r_{ik}^{out}(\tau) \le \beta_i^u(\tau) \oplus \left(\bigoplus_{(v_j,v_i)\in C} (r_{ji}^{out}(\tau) \otimes (\rho_{ji}(\tau) + \beta_i^u(\tau)))\right)$$

Unter Verwendung der Matrixmultiplikation $C = A \otimes B$ mit $c_{ij} := \bigoplus_k (a_{ik} \otimes b_{kj})$ können diese Ungleichungen kompakt als

$$r^l := \beta^l \oplus S^l \otimes r^l$$
$$r^u := \beta^u \oplus S^u \otimes r^u$$

mit

$$s_{ij}^{l,u} := \begin{cases} \beta_i^{l,u} + M_0(v_j,v_i) & \text{falls } (v_j,v_i)\in C \\ \infty & \text{sonst} \end{cases}$$

dargestellt werden. Dabei stellt S^l bzw. S^u die Systemmatrix, β^l bzw. β^u den Vektor der unteren bzw. oberen Servicekennlinien und r^l bzw. r^u den Vektor der unteren bzw. oberen Schranken an die Ankunftsfunktionen dar.

Für die Systemmatrizen S^l und S^u kann der Abschluss bestimmt werden, der wie folgt definiert ist:

$$S^* := \bigoplus_{k:=0}^{\infty} S^{(k)} \text{ mit } S^{(k)} := S \otimes S^{(k-1)}$$

Damit lassen sich schließlich die unteren und oberen Schranken der Ankunftsfunktion bestimmen [432]:

$$r^l := (S^l)^* \otimes \beta^l \tag{8.22}$$

$$r^u := (S^u)^* \otimes \beta^u \tag{8.23}$$

Maximale Latenzen in markierten Graphen

Um die maximalen Latenzen in einem markiertem Graphen mittels der modularen Zeitanalyse bestimmen zu können, werden zwei durch einen Pfad verbundene Aktoren des markierten Graphen ausgewählt: a_s sei der Quellaktor und a_d sei der Zielaktor. Es wird angenommen, dass der Quellaktor a_s mit der zusätzlichen Ankunftsfunktion $g_s(\tau)$ angeregt wird. Es soll dann die maximale Latenz einer Aktivierung des Quellaktors a_s bis zur Aktivierung des Zielaktors a_d bestimmt werden. Die zusätzliche Ankunftsfunktion kann dann als Diagonalmatrix G_s angegeben werden.

Beispiel 8.2.12. Gegeben ist der markierte Graph mit Eingangsankunftsfunktion am Aktor a_0 in Abb. 8.42. Die Matrix, welche die zusätzliche Ankunftsfunktion modelliert, lässt sich wie folgt beschreiben:

$$G_s := \begin{pmatrix} g_0 & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix}$$

Man sieht, dass auf der Diagonalen zusätzliche Eingangsfunktionen für Aktoren eingetragen werden können. Besitzt ein Aktor keinen Eingang, so ist der entsprechende Eintrag ∞. Die Nebendiagonalelemente erhalten den Wert null.

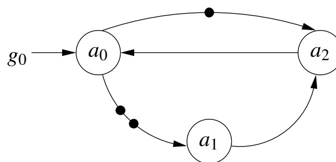


Abb. 8.42. Markierter Graph mit Eingangsankunftsfunktion

Mit der Matrix G_s lassen sich die Gleichungen (8.22) und (8.23) um den Eingang an Aktor a_s erweitern:

$$r^l := (S^l)^* \otimes G_s \otimes \beta^l \tag{8.24}$$

$$r^u := (S^u)^* \otimes G_s \otimes \beta^u \tag{8.25}$$

Betrachtet man nun lediglich den Zielaktor a_d, so ergibt sich für die untere Ankunftsfunktion nach Umformung folgende Gleichung:

$$\begin{aligned} r_d^l &:= \beta_{sd}^l \otimes G_s \oplus h_s d^l \text{ mit} \\ \beta_{sd}^l &:= (S^l)_{ds}^* \otimes \beta_s^l \text{ und } h_{sd}^l := \bigoplus_{i \neq j} \{(S^l)_{dj}^* \otimes \beta_j^l\} \end{aligned} \tag{8.26}$$

Darin beschreibt β_{sd}^l die untere kumulierte Servicekennlinie und h_{sd}^l beschreibt die Transferfunktion des markierten Graphen entlang des Pfades von a_s nach a_d, wenn die Ankunftsfunktion keine Marken enthält. Es handelt sich somit um einen konstanten Offset.

Für die Zeitanalyse wird schließlich noch die Eingangsankunftsfunktion g_s durch Ankunftskennlinien γ_s^l und γ_s^u begrenzt. Die maximale Latenz von a_s zu a_d lässt sich dann nach Gleichung (8.12) auf Seite 513 berechnen zu:

$$\begin{aligned} \Lambda_{sd} := \max\{&\inf\{\lambda \geq 0 : \beta_{sd}^l(\tau - \lambda) \geq \gamma_s^u(\tau) \forall \tau \geq 0\}, \\ &\inf\{\lambda \geq 0 : h_{sd}^l(\tau - \lambda) \geq \gamma_s^u(\tau) \forall \tau \geq 0\}\} \end{aligned}$$

8.3 Literaturhinweise

Die CTL-Modellprüfung für SysteMoC-Modelle ist in [191] beschrieben. Sie beruht auf der symbolischen Repräsentation des Zustandsraums eines SysteMoC-Modells auf Basis von IDDs und IMDs [193, 192]. Dabei handelt es sich um eine Verfeinerung der symbolischen Repräsentation des Zustandsraums für FunState-Modelle [415, 418, 414]. Der Zustandsraum eines FunState-Modells wird dabei als regulärer Zustandsautomat [434] modelliert. Die dort präsentierten Verfahren eignen sich auch zur Modellprüfung von Petri-Netzen [416, 442].

Die Modellprüfung von SystemC-Modellen ist in [271] vorgestellt. Diese basiert auf Vorarbeiten zur Verifikation von SpecC-Modellen [244] und der Äquivalenzprüfung von Verilog- und C-Programmen [270]. Die Verifikationsmethode verwendet dabei attributierte temporale Strukturen für jeden SystemC-Prozess [89]. Das Gesamtmodell erhält man über eine parallel Komposition der temporalen Strukturen, wie sie bereits in [14] vorgeschlagen wurde. Die verwendete Abstraktion ist identisch mit der „existentiellen Abstraktion“ für temporale Strukturen [101].

Die in [271] beschriebene formale Modellprüfung kann auch zur Verifikation von Transaktionsebenenmodellen (TLMs) eingesetzt werden. In [345, 347] ist ein

solcher Ansatz beschrieben. Eine Erweiterung der Transaktionen um die ACID-Eigenschaften (engl. *Atomicity, Consistency, Isolation, and Durability*) erlaubt es, das Modell des SystemC-Simulators vollständig zu eliminieren [346]. Die ACID-Eigenschaften sind aus Datenbanksystemen bekannt und beschreiben in TLMs folgende Eigenschaften:

- *Atomarität:* Entweder wird eine Transaktion exklusiv, d. h. ohne Verschränkung mit anderen Transaktionen, und vollständig auf einem Target ausgeführt oder gar nicht. Man beachte, dass nichtblockierende Transaktionen als mehrere atomare Transaktionen modelliert werden können.
- *Konsistenz:* Das Target, welches die Transaktion ausführt, ist zu Beginn und am Ende der Ausführung in einem gültigen Zustand.
- *Isolierung:* Es sind keine Zwischenzustände der Berechnung einer Transaktion außerhalb des Targets sichtbar.
- *Nachhaltigkeit:* Nachdem die erfolgreiche Abarbeitung einer Transaktion bestätigt wurde, wird diese nicht mehr rückgängig gemacht.

Weitere Ansätze zur formalen Modellprüfung von TLMs finden sich beispielsweise [210] und [337].

Der in Abschnitt 8.1.4 beschriebene zusicherungsbasierte Verifikationsansatz für TLMs basiert auf der Verschaltung elementarer Monitore und ist ausführlich in [361] beschrieben. Ein alternativer Ansatz, der spezielle SystemC-Kanäle benötigt, ist in [126, 278] beschrieben und verwendet FoCs [167] zur Generierung von C++-Monitoren. In [344] ist ein simulativer Ansatz zur Überprüfung von SystemVerilog-Zusicherungen (SVA) vorgestellt. Mittels Aspekt-orientierter Programmierung wird die Aufzeichnung von zeitbehafteten und nicht zeitbehafteten Transaktionen in die SystemC-TLM-Beschreibung eingeflochten. In einem ersten Simulationslauf werden somit Verläufe der Transaktionen aufgezeichnet und anschließend in einer RTL-Simulation bezüglich der Zusicherungen überprüft. Ein weiterer Ansatz basierend auf SVA wurde von der Firma Infineon entwickelt [143, 144]. Dieser Ansatz erweitert allerdings SVA dahingehend, dass für TLM-Zusicherungen eine neue Zusicherungssprache benötigt wird.

Eine Vielzahl an Ansätzen existieren zur Zeitanalyse auf Systemebene, wobei diese Entwurfsentscheidungen wie Prozessbindung und Transaktionsrouting berücksichtigen. Diese Ansätze lassen sich in simulative und formale Methoden klassifizieren, wobei simulative Methoden das Problem haben, dass sie unvollständig sind. In der Zeitanalyse bedeutet dies, dass die tatsächlichen besten und schlechtesten Fälle im Zeitverhalten einer Implementierung nicht in angemessener Zeit identifiziert werden können. Andererseits sind simulative Methoden breit einsetzbar. Viele simulative Methoden zur Zeitbewertung basieren auf Transaktionsebenenmodellen [133], wobei Transaktionsebenenmodelle die Möglichkeit bieten, die Zeitbewertung durch geeignete Wahl und Kombination von Abstraktionsebenen zu beschleunigen [387, 391]. Eine quantitative Bewertung der Analysegenauigkeit und -geschwindigkeit für Transaktionsebenenmodelle des AMBA-Busses findet sich in [386].

Der in diesem Buch vorgestellte Ansatz zur kombinierten Verhaltens- und Zeitsimulation mittels virtueller Komponenten ist in der VPC-Bibliothek implementiert und ausführlich in [419, 215, 420] beschrieben. Ein vergleichbarer Ansatz wurde in [256] vorgestellt. Um die Laufzeit der simulativen Zeitbewertung zu verringern, bietet es sich an, Daten, die während der Simulation aufgezeichnet werden, in einer späteren Analyse wiederzuverwenden. Ein solches Verfahren ist unter dem Namen *Trace-Driven-Simulation* bekannt geworden [279, 363].

Eine formale Analyse bei der ebenfalls die Kommunikation im Mittelpunkt steht, ist in [402] beschrieben. Dabei wird ein sog. *Kommunikations-Abhängigkeitsgraph* (engl. *communication dependency graph*) aus einer gegebenen Implementierung extrahiert, um den Einfluss der Abbildung auf die Architektur zu analysieren. Dort werden kooperative und nicht präemptive Software-Prozesse als Modell vorausgesetzt [455]. In [454] wird eine Erweiterung für zeitschlitzbasierte Ablaufplanung vorgestellt. Schließlich ist in [219] eine formale Modellierung von SystemC-Modellen mit zeitbehafteten Automaten beschrieben.

Bei den formalen Methoden zur Zeitanalyse können grob holistische und modulare Ansätze unterschieden werden. Holistische Ansätze sind eine direkte Weiterentwicklung der Planbarkeitsanalysen für Einprozessorsysteme hin zu Mehrprozessorsystemen. In einer grundlegenden Arbeit kombinierten Tindell und Clark präemptive Ablaufplanung auf Basis statischer Prioritäten mit TDMA-Ablaufplanung [441]. Eine Genauigkeitsverbesserung konnte in [470] erzielt werden. Weitere Busse, wie der CAN- [440] und der FlexRay-Bus [365], wurden in holistischen Verfahren betrachtet.

Bei den modularen Methoden zur formalen Zeitanalyse ist die symbolische Zeitanalyse ein wichtiger Vertreter. Die symbolische Zeitanalyse ist ausführlich in [378, 380, 379, 377] beschrieben. Wichtige Erweiterungen umfassen die Unterstützung verschiedener Prozessaktivierungen auf Basis mehrerer Ereignisströme [247, 217] und die Analyse von Datenflussmodellen [388]. Die symbolische Zeitanalyse wurde an der Technischen Universität Braunschweig entwickelt und wird mittlerweile als Werkzeug *SymTA/S* [423] von der Firma Symtavision vermarktet und weiterentwickelt.

Die modulare Zeitanalyse basiert auf den Netzwerkkalkül, der in [293] beschrieben ist und auf [121] zurück geht. Hieraus wurde an der ETH Zürich das Echtzeitkalkül entwickelt [431, 430]. Seitdem wurde das Echtzeitkalkül mit anderen Zeitanalysemethoden gekoppelt, u. a. simulativen Methoden [277] und zeitbehafteten Automaten [285].

A

Notation

In diesem Kapitel werden die in diesem Buch verwendeten mathematischen Notationen eingeführt. Dies erfolgt in Anlehnung an Definitionen in [426].

A.1 Mengen

Mengen werden mit Großbuchstaben, deren Elemente mit Kleinbuchstaben bezeichnet. $\{x \mid \mathcal{P}(x)\}$ bezeichne die Menge von Elementen, welche die Bedingung $\mathcal{P}(x)$ erfüllen. Das Symbol $\in$ bezeichne die *Mitgliedschaft* eines Elementes in einer Menge. Die Menge von Elementen, deren Elemente genau die Objekte der Liste $x_1, x_2, \ldots, x_m$ sind, wird mit $\{x_1, x_2, \ldots, x_m\}$ bezeichnet. Die leere Menge werde mit $\emptyset$ (oder $\{\ \}$) bezeichnet. Zwei Mengen heißen *disjunkt*, wenn sie kein Element gemeinsam haben. Die Anzahl der Elemente einer Menge S heißt *Mächtigkeit* von S und werde mit $|S|$ bezeichnet. Seien A und B Mengen. Man sagt A ist *Teilmenge* von B, bezeichnet mit $A \subseteq B$, genau dann, wenn jedes Element von A auch Element von B ist. Wenn $A \subseteq B$ und $A \neq B$, dann heißt A *echte Teilmenge* von B (Schreibweise $A \subset B$). Das *relative Komplement* von B in A, bezeichnet mit $A \setminus B$, ist die Menge $\{x \mid x \in A \wedge x \notin B\}$. Sei A eine Teilmenge einer Menge S. Dann ist das *Komplement* von A in S die Menge $\{x \mid x \in S \setminus A\}$. Schließlich werde die *Vereinigungsmenge* mit $\cup$ und die *Schnittmenge* mit $\cap$ bezeichnet.

Die *Überdeckung* einer Menge S ist eine Menge von Teilmengen von S mit der Eigenschaft, dass ihre Vereinigung gleich S ist. Eine *Partition* einer Menge S ist eine Überdeckung von S durch disjunkte Teilmengen, sog. *Partitionsblöcke*. Die Menge aller Teilmengen einer Menge S wird als *Potenzmenge* 2^S bezeichnet. Die leere Menge $\emptyset$ ist Element der Potenzmenge jeder beliebigen Menge S, d. h. $\emptyset \in 2^S$.

Die Menge der reellen Zahlen wird mit $\mathbb{R}$, die Menge der rationalen Zahlen mit $\mathbb{Q}$, die Menge der ganzen Zahlen mit $\mathbb{Z}$ und die Menge der natürlichen Zahlen mit $\mathbb{N}$ bezeichnet. Es sei S entweder $\mathbb{Z}$, $\mathbb{Q}$ oder $\mathbb{R}$. S^+ bezeichne die positive Teilmenge, S_0^+ oder $S_{\geq 0}$ die nichtnegative Teilmenge (insbesondere $S^+ \cup \{0\}$) und S^n das n-fache *kartesische Produkt* von S. Das *kartesische Produkt* zweier Mengen A und B, bezeichnet mit $A \times B$, ist die Menge von Paaren (a, b), wobei $a \in A$ und $b \in B$ gilt.

C. Haubelt, J. Teich, *Digitale Hardware/Software-Systeme*, eXamen.press,
DOI 10.1007/978-3-642-05356-6, © Springer-Verlag Berlin Heidelberg 2010

A.2 Relationen und Funktionen

Eine *Relation* R zwischen zwei Mengen A und B ist eine Teilmenge von $A \times B$. Es gelte die Schreibweise aRb, wenn $a \in A$ und $b \in B$ und $(a,b) \in R$. Eine Relation heißt *Äquivalenzrelation*, wenn sie *reflexiv* (insbesondere $(a,a) \in R$), *symmetrisch* (insbesondere $(a,b) \in R \Rightarrow (b,a) \in R$) und *transitiv* (insbesondere $(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$) ist. Eine *Partialordnung* ist eine reflexive, transitive und *antisymmetrische* (insbesondere $(a,b) \in R \wedge (b,a) \in R \Rightarrow a = b$) Relation.

Eine *Funktion* (oder Abbildung) f zwischen zwei Mengen A und B ist eine Relation mit der Eigenschaft, dass jedes Element von A immer als erstes Element und immer in genau einem Paar der Relation erscheint. A wird als *Definitionsmenge*, B als *Zielmenge* der Funktion f bezeichnet. Man schreibt $f : A \to B$. Die Funktion f weist jedem $x \in A$ eindeutig ein Element $f(x) \in B$ zu. Die Menge $f(A) = \{f(x) \mid x \in A\}$ heißt *Wertemenge* oder *Bildmenge* der Funktion. Eine Funktion heißt *surjektiv*, falls $f(A) = B$ und *injektiv* (oder Eins-zu-Eins-Abbildung), falls $\forall a,b \in A : f(a) = f(b) \Rightarrow a = b$. Ist die Funktion injektiv, so hat sie eine *Inverse* $f^{-1} : f(A) \to A$. Eine Funktion heißt *bijektiv*, wenn sie surjektiv und injektiv ist. Gegeben sei eine Funktion $f : A \to B$ und eine Teilmenge X von A. Dann heißt $f(X) = \{f(a) \mid a \in X\}$ *Bild* von X unter f. Sei nun $X \subseteq B$. Dann heißt die Menge $\{a \mid f(a) \in X\}$ *Urbild* von X unter f.

Funktionsrepräsentationen

Sei $\mathcal{F}$ eine Menge an Funktionen. Sei $\mathcal{R}$ eine endliche Menge und $\phi : \mathcal{R} \to \mathcal{F}$ eine Funktion, die jedem Element $r \in \mathcal{R}$ eine Funktion $f \in \mathcal{F}$ zuordnet. Das Paar $(\mathcal{R}, \phi)$ heißt *Repräsentation* aller Funktionen $f \in \mathcal{F}$ [329]. Die Funktion ϕ heißt *Interpretation*. Falls $\phi(r) = f$ gilt, so heißt r *Repräsentant* der Funktion f.

Eine Repräsentation $(\mathcal{R}, \phi)$ der Funktionen $f \in \mathcal{F}$ heißt *vollständig*, falls ϕ surjektiv ist. In diesem Fall heißt $\mathcal{R}$ Repräsentation von $\mathcal{F}$. Eine Repräsentation $(\mathcal{R}, \phi)$ der Funktionen $f \in \mathcal{F}$ heißt *eindeutig*, falls ϕ injektiv ist. Eine Repräsentation $(\mathcal{R}, \phi)$ der Funktionen $f \in \mathcal{F}$ heißt *kanonisch*, falls ϕ bijektiv ist.

Boolesche Funktionen

Boolesche Funktionen spielen eine zentrale Rolle im Bereich der Verifikation digitaler Systeme. Die folgende Einführung findet sich in ähnlicher Form in [272].

Gegeben sei die Menge $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$. Eine Boolesche Funktion ist eine Abbildung $f : \mathbb{B}^n \to \mathbb{B}$, d. h. die Wertemenge der Funktion enthält lediglich zwei Elemente. Eine *Belegung* ist eine Funktion β, die jeder Variable der Definitionsmenge von f den Wert T (*wahr*, engl. *true*) oder F (*falsch*, engl. *false*) zuweist. Bei n Variablen gibt es somit genau 2^n mögliche Belegungen. Eine Boolesche Funktion f heißt *gültig*, falls diese für alle Belegungen β zu T evaluiert. Sie heißt *erfüllbar*, falls eine Belegung β existiert, so dass f zu T evaluiert.

Funktionstabelle

Boolesche Funktionen können auf verschiedene Arten repräsentiert werden. Die einfachste Art, eine Boolesche Funktion zu repräsentieren, stellt eine *Funktionstabelle* dar. Das folgende Beispiel ist aus [272] entnommen.

Beispiel A.2.1. Gegeben ist die Boolesche Funktion $f : \mathbb{B}^3 \rightarrow \mathbb{B}$ mit drei Variablen x_1, x_2, x_3, repräsentiert durch Tabelle A.1.

Tabelle A.1. Funktionstabelle einer Booleschen Funktion

	x_1	x_2	x_3	$f(x_1,x_2,x_3)$
β_0	F	F	F	F
β_1	F	F	T	T
β_2	F	T	F	T
β_3	F	T	T	F
β_4	T	F	F	T
β_5	T	F	T	T
β_6	T	T	F	T
β_7	T	T	T	T

Der Vorteil von Funktionstabellen ist, dass diese eine *kanonische* Repräsentation Boolescher Funktionen darstellen, also *vollständig* und *eindeutig* sind. Somit existiert für jede Boolesche Funktion genau ein Repräsentant, der eindeutig ist. Der Nachteil von Funktionstabellen ist allerdings, dass diese exponentiell mit der Anzahl der Variablen wachsen, da jede Funktionstabelle für eine Boolesche Funktion mit n Variablen exakt 2^n Zeilen enthält (Anzahl möglicher Belegungen).

Boolesches Netzwerk

Boolesche Netzwerke dienen zur Modellierung von kombinatorischen Schaltungen oder zur Repräsentation von Booleschen Funktionen [329]. Sie werden oft als markierte gerichtete azyklische Graphen dargestellt. Im Folgenden bezeichne *IPAD* die Eingänge und *OPAD* die Ausgänge einer Schaltung. Ω sei eine endliche Teilmenge von Booleschen Funktionen mit genau einem Ausgang. $\Omega \cup IPAD \cup OPAD$ bezeichne eine sog. *Gatterbibliothek* zur Realisierung einer kombinatorischen Schaltung. Die Standardbibliothek $STD := \{IPAD, OPAD, AND, OR, XOR, NAND, NOR, NOT\}$ wird im Folgenden verwendet. Die graphische Darstellung der einzelnen Gatter wurde in einer DIN-Norm festgelegt. Die Symbole der Gatter aus obiger Standardbibliothek sind in Abb. A.1 dargestellt.

Definition A.2.1 (Boolesches Netzwerk). *Ein* Boolesches Netzwerk $\mathcal{N}$ *ist ein 4-Tupel* $\mathcal{N} = (G, \text{type}, \text{pe}, \text{pa})$ *mit*

- $G = (V, E)$ *ist ein gerichteter azyklischer Graph. V bezeichne die Menge der Knoten, welche die aktiven Elemente der Schaltung repräsentieren. E repräsentiert die Menge der Signale der Schaltung.*

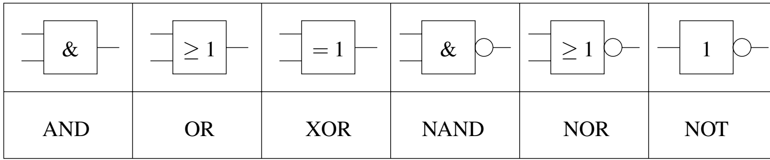


Abb. A.1. Symbole der Gatter der Standardbibliothek nach der DIN-Norm

- type : $V \rightarrow STD$ *ist eine Abbildung, die jedem Knoten* $v \in V$ *ein Element der Standardbibliothek* STD *zuordnet. Falls* $\text{type}(v) = IPAD$*, dann gilt* $\text{indeg}(v) = 0$*. Falls* $\text{type}(v) = OPAD$*, dann gilt* $\text{outdeg}(v) = 0$*. Falls* $\text{type}(v) \in \Omega$ *und* $\text{type}(v)$ *genau n Argumente benötigt, dann gilt* $\text{indeg}(v) = n$*.*
- pe : $\{0,1,\ldots,n-1\} \rightarrow \{v \in V \mid \text{type}(v) = IPAD\}$ *ist eine Abbildung, die den Eingängen der Schaltung eine Ordnung verleiht.* $\text{pe}(i)$ *wird als i-ter* primärer Eingang *der Schaltung bezeichnet.*
- *Analog dazu ist* pa : $\{0,1,\ldots,m-1\} \rightarrow \{v \in V \mid \text{type}(v) = OPAD\}$ *eine Abbildung, die den Ausgängen der Schaltung eine Ordnung verleiht.* $\text{pa}(j)$ *wird als j-ter* primärer Ausgang *der Schaltung bezeichnet.*

Die *Interpretation* $\phi(\mathcal{N})$ eines Boolesches Netzwerkes mit n Eingängen und m Ausgängen gibt die repräsentierte Boolesche Funktion für jeden Ausgang in Abhängigkeit von den Eingängen an: Mit jeder Kante $e \in E$ ist eine Boolesche Funktion f_e assoziiert. Sei im Folgenden $e = (v, \tilde{v})$. Falls v der i-te primäre Eingang ist, d. h. $\text{type}(v) = IPAD$ und $\text{pe}(i) = v$, dann ist mit der Kante e die Boolesche Funktion $f_e = x_i$ assoziiert.

Da die Standardbibliothek lediglich symmetrische Funktionen implementiert, ist die Reihenfolge der Eingänge an den Gattern nicht von Bedeutung. Betrachtet wird ein Knoten v mit eingehenden Kanten $e_0, e_1, \ldots, e_{\text{indeg}(v)-1}$ und ausgehender Kante e_a. Falls $\text{type}(v) = g$ und $g \in \Omega$, dann gilt:

$$f_{e_a} = g(f_{e_0}, f_{e_1}, \ldots, f_{e_{\text{indeg}(v)-1}})$$

Zuletzt werden die m primären Ausgänge des Booleschen Netzwerkes betrachtet. Sei y_j der j-te primäre Ausgang, dann implementiert das Boolesche Netzwerk $\mathcal{N}$ die Boolesche Funktion:

$$\phi(\mathcal{N}) = f_{y_0} \times f_{y_0} \times \cdots \times f_{y_{m-1}}$$

Aufgrund ihrer guten Verständlichkeit werden Boolesche Netzwerke überwiegend zur Repräsentation von kombinatorischen Schaltungen verwendet. Da allerdings Boolesche Netzwerke keine kanonische Repräsentation von Booleschen Funktionen sind, sind sie zur Verifikation von Schaltungen nur bedingt geeignet.

Beispiel A.2.2. Das Boolesche Netzwerk in Abb. A.2 repräsentiert einen Volladdierer, der die Berechnung $a_i + b_i + c_{i-1}$ für $a_i, b_i, c_{i-1} \in \mathbb{B}$ implementiert. Das Ergebnis ist das Summenbit s_i und das Übertragsbit c_i mit $s_i, c_i \in \mathbb{B}$. Die Eingänge und Ausgänge sind in diesem Beispiel nicht als Knoten dargestellt.

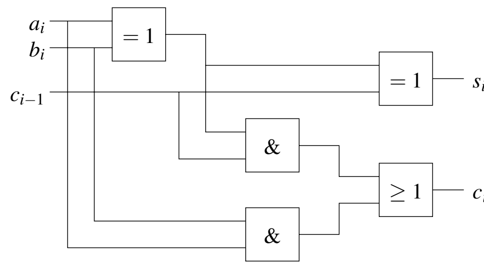


Abb. A.2. Volladdierer als Boolesches Netzwerk

A.3 Aussagenlogik

Aussagenlogik ist eine vollständige und nichteindeutige Repräsentation Boolescher Funktionen. Die Aussagenlogik studiert die Verknüpfungen von Aussagen, die entweder wahr oder falsch sein können. Die Symbole $\neg$, $\vee$, $\wedge$, $\Rightarrow$ und $\Leftrightarrow$ beschreiben die logischen Operatoren Negation, Oder, Und, Implikation und Beidseitige Implikation bzw. Äquivalenz (genau dann, wenn ...). Das Symbol $\forall$ bezeichne den *universellen Quantor* (für alle ...), das Symbol $\exists$ den *existentiellen Quantor* (es gibt ...).

Die *Syntax* der Aussagenlogik ist wie folgt definiert:

Definition A.3.1 (Formel). *Gegeben sei eine Menge* atomarer Formeln *(*Boolesche Variablen*) $x \in X$. Aussagenlogische* Formeln *lassen sich rekursiv definieren zu:*

1. *alle atomaren Formeln $x \in X$ sind Formeln.*
2. *wenn φ und ψ Formeln sind, dann sind auch $\varphi \wedge \psi$ und $\varphi \vee \psi$ Formeln.*
3. *ist φ eine Formel, so ist auch $\neg\varphi$ eine Formel.*

Die *Semantik* ist durch eine rekursive *Interpretation* definiert.

Definition A.3.2 (Interpretation). *Gegeben sei eine aussagenlogische Formel φ und eine Belegung β. Die* Interpretation *ϕ_β weist φ den Wert* T *(wahr) oder* F *(falsch) wie folgt zu:*

$$\phi_\beta(\varphi) := \beta(\varphi) \text{ falls } \varphi \in X \tag{A.1}$$

$$\phi_\beta(\varphi \wedge \psi) := \begin{cases} \mathsf{T} & \text{falls } \phi_\beta(\varphi) = \mathsf{T} \text{ und } \phi_\beta(\psi) = \mathsf{T} \\ \mathsf{F} & \text{sonst} \end{cases} \tag{A.2}$$

$$\phi_\beta(\varphi \vee \psi) := \begin{cases} \mathsf{T} & \text{falls } \phi_\beta(\varphi) = \mathsf{T} \text{ oder } \phi_\beta(\psi) = \mathsf{T} \\ \mathsf{F} & \text{sonst} \end{cases} \tag{A.3}$$

$$\phi_\beta(\neg\varphi) := \begin{cases} \mathsf{T} & \text{falls } \phi_\beta(\varphi) = \mathsf{F} \\ \mathsf{F} & \text{sonst} \end{cases} \tag{A.4}$$

Gegeben seien eine aussagenlogische Formel φ und eine Belegung β. Falls $\phi_\beta(\varphi) = \mathsf{T}$ ist, so sagt man, β ist ein *Modell* für φ, geschrieben als $\beta \models \varphi$. Man sagt auch β *erfüllt* φ. φ heißt *erfüllbar*, falls mindestens ein Modell für φ existiert,

andernfalls heißt φ *unerfüllbar*. Eine Menge an Formeln $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ heißt *erfüllbar*, genau dann, wenn eine Belegung β existiert, die für jede Formel ein Modell ist, d. h. $\forall \varphi_i \in \Phi : \beta \models \varphi_i$. Eine Formel φ heißt *gültig* (φ ist eine *Tautologie*), geschrieben $\models \varphi$, falls jede Belegung β ein Modell für φ ist. Hiermit lässt sich der Begriff *Deduktion* definieren [272]:

Definition A.3.3 (Deduktion). *Eine Formel φ ergibt sich aus einer Menge an Formeln Ψ, geschrieben als $\Psi \models \varphi$, wenn $\beta \models \varphi$ für jede Belegung β gilt, mit der Eigenschaft $\forall \psi \in \Psi : \beta \models \psi$.*

Mit anderen Worten: Die Formeln $\psi \in \Psi$ bestimmen diejenigen Belegungen β, die ein Modell für φ sind.

A.4 Prädikatenlogik erster Ordnung

Die Aussagenlogik basiert lediglich auf Fakten (aussagenlogischen Variablen). Dies erleichtert den Umgang mit der Aussagenlogik, schränkt diese aber auch in ihrer Möglichkeit ein, Wissen zu repräsentieren. Aus diesem Grund erweitert die *Prädikatenlogik erster Ordnung* (engl. *first order logic*) die Aussagenlogik um drei wesentliche Konzepte: Prädikate, Funktionen und Quantoren. In der Prädikatenlogik erster Ordnung werden *atomare Formel* hinsichtlich ihrer inneren Struktur untersucht, d. h. in der Prädikatenlogik sind atomare Formeln anders definiert als in der Aussagenlogik. Atomare Formeln der Prädikatenlogik werden durch Anwendung von Prädikatensymbolen auf Terme gebildet.

Die Elemente der Prädikatenlogik sind:

1. logische Zeichen: Klammern ("(", ")"), Quantoren ($\forall$ und $\exists$), logische Operatoren ($\neg$, $\wedge$ und $\vee$) sowie konstante Formeln (T und F).
2. Theoriezeichen: Variablen, konstante Symbole, Prädikatensymbole und Funktionssymbole.

Die logische Zeichen zusammen mit den Theoriezeichen bilden das *Alphabet* der Prädikatenlogik erster Ordnung.

Definition A.4.1 (Syntax der Prädikatenlogik erster Ordnung). *Gegeben sei das Alphabet der Prädikatenlogik erster Ordnung.* Terme *lassen sich wie folgt definieren:*

- *Jede Variable ist ein Term,*
- *konstante Symbole sind Terme, und*
- *falls $t_1, \ldots, t_n$ Terme und f_i ein Funktionssymbol für ein n-äre Funktion sind, dann ist $f_i(t_1, \ldots, t_n)$ ein Term.*

Formeln *sind wie folgt definiert:*

- *falls $t_1, \ldots, t_n$ Terme sind und p_i ein n-äres Prädikatensymbol ist, dann ist $p_i(t_1, \ldots, t_n)$ eine* atomare Formel,
- *atomare Formeln sind Formeln,*
- *konstante Formeln sind Formeln,*

- *falls φ eine Formeln ist, so ist auch $\neg\varphi$ eine Formel,*
- *falls φ und ψ Formeln sind, so sind auch $\varphi \wedge \psi$ und $\varphi \vee \psi$ Formeln und*
- *falls x eine Variable und φ eine Formel sind, so sind auch $\forall x : \varphi$ und $\exists x : \varphi$ Formeln. In beiden Fällen heißt x* gebunden *in φ. Eine nichtgebundene Variable heißt* frei.

Atomare Formeln der Prädikatenlogik erster Ordnung entsprechen den atomaren Formeln der Aussagenlogik. Somit können aus atomaren Formeln der Prädikatenlogik durch Verwendung logischer Operatoren und Quantoren komplexere Formeln gebildet werden. Aus diesem Grund kann die Aussagenlogik auch als Spezialfall der Prädikatenlogik gesehen werden. Dann dürfen allerdings weder allgemeine Variablen noch Funktionssymbole verwendet werden.

Ohne die Semantik der Prädikatenlogik erster Ordnung im Folgenden formal zu definieren, soll dennoch die Frage beantwortet werden, wann eine prädikatenlogische Formel φ erfüllt ist (den Wert T annimmt). Wie bei der Aussagenlogik bedient man sich hierfür der Interpretation ϕ_β bei gegebener Belegung β, die jeder Variablen der Formel einen Wert zuweist. Bei der Interpretation prädikatenlogischer Formeln müssen weiterhin konstante Symbole, Prädikatensymbole und Funktionssymbole interpretiert werden.

Beispiel A.4.1. Es wird ein Beispiel für die Anwendung einer einfachen Arithmetik betrachtet [30]. Die verwendeten Theoriezeichen sind:

- Variablen: x, y
- Konstante: 0
- Funktionssymbol: $+$
- Prädikatensymbole: $>, =$

Zusammen mit den logischen Zeichen lassen sich Eigenschaften der Arithmetik spezifizieren:

- Die Addition von 0 zu einer Zahl x ändert ihren Wert nicht: $\forall x : x + 0 = x$
- Addition ist kommutativ: $\forall x, y : (x + y) = (y + x)$
- Es gibt immer eine größere Zahl: $\forall x : \exists y : y > x$

A.5 Graphen

Ein Graph $G(V,E)$ ist ein Paar (V,E), wobei V eine Menge und $E \subseteq V \times V$ eine binäre Relation zwischen Elementen aus V darstellt. Die Elemente aus V heißen *Knoten* und die Elemente der Menge *E Kanten*. In einem *gerichteten Graphen* sind die Kanten geordnete Knotenpaare, in einem *ungerichteten Graphen* ungeordnete Paare. Eine gerichtete Kante von einem Knoten $v_i \in V$ zu einem Knoten $v_j \in V$ werde mit (v_i, v_j) und eine ungerichtete Kante zwischen v_i und v_j mit $\{v_i, v_j\}$ bezeichnet. Für eine Kante $e = \{v_i, v_j\}$ heißen v_i und v_j *Endpunkte* von e. Man sagt, dass v_i und v_j mit Kante *e inzident* sind und dass v_i und v_j *adjazent* sind. Der *Grad* $\deg(v_i)$ eines Knotens v_i ist die Zahl der mit ihm inzidenten Kanten.

Ungerichtete Graphen

Eine Kante mit gleichen Endpunkten heißt *Schleife*. Ein Graph ohne Schleifen und mit der Eigenschaft, dass zwischen jedem Knotenpaar nur maximal eine Kante verläuft, heißt *einfach*, sonst *Multigraph*. Bezeichne $(e_1, e_2, \ldots, e_n)$ eine Folge von Kanten eines Graphen $G(V,E)$. Wenn es Knoten $v_0, v_1, \ldots, v_n$ mit $e_i = \{v_{i-1}, v_i\}$ für alle $i = 1, \ldots, n$ gibt, dann heißt die Folge ein *Kantenzug* (engl. *walk*). Im Fall $v_0 = v_n$ spricht man von einem *geschlossenen Kantenzug*. Wenn die e_i paarweise verschieden sind, liegt ein *Weg* (engl. *trail*) bzw. im geschlossenen Fall ein *Kreis* (oder *Zyklus*, engl. *cycle*) vor. Falls auch die v_j paarweise verschieden sind, heißt der Weg *einfacher Weg* oder *Pfad*. Ein Graph heißt *zusammenhängend*, wenn es für je zwei Knoten $v_i, v_j \in V$ jeweils einen Pfad von v_i nach v_j gibt.

Beispiel A.5.1. Betrachtet wird der ungerichtete Graph in Abb. A.3. Es handelt sich um einen Multigraphen, da es Paare von Knoten gibt, zwischen denen mehr als eine Kante verläuft. Der Graph ist zusammenhängend. Die Folge von Kanten (a,b,e,e) ist ein Kantenzug, die Folge (a,g) ist kein Kantenzug. (a,a) ist ein geschlossener Kantenzug, (a,b,e) stellt einen Weg dar. Der Kantenzug (f,g,i,h) stellt einen Zyklus dar, allerdings keinen einfachen Zyklus, denn der Kantenzug ist kein Pfad. (f,h) ist ein einfacher Zyklus.

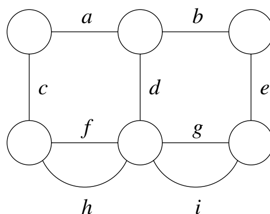


Abb. A.3. Ungerichteter Graph

Ein Graph ohne Zyklen heißt *azyklisch* oder *kreisfrei* oder *Wald* (engl. *forest*). Ein *Baum* ist ein zusammenhängender, azyklischer Graph. Ein *Spannbaum* (engl. *spanning tree*) eines Graphen $G(V,E)$ bezeichnet einen Baum $T(V,E')$ mit $E' \subseteq E$. Ein *vollständiger Graph* besitzt die Eigenschaft, dass zwischen jedem Paar von Knoten eine Kante verläuft. Das *Komplement* eines Graphen $G(V,E)$ bezeichnet einen Graphen mit Knotenmenge V, in dem zwei Knoten genau dann adjazent sind, wenn sie in G nicht adjazent sind. Ein Graph heißt *bipartit*, falls die Knotenmenge in zwei Teilmengen partitioniert werden kann, so dass jede Kante zwischen Knoten unterschiedlicher Partitionsblöcke verläuft.

Ein *Teilgraph* (oder *Untergraph*) eines Graphen $G(V,E)$ bezeichnet einen Graphen, dessen Knoten- und Kantenmenge in der Knoten- bzw. Kantenmenge von G enthalten sind. Gegeben seien ein Graph $G(V,E)$ und eine Teilmenge $U \subseteq V$, dann

bezeichnet der *von U induzierte Untergraph* den Teilgraphen von $G(V,E)$ mit Knotenmenge U und denjenigen Kanten aus E, deren beide Endpunkte in U sind.

Gerichtete Graphen

Die Definitionen für ungerichtete Graphen lassen sich direkt auf gerichtete Graphen erweitern. Gegeben sei eine gerichtete Kante $e = (v_i, v_j)$. Dann heißt v_i *Anfang* bzw. *Anfangsknoten* der Kante oder *direkter Vorgänger* von v_j, und v_j *Ende* bzw. *Endknoten* der Kante oder *direkter Nachfolger* von v_i. Man sagt auch: *e verlässt v_i* und *betritt v_j*. Für eine Kante $e \in E$ bezeichne $\mathrm{pred}(e)$ den Anfang der Kante und $\mathrm{succ}(e)$ das Ende der Kante. Der *Eingangsgrad* $\mathrm{indeg}(v)$ eines Knotens v ist gleich der Anzahl von Kanten, für die er der Endknoten ist, der *Ausgangsgrad* $\mathrm{outdeg}(v)$ eines Knotens v ist gleich der Anzahl von Kanten, für die er den Anfangsknoten darstellt. Die Definition eines *Kantenzugs* lässt sich analog zur Definition bei ungerichteten Graphen formulieren, indem man ungerichteten Kanten eine Orientierung verleiht. Genauso lassen sich Wege, Pfade und Zyklen definieren. Der *zugehörige ungerichtete Graph* (engl. *underlying graph*) eines gerichteten Graphen bezeichnet den ungerichteten Graphen, den man erhält, indem man die gerichteten Kanten durch ungerichtete Kanten ersetzt. Ein gerichteter Graph heißt *zusammenhängend*, wenn sein zugehöriger ungerichteter Graph zusammenhängend ist. Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es einen gerichteten Pfad von jedem Knoten zu jedem anderen Knoten gibt. Gegeben sei ein ungerichteter Graph. Dann bezeichnet eine *Orientierung* einen gerichteten Graphen, den man dadurch erhält, dass man den Kanten eine Richtung verleiht. Gerichtete azyklische Graphen (engl. *directed acyclic graphs*, DAGs) repräsentieren partiell geordnete Mengen. In einem DAG heißt ein Knoten v_j *Nachfolger* eines Knotens v_i, falls v_i Anfang eines Pfades ist, dessen Ende v_j darstellt. Man sagt auch v_j ist von v_i aus *erreichbar*. Entsprechend heißt ein Knoten v_j *Vorgänger* eines Knotens v_i, falls ein Pfad von v_j nach v_i existiert. Ein DAG heißt *polar*, falls er zwei ausgezeichnete Knoten besitzt, einen sog. *Quellknoten* und einen sog. *Senkeknoten*, und zusätzlich die Eigenschaften, dass alle Knoten vom Quellknoten aus erreichbar und der Senkeknoten von allen Knoten aus erreichbar ist.

Ein gerichteter Graph kann auch durch seine *Inzidenzmatrix* $C \in \{-1,0,1\}^{|V|\times|E|}$ dargestellt werden, die wie folgt definiert ist: $c_k = e_j - e_i$, mit $c_k, 1 \leq k \leq |E|$, bezeichnet dabei die k-te Spalte von C und stellt die Kante $(v_i, v_j) \in E$ dar. Ferner bezeichne $e_i \in \mathbb{Z}^{|V|\times 1}$ den i-ten Einheitsvektor, $1 \leq i \leq |V|$.

Gerichtete und ungerichtete Graphen können *gewichtet* sein. Dabei können Gewichte den Knoten, Kanten oder Knoten und Kanten zugeordnet sein. Ein Paar (G,w), bestehend aus einem Graphen $G(V,E)$ und einer Funktion $w : E \to \mathbb{R}$, heißt *Netzwerk*. Häufig benutzt man auch die Notation $G(V,E,w)$ zur Bezeichnung eines Netzwerks. In Zusammenhang mit Netzwerken bezeichnet man die Summe der Gewichte der Kanten eines Pfades als *Pfadgewicht*.

B

Binäre Entscheidungsdiagramme

Entscheidungsdiagramme (engl. *Decision Diagram*, DD) sind wichtige Datenstrukturen bei der Verifikation digitaler Hardware/Software-Systeme. In diesem Kapitel werden die Grundlagen dieser Funktionsrepräsentationen vorgestellt.

B.1 Entscheidungsdiagramme

Entscheidungsdiagramme sind Datenstrukturen für die Repräsentation von diskreten Funktionen. Besondere Bedeutung haben Entscheidungsdiagramme in der Form *binärer Entscheidungsdiagramme* zur Repräsentation Boolescher Funktionen und diskreter Mengen erlangt [294, 5, 427]. Der Durchbruch für Entscheidungsdiagrammen in der Verifikation kam allerdings mit der Publikation von R. E. Bryant im Jahr 1986 [62]. Die folgenden Definitionen sind in ähnlicher Form in [135, 36] zu finden.

Gegeben sei eine diskrete Funktion $f : A_1 \times \cdots \times A_\mathrm{n} \to B$, mit $A_i \subseteq \mathbb{R}, 1 \leq i \leq \mathrm{n}$. Weiterhin sei $B \subset \mathbb{Z}$ und endlich. Dann wird mit $\Delta(A_i) = \{\Delta_1(A_i), \ldots, \Delta_k(A_i), \ldots\}$ eine Partition der Definitionsmenge A_i bezeichnet. Weiterhin sei $a_i^{\Delta_k}$ ein *Literal* mit der folgenden Eigenschaft:

$$a_i^{\Delta_k} = \begin{cases} 0 \text{ falls } a_i \notin \Delta_k(A_i) \\ 1 \text{ falls } a_i \in \Delta_k(A_i) \end{cases} \tag{B.1}$$

Eine Definition eines allgemeinen Entscheidungsdiagramms sieht wie folgt aus:

Definition B.1.1 (Entscheidungsdiagramm). *Ein* Entscheidungsdiagramm *für eine Funktion $f : A_1, \ldots, A_n \to B$ ist ein gerichteter azyklischer Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E sowie den folgenden Eigenschaften:*

- *G ist ein Baum,*
- *V ist eine Partition von* Terminalknoten V_T *und* Nichtterminalknoten V_N,
- *eine Funktion* index : $V_\mathrm{N} \to \{1, \ldots, n\}$ *weist jedem Nichtterminalknoten $v \in V_\mathrm{N}$ eine Variablen $a_{\mathrm{index}(v)}$ zu,*

- *für jeden Nichtterminalknoten $v \in V_\mathrm{N}$ weist eine Funktion* child : $V_\mathrm{N} \times \{1, \ldots, |\Delta(A_{\mathrm{index}(v)})|\} \rightarrow V$ *dem Knoten v seine Nachfolger zu, d. h.* $(v, \mathrm{child}(v,k)) \in E$, *mit* $1 \leq k \leq |\Delta(A_{\mathrm{index}(v)})|$,
- *eine Funktion* value : $V_\mathrm{T} \rightarrow B$ *weist jedem Terminalknoten einen Wert aus der Zielmenge B zu.*

Ein Entscheidungsdiagramm besitzt genau einen ausgezeichneten *Quellknoten*, d. h. einen Knoten ohne Vorgänger, von dem aus alle Knoten $v \in V$ erreichbar sind. Terminalknoten sind Blätter und besitzen die Eigenschaft, dass sie keine Nachfolger haben. Die *Größe* eines Entscheidungsdiagramms ist gegeben durch die Anzahl seiner Knoten, d. h. $\mathrm{size}(G) = |V|$.

Ein Entscheidungsdiagramm heißt *komplett*, wenn jede Variable a_i auf jedem Pfad vom Quellknoten zu einem Terminalknoten *genau einmal* auftritt. Ein Entscheidungsdiagramm heißt *frei*, falls jede Variable a_i auf jedem Pfad vom Quellknoten zu einem Terminalknoten *höchstens* einmal auftritt. Ein Entscheidungsdiagramm G heißt *geordnet*, falls G frei ist und auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten.

B.2 Binäre Entscheidungsdiagramme

Boolesche Funktionen können als binäre Entscheidungsdiagramme repräsentiert werden. Grundlage hierzu bildet die *Shannon-Zerlegung* [393], welche auf dem Konzept von *Kofaktoren* basiert. Gegeben sei ein Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ mit Variablen $x_1, \ldots, x_n$. Die Funktion $f|_{x_i:=\mathsf{F}} = f(x_1, \ldots, x_{i-1}, \mathsf{F}, x_{i+1}, \ldots x_n)$ heißt *negativer Kofaktor* bezüglich x_i. Die Funktion $f|_{x_i:=\mathsf{T}} = f(x_1, \ldots, x_{i-1}, \mathsf{T}, x_{i+1}, \ldots x_n)$ heißt *positiver Kofaktor* bezüglich x_i. Sowohl $f|_{x_i:=\mathsf{F}}$ als auch $f|_{x_i:=\mathsf{T}}$ sind unabhängig von der Variablen x_i. Jede Boolesche Funktion f lässt sich mit Hilfe der *Shannon-Zerlegung* wie folgt mit Kofaktoren schreiben:

$$f = (\neg x_i \wedge f|_{x_i:=\mathsf{F}}) \vee (x_i \wedge f|_{x_i:=\mathsf{T}}) \tag{B.2}$$

Falls nun in einem Entscheidungsdiagramm in jedem Nichtterminalknoten $v \in V_\mathrm{N}$ die Shannon-Zerlegung nach der mit den Knoten assoziierten Variablen $x_{\mathrm{index}(v)}$ durchgeführt wird, so erhält man ein *binäres Entscheidungsdiagramm* (engl. *Binary Decision Diagram, BDD*). Mit $\forall v \in V_\mathrm{N} : \Delta(\mathbb{B}) = \{\Delta_0(\mathbb{B}) = \{\mathsf{F}\}, \Delta_1(\mathbb{B}) = \{\mathsf{T}\}\}$ kann dieses wie folgt definiert werden [135]:

Definition B.2.1 (Binäres Entscheidungsdiagramm). *Gegeben sei eine Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ mit den Variablen $x_1, \ldots, x_n$. Das zugehörige* binäre Entscheidungsdiagramm (V, E) *mit Quellknoten v_0 ist ein Entscheidungsdiagramm mit folgenden Eigenschaften:*

- *Der Quellknoten v_0 repräsentiert die Funktion f, d. h. seine Interpretation $\phi(v_0)$ ist f.*
- *Für jeden Knoten $v \in V$ gilt:*

– *Falls* $v \in V_\mathrm{T}$ *und* $\mathrm{value}(v) = \mathsf{F}$*, dann repräsentiert* v *die entsprechende konstante Boolesche Funktion, d. h.* $\phi(v) = \mathsf{F}$.
– *Falls* $v \in V_\mathrm{T}$ *und* $\mathrm{value}(v) = \mathsf{T}$*, dann repräsentiert* v *die entsprechende konstante Boolesche Funktion, d. h.* $\phi(v) = \mathsf{T}$.
– *Falls* $v \in V_\mathrm{N}$*, dann repräsentiert* v *die Funktion*

$$\phi(v) = (\neg x_{\mathrm{index}(v)} \wedge \phi(\mathrm{child}(v,1)) \vee (x_{\mathrm{index}(v)} \wedge \phi(\mathrm{child}(v,2)))$$

mit $\phi(\mathrm{child}(v,1)) = \phi(v)|_{x_{\mathrm{index}(v)}:=\mathsf{F}}$ *und* $\phi(\mathrm{child}(v,2)) = \phi(v)|_{x_{\mathrm{index}(v)}:=\mathsf{T}}$.

BDDs sind eine vollständige Repräsentation der Booleschen Funktionen. Ein BDD heißt *geordnetes binäres Entscheidungsdiagramm* (engl. *Ordered Binary Decision Diagram, OBDD*), falls es frei ist und auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten. Bei einer gegebenen Variablenordnung kann eine Boolesche Funktion noch durch unterschiedliche OBDDs repräsentiert werden.

Reduzierte OBDDs

Um zu einer *kanonischen* Repräsentation der Booleschen Funktionen zu gelangen, werden OBDDs reduziert. Das Ergebnis sind *reduzierte geordnete binäre Entscheidungsdiagramme* (engl. *Reduced Ordered Binary Decision Diagram, ROBDD*). Zur Reduktion eines OBDD gibt es zwei Regeln:

1. *Eliminationsregel*: Sind $\mathrm{child}(v,1) = \mathrm{child}(v,2) = v'$, so kann v aus dem Entscheidungsdiagramm eliminiert werden, und seine eingehenden Kanten zeigen auf v'.
2. *Verschmelzungsregel*: Knoten mit gleichem Index und gleichen Nachfolgern, d. h. $\mathrm{index}(v) = \mathrm{index}(v')$ und $\mathrm{child}(v,1) = \mathrm{child}(v',1)$ sowie $\mathrm{child}(v,2) = \mathrm{child}(v',2)$, können verschmolzen werden, indem alle ausgehenden Kanten von v gelöscht und alle eingehenden Kanten von v auf v' zeigen.

Ein OBDD, welches nicht weiter durch die Eliminations- oder Verschmelzungsregel reduziert werden kann, heißt *reduziertes OBDD*. Die Anwendung beider Regeln ist in Abb. B.1 dargestellt. Ausgehende gestrichelte Kanten zeigen $x_{\mathrm{index}(v)} = 0$ an, ausgehende durchgezogene Linien repräsentieren $x_{\mathrm{index}(v)} = 1$.

ROBDDs sind für eine gegebene Variablenordnung kanonische Repräsentationen Boolescher Funktionen, d. h. zu jeder Booleschen Funktion existiert genau ein eindeutiges ROBDD für diese Variablenordnung. Damit gilt: Zwei Boolesche Funktionen sind genau dann äquivalent, wenn ihre ROBDDs für eine gegebene Variablenordnung isomorph sind [62]. Zwei ROBDDs $G_1 = (V_1, E_1)$ und $G_2 := (V_2, E_2)$ heißen genau dann *isomorph*, wenn eine bijektive Funktion $\mathrm{m} : V_1 \rightarrow V_2$ existiert, so dass für alle Terminalknoten $v \in V_{1,\mathrm{T}}$ gilt $\mathrm{value}(v) = \mathrm{value}(\mathrm{m}(v))$ und für alle Nichtterminalknoten $v \in V_{1,\mathrm{N}}$ gilt $\mathrm{m}(\mathrm{child}(v,1)) = \mathrm{child}(\mathrm{m}(v),1)$ und $\mathrm{m}(\mathrm{child}(v,2)) = \mathrm{child}(\mathrm{m}(v),2)$.

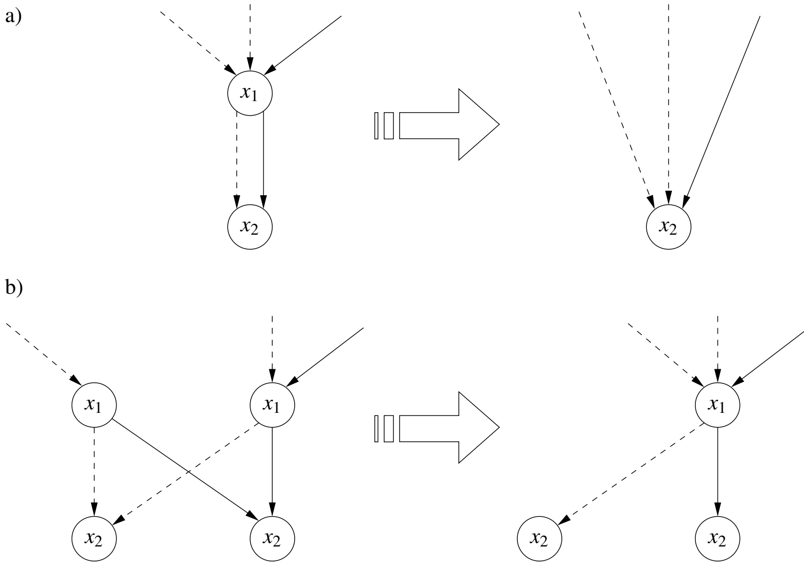


Abb. B.1. a) Eliminationsregel und b) Verschmelzungsregel

Auswirkung der Variablenordnung

Die Größe size(G) eines ROBDD G kann exponentiell mit der Anzahl der Variablen der repräsentierten Booleschen Funktion wachsen und hängt zudem von der Variablenordnung ab. Sie ist zwischen linear und exponentiell zu der Anzahl seiner Variablen, abhängig von der verwendeten Variablenordnung [62]. Es gibt zwar Funktionen, deren Darstellung unabhängig von der Variablenordnung immer eine Knotenanzahl exponentiell zu der Anzahl ihrer Variablen hat, z. B. die Multiplikation [63], aber für die meisten Funktionen lässt sich eine günstigere Ordnung finden. Eine optimale Variablenordnung zu finden, die mit der geringstmöglichen Anzahl an Knoten in dem Entscheidungsdiagramm auskommt, ist in den meisten in der Praxis vorkommenden Fällen zu rechenintensiv. Für BDDs gibt es verschiedene Ansätze, eine gute Variablenordnung zu finden, die entweder statisch festgelegt oder dynamisch zur Laufzeit bestimmt wird. Ein statischer Ansatz ist etwa FORCE [6], der die Knotenreihenfolge eines Hypergraphen und eine damit verbunden Ordnung der Variablen nach Vorbild physikalischer Kräfte ändert. Beispiele für dynamische Ansätze sind das sog. engl. *window permutation* (siehe auch [239]) und das sog. engl. *sifting* [383]. Beim *window permutation* werden für k Variablen die möglichen $k!$ Permutationen in der Variablenordnung getestet und das Ergebnis mit dem kleinsten ROBDD übernommen. Hingegen wird beim *sifting* eine einzelne der n Variablen betrachtet und diese versuchsweise an jede mögliche der n Positionen in der Variablenordnung platziert. Wiederum wird das beste Ergebnis, also dasjenige ROBDD mit geringster Größe, übernommen.

Rechnen mit ROBDDs

Zur Durchführung von Rechenoperationen (unäre oder binäre) auf ROBDDs wird üblicherweise auf den *if-then-else-Operator* (*ITE-Operator*) zurückgegriffen [57, 211]. Seien f, g, h Boolesche Funktionen. Der ITE-Operator ist wie folgt definiert:

$$\text{ITE}(f,g,h) := (f \wedge g) \vee (\neg f \wedge h) \tag{B.3}$$

Die wichtigsten Booleschen Rechenoperationen sind in Tabelle B.1 dargestellt.

Tabelle B.1. Berechnung wichtiger Boolescher Operationen mit dem ITE-Operator

Operator	ITE-Form
$\neg x$	$\text{ITE}(x, \mathsf{F}, \mathsf{T})$
$x_1 \wedge x_2$	$\text{ITE}(x_1, x_2, \mathsf{F})$
$x_1 \vee x_2$	$\text{ITE}(x_1, \mathsf{T}, x_2)$
$x_1 \Rightarrow x_2$	$\text{ITE}(x_1, x_2, \mathsf{T})$
$x_1 \Leftrightarrow x_2$	$\text{ITE}(x_1, x_2, \neg x_2)$
$f(x, g(x))$	$\text{ITE}(g(x), f(x, \mathsf{T}), f(x, \mathsf{F}))$
$\exists x : f(x)$	$\text{ITE}(f(\mathsf{T}), \mathsf{T}, f(\mathsf{F}))$
$\forall x : f(x)$	$\text{ITE}(f(\mathsf{T}), f(\mathsf{F}), \mathsf{F})$

Die Kofaktorberechnung bezüglich x_i erfolgt durch Umlenken aller zu einem Knoten v mit $\text{index}(v) = i$ eingehenden Kanten auf den Nachfolgerknoten $\text{child}(v, 1)$ bzw. $\text{child}(v, 2)$. Ob eine Boolesche Funktion f *gültig* oder *erfüllbar* ist, kann in konstanter Zeit mit Hilfe des zugehörigen ROBDD entschieden werden. Hierzu wird geprüft, ob dass BDD lediglich aus einem Terminalknoten v besteht, der mit $\text{value}(v) = \mathsf{T}$ bzw. $\text{value}(v) = \mathsf{F}$ markiert ist.

B.3 Verallgemeinerte binäre Entscheidungsdiagramme

Neben der Shannon-Zerlegung gibt es die *positive* und *negative Davio-Zerlegung* einer Booleschen Funktion. Die positive Davio-Zerlegung ist wie folgt definiert:

$$f := f|_{x_i:=\mathsf{F}} \oplus (x_i \wedge (f|_{x_i:=\mathsf{T}} \oplus f|_{x_i:=\mathsf{F}})) \tag{B.4}$$

Dabei stellt $\oplus$ die Exklusiv-Oder-Verknüpfung dar ($x_1 \oplus x_2 := (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$).

Verwendet man in einem binären Entscheidungsdiagramm an Stelle der Shannon-Zerlegung die positive Davio-Zerlegung, so erhält man ein sog. *positives funktionales Entscheidungsdiagramm* (engl. *positive Functional Decision Diagram, pFDD*) [253]. Die Interpretation ϕ eines Nichtterminalknoten $v \in V_\text{N}$ eines pFDD ist gegeben durch $\phi(v) := \phi(\text{child}(v,1)) \oplus (x_i \wedge \phi(\text{child}(v,2)))$. Die Interpretation der Terminalknoten bleibt unverändert.

Neben der Shannon- und positiven Davio-Zerlegung gibt es noch die *negative Davio-Zerlegung*:

$$f := f|_{x_i:=\mathsf{T}} \oplus (\neg x_i \wedge (f|_{x_i:=\mathsf{T}} \oplus f|_{x_i:=\mathsf{F}})) \tag{B.5}$$

Wird die negative Davio-Zerlegung in einem BDD verwendet, erhält man ein *negatives funktionales Entscheidungsdiagramm* (engl. *negative Functional Decision Diagram, nFDD*). Die Interpretation ϕ eines Nichtterminalknoten $v \in V_N$ eines nFDD ist gegeben durch $\phi(v) := \phi(\text{child}(v,1)) \oplus (\neg x_i \wedge \phi(\text{child}(v,2)))$. Die Interpretation der Terminalknoten bleibt unverändert.

Falls negative und positive Davio-Zerlegung in einem Entscheidungsdiagramm zugelassen ist, spricht man von einem *funktionalen Entscheidungsdiagramm* (*FDD*) [139]. In diesem Fall muss zu jeder Variable x_i der verwendete Zerlegungstyp festgelegt werden. Ein *geordnetes FDD* (engl. *Ordered FDD, OFDD*) besitzt die Eigenschaft, dass auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten. Ein *reduziertes OFDD* (engl. *Reduced OFDD, ROFDD*) erhält man durch wiederholte Anwendung der bereits bekannten *Verschmelzungsregel* in Abb. B.1b) und einer neuen *Eliminationsregel*: Ist $\text{child}(v,2) = \mathsf{F}$, so kann v aus dem Entscheidungsdiagramm eliminiert werden, und seine eingehenden Kanten auf $\text{child}(v,1)$ umgeleitet werden (siehe Abb. B.2).

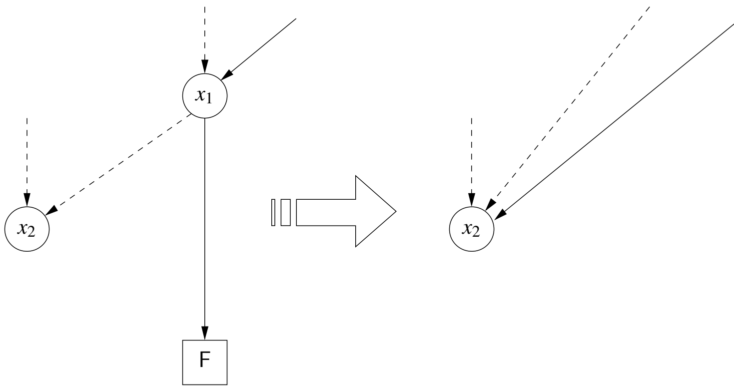


Abb. B.2. Eliminationsregel für OFDDs

Die drei Klassen binärer Entscheidungsdiagramme, BDD, pFDD und nFDD, lassen sich zu einer neuen Art von Entscheidungsdiagramm kombinieren: *Kronecker FDDs* basieren auf den drei Funktionszerlegungen Shannon-Zerlegung, positive Davio- und negative Davio-Zerlegung [138]. Hierbei muss für jede Variable festgelegt sein, welcher Zerlegungstyp verwendet wird. Shannon-, positive und negative Davio-Zerlegung sind die einzigen drei Zerlegungstypen die bei der Repräsentation von Booleschen Funktionen durch Entscheidungsdiagrammen berücksichtigt werden müssen, d. h. nur diese führen zu strukturell unterschiedlichen Entscheidungs-

diagrammen [34]. *Geordnete KFDDs* (engl. *Ordered KFDD, OKFDD*) besitzen die Eigenschaft, dass auf jedem Pfad vom Quellknoten zu einem Terminalknoten die Variablen in der selben Reihenfolge auftreten. *Reduzierte OKFDDs* (engl. *Reduced OKFDD, ROKFDD*) können aus OKFFDs durch wiederholte Anwendung der Verschmelzungs- und Eliminationsregel gebildet werden. Hierbei ist der verwendete Zerlegungstyp (Shannon- oder Davio-Zerlegung) bei Anwendung der Eliminationsregel zu berücksichtigen.

ROKFDDs sind für eine gegebene Variablenordnung kanonische Repräsentationen Boolescher Funktionen. Da ROBDDs und ROFDDs spezialisierte ROKFDDs sind, gilt, dass für eine gegebene Boolesche Funktion jedes minimal große ROKFDD nicht größer als das minimal große ROBDD oder minimal große ROFDD bei gleicher Variablenordnung ist. Das folgende Beispiel stammt aus [135].

Beispiel B.3.1. Gegeben ist die Boolesche Funktion $f := (x_1 \wedge x_2 \wedge x_3) \oplus (x_1 \wedge x_2 \wedge \neg x_3) \oplus (\neg x_1 \wedge x_2 \wedge x_3)$ sowie die Variablenordnung $x_1 < x_2 < x_3 < x_4$. Das zugehörige ROBDD ist in Abb. B.3a) zu sehen. Verwendet man für x_1 und x_4 die Shannon-Zerlegung, für x_2 die positive Davio-Zerlegung und für x_3 die negative Davio-Zerlegung, so erhält man das ROKFDD aus Abb. B.3b).

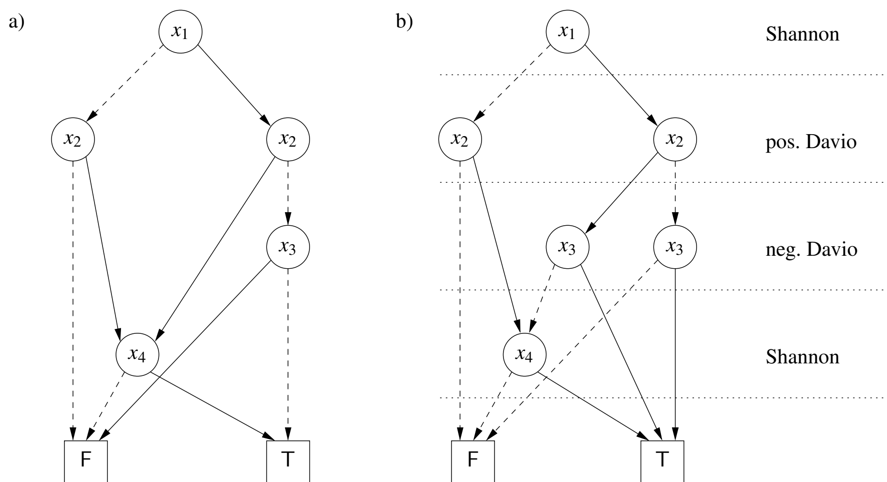


Abb. B.3. a) ROBDD und b)ROKFDD für die Boolesche Funktion aus Beispiel B.3.1

Man erkennt, dass das ROKFDD aus Abb. B.3b) größer als das ROBDD aus Abb. B.3a) ist. Da das ROBDD aber ebenfalls ein ROKFDD mit der Einschränkung auf die ausschließliche Nutzung der Shannon-Zerlegung ist, gilt somit, dass das kleinste ROKFDD bei gegebener Variablenordnung für eine Boolesche nicht größer sein kann als das ROBDD für diese Funktion unter Verwendung der selben Variablenordnung.

C

Algorithmen

In diesem Kapitel werden wichtige Algorithmen zur Verifikation digitaler Systeme beschrieben. Zur Bewertung der Algorithmen ist es jedoch zunächst notwendig, die Probleme, auf welche die Algorithmen angewendet werden, zu klassifizieren.

C.1 Klassifikation von Algorithmen

Ein *Algorithmus* bezeichnet eine Berechnungsvorschrift, die aus einer Menge von Eingaben, Ausgaben und einer *endlichen* Anzahl von *eindeutigen* Berechnungsschritten besteht und die in der Regel in einer endlichen Anzahl von Schritten *terminiert*. Probleme, die durch Algorithmen gelöst werden können, heißen auch *entscheidbar*. Algorithmen können nach a) Qualität der Lösung und b) Berechnungsaufwand klassifiziert werden. Um diese Merkmale näher zu quantifizieren, bedarf es einiger Definitionen. Ein Algorithmus heißt *exakt*, wenn er für alle Instanzen eines Problems eine exakte Lösung findet. Es lassen sich für alle entscheidbaren Probleme exakte Algorithmen finden. Jedoch ist der Berechnungsaufwand von exakten Algorithmen oft zu hoch, um in *vertretbarer Zeit* auf einem Computer gelöst zu werden. Folglich verwendet man sog. *Approximationsalgorithmen*, die nicht das Finden einer exakten Lösung garantieren, aber die in den meisten Fällen exakte Lösungen gut approximieren, also annähern. Approximationsalgorithmen heißen oft *Heuristiken*, da sie Strategien verwenden, die auf Vermutungen, plausiblen Annahmen und Erfahrungen beruhen.

Der Berechnungsaufwand (Komplexität) eines Algorithmus wird in Zeit- und Speicherbedarf im *schlimmsten Fall* (engl. *worst case complexity*) und im *Mittel* (engl. *average case complexity*) gemessen . Im Weiteren werden nur Komplexitäten für den schlimmsten Fall betrachtet, da dieser in der Praxis relevanter ist. Um den zeitlichen Berechnungsaufwand eines Algorithmus unabhängig vom Rechnertyp zu beurteilen, definiert man die Anzahl der elementaren Operationen des Algorithmus als Maß des Rechenaufwands. Da diese Zahl aber von der Problemgröße (die als Eingabeparameter des Algorithmus verstanden werden kann) abhängt, stellt man als Maß der Zeitkomplexität das Wachstum an elementaren Operationen als Funktion

der Problemgröße in Form der sog. $\mathcal{O}$-Notation dar. Bezeichnet man beispielsweise die Problemgröße mit n (z. B. „Sortiere eine Liste von n ganzen Zahlen in aufsteigender Reihenfolge"), dann besitzt die Zeitkomplexität *die Ordnung von* $f(n)$, wenn es eine Konstante c gibt, so dass $c \cdot f(n)$ eine obere Schranke der Anzahl elementarer Operationen zur Lösung des Problems darstellt. Die Bezeichnung der Zeitkomplexität des Algorithmus ist dann $\mathcal{O}(f(n))$. Oft sagt man, dass Algorithmen, bei denen $f(n)$ ein Polynom in n ist (= *polynomielle Algorithmen*, z. B. $\mathcal{O}(n^3 + \frac{1}{2} \cdot n)$), *effizient* sind. Im Gegensatz dazu sind *exponentielle Algorithmen* (z. B. $\mathcal{O}(2^n)$ oder $\mathcal{O}(n^{\frac{n}{2}})$) *ineffizient*. Vorsicht ist bei der Bewertung der Konstanten c geboten, da diese bei kleinen n die Rechenzeit dominieren können.

Die *Effizienz* eines exakten Algorithmus bewertet man durch Vergleich seiner Komplexität mit der dem Problem *inhärenten Komplexität*. Diese bildet eine untere Schranke für die Anzahl benötigter Operationen. Zum Beispiel ist die inhärente Komplexität des Problems, unter n ganzen Zahlen die maximale Zahl zu bestimmen, $\Omega(n)$, da auf jeden Fall $n-1$ Vergleiche notwendig sind. Ein Algorithmus heißt *optimal*, wenn seine Komplexität gleich der inhärenten Komplexität des Problems ist. Folglich ist ein Suchalgorithmus für die größte unter n Zahlen mit Komplexität $\mathcal{O}(n)$ optimal. Algorithmenoptimalität ist strikt von der Optimalität einer Lösung zu unterscheiden.

Manche Probleme lassen sich mit polynomiellen Algorithmen lösen. Diese Klasse von Problemen wird im Allgemeinen mit dem Symbol $\mathcal{P}$ bezeichnet. Leider umfasst diese Klasse nur wenige für die Verifikation von digitalen Hardware/Software-Systemen relevante Probleme. Andere Probleme lassen sich mit polynomiellen Algorithmen auf *nichtdeterministischen Maschinen* lösen. Dies sind hypothetische Computer, welche die Möglichkeit besitzen, Lösungen zu erraten und diese dann in polynomieller Zeit zu verifizieren. Diese Klasse von Problemen heißt $\mathcal{NP}$. Offensichtlich gilt $\mathcal{P} \subseteq \mathcal{NP}$. Die Frage, ob jedoch $\mathcal{P} = \mathcal{NP}$ gilt, ist ein immer noch ungelöstes Problem der Theoretischen Informatik [116, 180]. Es wurde jedoch gezeigt, dass es eine Klasse von Problemen mit der Eigenschaft gibt, dass wenn es irgendein Problem unter ihnen gibt, das in polynomieller Zeit gelöst werden kann, dann alle Probleme dieser Klasse in polynomieller Zeit lösbar sind. Die Klasse dieser Probleme heißt $\mathcal{NP}$*-schwer* und deren Teilklasse, die in der Menge der Probleme $\mathcal{NP}$ enthalten ist, heißt $\mathcal{NP}$*-vollständig*. Abbildung C.1 zeigt die Beziehung zwischen $\mathcal{P}$ und $\mathcal{NP}$. Oft gibt es für Probleme in $\mathcal{NP}$ Algorithmen mit exponentieller (oder höherer) Komplexität. Für manche Problemgrößen mögen diese Algorithmen in ihrer Laufzeit tolerierbar sein, für gewisse Problemgrößen ist man hingegen auf Heuristiken mit polynomieller Laufzeit angewiesen, um in überschaubarer Zeit eine Lösung zu erhalten.

C.2 SAT-Solver

Das *Boolesche Erfüllbarkeitsproblem*, oder kurz *SAT-Problem* (engl. *boolean satisfiability*), ist der Prototyp aller $\mathcal{NP}$-vollständigen Probleme [180]. Es besitzt

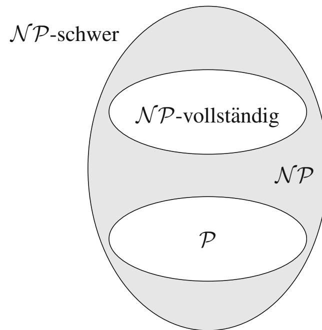


Abb. C.1. Beziehungen zwischen $\mathcal{P}$ und $\mathcal{NP}$

vielfältige Anwendungen im Bereich der Synthese und Verifikation von Computersystemen. Das Boolesche Erfüllbarkeitsproblem kann wie folgt definiert werden:

Definition C.2.1 (Boolesches Erfüllbarkeitsproblem). *Gegeben sei eine Boolesche Funktion f. Entscheide, ob f* erfüllbar *ist.*

Mit anderen Worten: Es soll gezeigt werden, dass mindestens eine Variablenbelegung β existiert, so dass $f(\beta) = \mathsf{T}$. Zur Lösung dieses Problems werden sog. *SAT-Solver* eingesetzt. Bei der Entwicklung von SAT-Solvern gab es in den vergangenen Jahren enorme Fortschritte [476, 366].

Die meisten SAT-Solver arbeiten auf der Repräsentation der Booleschen Funktion als aussagenlogische Formel φ, wobei diese in *konjunktiver Normalform* (*KNF*) gegeben ist. In KNF besteht φ aus der Konjunktion von sog. *Klauseln*. Jede Klausel besteht wiederum aus der Disjunktion von *Literalen*, wobei ein Literal eine Variable oder deren Negation ist. Um eine aussagenlogische Formel in KNF zu erfüllen, muss jede Klausel und somit mindestens ein Literal in jeder Klausel den Wert T annehmen.

Ein Literal, dem noch kein Wert zugewiesen wurde, heißt *unspezifiziert*. Wurde ihm der Wert T zugewiesen, so heißt es *erfüllt*, wurde ihm der Wert F zugewiesen, heißt es *verletzt*. Diese Begrifflichkeit kann auf Klauseln erweitert werden. Eine Klausel heißt *unspezifiziert*, solange sie keinen eindeutigen Wert angenommen hat. Sie heißt *erfüllt*, wenn sie den Wert T annimmt. Sie heißt *verletzt*, wenn sie den Wert F annimmt. Klauseln, bei denen lediglich ein Literal unspezifiziert ist, werden als *Einerklauseln* (engl. *unit clause*) bezeichnet. Diese implizieren direkt die Belegung des Literals.

Beispiel C.2.1. Gegeben ist die Boolesche Funktion $f := (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_5 \vee x_4)$. Weiterhin ist die Belegung $x_1 := \mathsf{T}$, $x_2 := \mathsf{T}$, $x_5 := \mathsf{T}$ und $x_3 := \mathsf{F}$ gegeben. Die Variable x_4 ist noch nicht belegt. Somit sind die Literale $\neg x_1, \neg x_2, x_3$ und $\neg x_5$ verletzt. Die Literale $x_2, \neg x_3$ und x_5 sind erfüllt, während das Literal x_4 unspezifiziert ist. Weiterhin gilt:

$$f = \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{\text{erfüllt}} \wedge \underbrace{(\neg x_2 \vee x_4)}_{\text{unspezifiziert}} \wedge \underbrace{(\neg x_3 \vee x_5)}_{\text{erfüllt}} \wedge \underbrace{(\neg x_5 \vee x_4)}_{\text{unspezifiziert}}$$

Die zweite und vierte Klausel sind Einerklauseln, da lediglich ein Literal (jeweils x_4) unspezifiziert ist. Die Funktion f kann durch die Belegung $x_4 := \mathsf{T}$ erfüllt werden.

Die meisten SAT-Solver basieren auf dem DPLL-Algorithmus [128], der nach seinen Erfindern Davis, Putnam, Longman und Loveland benannt ist. Der grundlegende Algorithmus SAT_SOLVE [174] ist im Folgenden beschrieben.

```
SAT_SOLVE(Φ) {
    IF (DEDUKTION(Φ, β) = F)
        RETURN F;
    WHILE (VERZWEIGUNG(Φ, β) != F)
        WHILE (DEDUKTION(Φ, β) = F)
            blevel := DIAGNOSE(Φ, β);
            IF (blevel = 0)
                RETURN F;
            ELSE
                BACKTRACK(blevel);
    RETURN T;
}
```

Der Algorithmus erhält als Eingabe eine Menge Φ an Klauseln und besteht im Wesentlichen aus den drei Funktionen DEDUKTION, VERZWEIGUNG und DIAGNOSE. Zu Beginn sind alle Variablen *frei*, d. h. nicht belegt, und somit alle Literale unspezifiziert. Im ersten Schritt wird eine Vorverarbeitung durch die Funktion DEDUKTION durchgeführt. Diese reduziert die Menge Φ durch sukzessives Auffinden von Einerklauseln und Belegung der zugehörigen unspezifizierten Variablen, so dass die jeweilige Einerklausel erfüllt ist und von der Menge der Klauseln entfernt werden kann. Durch diese Variablenbelegung zur Erfüllung einer Einerklausel kann es zu einem Konflikt kommen, indem dadurch eine andere Klausel verletzt wird. In diesem Fall ist die Boolesche Funktion nicht erfüllbar und DEDUKTION liefert F zurück. Andererseits können durch die Variablenbelegung neue Einerklauseln entstehen. Die sukzessive Reduktion durch Behandlung der Einerklauseln wird als engl. *unit propagation* oder engl. *Boolean Constraint Propagation* (*BCP*) bezeichnet. Neben BCP werden in der Funktion DEDUKTION oftmals auch sog. *reine Literale* (engl. *pure literals*) eliminiert. Hierbei wird eine Variable, die lediglich als positives (negatives) Literal in den Klauseln auftritt, mit T (F) belegt. Die Variablenbelegungen und deren Reihenfolge werden in der Belegung β gespeichert, um später eventuelle Fehlentscheidungen zu revidieren.

Der Hauptteil des SAT_SOLVE-Algorithmus besteht aus einer WHILE-Schleife, die mit dem Aufruf der Funktion VERZWEIGUNG beginnt. Hierin wird eine freie Variable gewählt und diese mit einem Wert, dem sog. *Verzweigungsliteral*, belegt. Die Auswahl des Verzweigungsliterals hat entscheidenden Einfluss auf die Geschwindigkeit des SAT-Solvers, da manche Entscheidungen schneller zu einer

Lösung führen als andere. Solange die Funktion VERZWEIGUNG freie Variablen zum Verzweigen findet, liefert diese T zurück. Gibt es allerdings keine freien Variablen mehr ist eine Belegung der Variablen gefunden, welche die Boolesche Funktion erfüllt.

Nach der Verzweigung wird die Funktion DEDUKTION aufgerufen und dabei alle Einerklauseln durch BCP eliminiert solange bis keine Einerklauseln mehr gefunden werden oder ein Konflikt durch eine verletzte Klausel entsteht. Im konfliktfreien Fall wird eine neue Verzweigung durchgeführt, während im Konfliktfall die Funktion DIAGNOSE den Grund für den Konflikt analysiert, das Ergebnis lernt (siehe unten) und schließlich eine *Zurückverfolgung* (engl. *backtracking*) durchgeführt wird. Die Zurückverfolgung in der Funktion BACKTRACK macht zuvor getroffene Entscheidungen, die in β gespeichert sind, bis zu einem bestimmten Punkt (blevel) rückgängig. Wird hierbei erkannt, dass bis zur Entscheidung blevel = 0 zurückgegangen werden muss, so ist die Funktion nicht erfüllbar. Fortschrittliche SAT-Solver führen dabei ein sog. *Rücksprungverfahren* (engl. *backjumping*) oder eine *nichtchronologische Zurückverfolgung* durch [474, 314, 336, 201] (siehe unten).

Die Effizienz von SAT-Solvern wird in der Anzahl der benötigten Zurückverfolgungsschritte gemessen, bis eine konsistente Belegung gefunden wurde oder die Formel als unerfüllbar identifiziert wurde. Wird eine schlechte Verzweigungsstrategie gewählt, so sind häufiger Zurückverfolgungsschritte durchzuführen. Der Algorithmus ist noch einmal in Abb. C.2 graphisch dargestellt. Im Folgenden werden die einzelnen Funktionen genauer betrachtet.

Verzweigungsstrategien

Die Wahl des Verzweigungsliterals hat einen großen Einfluss auf die Effizienz des SAT-Solvers. Sehr einfache Verzweigungsstrategien basieren auf der Häufigkeit des Auftretens von Literalen in den betrachteten Klauseln. Sei $h_m(a)$ die Häufigkeit mit der das Literal a in Klauseln der Länge m auftritt, dann lässt sich die absolute Häufigkeit $h(a)$ bestimmen zu:

$$h(a) := \sum_{m:=1}^{M} h_m(a)$$

wobei M die Länge der längsten Klausel ist. Diese Häufigkeiten aller Literale können einmal vor dem Start des SAT-Solvers oder nach jeder Verzweigung neu bestimmt werden. Daneben existieren auch Verfahren, die unabhängig von der Häufigkeit der Literale arbeiten. Ziel aller Verzweigungsstrategien ist es jedoch, bei der Verzweigung möglichst kurze, im besten Fall viele Einerklauseln, zu erzeugen. Im Folgenden werden einige einfache Strategien betrachtet (siehe auch [313]):

Die *DLIS-Strategie* (engl. *Dynamic Largest Individual Sum*) wählt dasjenige Literal a, für das $h(a)$ maximal wird. Setzt man $a := \mathsf{F}$, so wird eine maximale Anzahl an Klauseln verkürzt. Durch Setzen von $a := \mathsf{T}$ können andererseits möglichst viele Klauseln erfüllt werden und somit die Menge aller noch zu betrachtenden Klauseln verkleinert werden. Eine Erweiterung der DLIS-Strategie ist die *DLCS-Strategie*

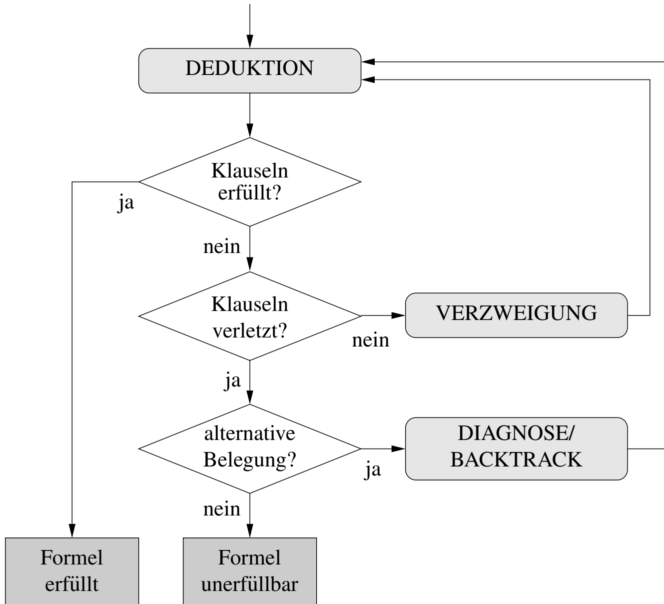


Abb. C.2. DPLL-Algorithmus

(engl. *Dynamic Largest Combined Sum*). Bei der DLCS-Strategie wird die maximale Summe aus negierten und positiven Literal einer Variable x bestimmt, d. h. $h(x) + h(\neg x)$. Anschließend wird $x := \mathsf{T}$ gesetzt, falls $h(x) \geq h(\neg x)$ ist, um so eine große Anzahl an Klauseln zu erfüllen. Andernfall ($h(x) < h(\neg x)$) wird $x := \mathsf{F}$ gesetzt.

Die *MOM-Strategie* (engl. *Maximum Occurrences in Minimal clauses*) wählt diejenige Variable, die am häufigsten in den kürzesten unspezifizierten Klausel vorkommt [141]. Gibt es mehrere Variablen, welche die gleiche Häufigkeit besitzen, wird das Produkt der Häufigkeiten von negierten und positiven Vorkommen dieser Variablen in den kürzesten Klauseln als zweitrangiges Kriterium herangezogen. Somit werden Variablen bevorzugt, die möglichst gleich häufig negiert und positiv auftreten. Dies lässt sich wie folgt berechnen:

$$(h_l(x) + h_l(\neg x)) \cdot 2^{\alpha} + h_l(x) \cdot h_l(\neg x)$$

Dabei ist l die Länge der kürzesten Klausel und α eine frei wählbare Konstante, die so gewählt wird, dass der erste den zweiten Term dominiert.

Eine Verzweigungsstrategie mit der Bezeichnung *VSIDS-Strategie* (engl. *Variable State Independent Decaying Sum*) [336] verzweigt anhand von Klauseln, die während der Diagnose gelernt werden (siehe unten). Bei der VSIDS-Strategie werden zu Beginn die Häufigkeiten $h(x)$ und $h(\neg x)$ für die Variable x ermittelt. Jedes Mal, wenn eine Klausel hinzugefügt wird (z. B. durch die Diagnose), werden $h(x)$ und $h(\neg x)$ in Abhängigkeit des Auftretens des negativen oder positiven Literals in der neuen Klausel um eine Konstante erhöht. Um neu hinzugefügte Klauseln stärker

zu gewichten, werden $h(x)$ und $h(\neg x)$ für jede Variable periodisch durch eine gegebene Konstante dividiert. Als Verzweigungsliteral wird dasjenige Literal mit dem höchsten Wert ausgewählt. Obwohl diese Werte nicht direkt die Häufigkeit von Literalen in unspezifizierten Klauseln widerspiegeln, ist die VSIDS-Strategie in der Praxis sehr effizient. Eine Erweiterung der VSIDS-Strategie verwendet anstatt der Literale in den neu hinzugefügten Klauseln die Literale aus den Klauseln, die verletzt worden sind [201].

Deduktion

Zu Beginn und nach jeder Verzweigung wird mittels Deduktion versucht, die aussagenlogische Formel zu reduzieren. Hierzu stehen unterschiedliche Verfahren zur Verfügung.

Eine Großteil der Rechenzeit (bis zu 80%) von SAT-Solvern verbraucht die BCP-Phase (engl. *Boolean Constraint Propagation*). Somit ist jede Verbesserung in BCP eine signifikante Verbesserung für den SAT-Solver. Ein Verfahren, das auf der Beobachtung von genau zwei Literalen basiert, ist in [336] vorgestellt. Die Vorteile dieses Verfahrens sind, dass nicht alle Literale in allen Klauseln beobachtet und Literale und Klauseln nicht gelöscht werden müssen sowie die Zurückverfolgung in konstanter Zeit möglich ist.

Zunächst werden für jede Klausel zwei Literale zur Beobachtung ausgewählt. Die Beobachtung heißt *zulässig*, wenn beide Literale unspezifiziert sind oder mindestens ein Literal erfüllt ist. Durch eine Verzweigung und anschließende BCP kann eine Beobachtung *unzulässig* werden. In diesem Fall müssen die verletzten beobachteten Literale durch neue beobachtete Literale ersetzt werden. Existiert keine zulässige Beobachtung für eine Klausel kann dies zwei Gründe haben: Erstens, die Klausel ist eine Einerklausel und lässt sich durch BCP erfüllen. Zweitens, die Klausel ist unter der gegebenen partiellen Variablenbelegung nicht erfüllbar. Es muss somit eine Zurückverfolgung durchgeführt werden. Dieses kann einfach durch Löschen der partielle Variablenbelegung ab der letzten Verzweigung erfolgen.

Beispiel C.2.2. Gegeben ist die Boolesche Funktion $f := (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Die Variablenbelegung β ist zu Beginn leer, d. h. $\beta = \emptyset$. Die Klauseln und die beobachteten Variablen v_1 und v_2 zu jeder Klausel sind in der folgenden Tabelle angegeben:

Klausel	v_1	v_2
$x_1 \vee x_2 \vee x_3$	$x_1 = -$	$x_2 = -$
$\neg x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 = -$	$x_3 = -$
$x_1 \vee \neg x_2 \vee x_4$	$x_1 = -$	$\neg x_2 = -$
$x_1 \vee x_3 \vee \neg x_4$	$x_1 = -$	$x_3 = -$

Hierbei zeigt der Wert $-$ ein unspezifiziertes Literal an. Bei der ersten Verzweigung wird $x_1 := \mathsf{F}$ gesetzt, d. h. $\beta = \{\neg x_1\}$. Hierdurch wird die zweite Klausel erfüllt und die zugehörige Beobachtung kann durch folgende Variablenbelegungen nicht mehr unzulässig werden. Die Beobachtungen für die anderen drei Klauseln werden unzulässig, weshalb neue beobachtete Literale anstelle von x_1 gewählt werden müssen.

Klausel	v_1	v_2
$x_1 \vee x_2 \vee x_3$	$x_3 = -$	$x_2 = -$
$\neg x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 = \mathsf{T}$	$x_3 = -$
$x_1 \vee \neg x_2 \vee x_4$	$x_4 = -$	$\neg x_2 = -$
$x_1 \vee x_3 \vee \neg x_4$	$\neg x_4 = -$	$x_3 = -$

In der nächsten Verzweigung wird die Variable $x_3 := \mathsf{F}$ gesetzt, d. h. $\beta = \{\neg x_1, \neg x_3\}$. Hierdurch werden die Beobachtungen der Klauseln eins und vier unzulässig.

Klausel	v_1	v_2
$x_1 \vee x_2 \vee x_3$	$x_3 = \mathsf{F}$	$x_2 = -$
$\neg x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 = \mathsf{T}$	$x_3 = \mathsf{F}$
$x_1 \vee \neg x_2 \vee x_4$	$x_4 = -$	$\neg x_2 = -$
$x_1 \vee x_3 \vee \neg x_4$	$\neg x_4 = -$	$x_3 = \mathsf{F}$

Beide Klauseln sind nun Einerklauseln und implizieren $x_2 := \mathsf{T}$ sowie $x_4 := \mathsf{F}$. Somit ergibt sich die Belegung $\beta = \{\neg x_1, \neg x_3, x_2, \neg x_4\}$. Dies verletzt allerdings Klausel drei, weshalb eine Zurückverfolgung durchgeführt werden muss. Ausgehend von der letzten Verzweigung wird nun $x_3 := \mathsf{T}$ gewählt, was zu einer partiellen Belegung $\beta = \{\neg x_1, x_3\}$ führt.

Weitere Reduktionen sind die Elimination reiner Literale (engl. *pure literals*) aus der Klauselmenge Φ. Dabei wird ein Literal x als *rein* bezeichnet, wenn $\neg x$ in keiner Klausel in Φ vorkommt. Die Belegung $x := \mathsf{F}$ muss in diesem Fall nicht betrachtet werden. Dies liegt daran, dass wenn Φ unter der Belegung $x := \mathsf{F}$ erfüllbar ist, so ist Φ auch unter $x := \mathsf{T}$ erfüllbar.

Ein Reduktionsverfahren, welches keine Variablenbelegung benötigt, ist die *Subsumtion*. Eine Klausel c_1 subsumiert eine Klausel c_2, wenn jedes Literal in c_1 auch in c_2 vorkommt. Eine Variablenbelegung die c_1 erfüllt, erfüllt in diesem Fall automatisch auch c_2. Die Klausel c_2 kann aus der Menge aller Klauseln gelöscht werden.

Die *Resolution* ist ein Reduktionsverfahren, welches die Klauselmenge vergrößert. Enthält Φ zwei Klauseln mit komplementären Literalen, z. B. $x \vee c_1$ und $\neg x_1 \vee c_2$, so kann die Klausel $c_1 \vee c_2$ zu Φ hinzugefügt werden. Dies verändert die Erfüllbarkeit von Φ nicht, da $(x := \mathsf{T}) \Rightarrow (c_2 = \mathsf{T})$ und $(x_1 := \mathsf{F}) \Rightarrow (c_1 = \mathsf{T})$. Hieraus können weitere Regeln abgeleitet werden. Enthält die Menge Φ die Klauseln $x_1 \vee x_2$ und $\neg x_1 \vee x_2$, so muss $x_2 := \mathsf{T}$ sein. Enthält Φ die Klauseln $x \vee c_1$ und $\neg x_1 \vee c_2$, wobei gilt, dass $c_1 \vee c_2$ eine Klausel c subsumiert, so kann c aus Φ gelöscht werden.

Diagnose

Ein Konflikt während der Erfüllbarkeitsanalyse tritt auf, wenn während des BCP eine Variable gleichzeitig mit T und F, also gegensätzlichen Werten, belegt werden soll. Eine Möglichkeit diesen Konflikt zu beseitigen, besteht darin, die gewählte Belegung aus der letzten Verzweigung rückgängig zu machen. Dies wird als *chronologische Zurückverfolgung* bezeichnet. Die chronologische Zurückverfolgung hat sich allerdings nicht als besonders effektiv herausgestellt, wenn es darum geht, Bereiche, die lediglich ungültige Lösungen enthalten, von der Suche auszuschließen.

Größere Teile des Suchraums können beschnitten werden, wenn vor der Zurückverfolgung eine *Konfliktanalyse* durchgeführt wird. Hierbei wird der Grund für den Konflikt ermittelt und entsprechend frühere Entscheidungen revidiert. Solche Verfahren werden als *konfliktgetriebenes Rücksprungverfahren* [367, 33] bezeichnet. Es wird also eine *nichtchronologische Zurückverfolgung* durchgeführt.

Zur Konfliktanalyse wird häufig ein sog. *Implikationsgraph* eingesetzt. Ein Implikationsgraph ist ein gerichteter azyklischer Graph. Knoten repräsentieren Wertzuweisungen an Variablen, wobei Knoten ohne Vorgänger Entscheidungen aus den Verzweigungen des SAT-Solvers darstellen. Kanten repräsentieren Implikationen. Ein *konfliktfreier Implikationsgraph* enthält für jede Variable höchstens einen Knoten. Existieren in einem Implikationsgraphen zwei Knoten mit unterschiedlichen Zuweisungen für eine Variable x, so enthält der Implikationsgraph einen Konflikt. Die Variable x wird als widersprüchlich bezeichnet. Die zugehörigen Knoten heißen Konfliktknoten.

Neben der Zurückverfolgung führen SAT-Solver häufig auch ein *konfliktgetriebenes Lernen* durch [475], d. h. der SAT-Solver speichert den Grund für den Konflikt ab, damit der selbe Fehler an einer anderen Stelle nicht wiederholt wird. Hierzu wird eine Bipartition des Implikationsgraphen erstellt, wobei die Konfliktknoten in dem einen Partitionsblock und die Knoten, die Entscheidungen des SAT-Solvers aus Verzweigungen repräsentieren, in dem anderen Partitionsblock liegen. Die Bipartition erzeugt einen *Schnitt* im Implikationsgraphen. Alle Kanten, die diesen Schnitt passieren und die auf einem Pfad zu den Konfliktknoten liegen, repräsentieren die Variablenbelegung, die zu dem Konflikt geführt hat. Um diesen Konflikt zukünftig zu vermeiden, wird eine zusätzliche Klausel, die den Konflikt repräsentiert, der Klauselmenge hinzugefügt. Dies wird an einem Beispiel verdeutlicht [174].

Beispiel C.2.3. Gegeben sind die folgenden sieben Klauseln einer Booleschen Funktion $f := c_1 \wedge \cdots \wedge c_7$ in konjunktiver Normalform:

$$\begin{aligned}
c_1 &: \neg x_1 \vee x_2 \vee x_6 \\
c_2 &: x_2 \vee x_3 \vee \neg x_7 \\
c_3 &: x_3 \vee \neg x_4 \vee x_8 \\
c_4 &: \neg x_1 \vee \neg x_5 \vee \neg x_6 \\
c_5 &: \neg x_6 \vee x_7 \vee \neg x_8 \vee \neg x_9 \\
c_6 &: x_5 \vee x_9 \vee x_{10} \\
c_7 &: x_9 \vee \neg x_{10}
\end{aligned}$$

Der resultierende Implikationsgraph nach den Wertzuweisungen $x_1 = x_4 := \mathsf{T}$ und $x_2 = x_3 := \mathsf{F}$ ist in Abb. C.3 zusehen. Der Implikationsgraph enthält einen Konflikt für die Variable x_{10}. Die Kanten sind zusätzlich mit den Klauseln markiert, welche die Implikation verursachen.

In Abb. C.3 ist weiterhin eine mögliche Bipartition zum konfliktgetriebenen Lernen gezeigt. Die Kanten, die den Schnitt passieren, repräsentieren das Weiterleiten der Variablenbelegung $x_1 = x_8 := \mathsf{T}$ und $x_2 = x_3 := \mathsf{F}$. Um diesen Konflikt zu lernen, wird die folgende Klausel erzeugt:

$$c_8 := \neg x_1 \vee x_2 \vee x_3 \vee \neg x_8$$

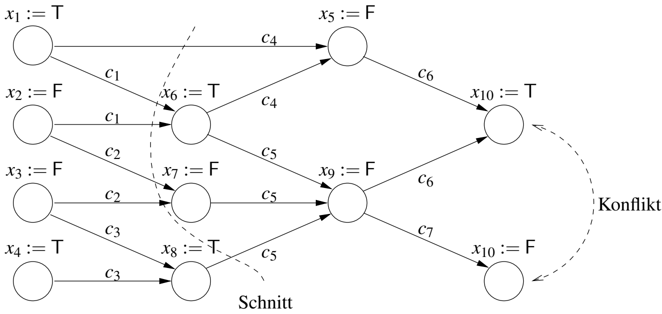


Abb. C.3. Konfliktbehafteter Implikationsgraph [174]

und konjunktiv mit der Funktion f verknüpft, d. h. $f := f \wedge c_8$.

An diesem Beispiel sieht man bereits, dass man mehrere Bipartitionen bilden und somit unterschiedliche Klauseln zum Lernen des Konflikts hinzufügen kann. Besonders interessant sind hierbei der sog. *UIP* (engl. *Unique Implication Point*). Ein UIP ist ein Knoten im Konfliktgraphen, durch den alle Pfade von Knoten, die Entscheidungen aus Verzweigungen repräsentieren, zu Konfliktknoten verlaufen.

Verbesserungen von SAT-Solvern

Neben den oben beschriebenen Techniken gibt es eine Vielzahl von möglichen Verbesserungen für SAT-Solver. Die Wichtigsten werden im Folgenden kurz vorgestellt.

Die ersten Entscheidungen, die ein SAT-Solver trifft, können einen großen Einfluss auf die Laufzeit des SAT-Solvers haben. So kann beispielsweise eine schlechte Wahl zu Beginn den SAT-Solver in einen Bereich des Suchraums versetzen, in dem keine konsistente Variablenbelegung existiert und den er nur schwer verlassen kann. Dies a priori zu erkennen, ist im Allgemeinen nicht möglich. Aus diesem Grund werden heutige SAT-Solver wiederholt zufällig neu gestartet, wobei alle Entscheidungen rückgängig gemacht werden [336]. Allerdings werden gelernte Klauseln dabei nicht wieder vergessen.

Eine Variation der zufälligen Neustarts ist ein Rücksprungverfahren, das eingeleitet wird, ohne dass ein Konflikt aufgetreten ist [362]. Die Motivation ist, dass der SAT-Solver schnell Bereiche des Suchraums verlassen soll, die viele Konflikte enthalten. Treten also Konflikte gehäuft auf, wird eine große Anzahl an getroffen Entscheidungen rückgängig gemacht.

Auf der anderen Seite können gelernte Klauseln die Laufzeit des SAT-Solvers verlängern. Dies liegt daran, dass gelernte Klauseln während der BCP ebenfalls aktualisiert werden müssen. Da gelernte Klauseln aber redundant sind, sie also nicht die Erfüllbarkeit der gegebenen Booleschen Funktion ändern, können sie auch wieder gelöscht werden, ohne das Ergebnis zu verändern. Aus diesem Grund berechnen viele SAT-Solver die Relevanz gelernter Klauseln. Als Relevanzkriterien kommen z. B.

die Anzahl unspezifizierter Literale in einer Klausel [336] oder aber die Häufigkeit mit der eine Klauseln in einen Konflikt verwickelt ist [201] zum Einsatz.

Eine weitere Verbesserung von SAT-Solvern kann erreicht werden, wenn gelernte Klauseln möglichst kurz sind. Dies liegt daran, dass kürzere Klauseln den Suchraum stärker beschneiden. Ein Verfahren, welches eine notwendige Teilmenge an Literalen bestimmt, welche den Konflikt erzeugt, ist in [340] vorgestellt. Wenn die zugehörigen Variablen als Verzweigungsliterale verwendet werden, kann ein Konflikt frühzeitig detektiert werden.

C.3 SMT-Solver

SAT-Solver werden u. a. vermehrt zur Verifikation von Hardware und Software auf höheren Abstraktionsebenen eingesetzt. Hierzu werden Eigenschaften der betrachteten Systeme häufig in Aussagenlogik übersetzt, um diese z. B. mittels SAT-Solvern zu beweisen. Dies kann allerdings zu sehr großen aussagenlogischen Formeln führen, sofern überhaupt eine Übersetzung existiert, da nicht alle Eigenschaften in Aussagenlogik dargestellt werden können. Obwohl sich die verfügbaren SAT-Solver enorm verbessert haben, wäre es natürlicher, die Eigenschaften der Systeme in Logiken zu beschreiben, die expressiver als Aussagenlogik sind.

Eine Möglichkeit hierzu ist die Prädikatenlogik erster Ordnung mit Äquivalenz. Um die Erfüllbarkeit prädikatenlogischer Formeln zu zeigen, können beispielsweise Theorembeweiser eingesetzt werden. Leider ist der Einsatz allgemeiner Theorembeweiser für viele praktische Probleme oft inadäquat.

Dies kann am folgendem Beispiel verdeutlicht werden. Gegeben sei die folgende Formel [397]:

$$\begin{aligned}\varphi := {} & [a_1 \vee (u - w \leq 5)] \wedge \\ & [a_2 \vee (v + w) \leq 6)] \wedge \\ & [a_3 \vee (z = 0)] \wedge \\ & [a_4 \vee (u + v \geq 12)] \wedge \\ & [\neg a_3 \vee \neg a_4] \wedge \\ & [(x = z + 1) \vee (x = z + 3) \vee (x = z + 5) \vee (x = z + 7)] \wedge \\ & [(y = z + 2) \vee (y = z + 4) \vee (y = z + 6)] \wedge \\ & [(u + v - 4 \cdot x - 4 \cdot y = 0)] \end{aligned} \tag{C.1}$$

für $a_1, a_2, a_3, a_4 \in \mathbb{B}$ und $u, v, w, x, y, z \in \mathbb{R}$. Dabei ist man typischerweise nicht daran interessiert, dass die Formel für alle möglichen Interpretationen der Theoriezeichen $\leq, =$ und $+, \cdot$ erfüllbar ist, sondern lediglich an dem Fall, wo $\leq$ ($\geq$) die „kleiner gleich“-Relation („größer gleich“), $=$ die Gleichheitsrelation und $+$ die Addition und $\cdot$ die Multiplikation von reellen Zahlen beschreiben. Dies bedeutet, die Erfüllbarkeit der Formel wird bezüglich einer sog. *Hintergrundtheorie* $\mathcal{T}$ entschieden. In obigem Beispiel handelt es sich bei der Hintergrundtheorie um Lineare Arithmetik

reellwertiger Zahlen (engl. *Linear Real Arithmetics, LRA*). Um die Hintergrundtheorie durch einen allgemeinen Theorembeweiser berücksichtigen zu lassen, müssen die Axiome, welche die verwendete Hintergrundtheorie definieren, konjunktiv mit der Formel verknüpft werden. Dies ist allerdings nicht immer möglich. Falls es dennoch möglich ist, wird die Geschwindigkeit des Theorembeweisers oft stark gebremst.

Bei dem obigen Problem wird also versucht, die Frage zu beantworten, ob eine prädikatenlogische Formel φ bezüglich einer entscheidbaren Hintergrundtheorie $\mathcal{T}$ erfüllbar ist. Aus diesem Grund spricht man auch vom *Erfüllbarkeitsproblem modulo Theorien* (engl. *Satisfiability Modulo Theories, SMT*). Dabei können durchaus mehrere Hintergrundtheorien kombiniert werden. Beispiele für Hintergrundtheorien sind: Gleichheit und uninterpretierte Funktionen, Lineare Arithmetik über reelle und ganze Zahlen, Divergenzlogik, Bitvektor-Theorie und Array-Theorie.

Programme zum Lösen des SMT-Problems werden als *SMT-Solver* bezeichnet. Die Kombination aus Theorien wurde bereits in den wegweisenden Arbeiten von Nelson und Oppen [342] und Shostak [399, 400] diskutiert. Ein wirklich starkes Interesse an der Entwicklung von SMT-Solvern ist allerdings erst in den letzten Jahren entstanden [392].

Die meisten heute verwendeten SMT-Solver sind sog. *indirekte* SMT-Solver. Indirekte SMT-Solver kombinieren DPLL-SAT-Solver mit bestehenden Theorielösern. Der SAT-Solver steuert den kombinierten Lösungsansatz, indem er atomare Formeln (aussagenlogische Variablen und Formeln in der Hintergrundtheorie) mit Booleschen Werten belegt und den Löser für die Hintergrundtheorie regelmäßig beauftragt, die Erfüllbarkeit der Formeln der Hintergrundtheorie zu überprüfen. Dies steht im Gegensatz zu den oben beschriebenen direkten SMT-Solvern, welche die Eigenschaften in aussagenlogische Formeln übersetzen und somit die Hintergrundtheorie mit der aussagenlogischen Formel verknüpfen. Abbildung C.4 zeigt die Arbeitsweise eines indirekten SMT-Solver für prädikatenlogische Formeln.

Ausgehend von der prädikatenlogischen Formel φ, wird zunächst eine aussagenlogische Abstraktion durchgeführt. Dabei werden die atomaren prädikatenlogischen Formeln durch Boolesche Variablen repräsentiert. Die resultierende aussagenlogische Formel φ_a ist die Eingabe des verwendeten SAT-Solvers. Ist das Ergebnis des SAT-Solvers, dass φ_a nicht erfüllbar ist, so ist auch die prädikatenlogische Formel φ unerfüllbar. Findet der SAT-Solver hingegen eine konsistente Belegung für die aussagenlogische Formel φ_a, so werden die atomaren prädikatenlogischen Formeln Φ, die durch die Belegung impliziert werden, an den Theorielöser weitergereicht. Der Theorielöser sucht nun seinerseits für die Konjunktion der übergebenen Formeln eine konsistente Belegung entsprechend der verwendeten Hintergrundtheorie $\mathcal{T}$. Wird eine solche konsistente Belegung gefunden, bildet die Vereinigungsmenge dieser Belegung zusammen mit der Belegung der atomaren aussagenlogischen Formeln aus φ die konsistente Belegung für φ. Wird eine solche Belegung hingegen nicht gefunden, führt der Theorielöser eine Konfliktanalyse durch und verfeinert die abstrahierte aussagenlogische Formel entsprechend.

Beispiel C.3.1. Die Arbeitsweise eines indirekten SMT-Solvers wird anhand des Beispiels aus Gleichung (C.1) demonstriert [397]. Zunächst wird die aussagenlogische

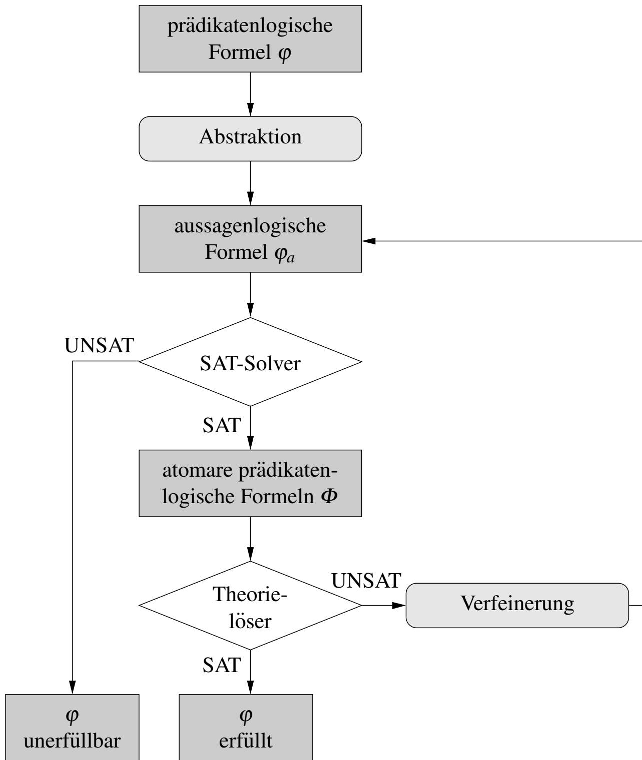


Abb. C.4. Indirekter SMT-Solver

Abstraktion durchgeführt. Hierzu werden zusätzliche Boolesche Variablen b_i für die atomaren prädikatenlogischen Formeln eingeführt. Die resultierende aussagenlogische Formel φ_a sieht wie folgt aus:

$$\begin{aligned}\varphi_a = {} & [a_1 \vee b_1] \wedge [a_2 \vee b_2] \wedge [a_3 \vee b_3] \wedge [a_4 \vee b_4] \wedge \\ & [\neg a_3 \vee \neg a_4] \wedge [b_{5,1} \vee b_{5,2} \vee b_{5,3} \vee b_{5,4}] \wedge [b_{6,1} \vee b_{6,2} \vee b_{6,3}] \wedge [\mathsf{T}]\end{aligned}$$

Man beachte, dass die letzte atomare prädikatenlogische Formel aus Gleichung (C.1) unbedingt erfüllt sein muss, weshalb keine neue Boolesche Variable eingeführt wurde. Die neuen Booleschen Variablen b_i implizieren das Ergebnis der einzelnen Relationen:

$$\begin{array}{ll} b_1 \Rightarrow (u - w \leq 5) & b_{5,3} \Rightarrow (x = z + 5) \\ b_2 \Rightarrow (v + w \leq 6) & b_{5,4} \Rightarrow (x = z + 7) \\ b_3 \Rightarrow (z = 0) & b_{6,1} \Rightarrow (y = z + 2) \\ b_4 \Rightarrow (u + v \geq 12) & b_{6,2} \Rightarrow (y = z + 4) \\ b_{5,1} \Rightarrow (x = z + 1) & b_{6,3} \Rightarrow (y = z + 6) \\ b_{5,2} \Rightarrow (x = z + 3) & \mathsf{T} \Rightarrow (u + v - 4 \cdot x - 4 \cdot y = 0) \end{array}$$

Nach Abb. C.4 wird zunächst der SAT-Solver aufgerufen, um die aussagenlogische Formel φ_a auf Erfüllbarkeit zu testen. Eine mögliche Belegung β, die φ_a erfüllt, wäre z. B.

$$\beta = \{\neg a_1, \neg a_2, \neg a_3, \neg a_4, b_{5,1}, b_{6,1}\} .$$

Durch BCP vervollständigt sich die Belegung β zu:

$$\beta = \{\neg a_1, b_1, \neg a_2, b_2, \neg a_3, b_3, \neg a_4, b_4, b_{5,1}, b_{6,1}\}$$

Dies impliziert die folgende Menge atomarer prädikatenlogischer Formeln:

$$\begin{aligned}\Phi = \{&(u - w \leq 5), (v + w \leq 6), (z = 0), (u + v \geq 12), (x = z + 1),\\ &(y = z + 2), (u + v - 4 \cdot x - 4 \cdot y = 0)\}\end{aligned}$$

Die Konjunktion der Formeln wird vom Theorielöser (in diesem Fall ein LRA-Solver, engl. *Linear Reel Arithmetic*) auf Erfüllbarkeit überprüft. Die oben gegebene Formelmenge Φ ist nicht erfüllbar. Die anschließende Konfliktanalyse (siehe Abb. C.5) ergibt, dass die Belegung $b_1 = b_2 = b_4 := \top$ zu einem Konflikt geführt hat. Aus diesem Grund wird zur Verfeinerung der aussagenlogischen Formel φ_a die Klausel $\neg b_1 \vee \neg b_2 \vee \neg b_4$ ($\neg(b_1 \wedge b_2 \wedge b_4)$) hinzugefügt.

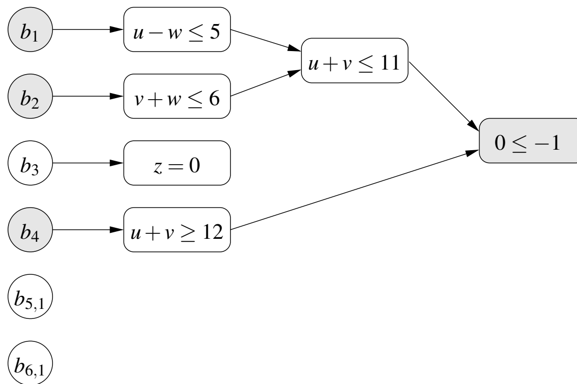


Abb. C.5. Konfliktanalyse im LRA-Solver [397]

In Beispiel C.3.1 konnte der Theorielöser keine konsistente Belegung der Variablen der prädikatenlogischen Formeln finden. In diesem Fall sagt man, dass das Modell der aussagenlogischen Formel $\mathcal{T}$*-inkonsistent* ist. Andernfalls heißt das Modell $\mathcal{T}$*-konsistent*.

In dem oben skizzierten Vorgehen werden lediglich vollständige Modelle der aussagenlogischen Formel auf $\mathcal{T}$-Konsistenz geprüft. Eine mögliche Verbesserung des SMT-Solvers besteht darin, auch partielle Belegungen auf $\mathcal{T}$-Konsistenz zu prüfen.

Beispiel C.3.2. Dies wird anhand des Beispiels C.3.1 verdeutlicht, nachdem die Klausel $\neg b_1 \vee \neg b_2 \vee \neg b_4$ vom SMT-Solver gelernt wurde. Abbildung C.6 zeigt die einzelnen Entscheidungen des SAT-Solvers sowie die Interaktion des SAT-Solvers mit dem Theorielöser.

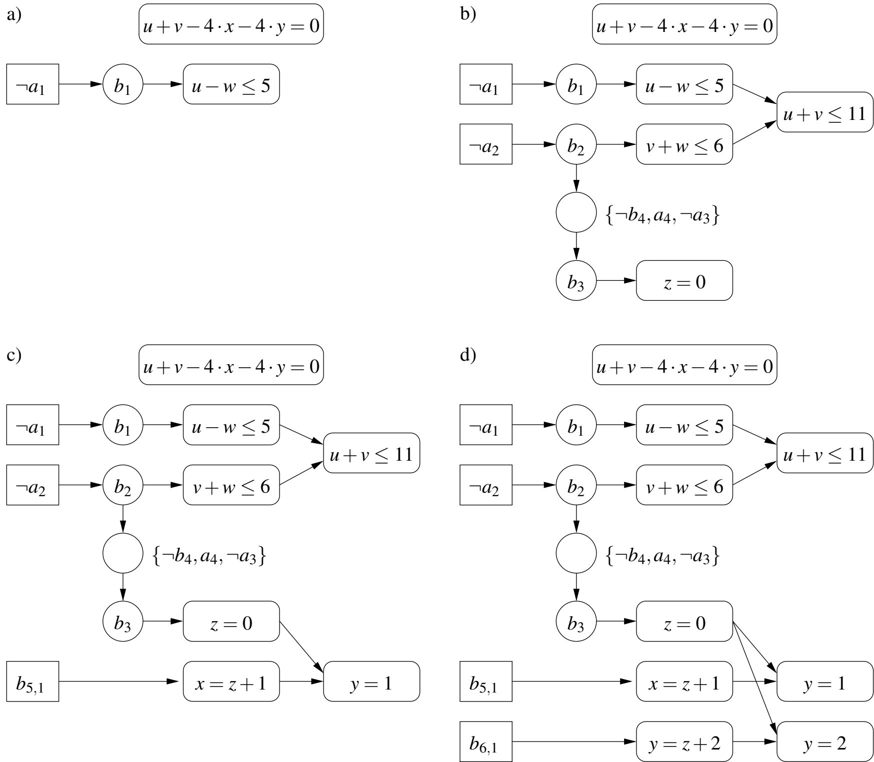


Abb. C.6. Schnelle Konfliktanalyse

Im ersten Schritt (Abb. C.6a)) belegt der SAT-Solver die Boolesche Variable a_1 mit dem Wert F. Durch BCP wird die Variable b_1 mit dem Wert T belegt. Da $b_1 = \mathsf{T}$ eine atomare prädikatenlogische Formel impliziert, wird sofort der LRA-Solver aufgerufen. Neben der Formel $(u - w \leq 5)$ muss der LRA-Solver auch die unbedingte Formel $(u + v - 4 \cdot x - 4 \cdot y = 0)$ erfüllen. Da dies möglich ist, wird die Kontrolle an den SAT-Solver zurück gegeben.

Im zweiten Schritt weist der SAT-Solver der Booleschen Variablen a_2 den Wert F zu (Abb. C.6b)). BCP führt zur Zuweisung $b_2 := \mathsf{T}$. Diese Belegung führt wiederum zu einem Aufruf des LRA-Solver, der nun zusätzlich die Formel $(u + w \leq 6)$ erfüllen muss. Da auch dies möglich ist, wird die Kontrolle an den SAT-Solver zurück gegeben. Noch im selben Schritt führen die Implikationen dazu, dass $\{\neg b_4, a_4, \neg a_3, b_3\}$

zur Belegung der Booleschen Variablen hinzugefügt wird, wobei b_3 wiederum zu einer Implikation führt und den Aufruf des Theorielösers erzwingt. Dieser bekommt als zusätzliche Formel $(z = 0)$, welche zusammen mit den bereits implizierten Formeln erfüllbar ist.

Im dritten Schritt (Abb. C.6c)) wird der Booleschen Variablen $b_{5,1}$ der Wert T durch den SAT-Solver zugewiesen. Dies führt dazu, dass zu Φ die Formel $(x = z + 1)$ hinzugefügt wird. Diese Formel vereinfacht sich mit $(z = 0)$ zu $(x = 1)$. Auch jetzt ist weiterhin die konjunktive Verknüpfung der atomaren prädikatenlogischen Formeln erfüllbar.

Im vierten Schritt (Abb. C.6d) schließlich weist der SAT-Solver der Variablen $b_{6,1}$ den Wert T zu, was wiederum eine Implikation und damit einen Aufruf des LRA-Solver nach sich zieht. Auch in diesem Fall ist die Konjunktion der Formeln durch den LRA-Solver erfüllbar. Da auch ebenfalls ein Modell der aussagenlogischen Formel φ_a gefunden wurde, ist dieses auch LRA-konsistent. Dies bedeutet, dass die prädikatenlogische Formel φ aus Gleichung (C.1) erfüllbar ist.

Viele effiziente SMT-Solver sind in den letzten Jahren entstanden, z. B. Ario [397], BarceLogic [348], CVCLite/CVC3 [29], DLSAT [310], haRVey [130], MathSAT [56], Sateen [261], SDSAT [178], Simplify [131], TSAT++ [16], UCLID [280], Yices [142], Verifun [164], Zapato [24].

C.4 CTL-Fixpunktberechnung

Zur formalen Spezifikation funktionaler Eigenschaften wird häufig die temporale Aussagenlogik *CTL* (engl. *Computation Tree Logic*) verwendet. Diese ist in Abschnitt 2.4.2 eingeführt. Jeder der acht Operatoren mit Verzweigungslogik in CTL lässt sich als ein *Fixpunkt* charakterisieren. Ein Fixpunkt einer Funktion f lässt sich charakterisieren als $f(a) = a$, d. h. dass die Anwendung einer Funktion auf ein Argument a das Ergebnis a liefert.

Für CTL-Fixpunkte sind sog. *Funktionale* der Ausgangspunkt. Ein Funktional ist eine Abbildung zwischen Mengen von Abbildungen. Ein Funktional τ wird im Folgenden mit $\lambda y : \varphi$ bezeichnet. Dabei ist φ eine CTL-Formel und y eine in φ enthaltene Variable. Wird das Funktional auf eine Formel p angewandt, so wird in φ jedes Vorkommen von y durch p ersetzt.

Beispiel C.4.1. Gegeben ist das Funktional $\tau := \lambda y : x \vee y$ sowie die Formel $p := \mathsf{F}$. Die Anwendung von τ auf p liefert:

$$\tau(p) = \tau(\mathsf{F}) = x \vee \mathsf{F} = x$$

Definition C.4.1 (Fixpunkt eines Funktionals). *Eine Formel p heißt* Fixpunkt *eines Funktionals τ, genau dann, wenn gilt:*

$$\tau(p) = p$$

Beispiel C.4.2. Gegeben ist das Funktional $\tau := \lambda y : x \vee y$ sowie die Formel $p := x \vee z$. Die Anwendung von τ auf die Formel p liefert:

$$\tau(p) = \tau(x \vee z) = x \vee (x \vee z) = x \vee z = p$$

Somit ist p Fixpunkt von τ.

Definition C.4.2 (Monotonie eines Funktionals). *Ein Funktional τ heißt* monoton*, genau dann, wenn gilt:*

$$p \subseteq q \Rightarrow \tau(p) \subseteq \tau(q)$$

Anschaulich bedeutet dies für aussagenlogische Formeln, dass wenn q die selben und eventuelle zusätzliche Einsstellen enthält wie p, so enthält $\tau(q)$ die selben und eventuell zusätzliche Einsstellen zu $\tau(p)$.

Beispiel C.4.3. Gegeben ist das Funktional $\tau := \lambda y : x \vee y$ und die Formeln $p := x \wedge z$ und $q := z$, d. h. $p \subset q$, dann ergibt sich $\tau(p)$ zu $\tau(p) = x \vee (x \wedge z) = x$ und $\tau(q) = x \vee z$.

Allgemein gilt, dass jedes monotone Funktional $\tau = \lambda y : \varphi$ einen *kleinsten Fixpunkt* $\mu y : \varphi$ und einen *größten Fixpunkt* $\nu y : \varphi$ besitzt.

Im Folgenden bezeichne $\tau^i(p)$ diejenige Formel, die sich nach i-facher Iteration des Funktionals für die Anfangsformel p ergibt, d. h.

$$\tau^i(p) := \underbrace{\tau(\tau(\dots\tau(p)))}_{i\text{-fach}}.$$

Weiterhin bezeichne $\bigcup_i \tau^i(p)$ die Vereinigung aller Formeln, die durch i-fache Iteration des Funktionals τ ausgehend von der Anfangsformel p entstehen, d. h.

$$\bigcup_i \tau^i(p) := \tau(p) \cup \tau(\tau(p)) \cup \dots \cup \underbrace{\tau(\tau(\dots\tau(p)))}_{(i-1)\text{-fach}} \cup \underbrace{\tau(\tau(\dots\tau(p)))}_{i\text{-fach}}.$$

Schließlich bezeichne $\bigcap_i \tau^i(p)$ die Schnittmenge aller Formeln, die durch i-fache Iteration des Funktionals τ ausgehend von der Anfangsformel p entstehen, d. h.

$$\bigcap_i \tau^i(p) := \tau(p) \cap \tau(\tau(p)) \cap \dots \cap \underbrace{\tau(\tau(\dots\tau(p)))}_{(i-1)\text{-fach}} \cap \underbrace{\tau(\tau(\dots\tau(p)))}_{i\text{-fach}}.$$

Definition C.4.3 (Vereinigungs- und Schnittstetigkeit). *Ein Funktional τ heißt* vereinigungsstetig*, wenn für eine beliebige unendliche Folge $(p_1, p_2, p_3, \dots)$ mit $p_1 \subseteq p_2 \subseteq p_3 \subseteq \dots$ gilt:*

$$\bigcup_i \tau(p_i) = \tau\left(\bigcup_i p_i\right)$$

τ heißt schnittstetig*, wenn für eine beliebige unendliche Folge $(p_1, p_2, p_3, \dots)$ mit $p_1 \subseteq p_2 \subseteq p_3 \subseteq \dots$ gilt:*

$$\bigcap_i \tau(p_i) = \tau\left(\bigcap_i p_i\right)$$

Es gilt, dass wenn $\tau = \lambda y : \varphi$ ein monotones Funktional ist und φ eine CTL-Formel, dann ist φ vereinigungs- und schnittstetig. Der kleinste und größte Fixpunkt lässt sich wie folgt berechnen:

$$\mu y : \varphi := \bigcup_i \tau^i(\mathsf{F}) \tag{C.2}$$

$$\nu y : \varphi := \bigcap_i \tau^i(\mathsf{T}) \tag{C.3}$$

Für eine beliebige temporale Struktur M (siehe Abschnitt 2.4.1) gilt:

$$\text{EG } p := \nu y : p \wedge \text{EX } y \tag{C.4}$$

$$\text{E } p \text{ U } q := \mu y : q \vee (p \wedge \text{EX } y) \tag{C.5}$$

Im Folgenden werden für eine gegebene temporale Struktur M Zustandsmengen mit CTL-Formeln φ assoziiert, d. h. $\{s \in S \mid M, s \models \varphi\}$.

Beispiel C.4.4. Gegeben ist die temporale Struktur M aus Abb. C.7a). Es soll durch Fixpunktberechnung gezeigt werden, dass im Zustand s_0 der temporalen Struktur M die CTL-Formel EG p gilt. p gilt in den Zuständen s_0, s_1 und s_2. Nach Gleichung (C.4) muss der größte Fixpunkt mit Anfangsformel T berechnet werden. Der erste Schritt besteht darin, die Menge aller Zustände $S[0]$ zu bestimmen. Da $\tau^0 = \mathsf{T}$, gilt $S[0] = \{s_0, s_1, s_2, s_3\}$. Dies ist in Abb. C.7b) dargestellt.

Nun erfolgt die Iteration:

1. Im ersten Schritt der Iteration erhält man $\tau^1 = \tau(\mathsf{T}) = p \wedge \text{EX } \mathsf{T} = p$. Die zugehörige Zustandsmenge ist diejenige Menge an Zuständen, die mit p markiert sind, d. h. $S[1] = \{s_0, s_1, s_2\}$. Dies ist in Abb. C.7c) dargestellt.
2. Im zweiten Schritt der Iteration ergibt sich $\tau^2 = \tau(p) = p \wedge \text{EX}p$. Dies repräsentiert diejenigen Zustände, die mit p markiert sind und die einen mit p markierten Zustand in einem Schritt erreichen können. Dies ergibt die Zustandsmenge $S[2] = \{s_0, s_1\}$, die in Abb. C.7d) dargestellt ist.
3. Im dritten Schritt ergibt sich $\tau^3 = \tau(p \wedge \text{EX } p) = p \wedge \text{EX } (p \wedge \text{EX } p)$. Die repräsentierte Zustandsmenge besteht aus den Zuständen, die mit p markiert sind, einen mit p markierten Zustand in einem Zustandsübergang erreichen können und von dort einen mit p markierten Zustand in einem weiteren Zustandsübergang erreichen können. Dies ist die Menge $S[3] = \{s_0\}$, die in Abb. C.7e) zu sehen ist.
4. Im vierten Schritt ergibt sich $\tau^4 = \tau(\tau^3) = p \wedge \text{EX } (p \wedge \text{EX } (p \wedge \text{EX } p))$. Die repräsentierte Zustandsmenge besteht aus denjenigen Zuständen aus der Menge $S[3]$, die zusätzlich in einen weiteren Schritt wieder einen mit p markierten Zustand erreichen können. Aufgrund der Schleife an Zustand s_0 ist dies die Menge $S[4] = \{s_0\} = S[3]$. Hiermit ist der größte Fixpunkt erreicht.

Für die Berechnung der Menge an Zuständen, in denen auf der gegebenen temporalen Struktur M die CTL-Formel EG p gilt, muss nach Gleichung (C.3) die Schnittmenge aller Iterationsschritte bestimmt werden:

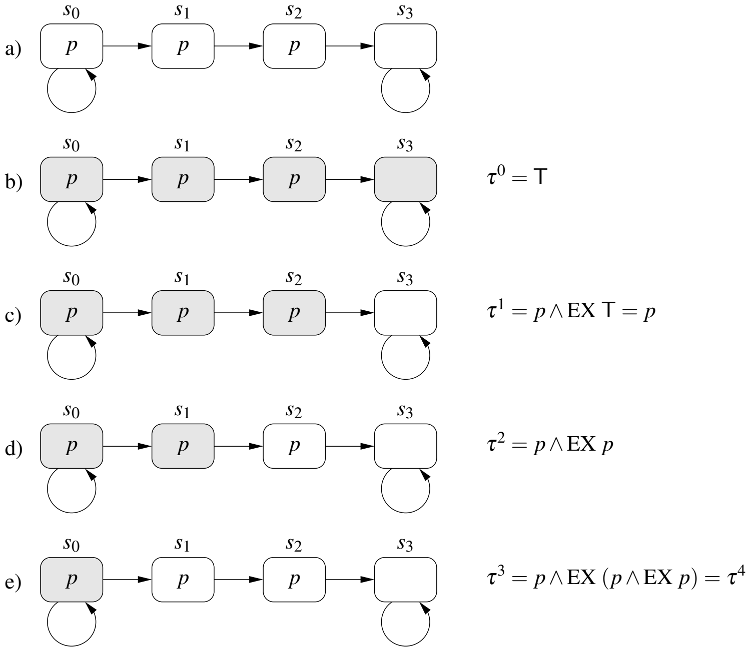


Abb. C.7. Berechnung von EG p

$$\mathrm{EG}\,p = \bigcap_i (\lambda y : p \wedge \mathrm{EX}\,y)^i\,(\mathsf{T})$$

Dies kann direkt auf den assoziierten Zustandsmengen erfolgen:

$$S[0] \cap S[1] \cap S[2] \cap S[3] = \{s_0\}$$

Da s_0 Element dieser Menge ist, gilt EG p im Zustand s_0 der temporalen Struktur M.

Beispiel C.4.5. Gegeben ist die temporale Struktur M aus Abb. C.8a). Es soll durch Fixpunktberechnung gezeigt werden, dass im Zustand s_0 der temporalen Struktur M die CTL-Formel E p U q gilt. Nach Gleichung (C.5) muss der kleinste Fixpunkt berechnet werden. Der Anfangszustand $\tau^0 = \mathsf{F}$ impliziert $S[0] = \emptyset$. Dies ist in Abb. C.8a) dargestellt.

Nun erfolgt die Iteration:

1. Im ersten Schritt der Iteration erhält man $\tau^1 = \tau(\mathsf{F}) = q \vee (p \wedge \mathrm{EX}\,\mathsf{F}) = q$. Die zugehörige Zustandsmenge ist diejenige Menge an Zuständen, die mit q markiert sind, d. h. $S[1] = \{s_2\}$. Dies ist in Abb. C.8b) dargestellt.
2. Im zweiten Schritt der Iteration ergibt sich $\tau^2 = \tau(q) = q \vee (p \wedge \mathrm{EX}\,q)$. Dies repräsentiert diejenigen Zustände, die mit q markiert sind oder die mit p markiert sind und einen mit q markierten Zustand in einem Schritt erreichen können. Dies ergibt die Zustandsmenge $S[2] = \{s_1, s_2\}$, die in Abb. C.8c) dargestellt ist.

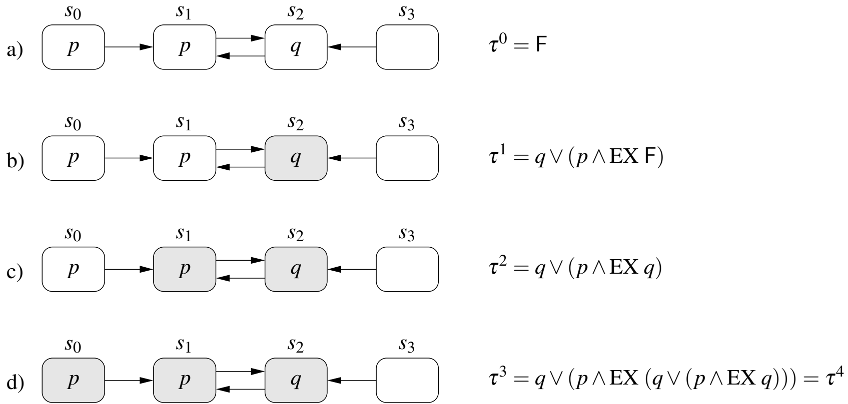


Abb. C.8. Berechnung von E p U q

3. Im dritten Schritt ergibt sich $\tau^3 = \tau(q \vee (p \wedge \mathrm{EX}\ q)) = q \vee (p \wedge \mathrm{EX}\ (q \vee (p \wedge \mathrm{EX}\ q)))$. Die repräsentierte Zustandsmenge besteht aus den Zuständen,
   - die mit q markiert sind,
   - die mit p markiert sind und einen mit q markierten Zustand in einem Zustandsübergang erreichen können, oder
   - die mit p markiert sind, in einem Zustandsübergang einen mit p markierten Zustand und in einem weiteren Zustandsübergang einen mit q markierten Zustand erreichen können.

   Dies ist die Menge $S[3] = \{s_0, s_1, s_2\}$, die in Abb. C.7d) zu sehen ist.
4. Im vierten Schritt ergibt sich $\tau^4 = \tau(\tau^3) = \tau^3$. Die repräsentierte Zustandsmenge ist identisch mit $S[3]$. Hiermit ist der kleinste Fixpunkt erreicht.

Für die Berechnung der Zustände, in denen auf der gegebenen temporalen Struktur M die CTL-Formel E p U q gilt, muss nach Gleichung (C.2) noch die Vereinigungsmenge aller Iterationsschritte bestimmt werden:

$$\mathrm{E}\ p\ \mathrm{U}\ q = \bigcup_i (\lambda y : q \vee (p \wedge \mathrm{EX}\ y))^i\,(\mathsf{F})$$

Dies kann direkt auf den berechneten Zustandsmengen erfolgen:

$$S[0] \cup S[1] \cup S[2] \cup S[3] = \{s_0, s_1, s_2\}$$

Da s_0 Element dieser Menge ist, gilt E p U q im Zustand s_0 der temporalen Struktur M.

Literatur

1. ABRAMOVICI, M., M. A. BREUER und A. D. FRIEDMAN: *Digital Systems Testing and Testable Design*. John Wiley & Sons, Inc., Hoboken, New Jersey, U.S.A., 1994.
2. ACKERMANN, WILHELM: *Solvable Cases of the Decision Problem*. North-Holland Pub. Co, Amsterdam, The Netherlands, 1954.
3. AGHA, G.: *Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems*. In: NAJM, E. und J-B. STEFANI (Herausgeber): *Formal Methods for Open Object-based Distributed Systems*, Seiten 135—153. Springer, 1997.
4. AHO, A. V., R. SETHI und J. D. ULLMAN: *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 2006. 2. Auflage.
5. AKERS, S. B.: *Binary Decision Diagrams*. IEEE Transactions on Computers, C-27(6):509–516, 1978.
6. ALOUL, F. A., I. L. MARKOV und K. A. SAKALLAH: *FORCE: A Fast & Easy-to-Implement Variable-Ordering Heuristic*. In: *Proceedings of the Great Lakes symposium on VLSI (GLSVLSI)*, Seiten 116–119, 2003.
7. ALUR, R.: *Timed Automata*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 8–22, 1999.
8. ALUR, R., C. COURCOUBETIS und D. L. DILL: *Model Checking in Dense Real-Time*. Information and Computation, 104(1):2–34, 1993.
9. ALUR, R. und D. L. DILL: *Automata for Modeling Real-Time Systems*. In: *Proceedings of the International Colloquium on Automata, Languages and Programming*, Seiten 322–335, 1990.
10. ALUR, R. und D. L. DILL: *A Theory of Timed Automata*. Theoretical Computer Science, 126(2):183–235, 1994.
11. ALUR, R. und T. A. HENZINGER: *Logics and Models of Real Time: A Survey*. In: *Real-Time: Theory in Practice*, Seiten 74–106, 1992.
12. ALUR, R. und T. A. HENZINGER: *A Really Temporal Logic*. Journal of the ACM, 41(1):181–203, 1994.
13. AMBLER, S. W.: *The Elements of UML(TM) 2.0 Style*. Cambridge University Press, New York, NY, U.S.A., 2005.
14. ANANTHARAMAN, T. S., E. M. CLARKE, M. J. FOSTER und B. MISHRA: *Compiling Path Expressions into VLSI Circuits*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 191–204, 1985.
15. ANDREWS, T., S. QADEER, S. K. RAJAMANI, J. REHOF und Y. XIE: *Zing: Exploiting Program Structure for Model Checking Concurrent Software*. In: *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, Seiten 1–15, 2004.

16. ARMANDO, A., C. CASTELLINI, E. GIUNCHIGLIA und M. MARATEA: *A SAT-Based Decision Procedure for the Boolean Combination of Difference Constraints*. In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Seiten 16–29, 2004.
17. ARMANDO, A., J. MANTOVANI und L. PLATANIA: *Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers*. In: *Proceedings of the International SPIN Workshop (SPIN)*, Seiten 146–162, 2006.
18. ARMANDO, A., J. MANTOVANI und L. PLATANIA: *Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers*. International Journal on Software Tools for Technology Transfer (STTT), 11(1):69–83, 2009.
19. ASHCROFT, E. A.: *Proving Assertions about Parallel Programs*. Journal of Computer and Systems Science, 10(1):110–135, 1975.
20. ASHENDEN, P. J.: *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, 1991.
21. AUDSLEY, N. C., A. BURNS, M. F. RICHARDSON, K. TINDELL und A. J. WELLINGS: *Applying new Scheduling Theory to Static Priority Preemptive Scheduling*. Real-Time Systems, 8(5):284–292, 1993.
22. BACCELLI, F., G. COHEN, G. J. OLSDER und J.-P. QUADRAT: *Synchronization and Linearity: An Algebra for Discrete Event Systems*. John Wiley & Sons, Inc., Chichester, West Sussex, England, 1992.
23. BALARIN, F. und A. L. SANGIOVANNI-VINCENTELLI: *An Iterative Approach to Language Containment*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 29–40, 1993.
24. BALL, T., B. COOK, S. K. LAHIRI und L. ZHANG: *Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 457–461, 2004.
25. BALL, T., B. COOK, V. LEVIN und S. K. RAJAMANI: *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft*. In: *Proceedings of the International Conference on Integrated Formal Methods (IFM)*, Seiten 1–20, 2004.
26. BALL, T., R. MAJUMDAR, T. MILLSTEIN und S. K. RAJAMANI: *Automatic Predicate Abstraction of C Programs*. ACM SIGPLAN Notices, 36(5):203–213, 2001.
27. BALL, T. und S. K. RAJAMANI: *Bebop: A Symbolic Model Checker for Boolean Programs*. In: *Proceedings of the International SPIN Workshop (SPIN)*, Seiten 113–130, 2000.
28. BALL, T. und S. K. RAJAMANI: *Formal Methods Specification and Verification Guidebook for Software and Computer Systems*. Technischer Bericht MSR-TR-2002-09, Microsoft Research, Redmond, WA, 2002.
29. BARRETT, C. und S. BEREZIN: *CVC Lite: A New Implementation of the Cooperating Validity Checker*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 515–518, 2004.
30. BARRETT, C. W.: *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. Doktorarbeit, Stanford University, CA, U.S.A., 2003.
31. BARRETT, C. W., D. L. DILL und J. R. LEVITT: *A Decision Procedure for Bit-Vector Arithmetic*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 522–527, 1998.
32. BAUMGARTEN, B.: *Petri-Netze Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Mannheim, 1996. 2. Auflage.
33. BAYARDO, R. J. und R. C. SCHRAG: *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*. In: *Proceedings of the National Conference on Artificial Intelligence*, Seiten 203–208, 1997.

34. BECKER, B. und R. DRECHSLER: *How Many Decomposition Types do we Need?* In: *Proceedings of the European Conference on Design and Test (EDTC)*, Seiten 438–443, 1995.
35. BECKER, B., R. DRECHSLER und R. ENDERS: *On the Representational Power of Bit-Level and Word-Level Decision Diagrams*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 461–467, 1997.
36. BECKER, B., R. DRECHSLER und P. MOLITOR: *Technische Informatik: Eine Einführung*. Pearson Studium, Deutschland, 2005.
37. BEN-ARI, M., Z. MANNA und A. PNUELI: *The Temporal Logic of Branching Time*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 164–176, 1981.
38. BENVENISTE, A. und P. LEGUERNIC: *Hybrid Dynamical Systems Theory and the SIGNAL Language*. IEEE Transactions on Automatic Control, 35(5):535–546, 1990.
39. BÉRARD, B., M. BIDOIT, A. FINKEL, F. LAROUSSINIE, A. PETIT, L. PETRUCCI, P. SCHNOEBELEN und P. MCKENZIE: *Systems and Software Verification*. Springer, Berlin, Heidelberg, 2001.
40. BEREZIN, S., A. BIERE, E. M. CLARKE und Y. ZHU: *Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification*. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Seiten 369–386, 1998.
41. BERGERON, J.: *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, Dordrecht, 2003. 2. Auflage.
42. BERGERON, J., E. CERNY, A. HUNTER und A. NIGHTINGALE: *Verification Methodology Manual for SystemVerilog*. Springer, New York, Berlin, Heidelberg, 2005.
43. BERMAN, C. L. und L. H. TREVILLYAN: *Functional Comparison of Logic Designs for VLSI Circuits*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 456–459, 1989.
44. BERRY, G. und G. GONTHIER: *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*. Science of Computer Programming, 19(2):87–152, 1992.
45. BERTACCO, V.: *Scalable Hardware Verification with Symbolic Simulation*. Springer, New York, NY, U.S.A., 2006.
46. BEYER, D., T. A. HENZINGER und G. THÉODULOZ: *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 504–518, 2007.
47. BHASKER, J.: *A SystemC Primer*. Star Galaxy Publishing, Allentown, Pennsylvania, 2004. 2. Auflage.
48. BIERE, A., A. CIMATTI, E. M. CLARKE, M. FUJITA und Y. ZHU: *Symbolic Model Checking Using SAT Procedures instead of BDDs*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 317–320, 1999.
49. BIERE, A., A. CIMATTI, E. M. CLARKE und Y. ZHU: *Symbolic Model Checking without BDDs*. In: *Tools and Algorithms for Construction and Analysis of Systems*, Seiten 193–207. Springer, Berlin, Heidelberg, 1999.
50. BIERE, A. und W. KUNZ: *SAT and ATPG: Boolean Engines for Formal Hardware Verification*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 782–785, 2002.
51. BIROLINI, A.: *Reliability Engineering: Theory and Practice*. Springer, Berlin, Heidelberg, 2004. 2. Auflage.

52. BJESSE, P. und K. CLAESSEN: *SAT-Based Verification without State Space Traversal*. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Seiten 372–389, 2000.
53. BOULÉ, M. und Z. ZILIC: *Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 69–76, 2006.
54. BOULÉ, M. und Z. ZILIC: *Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 324–329, 2007.
55. BOULÉ, M. und Z. ZILIC: *Automata-Based Assertion-Checker Synthesis of PSL Properties*. ACM Transactions on Design Automation of Electronic Systems (TODAES), 13(1):1–21, 2008.
56. BOZZANO, M., R. BRUTTOMESSO, A. CIMATTI, T. JUNTTILA, P. VAN ROSSUM, S. SCHULZ und R. SEBASTIANI: *An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Seiten 317–333. Springer, Berlin, Heidelberg, 2005.
57. BRACE, K. S., R. L. RUDELL und R. E. BRYANT: *Efficient Implementation of a BDD Package*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 40–45, 1990.
58. BRAND, D.: *Verification of Large Synthesized Designs*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 534–537, 1993.
59. BRINKMANN, R. und R. DRECHSLER: *RTL-Datapath Verification using Integer Linear Programming*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 741–746, 2002.
60. BRYANT, R., D. KROENING, J. OUAKNINE, S. A. SESHIA, O. STRICHMAN und B. BRADY: *Deciding Bit-Vector Arithmetic with Abstraction*. In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Seiten 358–372, 2007.
61. BRYANT, R. E.: *Symbolic Verification of MOS Circuits*. In: *Chapel Hill Conference on VLSI*, Seiten 419–438, 1985.
62. BRYANT, R. E.: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 35(8):677–691, 1986.
63. BRYANT, R. E.: *On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication*. IEEE Transactions on Computers, 40(2):205–213, 1991.
64. BRYANT, R. E., D. BEATTY, K. BRACE, K. CHO und T. SHEFFLER: *COSMOS: A Compiled Simulator for MOS Circuits*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 9–16, 1987.
65. BRYANT, R. E. und Y.-A. CHEN: *Verification of Arithmetic Functions with Binary Moment Diagrams*. Technischer Bericht CS-94-160, Carnegie Mellon University, 1994.
66. BRYANT, R. E. und Y.-A. CHEN: *Verification of Arithmetic Circuits with Binary Moment Diagrams*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 535–541, 1995.
67. BRYANT, R. E., S. GERMAN und M. N. VELEV: *Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic*. ACM Transactions on Computational Logic (TOCL), 2(1):93–134, 2001.
68. BRYANT, R. E. und M. N. VELEV: *Verification of Pipelined Microprocessors by Comparing Memory Execution Sequences in Symbolic Simulation*. In: *Proceedings of the Asian Computing Science Conference on Advances in Computing Science (ASIAN)*, Seiten 18–31, 1997.

69. BUCK, J., S. HA, E. A. LEE und D. G. MESSERSCHMITT: *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems.* International Journal on Computer Simulation, 4(2):155–182, 1994.
70. BUCK, J. T.: *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* Doktorarbeit, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.
71. BURCH, J. R.: *Techniques for Verifying Superscalar Microprocessors.* In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 552–557, 1996.
72. BURCH, J. R., E. M. CLARKE und D. E. LONG: *Symbolic Model Checking with Partitioned Transition Relations.* In: *Proceedings of the International Conference on Very Large Scale Integration (VLSI)*, Seiten 49–58, 1991.
73. BURCH., J. R., E. M. CLARKE, D. E. LONG, K. L. MCMILLAN und D. L. DILL: *Symbolic Model Checking for Sequential Circuit Verification.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(4):401–424, 1994.
74. BURCH, J. R., E. M. CLARKE, K. L. MCMILLAN, D. L. DILL und L. J. HWANG: *Symbolic Model Checking:* 10^{20} *States and Beyond.* In: *Proceedings of the Symposium on Logic in Computer Science (LICS)*, Seiten 428–439, 1990.
75. BURCH, J. R. und D. L. DILL: *Automatic Verification of Pipelined Microprocessor Control.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 68–80, 1994.
76. BUSHNELL, M. und V. AGRAWAL: *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits.* Kluwer Academic Publishers, Norwell, Massachusetts, U.S.A., 2000.
77. BUSTAN, D., D. FISMAN und J. HAVLICEK: *Automata Construction for PSL.* Technischer Bericht, The Weizmann Institute of Science, 2005. Technical Report MCS05-04.
78. BUTTAZZO, G.: *Rate Monotonic vs. EDF: Judgment Day.* In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Seiten 67–83, 2003.
79. BUTTAZZO, G.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, New York, NY, U.S.A., 2004.
80. CABODI, G., P. CAMURATI, L. LAVAGNO und S. QUER: *Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits.* In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 728–733, 1997.
81. CABODI, G., P. CAMURATI und S. QUER: *Improved Reachability Analysis of Large Finite State Machines.* In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 354–360, 1996.
82. CADAR, C., V. GANESH, P. M. PAWLOWSKI, D. L. DILL und D. R. ENGLER: *EXE: Automatically Generating Inputs of Death.* In: *Proceedings of the Conference on Computer and Communications Security (CCS)*, Seiten 322–335, 2006.
83. CARLONI, L. P., K. L. MCMILLAN, A. SALDANHA und A. L. SANGIOVANNI-VINCENTELLI: *A Methodology for Correct-by-Construction Latency Insensitive Design.* In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 309–315, 1999.
84. CARLONI, L. P., K. L MCMILLAN und A. L. SANGIOVANNI-VINCENTELLI: *Theory of Latency-Insensitive Design.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(9):1059–1076, 2001.
85. CARLONI, L. P. und A. L. SANGIOVANNI-VINCENTELLI: *Performance Analysis and Optimization of Latency Insensitive Systems.* In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 361–367, 2000.

86. CARTER, W. C., W. H. JOYNER und D. BRAND: *Symbolic simulation for correct machine design*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 280–286, 1979.
87. CASSANDRAS, C. G. und S. LAFORTUNE: *Introduction to Discrete Event Systems*. Springer, New York, NY, U.S.A., 1999.
88. CASSEZ, F. und O.-H. ROUX: *Structural Translation from Time Petri Nets to Timed Automata*. Electronic Notes in Theoretical Computer Science, 128(6):145–160, 2005.
89. CHAKI, S., E. M. CLARKE, J. OUAKNINE, N. SHARYGINA und N. SINHA: *State/Event-Based Software Model Checking*. In: *In Proceeding of the International Conference on Integrated Formal Methods*, Seiten 128–147, 2004.
90. CHETTO, H., M. SILLY und T. BOUCHENTOUF: *Dynamic Scheduling of Real-Time Tasks under Precedence Constraints*. Real-Time Systems, 2:325–346, 1990.
91. CHUSHO, T.: *Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing*. IEEE Transactions on Software Engineering, SE-13(5):509–517, 1987.
92. CIESIELSKI, M., P. KALLA und S. ASKAR: *Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs*. IEEE Transactions on Computers, 55(9):1188–1201, 2006.
93. CIESIELSKI, M., P. KALLA, Z. ZENG und B. ROUZEYRE: *Taylor Expansion Diagrams: A New Representation for RTL Verification*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 70–75, 2001.
94. CIESIELSKI, M., P. KALLA, Z. ZHENG und B. ROUZEYRE: *Taylor Expansion Diagrams: A Compact, Canonical Representation with Applications to Symbolic Verification*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 285–289, 2002.
95. CLAESSEN, K. und J. MÅRTENSSON: *An Operational Semantics for Weak PSL*. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Seiten 337–351, 2004.
96. CLARISÓ, R. und J. CORTADELLA: *The Octahedron Abstract Domain*. Science of Computer Programming, 64(1):115–139, 2006.
97. CLARKE, E. M. und E. A. EMERSON: *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: *Processings of the Workshop on Logic of Programs*, Seiten 52–71, 1982.
98. CLARKE, E. M., E. A. EMERSON und A. P. SISTLA: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems (TOPLAS), 8(2):244–263, 1986.
99. CLARKE, E. M., M. FUJITA, P. MCGEER, K. L. MCMILLAN, J. YANG und X. ZHAO: *Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation*. In: *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, Seiten 1–15, 1993.
100. CLARKE, E. M., O. GRUMBERG, S. JHA, Y. LU und H. VEITH: *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. Journal of the ACM, 50(5):752–794, 2003.
101. CLARKE, E. M., O. GRUMBERG und D. E. LONG: *Model Checking and Abstraction*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 343–354, 1992.
102. CLARKE, E. M., D. KROENING und F. LERDA: *A Tool for Checking ANSI-C Programs*. In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Seiten 168–176, 2004.

103. Clarke, E. M., D. Kroening, N. Sharygina und K. Yorav: *Predicate Abstraction of ANSI-C Programs Using SAT*. Journal of Formal Methods in System Design, 25(2–3):105–127, 2004.
104. Clarke, E. M., D. Kroening und K. Yorav: *Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 368–371, 2003.
105. Clarke, E. M., D. Kroening und K. Yorav: *Specifying and Verifying Systems with Multiple Clocks*. In: *Proceedings of the International Conference on Computer Design (ICCD)*, Seiten 48–55, 2003.
106. Clarke, E. M., K. L. McMillan, X. Zhao, M. Fujita und J. Yang: *Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 54–60, 1993.
107. Clarke, E. M., O.Grumberg und D. A. Peled: *Model Checking*. MIT Press, Cambridge, MA, U.S.A., 1999.
108. Clarke, L. A., J. Hassell und D. J. Richardson: *A Close Look at Domain Testing*. IEEE Transactions on Software Engineering, 8(4):380–390, 1982.
109. Clarke, L. A., A. Podgurski, D. J. Richardson und S. J. Zeil: *A Comparison of Data Flow Path Selection Criteria*. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 28–30, 1985.
110. Cochet-Terrasson, J., G. Cohen, S. Gaubert, M. Mc Gettrick und J.-P. Quadrat: *Numerical Computation of Spectral Elements in Max-Plus Algebra*. In: *Proceedings of the Conference on System Structure and Control*, Seiten 667–674, 1998.
111. Coelho Jr., C. N. und H. D. Foster: *Assertion-Based Verification – Property Specification*. In: *Advanced Formal Verification*, Seiten 167–204. Kluwer Academic Publishers, Boston, 2004.
112. Cohen, G., D. Dubois, J. Quadrat und M. Viot: *A Linear-System-Theoretic View of Discrete-Event Processes and its Use for Performance Evaluation in Manufacturing*. IEEE Transactions on Automatic Control, 30(3):210–220, 1985.
113. Commoner, F., A. W. Holt, S. Even und A. Pnueli: *Marked Directed Graphs*. Journal of Computer and System Sciences, 5:511–523, 1971.
114. Cook, B., D. Kroening und N. Sharygina: *Cogent: Accurate Theorem Proving for Program Verification*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 296–300, 2005.
115. Cook, B., D. Kroening und N. Sharygina: *Symbolic Model Checking for Asynchronous Boolean Programs*. In: *Proceedings of the International SPIN Workshop (SPIN)*, Seiten 75–90, 2005.
116. Cook, S. A.: *The Complexity of Theorem-Proving Procedures*. In: *Proceedings of the Symposium on Theory of Computing (STOC)*, Seiten 151–158, 1971.
117. Coudert, O., C. Berthet und J. C. Madre: *Verification of Synchronous Sequential Machines Based on Symbolic Execution*. In: *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Seiten 365–373, 1990.
118. Cousot, P. und R. Cousot: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 238–252, 1977.
119. Cousot, P. und R. Cousot: *Systematic Design of Program Analysis Frameworks*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 269–282, 1979.

120. COUSOT, P. und N. HALBWACHS: *Automatic Discovery of Linear Restraints Among Variables of a Program*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 84–96, 1978.
121. CRUZ, R. L.: *A Calculus for Network Delay, Part I: Network Elements in Isolation*. IEEE Transactions on Information Theory, 37(1):114–131, 1991.
122. CURRIE, D. W., X. FENG, M. FUJITA, A. J. HU, M. KWAN und S. RAJAN: *Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions*. International Journal of Parallel Programming, 34(1):61–91, 2006.
123. CURRIE, D. W., A. J. HU, S. RAJAN und M. FUJITA: *Automatic Formal Verification of DSP Software*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 130–135, 2000.
124. CYRLUK, D., O. MÖLLER und H. RUESS: *An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 60–71, 1997.
125. CYTRON, R., J. FERRANTE, B. K. ROSEN, M. N. WEGMAN und F. K. ZADECK: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4):451–490, 1991.
126. DAHAN, A., D. GEIST, L. GLUHOVSKY, D. PIDAN, G. SHAPIR, Y. WOLFSTHAL, L. BENALYCHERIF, R. KAMIDEM und Y. LAHBIB: *Combining System Level Modeling with Assertion Based Verification*. In: *Proceedings of the International Symposium on Quality of Electronic Design (ISQED)*, Seiten 310–315, 2005.
127. DANIELE, M., F. GIUNCHIGLIA und M. Y. VARDI: *Improved Automata Generation for Linear Temporal Logic*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 249–260, 1999.
128. DAVIS, M., G. LOGEMANN und D. LOVELAND: *A Machine Program for Theorem-Proving*. Communications of the ACM, 5(7):394–397, 1962.
129. DAVIS, M. und H. PUTNAM: *A Computing Procedure for Quantification Theory*. Journal of the ACM, 7(3):201–215, 1960.
130. DEHARBE, D. und S. RANISE: *Light-Weight Theorem Proving for Debugging and Verifying Units of Code*. In: *Proceedings of the Conference on Software Engineering and Formal Methods*, Seiten 220–228, 2003.
131. DETLEFS, D., G. NELSON und J. B. SAXE: *Simplify: A Theorem Prover for Program Checking*. Journal of the ACM, 52(3):365–473, 2005.
132. DILL, D. L.: *Timing Assumptions and Verification of Finite-State Concurrent Systems*. In: *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Seiten 197–212, 1990.
133. DONLIN, A.: *Transaction Level Modeling: Flows and Use Models*. In: *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seiten 75–80, 2004.
134. DRECHSLER, R. und B. BECKER: *Overview of Decision Diagrams*. IEE Proceedings on Computers and Digital Techniques, 144(3):187–193, 1997.
135. DRECHSLER, R. und B. BECKER: *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.
136. DRECHSLER, R., B. BECKER und S. RUPPERTZ: *K*BMDs: A New Data Structure for Verification*. In: *Proceedings of the European Conference on Design and Test (EDTC)*, Seiten 2–8, 1996.
137. DRECHSLER, R., S. EGGERSGLÜSS, G. FEY, J. SCHLÖFFEL und D. TILLE: *Effiziente Erfüllbarkeitsalgorithmen für die Generierung von Testmustern*. it – information technology, 51(2):102–111, 2009.

138. Drechsler, R., A. Sarabi, M. Theobald, B. Becker und M. A. Perkowski: *Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 415–419, 1994.
139. Drechsler, R., M. Theobald und B. Becker: *Fast OFDD based Minimization of Fixed Polarity Reed-Muller Expressions*. In: *Proceedings of the European Conference on Design Automation (ECDA)*, Seiten 2–7, 1994.
140. D'Silva, V., D. Kroening und G. Weissenbacher: *A Survey of Automated Techniques for Formal Software Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(7):1165–1178, 2008.
141. Dubois, O., P. Andre, Y. Boufkhad und J. Carlier: *SAT versus UNSAT*. In: Johnson, D. S. und M. A. Trick (Herausgeber): *Second DIMACS Implementation Challenge*, Band 26 der Reihe *Series in Discrete Mathematics and Theoretical Computer Science (DIMACS)*, Seiten 415–434. American Mathematical Society, 1996.
142. Dutertre, B. und L. de Moura: *A Fast Linear-Arithmetic Solver for DPLL(T)*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 81–94, 2006.
143. Ecker, W., V. Esen, T. Steininger, M. Velten und M. Hull: *Execution Semantics and Formalisms for Multi-Abstraction TLM Assertions*. In: *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Seiten 93–102, 2006.
144. Ecker, W., V. Esen, T. Steininger, M. Velten und M. Hull: *Implementation of a Transaction Level Assertion Framework in SystemC*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 1–6, 2007.
145. Eijk, C. A. J. van: *Formal Methods for the Verification of Digital Circuits*. Doktorarbeit, Eindhoven University of Technology, The Netherlands, 1997.
146. Eijk, C. A. J. van: *Sequential Equivalence Checking based on Structural Similarities*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(7):814–819, 2000.
147. Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs und Y. Xiong: *Taming Heterogeneity – the Ptolemy Approach*. Proceedings of the IEEE, 91(1):127–144, 2003.
148. Emerson, E. A.: *Temporal and Modal Logic*. In: *Formal Models and Semantics*, Band B der Reihe *Handbook of Theoretical Computer Science*, Seiten 995–1072. MIT Press, Cambridge, MA, U.S.A., 1990.
149. Emerson, E. A. und E. M. Clarke: *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*. In: *Proceedings of the Colloquium on Automata, Languages and Programming*, Seiten 169–181, 1980.
150. Emerson, E. A. und J. Y. Halpern: *"Sometimes" and "Not Never" Revisited: On Branching versus Linear Time*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 127–140, 1983.
151. Emerson, E. A., A. K. Mok, A. P. Sistla und J. Srinivasan: *Quantitative Temporal Reasoning*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 136–145, 1990.
152. Emerson, E. A. und R. J. Trefler: *Parametric Quantitative Temporal Reasoning*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS)*, Seiten 336–343, 1999.
153. Engels, M., G. Bilsen, R. Lauwereins und J. Peperstraete: *Cyclo-Static Data Flow: Model and Implementation*. In: *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, Seiten 503–507, 1994.

154. ERNST, R. und W. YE: *Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 598–604, 1997.
155. ESPARZA, J.: *Model Checking Using Net Unfoldings*. In: *Proceedings of the Conference on Theory and Practice of Software Development (TAPSOFT)*, Seiten 613–628, 1993.
156. EVANS, A., A. SILBURT, G. VRCKOVNIK, T. BROWN, M. DUFRESNE, G. HALL, T. HO und Y. LIU: *Functional Verification of Large ASICs*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 650–655, 1998.
157. FALK, J., C. HAUBELT und J. TEICH: *Efficient Representation and Simulation of Model-Based Designs in SystemC*. In: *Proceedings of the Forum on Design Languages (FDL)*, Seiten 129–134, 2006.
158. FEAUTRIER, P.: *Array Expansion*. In: *Proceedings of the International Conference on Supercomputing (ICS)*, Seiten 429–441, 1988.
159. FENG, X. und A. J. HU: *Automatic Formal Verification for Scheduled VLIW Code*. In: *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems (SCOPES)*, Seiten 85–92, 2002.
160. FENG, X. und A. J. HU: *Cutpoints for Formal Equivalence Verification of Embedded Software*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Seiten 307–316, 2005.
161. FENG, X. und A. J. HU: *Early Cutpoint Insertion for High-Level Software vs. RTL Formal Combinational Equivalence Verification*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 1063–1068, 2006.
162. FETTWEIS, A.: *Realizability of Digital Filter Networks*. Archiv Elek. Übertragung, 30(2):90–96, 1976.
163. FEY, G., R. DRECHSLER und M. CIESIELSKI: *Algorithms for Taylor Expansion Diagrams*. In: *Proceedings of the International Symposium on Multiple-Valued Logic (ISMVL)*, Seiten 235–240, 2004.
164. FLANAGAN, C., R. JOSHI, X. OU und J. B. SAXE: *Theorem Proving Using Lazy Proof Explication*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 355–367, 2003.
165. http://www.flexray.com.
166. FLOYD, R. W.: *Assigning Meaning to Programs*. In: *Proceedings of the Symposium of Applied Mathematics*, Seiten 19–32, 1967.
167. http://www.haifa.ibm.com/projects/verification/focs/.
168. FOSTER, H. D., A. C. KROLNIK und D. J. LACEY: *Assertion-Based Design*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004. 2. Auflage.
169. FOWLER, M. und K. SCOTT: *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, U.S.A., 1997.
170. GAJSKI, D. D. und R. H. KUHN: *New VLSI Tools*. IEEE Computer, 16(12):11–14, 1983.
171. GAJSKI, D. D., F. VAHID, S. NARAYAN und J. GONG: *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., 1994.
172. GAJSKI, D. D., J. ZHU, R. DÖMER, A. GERSTLAUER und S. ZHAO: *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.
173. GANAI, M. K. und A. AZIZ: *Improved SAT-Based Bounded Reachability Analysis*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 729–734, 2002.
174. GANAI, M. K. und A. GUPTA: *SAT-based Scalable Formal Verification Solutions*. Springer, New York, NY, U.S.A., 2007.

175. GANAI, M. K., A. GUPTA und P. ASHAR: *Efficient Modeling of Embedded Memories in Bounded Model Checking*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 440–452, 2004.
176. GANAI, M. K., A. GUPTA und P. ASHAR: *Beyond Safety: Customized SAT-Based Model Checking*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 738–743, 2005.
177. GANAI, M. K., A. GUPTA und P. ASHAR: *Verification of Embedded Memory Systems using Efficient Memory Modeling*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 1096–1101, 2005.
178. GANAI, M. K., M. TALUPUR und A. GUPTA: *SDSAT: Tight Integration of Small Domain Encoding and Lazy Approaches in a Separation Logic Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Seiten 135–150. Springer, Berlin, Heidelberg, 2006.
179. GANESH, V. und D. L. DILL: *A Decision Procedure for Bit-Vectors and Arrays*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 524–536, 2007.
180. GAREY, M. R. und D. S. JOHNSON: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, NY, U.S.A., 1979.
181. GASTIN, P. und D. ODDOUX: *Fast LTL to Büchi Automata Translation*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 53–65, 2001.
182. GERSTLAUER, A., C. HAUBELT, A. D. PIMENTEL, T. P. STEFANOV, D. D. GAJSKI und J. TEICH: *Electronic System-Level Synthesis Methodologies*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(10):1517–1530, 2009.
183. GERTH, R., D. PELED, M. Y. VARDI und P. WOLPER: *Simple On-the-Fly Automatic Verification of Linear Temporal Logic*. In: *Proceedings of the International Symposium on Protocol Specification, Testing and Verification*, Seiten 3–18, 1996.
184. GHAMARIAN, A. H.: *Timing Analysis of Synchronous Data Flow Graphs*. Doktorarbeit, Eindhoven University of Technology, The Netherlands, 2008.
185. GHAMARIAN, A. H., M. C. W. GEILEN, S. STUIJK, T. BASTEN, A. J. M. MOONEN, M. J. G. BEKOOIJ, B. D. THEELEN und M. R. MOUSAVI: *Throughput Analysis of Synchronous Data Flow Graphs*. In: *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, Seiten 25–36, 2006.
186. GHAMARIAN, A. H., S. STUIJK, T. BASTEN, M. C. W. GEILEN und B. D. THEELEN: *Latency Minimization for Synchronous Data Flow Graphs*. In: *Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, Seiten 189–196, 2007.
187. GHEORGHITA, S. V. und R. GRIGORE: *Constructing Checkers from PSL Properties*. In: *In Proceedings of the International Conference on Control Systems and Computer Science*, Seiten 757–762, 2005.
188. GIANNAKOPOULOU, D. und F. LERDA: *From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata*. In: *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Seiten 308–326, 2002.
189. GIRAULT, C. und R. VALK: *Petri Nets for Systems Engineering – A Guide to Modeling, Verification, and Application*. Springer, Berlin, Heidelberg, New York, 2003.
190. GIRGIS, M. und M. WOODWARD: *An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria*. In: *Proceedings of the Workshop on Software Testing*, Seiten 64–73, 1986.

191. GLADIGAU, J., F. BLENDINGER, C. HAUBELT und J. TEICH: *Symbolische Modellprüfung Aktor-orientierter High-level SystemC-Modelle mit Intervalldiagrammen*. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Seiten 109–118, 2008.
192. GLADIGAU, J., C. HAUBELT und J. TEICH: *Symbolic Scheduling of SystemC Dataflow Designs*. In: *Languages for Embedded Systems and their Applications*, Band 36 der Reihe *Lecture Notes in Electrical Engineering*, Seiten 183–199. Springer, 2009.
193. GLADIGAU, JENS: *Symbolische Ablaufplanung von SysteMoC-Beschreibungen*. Diplomarbeit, Department of Computer Science, University of Erlangen-Nuremberg, 2006.
194. GODEFROID, P.: *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, New York, NY, U.S.A., 1996.
195. GODEFROID, P.: *Model Checking for Programming Languages Using VeriSoft*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 174–186, 1997.
196. GODEFROID, P. und P. WOLPER: *A Partial Approach to Model Checking*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS)*, Seiten 406–415, 1991.
197. GODEFROID, P. und P. WOLPER: *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*. Journal of Formal Methods in System Design, 2(2):149–164, 1993.
198. GÖDEL, KURT: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. Monatshefte für Mathematik und Physik, 38:173–198, 1931.
199. GOEL, A. und R. E. BRYANT: *Set Manipulation with Boolean Functional Vectors for Symbolic Reachability Analysis*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 10816–10821, 2003.
200. GOEL, A., K. SAJID, H. ZHOU, A. AZIZ und V. SINGHAL: *BDD Based Procedures for a Theory of Equality with Uninterpreted Functions*. Journal of Formal Methods in System Design, 22(3):205–224, 2003.
201. GOLDBERG, E. und Y. NOVIKOV: *BerkMin: A Fast and Robust SAT-solver*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 142–149, 2002.
202. GOMEZ-PRADO, D., Q. REN, S. ASKAR, M. CIESIELSKI und E. BOUTILLON: *Variable Ordering for Taylor Expansion Diagrams*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 55–59, 2004.
203. GORDON, M., JOE HURD und K. SLIND: *Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving*. In: *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, Seiten 200–215, 2003.
204. GOVINDARAJAN, R. und G. R. GAO: *Rate-Optimal Schedule for Multi-Rate DSP Computations*. Journal of VLSI Signal Processing Systems, 9(3):211–232, 1995.
205. GRAF, S. und H. SAÏDI: *Construction of Abstract State Graphs with PVS*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 72–83, 1997.
206. GROSSE, D. und R. DRECHSLER: *Ein Ansatz zur formalen Verifikation von Schaltungsbeschreibungen in SystemC*. it - Information Technology, 45(4):219–226, 2003.
207. GRÖTKER, T., S. LIAO, G. MARTIN und S. SWAN: *System Design with SystemC*. Kluwer Academic Publishers, Norwell, Massachusetts, Dordrecht, 2002.
208. GUPTA, A. und P. ASHAR: *Integrating a Boolean Satisfiability Checker and BDDs for Combinational Equivalence Checking*. In: *Proceedings of the International Conference on VLSI Design (VLSID)*, Seiten 222–225, 1998.

209. GUPTA, A., M. GANAI, Z. YANG und P. ASHAR: *Iterative Abstraction using SAT-based BMC with Proof Analysis*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 416–423, 2003.
210. HABIBI, A. und S. TAHAR: *Design and Verification of SystemC Transaction-Level Models*. IEEE Transactions on Very Large Scale Integrated Systems, 14(1):57–68, 2006.
211. HACHTEL, G. D. und F. SOMENZI: *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 U.S.A., 1996.
212. HALBWACHS, N., P. CASPI, P. RAYMOND und D. PILAUD: *The Synchronous Data Flow Programming Language LUSTRE*. Proceedings of the IEEE, 79(9):1305–1320, 1991.
213. HALBWACHS, N., Y.-E. PROY und P. ROUMANOFF: *Verification of Real-Time Systems using Linear Relation Analysis*. Journal of Formal Methods in System Design, 11(2):157–185, 1997.
214. HAMAGUCHI, K., A. MORITA und S. YAJIMA: *Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 78–82, 1995.
215. HAUBELT, C., J. FALK, J. KEINERT, T. SCHLICHTER, M. STREUBÜHR, A. DEYHLE, A. HADERT und J. TEICH: *A SystemC-based Design Methodology for Digital Signal Processing Systems*. EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems, 2007.
216. HAUBELT, C., M. MEREDITH, T. SCHLICHTER und J. KEINERT: *SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 580–585, 2008.
217. HENIA, R., A. HAMANN, M. JERSAK, R. RACU, K. RICHTER und R. ERNST: *System Level Performance Analysis – The SymTA/S Approach*. IEE Proceedings on Computers and Digital Techniques, 152(2):148–166, 2005.
218. HENZINGER, T. A., X. NICOLLIN, J. SIFAKIS und S. YOVINE: *Symbolic Model Checking for Real-Time Systems*. Information and Computation, 111(2):193–244, 1994.
219. HERBER, P., J. FELLMUTH und S. GLESNER: *Model Checking SystemC Designs Using Timed Automata*. In: *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seiten 131–136, 2008.
220. HIND, M.: *Pointer Analysis: Haven't We Solved this Problem Yet?* In: *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Seiten 54–61, 2001.
221. HOARE, C. A. R.: *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10):576–580, 1969.
222. HOLZMANN, G. J.: *The Model Checker SPIN*. IEEE Transactions on Software Engineering, 23(5):279–295, 1997.
223. HOPCROFT, J. E., R. MOTWANI und J. D. ULLMAN: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, Deutschland, München, 2002. 2. Auflage.
224. HÖRETH, S. und R. DRECHSLER: *Formal Verification of Word-Level Specifications*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 52–58, 1999.
225. HORN, W. A.: *Some Simple Scheduling Algorithms*. Naval Research Logistics Quarterly, 21:177–185, 1974.
226. HORWITZ, S., T. REPS und D. BINKLEY: *Interprocedural Slicing Using Dependence Graphs*. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Seiten 35–46, 1988.
227. HOWDEN, W. E.: *Theoretical and Empirical Studies of Program Testing*. IEEE Transactions on Software Engineering, SE-4(4):293–298, 1978.

228. HOWDEN, W. E.: *Theoretical and Empirical Studies of Program Testing*. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 305–311, 1978.
229. HU, A. J.: *High-Level vs. RTL Combinational Equivalence: An Introduction*. In: *Proceedings of the International Conference on Computer Design (ICCD)*, Seiten 274–279, 2007.
230. HUANG, C.-Y. und K.-T. CHENG: *Assertion Checking by Combined Word-Level ATPG and Modular Arithmetic Constraint-Solving Techniques*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 118–123, 2000.
231. IBARRA, O. H. und S. MORAN: *Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs*. Journal of the ACM, 30(1):217–228, 1983.
232. IEEE: *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990, 1990.
233. IEEE: *IEEE Standard VHDL Language Reference Manual*. IEEE Std. 1076-1993, 1993.
234. IEEE: *IEEE Standard for Property Specification Language (PSL)*. IEEE Std 1850, 2005.
235. IEEE: *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*. IEEE Std 1800, 2005.
236. IEEE: *IEEE Standard SystemC Language Reference Manual*. IEEE Std 1666, 2006.
237. http://www.cadence.com/products/fv/design_team_simulator/.
238. http://www.cadence.com/products/fv/formal_verifier/.
239. ISHIURA, N., H. SAWADA und S. YAJIMA: *Minimization of Binary Decision Diagrams based on Exchanges of Variables*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 472–475, 1991.
240. ITRS: *International Technology Roadmap for Semiconductors – System Drivers*. Technischer Bericht, ITRS, 2007. http://www.itrs.net/.
241. IVANČIĆ, F., I. SHLYAKHTER, A. GUPTA, M. K. GANAI, V. KAHLON, C. WANG und Z. YANG: *Model Checking C Programs using F-Soft*. In: *Proceedings of the International Conference on Computer Design (ICCD)*, Seiten 297–308, 2005.
242. IVANČIĆ, F., Z. YANG, M. K. GANAI, A. GUPTA und P. ASHAR: *Efficient SAT-Based Bounded Model Checking for Software Verification*. Theoretical Computer Science, 404(3):256–274, 2008.
243. JACKSON, J. R.: *Scheduling a Production Line to Minimize Maximum Tardiness*. Technischer Bericht 43, University of California, Los Angeles, 1955.
244. JAIN, H., D. KROENING und E. M. CLARKE: *Verification of SpecC Using Predicate Abstraction*. In: *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Seiten 7–16, 2004.
245. JAIN, J., A. NARAYAN, C. COELHO, S. P. KHATRI, A. L. SANGIOVANNI-VINCENTELLI, R. K. BRAYTON und M. FUJITA: *Decomposition Techniques for Efficient ROBDD Construction*. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Seiten 419–434, 1996.
246. JALOTE, P.: *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., 1994.
247. JERSAK, M.: *Compositional Performance Analysis for Complex Embedded Applications*. Doktorarbeit, Technische Universität Braunschweig, Deutschland, 2005.
248. JONES, N. D. und S. S. MUCHNICK: *Flow Analysis and Optimization of LISP-like Structures*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 244–256, 1979.
249. JOSEPH, M. und P. PANDYA: *Finding Response Times in a Real-Time System*. The Computer Journal, 29(5):390–395, 1986.

250. KAHN, G.: *The Semantics of a Simple Language for Parallel Programming*. In: *Proceedings of the IFIP Congress*, Seiten 471–475, Stockholm, Sweden, 1974.
251. KALLA, P., M. CIESIELSKI, E. BOUTILLON und E. MARTIN: *High-Level Design Verification Using Taylor Expansion Diagrams: First Results*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 13–17, 2002.
252. KATZ, S. und D. PELED: *Verification of Distributed Programs Using Representative Interleaving Sequences*. Journal of Distributed Computing, 6(2):107–120, 1992.
253. KEBSCHULL, U., E. SCHUBERT und W. ROSENSTIEL: *Multilevel Logic Synthesis based on Functional Decision Diagrams*. In: *Proceedings of the European Conference on Design Automation (ECDA)*, Seiten 43–47, 1992.
254. KEINERT, J., M. STREUBÜHR, T. SCHLICHTER, J. FALK, J. GLADIGAU, C. HAUBELT, J. TEICH und M. MEREDITH: *SystemCoDesigner - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications*. ACM Transactions on Design Automation of Electronic Systems (TODAES), 14(1):1–23, 2009.
255. KELLY, J. C. und K. KEMP: *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*. Technischer Bericht NASA-GB-002-95, NASA, Office of Safety and Mission Assurance, 1995.
256. KEMPF, T., M. DOERPER, R. LEUPERS, G. ASCHEID, H. MEYR, T. KOGEL und B. VANTHOURNOUT: *A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 876–881, 2005.
257. KERNIGHAN, B. W. und D. RITCHIE: *The C Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., 1988. 2. Auflage.
258. KIENHUIS, B., E. DEPRETTERE, K. VISSERS und P. VAN DER WOLF: *An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures*. In: *Proceedings of the Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Seiten 338–349, 1997.
259. KINDLER, E. und T. VESPER: *ESTL: A Temporal Logic for Events and States*. In: *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN)*, Seiten 365–384, 1998.
260. KING, J. C.: *Symbolic Execution and Program Testing*. Communications of the ACM, 19(7):385–394, 1976.
261. KIRCHNER, H., S. RANISE, C. RINGEISSEN und D. K. TRAN: *On Superposition-Based Satisfiability Procedures and Their Combination*. In: *Proceedings of the International Conference on Theoretical Aspects of Computing (ICTAC)*, Seiten 594–608, 2005.
262. KLEIN, M.: *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, Boston, MA, U.S.A., 1993.
263. KLINGERMAN, E. und A. D. STOYENKO: *Real-Time Euclid: A Language for Reliable Real-Time Systems*. IEEE Transactions on Software Engineering, SE-12(9):941–949, 1986.
264. KOELBL, A., Y. LU und A. MATHUR: *Embedded Tutorial: Formal Equivalence Checking Between System-Level Models and RTL*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 965–971, 2005.
265. KONDRATYEV, A., M. KISHINEVSKY, A. TAUBIN und S. TEN: *A Structural Approach for the Analysis of Petri Nets by Reduced Unfoldings*. In: *Proceedings of the Conference on Application and Theory of Petri Nets*, Seiten 346–365, 1996.

266. KOPETZ, H.: *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, MA, U.S.A., 1997.
267. KOREN, I. und C. M. KRISHNA: *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, U.S.A., 2007.
268. KRIPKE, S. A.: *A Completeness Theorem in Modal Logic*. The Journal of Symbolic Logic, 24(1):1–14, 1959.
269. KRIPKE, S. A.: *Semantical Considerations on Modal Logic*. Acta Philosophica Fennica, 16:83–94, 1963.
270. KROENING, D. und E. CLARKE: *Checking Consistency of C and Verilog Using Predicate Abstraction and Induction*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 66–72, 2004.
271. KROENING, D. und N. SHARYGINA: *Formal Verification of SystemC by Automatic Hardware/Software Partitioning*. In: *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Seiten 101–110, 2005.
272. KROPF, THOMAS: *Introduction to Formal Hardware Verification*. Springer, Berlin, Heidelberg, 1999.
273. KUEHLMANN, A. und F. KROHM: *Equivalence Checking Using Cuts and Heaps*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 263–268, 1997.
274. KUEHLMANN, A., V. PARUTHI, F. KROHM und M. K. GANAI: *Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21(12):1377–1394, 2002.
275. KUNZ, W., D. K. PRADHAN und S. M. REDDY: *A Novel Framework for Logic Verification in a Synthesis Environment*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 15(1):20–32, 1996.
276. KUNZ, W. und D. STOFFEL: *Reasoning in Boolean Networks*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.
277. KÜNZLI, S., F. POLETTI, L. BENINI und L. THIELE: *Combining Simulation and Formal Methods for System-Level Performance Analysis*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 236–241, 2006.
278. LAHBIB, Y., A. PERRIN, L. MAILLET-CONTOZ, A. CLOUARD, F. GHENASSIA und R. TOURKI: *Enriching the Boolean and the Modeling Layers of PSL with SystemC and TLM Flavors*. In: *Proceedings of the Forum on Design Languages (FDL)*, Seiten 273–278, 2006.
279. LAHIRI, K., A. RAGHUNATHAN und S. DEY: *System-Level Performance Analysis for Designing On-Chip Communication Architectures*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(6):768–783, Juni 2001.
280. LAHIRI, S. K. und S. A. SESHIA: *The UCLID Decision Procedure*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 475–478, 2004.
281. LAI, Y.-T. und S. SASTRY: *Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 608–613, 1992.
282. LALA, P. K.: *Fault Tolerant and Fault Testable Hardware Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, U.S.A., 1985.
283. LALA, P. K. (Herausgeber): *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, U.S.A., 2001.
284. LAM, W. K.: *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Pearson Studium, Deutschland, Upper Saddle River, NJ, U.S.A., 2005.

285. LAMPKA, K., S. PERATHONER und L. THIELE: *Analytic Real-Time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-Time Systems*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Seiten 107–116, 2009.
286. LAMPORT, L.: *"Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 174–185, 1980.
287. LAROUSSINIE, F., N. MARKEY und P. SCHNOEBELEN: *On Model Checking Durational Kripke Structures*. In: *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, Seiten 264–279, 2002.
288. LAROUSSINIE, F., P. SCHNOEBELEN, und M. TURUANI: *On the Expressivity and Complexity of Quantitative Branching-Time Temporal Logics*. In: *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*, Seiten 437–446, 2000.
289. LARRABEE, T.: *Test Pattern Generation Using Boolean Satisfiability*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 11(1):4–15, 1992.
290. LASKI, J. W.: *On Data Flow Guided Program Testing*. ACM SIGPLAN Notices, 17(9):62–71, 1982.
291. LASKI, J. W. und B. KOREL: *A Data Flow Oriented Program Testing Strategy*. IEEE Transactions on Software Engineering, SE-9(3):347–354, 1983.
292. LAWLER, E. L.: *Optimal Sequencing of a Single Machine Subject to Precedence Constraints*. Management Science, 19:544–546, 1973.
293. LE BOUDEC, J.Y. und P. THIRAN: *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, New York, NY, U.S.A., 2001.
294. LEE, C. Y.: *Representation of Switching Circuits by Binary-Decision Programs*. Bell Systems Technical Journal, 38:985–999, 1959.
295. LEE, E. A.: *Dataflow Process Networks*. Technischer Bericht UCB/ERL 94/53, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.
296. LEE, E. A. und D. G. MESSERSCHMITT: *Synchronous Data Flow*. Proceedings of the IEEE, 75(9):1235–1245, 1987.
297. LEE, E. A., S. NEUENDORFFER und M. J. WIRTHLIN: *Actor-Oriented Design of Embedded Hardware and Software Systems*. Journal of Circuits, Systems, and Computers, 12(3):231–260, 2003.
298. LEE, E. A. und A. L. SANGIOVANNI-VINCENTELLI: *A Framework for Comparing Models of Computation*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(12):1217–1229, 1998.
299. LEHOCZKY, J.: *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*. In: *Proceedings of the Real-Time Systems Symposium (RTSS)*, Seiten 201–209, 1990.
300. LEHOCZKY, J. P. und L. SHA: *Performance of Real-Time Bus Scheduling Algorithms*. In: *International Conference on Measurement and Modeling of Computer Systems*, Seiten 44–53, 1986.
301. LEUNG, J. und J. WHITEHEAD: *On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks*. Performance Evaluation, 2(4):237–250, 1982.
302. LI, Y.-T. S. und S. MALIK: *Performance Analysis of Embedded Software Using Implicit Path Enumeration*. ACM SIGPLAN Notices, 30(11):88–98, 1995.
303. LI, Y.-T. S., S. MALIK und A. WOLFE: *Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software*. In: *Proceedings of the Real-Time Systems Symposium (RTSS)*, Seiten 298–307, 1995.

304. LICHTENSTEIN, O. und A. PNUELI: *Checking that Finite State Concurrent Programs Satisfy their Linear Specification.* In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 97–107, 1985.
305. LIGGESMEYER, P.: *Software-Qualität – Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.
306. LIGGESMEYER, P. und D. ROMBACH: *Software Egineering eingebetteter Systeme – Grundlagen - Methodik - Anwendungen.* Elsevier GmbH, München, 2005.
307. LIU, C. L. und J. W. LAYLAND: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, 20(1):46–61, 1973.
308. LIU, J.: *Real-Time Systems.* Prentice-Hall, Inc., Boston, MA, U.S.A., 2000.
309. LU, R. und C.-K. KOH: *Performance Analysis of Latency-Insensitive Systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(3):469–483, 2006.
310. MAHFOUDH, M., P. NIEBERT, E. ASARIN und O. MALER: *A Satisfiability Checker for Difference Logic.* In: *Proceedings of the Symposium on Theory and Applications of Satisfiability Testing (SAT)*, Seiten 222–230, 2002.
311. MANO, M. M. und C. R. KIME: *Logic and Computer Design Fundamentals.* Pearson Studium, Deutschland, Upper Saddle River, NJ, U.S.A., 2008. 4. Auflage.
312. MARKEY, N. und P. SCHNOEBELEN: *Symbolic Model Checking of Simply-Timed Systems.* In: *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systmes (FORMATS)*, Seiten 102–117, 2004.
313. MARQUES-SILVA, J. P.: *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms.* In: *Proceedings of the Portuguese Conference on Artificial Intelligence (EPIA)*, Seiten 62–74, 1999.
314. MARQUES-SILVA, J. P. und K. A. SAKALLAH: *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transactions on Computers, 48(5):506–521, 1999.
315. MATSUMOTO, T., H. SAITO und M. FUJITA: *Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs.* In: *Proceedings of the International Symposium on Quality of Electronic Design (ISQED)*, Seiten 370–375, 2006.
316. MCMILLAN, K. L.: *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, Norwell, MA, U.S.A., 1993.
317. MCMILLAN, K. L.: *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 164–177, 1993.
318. MCMILLAN, K. L.: *Trace Theoretic Verification of Asynchronous Circuits Using Unfoldings.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 180–195, 1995.
319. MCMILLAN, K. L.: *Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 110–121, 1998.
320. MCMILLAN, K. L.: *Interpolation and SAT-Based Model Checking.* In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 1–13, 2003.
321. MCMILLAN, K. L. und N. AMLA: *Automatic Abstraction without Counterexamples.* In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Seiten 2–17, 2003.
322. MCNAUGHTON, R.: *Testing and Generating Infinite Sequences by a Finite Automaton.* Information and Control, 9(5):521–530, 1966.
323. MCNAUGHTON, R. und H. YAMADA: *Regular Expressions and State Graphs for Automata.* IRE Transactions on Electronic Computers, EC-9(1):39–47, 1960.

324. MEINEL, C. und T. THEOBALD: *Algorithmen und Datenstrukturen im VLSI-Design – OBDD-Grundlagen und -Anwendungen*. Springer, Berlin, Heidelberg, 1998.
325. MELHAM, T.: *Abstraction Mechanisms for Hardware Verification*. In: *VLSI Specification, Verification and Synthesis*, Seiten 129–157, 1988.
326. MERLIN, P. M.: *A Study of the Recoverability of Computing Systems*. Doktorarbeit, University of California, Irvine, Irvine, CA, U.S.A., 1974.
327. MINÉ, A.: *The Octagon Abstract Domain*. Higher Order and Symbolic Computation, 19(1):31–100, 2006.
328. http://www.model.com/.
329. MOLITOR, P. und J. MOHNKE: *Equivalence Checking of Digital Circuits*. Kluwer Academic Publishers, Boston, 2004.
330. MÖLLER, K.-H.: *Ausgangsdaten für Qualitätsmetriken – Eine Fundgrube für Analysen*. In: EBERT, C. und R. DUMKE (Herausgeber): *Software-Metriken in der Praxis*, Seiten 105 – 116. Springer, Berlin, 1996.
331. MOON, I.-H., J.H. KUKULA, K. RAVI und F. SOMENZI: *To Split or to Conjoin: The Question in Image Computation*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 23–28, 2000.
332. MOORE, G.: *Cramming More Components onto Integrated Circuits*. Electronics, 38:114–117, 1965.
333. MORIN-ALLORY, K. und D. BORRIONE: *A Proof of Correctness for the Construction of Property Monitors*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 237–244, 2005.
334. MORIN-ALLORY, K. und D. BORRIONE: *On-Line Monitoring of Properties Built on Regular Expressions*. In: *Proceedings of the Forum on Design Languages (FDL)*, Seiten 249–254, 2006.
335. MORIN-ALLORY, K. und D. BORRIONE: *Proven Correct Monitors from PSL Specifications*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 1246–1251, 2006.
336. MOSKEWICZ, M. W., C. F. MADIGAN, Y. ZHAO, L. ZHANG und S. MALIK: *Chaff: Engineering an Efficient SAT Solver*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 530–535, 2001.
337. MOY, M., F. MARANINCHI und L. MAILLET-CONTOZ: *LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level*. In: *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, Seiten 26–35, 2005.
338. MURATA, T.: *Petri Nets: Properties, Analysis, and Applications*. Proceedings of the IEEE, 77(4):541–580, 1989.
339. MUSUVATHI, M., D. Y. W. PARK, A. CHOU, D. R. ENGLER und D. L. DILL: *CMC: A Pragmatic Approach to Model Checking Real Code*. ACM SIGOPS Operating Systems Review, 36(SI):75–88, 2002.
340. NADEL, ALEXANDER: *Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations*. Master Thesis, The Hebrew University of Jerusalem, Israel, 2002.
341. NAUR, P.: *Checking of Operand Types in ALGOL Compilers*. BIT Numerical Mathematics, 5(3):151–163, 1965.
342. NELSON, C. G. und D. C. OPPEN: *Simplification by Cooperating Decision Procedures*. ACM Transactions on Programming Languages and Systems (TOPLAS), 1(2):245–257, 1979.
343. NIELSON, F., H. R. NIELSON und C. HANKIN: *Principles of Program Analysis*. Springer, Heidelberg, Berlin, New York, 2005. 2. Auflage.

344. NIEMANN, B. und C. HAUBELT: *Assertion-Based Verification of Transaction Level Models*. In: *Proceedings of Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, Seiten 232–236, 2006.
345. NIEMANN, B. und C. HAUBELT: *Formalizing TLM with Communicating State Machines*. In: *Proceedings of the Forum on Design Languages (FDL)*, Seiten 285–292, 2006.
346. NIEMANN, B. und C. HAUBELT: *Towards a Unified Execution Model for Transactions in TLM*. In: *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Seiten 103–112, 2007.
347. NIEMANN, B., C. HAUBELT, M. URIBE und J. TEICH: *Formalizing TLM with Communicating State Machines*. In: *Advances in Design and Specification Languages for Embedded Systems*, Seiten 225–242. Springer, 2007.
348. NIEUWENHUIS, R. und A. OLIVERAS: *DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 321–334, 2005.
349. NTAFOS, S. C.: *On Required Element Testing*. IEEE Transactions on Software Engineering, SE-10(6):795–803, 1984.
350. NTAFOS, S. C.: *A Comparison of some Structural Testing Strategies*. IEEE Transactions on Software Engineering, 14(6):868–874, 1988.
351. http://www.accellera.org/.
352. OSCI TLM WORKING GROUP: *OSCI TLM-2.0 Language Reference Manual*, 2009. Version JA32, http://www.systemc.org.
353. PARTHASARATHY, G., M. K. IYER, K.-T. CHENG und L.-C. WANG: *An Efficient Finite-Domain Constraint Solver for Circuits*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 212–217, 2004.
354. PASRICHA, S., N. DUTT, E. BOZORGZADEH und M. BEN-ROMDHANE: *FABSYN: Floorplan-aware Bus Architecture Synthesis*. IEEE Transactions on Very Large Scale Integrated Systems, 14(3):241–253, 2006.
355. PATTERSON, D. A. und J. L. HENNESSY: *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, U.S.A., 1997. 2. Auflage.
356. PCI SPECIAL INTEREST GROUP: *PCI Local Bus Specification*, 1998. Version 2.2.
357. PELED, D.: *All from One, One for All: On Model Checking Using Representatives*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 409–423, 1993.
358. PELED, D.: *Combining Partial Order Reductions with On-the-Fly Model-Checking*. Journal of Formal Methods in System Design, 8(1):39–64, 1996.
359. PETERSON, J. L.: *Petri Net Theory and Modeling of Systems*. Prentice-Hall, Inc., Reading, MA, 1981.
360. PETRI, C. A.: *Interpretations of a Net Theory*. Technischer Bericht 75–07, GMD, Bonn, Germany, 1975.
361. PIERRE, L. und L. FERRO: *A Tractable and Fast Method for Monitoring SystemC TLM Specifications*. IEEE Transactions on Computers, 57(10):1346–1356, 2008.
362. PILARSKI, S. und G. HU: *SAT with Partial Clauses and Back-Leaps*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 743–746, 2002.
363. PIMENTEL, A. D., C. ERBAS und S. POLSTRA: *A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels*. IEEE Transactions on Computers, 55(2):99–112, 2006.
364. PNUELI, A.: *The Temporal Logic of Programs*. In: *Proceedings of the Symposium on Foundations of Computer Science*, Seiten 46–57, 1977.

365. POP, T., P. ELES und Z. PENG: *Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems*. In: *Proceedings of the Conference on Hardware/Software Codesign (CODES)*, Seiten 187–192, 2002.
366. PRASAD, M., A. BIERE und A. GUPTA: *A Survey of Recent Advances in SAT-Based Formal Verification*. International Journal on Software Tools for Technology Transfer (STTT), 7(2):156–173, 2005.
367. PROSSER, PATRICK: *Hybrid Algorithms for the Constraint Satisfaction Problem*. Computational Intelligence, 9(3):268–299, 1993.
368. PUSCHNER, P. und C. KOZA: *Calculating the Maximum, Execution Time of Real-Time Programs*. Real-Time Systems, 1(2):159–176, 1989.
369. QUEILLE, J.-P. und J. SIFAKIS: *Specification and Verification of Concurrent Systems in CESAR*. In: *Proceedings of the Symposium on Programming*, Seiten 337–351, 1982.
370. RAMCHANDANI, C.: *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Doktorarbeit, Massachusetts Institute of Technology, Cambridge, MA, U.S.A., 1974.
371. RAVI, K. und F. SOMENZI: *High-Density Reachability Analysis*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 154–158, 1995.
372. REDA, S. und A. SALEM: *Combinational Equivalence Checking Using Boolean Satisfiability and Binary Decision Diagrams*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 122–126, 2001.
373. REISIG, W.: *A Primer in Petri Net Design*. Springer, Berlin, Heidelberg, New York, Tokyo, 1992.
374. REISIG, W.: *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, Berlin, 1998.
375. REITER, R.: *Scheduling Parallel Computations*. Journal of the ACM, 15(4):590–599, 1968.
376. REYNOLDS, J. C.: *Automatic Computation of Data Set Definitions*. In: *Proceedings of IFIP Congress*, Band 1, Seiten 456–461, 1968.
377. RICHTER, K.: *Compositional Scheduling Analysis Using Standard Event Models*. Doktorarbeit, Technische Universität Braunschweig, Deutschland, 2004.
378. RICHTER, K. und R. ERNST: *Event Model Interfaces for Heterogeneous System Analysis*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 506–513, 2002.
379. RICHTER, K., M. JERSAK und R. ERNST: *A Formal Approach to MPSoC Performance Verification*. IEEE Computer, 36(4):60–67, 2003.
380. RICHTER, K., D. ZIEGENBEIN, M. JERSAK und R. ERNST: *Model Composition for Scheduling Analysis in Platform Design*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 287–292, 2002.
381. ROTH, J. P.: *Diagnosis of Automata Failures: A Calculus and a Method*. IBM Journal of Research and Development, 10(4):278–291, 1966.
382. RTCA: *Software Considerations in Airborne Systems and Equipment Certification*. DO-178B, 1992.
383. RUDELL, R.: *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 42–47, 1993.
384. http://www.haifa.ibm.com/projects/verification/RB_Homepage/.
385. SAFRA, S.: *On the Complexity of ω-Automata*. In: *Proceedings of the Symposium on Foundations of Computer Science*, Seiten 319–327, 1988.

386. SCHIRNER, G. und R. DÖMER: *Quantitative Analysis of Transaction Level Models for the AMBA Bus*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 230–235, 2006.
387. SCHIRNER, G., A. GERSTLAUER und R. DÖMER: *Abstract, Multifaceted Modeling of Embedded Processors for System Level Design*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 384–389, 2007.
388. SCHLIECKER, S., S. STEIN und R. ERNST: *Performance Analysis of Complex Systems by Integration of Dataflow Graphs and Compositional Performance Analysis*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 273–278, 2007.
389. SCHMIDT, D. A.: *Data Flow Analysis is Model Checking of Abstract Interpretations*. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Seiten 38–48, 1998.
390. SCHNEIDER, K.: *Verification of Reactive Systems – Formal Methods and Algorithms*. Springer, Berlin, Heidelberg, 2004.
391. SCHNERR, J., O. BRINGMANN, A. VIEHL und W. ROSENSTIEL: *High-Performance Timing Simulation of Embedded Software*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 290–295, 2008.
392. SEBASTIANI, ROBERTO: *Lazy Satisfiability Modulo Theories*. Journal on Satisfiability, Boolean Modeling and Computation, 3:141–224, 2007.
393. SHANNON, C. E.: *The Synthesis of Two-Terminal Switching Circuits*. Bell Systems Technical Journal, 28:59–98, 1949.
394. SHASHIDHAR, K. C., M. BRUYNOOGHE, F. CATTHOOR und G. JANSSENS: *Automatic Functional Verification of Memory Oriented Global Source Code Transformations*. In: *Proceedings of the High-Level Design Validation and Test Workshop (HLDVT)*, Seiten 31–36, 2003.
395. SHASHIDHAR, K. C., M. BRUYNOOGHE, F. CATTHOOR und G. JANSSENS: *Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 1310–1315, 2005.
396. SHASHIDHAR, K. C., M. BRUYNOOGHE, F. CATTHOOR und G. JANSSENS: *Verification of Source Code Transformations by Program Equivalence Checking* . In: *Compiler Construction*, Seiten 221–236. Springer, 2005.
397. SHEINI, H. M. und K. A. SAKALLAH: *A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic*. In: *Theory and Applications of Satisfiability Testing*, Seiten 241–256, 2005.
398. SHEKHAR, N., P. KALLA und F. ENESCU: *Equivalence Verification of Polynomial Datapaths Using Ideal Membership Testing*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(7):1320–1330, 2007.
399. SHOSTAK, R. E.: *A Practical Decision Procedure for Arithmetic with Function Symbols*. Journal of the ACM, 26(2):351–360, 1979.
400. SHOSTAK, R. E.: *Deciding Combinations of Theories*. Journal of the ACM, 31(1):1–12, 1984.
401. SHTRICHMAN, O.: *Pruning Techniques for the SAT-Based Bounded Model Checking Problem*. In: *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, Seiten 58–70, 2001.
402. SIEBENBORN, A., A. VIEHL, O. BRINGMANN und W. ROSENSTIEL: *Control-Flow Aware Communication and Conflict Analysis of Parallel Processes*. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seiten 32–37, 2007.

403. SKAKKEBÆK, J. U., R. B. JONES und D. L. DILL: *Formal Verification of Out-of-Order Execution Using Incremental Flushing*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 98–109, 1998.
404. http://www.averant.com/products-solidify.html.
405. SOMENZI, F. und R. BLOEM: *Efficient Büchi Automata from LTL Formulae*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 248–263, 2000.
406. SPURI, M.: *Earliest Deadline Scheduled in Real-Time Systems*. Doktorarbeit, INRIA, Le Chesnay, Cedex, France, 1995.
407. SPURI, M.: *Analysis of Deadline Scheduled Real-Time Tasks*. Technischer Bericht, INRIA, Le Chesnay, Cedex, France, 1996.
408. STANKOVIC, J., M. SPURI, K. RAMAMRITHAM und G. BUTTAZZO: *Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms*. Kluwer Academic Publishers, Boston, MA, U.S.A., 1998.
409. STARKE, P. H.: *Analyse von Petri-Netz-Modellen*. Teubner, Stuttgart, 1990.
410. STEFFEN, B.: *Data Flow Analysis as Model Checking*. In: *'Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*, Seiten 346–365, 1991.
411. STOFFEL, D. und W. KUNZ: *Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 394–399, 1997.
412. STOFFEL, D., M. WEDLER, P. WARKENTIN und W. KUNZ: *Structural FSM traversal*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(5):598–619, 2004.
413. STRAHM, T.: *Logik in Informatik, Mathematik und Philosophie*, 1999. Vortrag anlässlich der Veranstaltung Theodor-Kocher-Preis 1998 der Universität Bern.
414. STREHL, K.: *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*. Doktorarbeit, Swiss Federal Institute of Technology Zurich, Switzerland, 2000.
415. STREHL, K. und L. THIELE: *Symbolic Model Checking of Process Networks Using Interval Diagram Techniques*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 686–692, 1998.
416. STREHL, K. und L. THIELE: *Interval Diagram Techniques for Symbolic Model Checking of Petri Nets*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 756–757, 1999.
417. STREHL, K., L. THIELE, M. GRIES, D. ZIEGENBEIN, R. ERNST und J. TEICH: *FunState – An Internal Design Representation for Codesign*. IEEE Transactions on Very Large Scale Integrated Systems, 9(4):524–544, 2001.
418. STREHL, K., L. THIELE, D. ZIEGENBEIN, R. ERNST und J. TEICH: *Scheduling Hardware/Software Systems Using Symbolic Techniques*. In: *Proceedings of the Conference on Hardware/Software Codesign (CODES)*, Seiten 173–177, 1999.
419. STREUBÜHR, M., J. FALK, C. HAUBELT, J. TEICH, R. DORSCH und T. SCHLIPF: *Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 480–481, 2006.
420. STREUBÜHR, M., J. GLADIGAU, C. HAUBELT und J. TEICH: *Efficient Approximately-Timed Performance Modeling for Architectural Exploration of MPSoCs*. In: *Proceedings of the Forum on Design Languages (FDL)*, 2009.
421. STROUSTRUP, B.: *The C++ Programming Language: Language Library and Design Tutorial*. Addison-Wesley, Amsterdam, 1997.

422. SUTHERLAND, S., S. DAVIDMANN, P. FLAKE und P. MOORBY: *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Kluwer Academic Publishers, Norwell, MA, U.S.A., 2006. 2. Auflage.
423. http://www.symtavision.com/symtas.html.
424. SYSTEMC VERIFICATION WORKING GROUP: *SystemC Verification Standard Specification*, 2006. Version 1.0e, http://www.systemc.org.
425. TEICH, J.: *Embedded System Synthesis and Optimization*. In: *Proceedings of the Workshop on System Design Automation (SDA)*, Seiten 9–22, 2000.
426. TEICH, J. und C. HAUBELT: *Digitale Hardware/Software-Systeme – Synthese und Optimierung*. Springer, Berlin, Heidelberg, 2007. 2. erweiterte Auflage.
427. THAYSE, A., M. DAVIO und J.-P. DESCHAMPS: *Optimization of Multivalued Decision Algorithms*. In: *Proceedings of the International Symposium on Multiple-Valued Logic*, Seiten 171–178, 1978.
428. THEILING, H., C. FERDINAND und R. WILHELM: *Fast and Precise WCET Prediction by Separated Cache and Path Analyses*. Real-Time Systems, 18(2–3):157–179, 2000.
429. THIELE, L., S. CHAKRABORTY, M. GRIES und S. KÜNZLI: *Design Space Exploration of Network Processor Architectures*. Network Processor Design: Issues and Practices, 1:55–89, 2002.
430. THIELE, L., S. CHAKRABORTY, M. GRIES, A. MAXIAGUINE und J. GREUTERT: *Embedded Software in Network Processors – Models and Algorithms*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Seiten 416–434, 2001.
431. THIELE, L., S. CHAKRABORTY und M. NAEDELE: *Real-Time Calculus for Scheduling Hard Real-Time Systems*. In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, Seiten 101–104, 2000.
432. THIELE, L. und N. STOIMENOV: *Modular Performance Analysis of Cyclic Dataflow Graphs*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Seiten 127–136, 2009.
433. THIELE, L., K. STREHL, D. ZIEGENBEIN, R. ERNST und J. TEICH: *FunState – An Internal Design Representation for Codesign*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 558–565, 1999.
434. THIELE, L., J. TEICH und K. STREHL: *Regular State Machines*. Journal of Parallel Algorithms and Applications, 15(3-4):265–300, 2000.
435. THIELE, L. und E. WANDELER: *Performance Analysis of Distributed Embedded Systems*. In: *Embedded Systems Handbook*, Seiten 15.1–15.18. CRC Press, Boca Raton, FL, 2006.
436. THIRIOUX, X.: *Simple and Efficient Translation from LTL Formulas to Büchi Automata*. In: *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Seiten 145–159, 2002.
437. THOMPSON, K.: *Programming Techniques: Regular Expression Search Algorithm*. Communications of the ACM, 11(6):419–422, 1968.
438. TIETZE, U. und C. SCHENK: *Halbleiter-Schaltungstechnik*. Springer, Berlin, Heidelberg, 2002. 12. Auflage.
439. TINDELL, K., A. BURNS und A. WELLINGS: *An Extendible Approach for Analysing Fixed Priority Hard Real-Time Systems*. Real-Time Systems, 6(2):133–152, 1994.
440. TINDELL, K., A. BURNS und A. J. WELLINGS: *Calculating Controller Area Network (CAN) Message Response Times*. Control Engineering Practice, 3(8):1163–1169, 1995.
441. TINDELL, K. und J. CLARK: *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems*. Microprocessing and Microprogramming, 40(2–3):117–134, 1994.

442. TOVCHIGRECHKO, A. A.: *Efficient symbolic analysis of bounded Petri nets using Interval Decision Diagrams*. Doktorarbeit, Technische Universität Cottbus, Deutschland, 2008.
443. http://www.tttech.com/.
444. TURING, A.: *On Computable Numbers With An Application To The Entscheidungs Problem*. In: *Proceedings of The London Mathematical Society*, Band 42 der Reihe *2*, Seiten 230–265, 1937.
445. http://www.uml.org.
446. VALMARI, A.: *Stubborn Sets for Reduced State Space Generation*. In: *Proceedings of the International Conference on Applications and Theory of Petri Nets*, Seiten 491–515, 1991.
447. VALMARI, A.: *A Stubborn Attack on State Explosion*. Journal of Formal Methods in System Design, 1(4):297–322, 1992.
448. VALMARI, A.: *On-the-Fly Verification with Stubborn Sets*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 397–408, 1993.
449. VARDI, M. Y. und P. WOLPER: *An Automata-Theoretic Approach to Automatic Program Verification*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS)*, Seiten 332–344, 1986.
450. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx.
451. VELEV, M. N. und R. E. BRYANT: *Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking*. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Seiten 18–35, 1998.
452. VELEV, M. N. und R. E. BRYANT: *Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic*. In: *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, Seiten 37–53, 1999.
453. VELEV, M. N. und R. E. BRYANT: *Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 112–117, 2000.
454. VIEHL, A., M. PRESSLER und O. BRINGMANN: *Bottom-Up Performance Analysis Considering Time Slice Based Software Scheduling at System Level*. In: *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seiten 423–432, 2009.
455. VIEHL, A., M. PRESSLER, O. BRINGMANN und W. ROSENSTIEL: *White Box Performance Analysis Considering Static Non-Preemptive Software Scheduling*. In: *Proceedings of the Design, Automation and Test in Europe (DATE)*, Seiten 513–518, 2009.
456. VISSER, W., K. HAVELUND, G. BRAT, S. PARK und F. LERDA: *Model Checking Programs*. Automated Software Engineering (ASE), 10(2):203–232, 2003.
457. WANG, C., G. D. HACHTEL und F. SOMENZI: *Abstraction Refinement for Large Scale Model Checking*. Springer, New York, NY, U.S.A., 2006.
458. WANG, H. und H. PHAM: *Reliability and Optimal Maintenance*. In: *Springer Series in Reliability Engineering*. Springer, London, 2006.
459. WEDLER, M., D. STOFFEL, R. BRINKMANN und W. KUNZ: *A Normalization Method for Arithmetic Data-Path Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(11):1909–1922, 2007.
460. WEDLER, M., D. STOFFEL und W. KUNZ: *Normalization at the Arithmetic Bit Level*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 457–462, 2005.

461. WEYUKER, E. J. und T. J. OSTRAND: *Theories of Program Testing and the Application of Revealing Subdomains*. IEEE Transactions on Software Engineering, 6(3):236–246, 1980.
462. WHITE, L. J. und E. I. COHEN: *A Domain Strategy for Computer Program Testing*. IEEE Transactions on Software Engineering, 6(3):247–257, 1980.
463. WHITTEMORE, J., J. KIM und K. SAKALLAH: *SATIRE: A New Incremental Satisfiability Engine*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 542–545, 2001.
464. WILHELM, R., J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, F. MUELLER, I. PUAUT, P. PUSCHNER, J. STASCHULAT und P. STENSTRÖM: *The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools*. ACM Transactions on Embedded Computing Systems (TECS), 7(3):1–53, 2008.
465. WILLIAMS, P. F., A. BIERE, E. M. CLARKE und A. GUPTA: *Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 124–138, 2000.
466. WOLF, WAYNE, AHMED AMINE JERRAYA und GRANT MARTIN: *Multiprocessor System-on-Chip (MPSoC) Technology*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(10):1701–1713, 2008.
467. WOLPER, P., M. Y. VARDI und A. P. SISTLA: *Reasoning about Infinite Computation Paths*. In: *Proceedings of the Symposium on Foundations of Computer Science*, Seiten 185–194, 1983.
468. XU, J. und D. L. PARNAS: *Priority Scheduling versus Pre-Run-Time Scheduling*. Real-Time Systems, 18(1):7–24, 2000.
469. YANG, C. H. und D. L. DILL: *Validation with Guided Search of the State Space*. In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 599–604, 1998.
470. YEN, T.-Y. und W. WOLF: *Performance Estimation for Real-Time Distributed Embedded Systems*. IEEE Transactions on Parallel and Distributed Systems, 9(11):1125–1136, 1998.
471. YOVINE, S.: *Model Checking Timed Automata*. In: *European Educational Forum: School on Embedded Systems*, Seiten 114–152, 1996.
472. YUAN, J., C. PIXLEY und A. AZIZ: *Constrained-Based Verification*. Springer, New York, NY, U.S.A., 2006.
473. YUAN, J., J. SHEN, J. A. ABRAHAM und A. AZIZ: *On Combining Formal and Informal Verification*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 376–387, 1997.
474. ZHANG, H.: *SATO: An Efficient Propositional Prover*. In: *Proceedings of the International Conference on Automated Deduction (CADE)*, Seiten 272–275, 1997.
475. ZHANG, L., C. F. MADIGAN, M. H. MOSKEWICZ und S. MALIK: *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Seiten 279–285, 2001.
476. ZHANG, L. und S. MALIK: *The Quest for Efficient Boolean Satisfiability Solvers*. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Seiten 17–36, 2002.
477. ZHU, C., Z. P. GU, R. P. DICK und L. SHANG: *Reliable Multiprocessor System-On-Chip Synthesis*. In: *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Seiten 239–244, 2007.

Sachverzeichnis

Abarbeitungszeit 439
Abbildung 524
Abbildungsgraph 461
Abhängigkeitsabbildung 382
Ablaufplan 348
  iterativer, dynamischer, 442
  Latenz eines, 349
  periodischer, 349
  statischer, 348
Ablaufplanung 18
Abstraktion 21, 109
  Daten-, 110
  existentielle, 427
  funktionale, 110
  strukturelle, 109
  Zeit-, 110
Abstraktionsebene 14
Abstraktionsverfeinerung 98–99, 425–476
ADD 275
ADDG 381, 383
Adjazenz von Knoten 529
Äquivalenz
  Eingabe/Ausgabe-, 381
  -relation, 524
  von Mikroarchitektur und ISA, 295
  zweier Assemblerprogramme, 365
  zweier C-Programme, 376
  zweier LTS, 467
  zweier Schaltungen, 236
Äquivalenzklasse 133, 366, 373, 392
  Bildung von, 133
    funktionsorientierte 133, 392–396
    strukturorientierte 133
  gültige, 133, 392
  ungültige, 133, 392
Äquivalenzprüfung 21, 95, 115–154
  auf Architekturebene, 273–280, 283–322
  auf Blockebene, 362–391
  auf Logikebene, 236–273
  explizite, 116, 129–140, 236, 246
  Hardware-, 236–322
  implizite, 116–129, 236, 242
  kombinatorische, 117
  sequentielle, 117, 141–152
  Software-, 362–391
  strukturelle, 116, 152–154, 236, 263, 368
  von Assemblerprogrammen, 362–372
  von C-Programmen, 373–391
  von Prozessoren, 291–322
  zwischen Architektur- und Logikebene, 280–283
Aktionsintervall 462
Aktivierungsfront 255
Aktivität 51
Algorithmus
  Approximations-, 541
  Definition, 541
  Effizienz, 542
  exakter, 541
  exponentieller, 542
  Komplexität, 541
  Optimalität, 542
  polynomieller, 542
  Problemgröße, 541
  von Tomasulo, 314
Alias-Analyse 449

Allokation 18
Alphabet 190
alternativer Programmfluss 56
Anfangszustand 47
Anforderung 16, 17, 37, 40
  funktionale, 40, 72
  nichtfunktionale, 40, 72
Anforderungsanalyse 4
Ankunftsfunktion 515
Ankunftskennlinie 509
  obere, 509
  untere, 509
Ankunftszeit 438
Annahme 88
Antwortzeit 439
Architekturebene 14, 15
Array-Datenabhängigkeitsgraph *siehe* ADDG
atomare Bedingungen 404
ATPG 113, 246, 356
Ausdruck 298
  ω-regulärer, 186, 190
  regulärer, 186, 190
Ausdruckskraft 56
Ausführungssemantik 39
Ausfall 5
Ausgabealphabet 47
Ausgabefunktion 48
  erweiterte, 142
Ausgaberaum 108
Ausgaberelation 47
Aussagenlogik 23, 73, 527
  atomare Formel, 527
  Definitionen, 527–528
  Formel, 527
    erfüllbare 527
    gültige 528
  Interpretation, 527
  Modell, 527
  Semantik, 527
  Syntax, 527
  temporale, 73, 75–83
Automat 15, 47
  endlicher, 51
    deterministischer 47, 141, 258
    kommunizierender 480
    nichtdeterministischer 47, 48, 190
    Zustand 47
  sequentielle Tiefe, 146
  verhaltensgleich, 142
  zeitbehafteter, 49–51, 214, 220
    Anfangszustand 49, 220
    Invariante 49
    Modellprüfung 214–222
    Prüfung des Zeitverhaltens 214–222
    Region 215–218
    Simulation 220–222
    Zeitvariable 49
    Zone 218–219
    Zustand 49, 220
Automaten-Äquivalenz 142, 143, 145
Automatentabelle 399
Axiom 23

back-to-back test *siehe* Ende-zu-Ende-Test
backjumping 545
backtracking *siehe* Zurückverfolgung
Baum 530
BCET-Analyse 432–438
BCP 249, 544
BDD 113, 533–539
  Definition, 534
  geordnetes, *siehe* OBDD
  Multi-Terminal, *siehe* MTBDD
  verallgemeinertes, 537–539
BDF-Graph 55
Beobachtbarkeit 109, 110
Berechnungsbaum 73
Berechnungsfehler 134
Berechnungsmodell 16, 39
Bereichsfehler 134
Bereichstest 133–134
Beschränktheit
  eines Petri-Netzes, 44
  von dynamischen Datenflussmodellen, 55
Beschränkung 111
  Umgebungs-, 112
Beschränkungs-Erfüllbarkeitsproblem 112
Betrieb 4
Betriebsfehler 6
Beweis 97–99
Beweisassistent 107
Bindung 18
bipartiter Graph 41
bit flattening 341
Bitvektor 341
Bitvektor-Arithmetik 341
  Syntax, 341

Black-Box 105
Blockebene 14, 15
BMD 275
  Addition, 283
  Definition, 276
  geordneter, *siehe* OBMD
  Multiplikation, 283
  multiplikative, *siehe* *BMD
*BMD 277
  Interpretation von, 277
  Kronecker, *siehe* K*BMD
boolean constraint propagation *siehe* BCP
Boolesche Funktion 15, 524–526
  Belegung, 524
  Erfüllbarkeit, 524, 537
  Gültigkeit, 524, 537
  Konstruktion aus einer kombinatorischen Schaltung, 237
    Rückwärtskonstruktion 237
    Vorwärtskonstruktion 237
Boolescher Ausdruck 56
Boolescher Datenfluss (BDF) 55
Boolesches Netzwerk 525–526
Boolesches Programm 427
bridging fault 5
Büchi-Automat 186, 190
Burst 450

C 27
c-use 410, 412
CDF-Graph 56–57
CEGAR 426
CET 432
CF-Graph 57
  datenflussattributierter, 410
CLP 113
Codierung
  2er-Komplement, 281
  Vorzeichen-Betrag-Darstellung, 281
  vorzeichenlos, 281
computation tree *siehe* Berechnungsbaum
computation tree logic *siehe* CTL
constrained random simulation 102, 111
constraint logic programming *siehe* CLP
constraint satisfaction programming *siehe* Beschränkungs-Erfüllbarkeitsproblem
control flow graph *siehe* CF-Graph
control/dataflow graph *siehe* CDF-Graph
core execution time *siehe* CET
CSDF-Graph 53–55
CTL 72, 77–80, 556
  Erfüllbarkeit, 78
  Fixpunktberechnung, 556–560
  Gültigkeit, 78
  Modellprüfung, 179–185
  Normalform, 80
  Semantik, 78
  Syntax, 78
CTL* 80–82
  Erfüllbarkeit, 82
  Gültigkeit, 82
  Pfadformel, 81
  Semantik, 82
  Syntax, 81
  Zustandsformel, 81
cyclo-static dataflow graph *siehe* CSDF-Graph

D-Algorithmus 254
DAG 531
Datenabhängigkeit 56
Datenfluss 51, 56
  dynamischer, 55
Datenflussanalyse 57
Datenflussgraph 51–57
  Aktor, 51
  dynamischer, 55–56
  Kommunikationsregel, 51
  regulärer, 55
  synchroner , *siehe* SDF-Graph
  zyklostatischer, *siehe* CSDF-Graph
Datenflussmodell
  deterministisches, 55
Datenflusssprache 55
Datenflusstransformation
  globale, 380
Davio-Zerlegung
  negative, 243, 279, 537, 538
  positive, 243, 276, 537
dcu 412
De Morgansches Gesetz 30
deadline 431, 438
deadlock *siehe* Verklemmung
Deduktion 528
def 410, 412
Definitionsbereich 381
DFA 47
  vollständig spezifiziert, 48

Dispatchlatenz 438
domain testing *siehe* Bereichstest
Doppeldachmodell 14–22, 34
  für den Entwurfsprozess, 14
  für den Verifikationsprozess, 18
DPLL-Algorithmus 26
dpu 412

earliest deadline first *siehe* EDF-Algorithmus
*earliest deadline first** *siehe* EDF*-Algorithmus
earliest due date *siehe* EDD-Algorithmus
Echtzeitanalyse 432, 438–448
Echtzeitanforderung 207
  harte, 207
  weiche, 207
Echtzeitkalkül *siehe* RTC
EDD-Algorithmus 440
EDF*-Algorithmus 441
EDF-Algorithmus 440
Eigenschaft 37
  funktionale, 40, 72
    in PSL 84
    Spezifikation 72–88
  nichtfunktionale, 40, 72
    Spezifikation 88–91
Eigenschaftsprüfung 96, 155–232
  funktionale, 21, 96, 156–207
    auf Systemebene 452–490
    von Hardware 323–345
    von Programmen 416–431
  nichtfunktionale, 21, 96, 207–232
    auf Systemebene 490–520
    von Hardware 345–356
    von Programmen 431–448
  zusicherungsbasierte, 28, 104, 111, 188–197, 233
    TLM 484–490
    von Hardware 323–331
Einerklausel 249, 543
Eingabealphabet 47, 49, 220
Eingaberaum 108
eingebettetes System 1
Einmalzuweisung
  dynamische, 381
  statische, 381
Eins-zu-Eins-Abbildung 524
electronic system level *siehe* Systemebene
Eliminationsregel 535, 538
Ende-zu-Ende-Test 130
Entscheidbarkeit 541
Entscheidungsdiagramm 533–534
  algebraisches, *siehe* ADD
  binäres, *siehe* BDD
  freies, 534
  funktionales, *siehe* FDD
  geordnetes, 534
  Größe, 534
  komplettes, 534
  Nichtterminalknoten, 533
  Quellknoten, 534
  Terminalknoten, 533
Entscheidungstabelle 393, 395
Entwicklung 4
Entwurf 4
Entwurfsdreieck 10
Entwurfsentscheidung 17
Entwurfsfehler 5
Entwurfsraumexploration 18
Ereignis
  periodisches, 500
    mit Bursts 500
    mit Jitter 500
  sporadisches, 500
Ereignisadaptierungsfunktion 503
Ereignismodell 500
Ereignismodellschnittstelle 503
Ereignisstrom 499–505, 515
  Ausgabe-, 501
  Eingabe-, 501
  -kopplung, 501–505
Ereigniszähler 220
Erfüllbarkeitsproblem
  Boolesches, 25, 542, 543
    unerfüllbarer Kern 344
Ergebnis
  falschnegatives, 154, 265, 368, 371, 417, 476, 491
  falschpositives, 130, 368, 372, 491
Erreichbarkeitseigenschaft 83
Erreichbarkeitsgraph 158, 209
  reduzierter, 173
error 5
ESL *siehe* Systemebene
EUF 295, 297, 357
Euler-Methode 51

failure 5
false negative *siehe* Ergebnis, falschnegatives
false positive *siehe* Ergebnis, falschpositives
Falsifikation 97–98
fault 5
FCFS-Algorithmus 447, 494
FDD 538
  geordnetes, *siehe* OFDD
  Kronecker, *siehe* KFDD
  negatives, 538
  positives, 537
Fehler 5
  Übertragung, 109, 253
  Aktivierung, 109, 253
Fehlerkorrektur 6
fehleroffenbarende Unterbereiche 139–140
Fehlverhalten 5
first come, first served *siehe* FCFS-Algorithmus
Fixpunkt 146, 147, 556
  eines Funktionals, 556
Flushing 295
  beschleunigtes, 309
Flussbeschränkung 434
Flussrelation eines Petri-Netzes 41
Flusszeit 439
Folgemarkierung 43
FPGA 100
frame by frame simulation 259
Funktion 524
  abstrakte, 419
  bijektive, 524
  Bildmenge einer, 524
  Boolesche, *siehe* Boolesche Funktion
  charakteristische, 148
  Definitionsmenge einer, 524
  injektive, 524
  inverse, 524
  Pseudo-Boolesche, *siehe* Pseudo-Boolesche Funktion
  Repräsentant, 117, 524
  Repräsentation, 524
    eindeutige 117, 524
    Interpretation 117, 524
    kanonische 117, 524
    vollständige 117, 524
  surjektive, 524
  uninterpretierte, 363, 366
  wertebereichäquivalente, 268
  Wertemenge einer, 524
  Zielmenge einer, 524
Funktional 556
  Fixpunkt, 556
    größter 557
    kleinster 557
  Monotonie, 557
  schnittstetiges, 557
  vereinigungsstetiges, 557
Funktionssymbol 297
  Ordnung eines, 298
  uninterpretiertes, 298
Funktionstabelle 525
FunState 58–59
  Definition, 58

Gatterbibliothek 525
Gefahrlosigkeit 40
Gefahrlosigkeitseigenschaft 72, 73, 83, 163, 179
Gegenbeispiel 97
  unzulässiges, 98, 417
  zulässiges, 98
Generator 189
Gleichung 298
  generelle, 357
  positive, 357
Graph 529–531
  azyklischer, 530
  Baum, 530
    Spann- 530
  bipartiter, 530
  einfacher, 530
  Endpunkte einer Kante, 529
  gerichteter, 529, 531
    Ausgangsgrad eines Knotens 531
    azyklischer *siehe* DAG
    direkter Nachfolger eines Knotens 531
    direkter Vorgänger eines Knotens 531
    Eingangsgrad eines Knotens 531
    Inzidenzmatrix 531
    Kantenzug 531
    Nachfolger eines Knotens 531
    polarer 531
    stark zusammenhängender 531

Vorgänger eines Knotens 531
zugehöriger ungerichteter Graph 531
zusammenhängender 531
gewichteter, 531
Kante, 529
Kantenzug, 530
geschlossener 530
Knoten, 529
Grad 529
Komplement, 530
Kreis, 530
kreisfreier, 530
Pfad, 530
Pfadgewicht, 531
Schleife, 530
SDF-, *siehe* SDF-Graph
Teilgraph, 530
ungerichteter, 529–531
Orientierung 531
Untergraph, 530
vollständiger, 530
Wald, 530
Weg, 530
einfacher 530
zusammenhängender, 530
Zyklus, 530
Grundblock 57
Gültigkeit
allgemeine, 299

Haftfehler 5, 246
Haftfehlermodell 253
Halteproblem 24
Hardware-Entwurfsprozess 14
hardware-in-the-loop simulation 105
Herstellungsfehler 5
Heuristik 541
Hoare-Kalkül 26
Hoare-Tripel 26

IDD 456–460
Definition, 458
geordnetes, *siehe* OIDD
Ideal 291
Ignoranzproblem 174
II 457
IIP 457
geordnetes, 458
minimales, 458
reduziertes, 458
ILP 113
0-1-, 113
IMD 456, 460–464
Implementierung 5, 15, 17
prototypische, 100
Implikationsgraph 549
konfliktfreier, 549
Schnitt im, 549
Indexverschiebung 349
instruction set architecture *siehe* ISA
instruction timing addition 436
Instruktionssatzarchitektur *siehe* ISA
Interferenz 446
Interpretation
abstrakte, 419–422
konkrete, 419
Intervall
benachbartes, 457
geschlossenes, 457
halboffenes, 457
offenes, 457
Intervall-Abbildungsdiagramm *siehe* IMD
Intervall-Abbildungsfunktion 461
neutrale, 461
Intervall-Entscheidungsdiagramm *siehe* IDD
Intervallüberdeckung 457
disjunkte, 457
Intervalladdition 461
Intervallpartition 457
Inzidenz von Knoten und Kanten 529
Inzidenzmatrix 531
Irritationsproblem 174
Irrtum 5
ISA 15, 291, 293
Iteration
als Kontrollstruktur, 56
Iterationsbereich 381
Iterationsintervallschranke 231

Jitter 450, 500

Kahn-Prozessnetzwerk 55
Kalkül 23
Kante
gerichtete, 529
Anfangsknoten einer 531
Endknoten einer 531

ungerichtete, 529
Kantenmarkierungsfunktion 158
kartesisches Produkt zweier Mengen 523
K*BMD 280
KFDD 538
geordnetes, *siehe* OKFDD
Klausel 247, 543
erfüllte, 543
leere, 247
unspezifizierte, 543
verletzte, 543
Knotenmarkierungsfunktion 158
Kofaktor 534
negativer, 534
positiver, 534
Kommunikationskanal 55
Komplement eines Graphen 530
Konfiguration 100
Konflikt
von Transitionen, 43
Konfliktanalyse 549
Konservativität 45
Kontext 303
Kontroll-Datenflussgraph *siehe* CDF-Graph
Kontrollfluss 56, 57
alternativer, 56
Kontrollflussgraph *siehe* CF-Graph
Kontrollstruktur 56
Iteration, 56
Verzweigung, 56
konvexe Hülle 421
Korrektheit 9
funktionale, 5
nichtfunktionaler Eigenschaften, 5
Kripke-Struktur *siehe* Temporale Struktur

lateness 440
Latenz 349
latest deadline first *siehe* LDF-Algorithmus
LDF-Algorithmus 440
Lebendigkeit 40, 44
eines Petri-Netzes, 44, 156
Lebendigkeitseigenschaft 72, 73, 83, 163, 179
Lernen
iteratives, 358
konfliktgetriebenes, 549
linear time propositional logic *siehe* LTL
lineare Programmierung 113
ganzzahlige, *siehe* ILP
LIS-Graph 352
Definition, 352
Literal 247, 543
erfülltes, 543
reines, 544, 548
unspezifiziertes, 543
liveness *siehe* Lebendigkeit
Logik 22
Logikebene 14, 15
LTL 72, 75–77
Überdeckung, 192
Codierung, 203–205
Erfüllbarkeit, 77
Gültigkeit, 77
existentielle 200
universelle 200
Modell, 77
Modellprüfung, 185–188
Normalform, 77
SE-, *siehe* SE-LTL
Semantik, 76
beschränkte 201, 202
Syntax, 76
LTS 466–468
Äquivalenz, 467
Abstraktion, 467
parallele Komposition, 467, 468

markierter Graph 51
äquivalentes Petri-Netz, 51
Markierungsprädikat 163
Matrix
differenzenbeschränkte, *siehe* Wertebereich, abstrakter, DBM
Max-Plus-Algebra 230
Mehrzieloptimierungsproblem 18
Menge 523
Überdeckung, 523
der ganzen Zahlen, 523
der natürlichen Zahlen, 523
der rationalen Zahlen, 523
der reellen Zahlen, 523
disjunkte, 523
kartesisches Produkt, 523
Komplement, 523
Mächtigkeit, 523

relatives Komplement, 523
Mikroarchitektur 291
mit Ausnahmebehandlung, 293, 311–312
mit dynamischer Instruktionsablaufplanung, 293, 313–322
mit Fließbandverarbeitung, 293–308
mit Multizyklen-Funktionseinheit, 293, 308–311
mit Sprungvorhersage, 293, 312–313
superskalare, 293
Miter 246
MoC 16
Modallogik 24
model of computation *siehe* MoC
Modell
heterogenes, 56
Modellprüfer
CBMC, 449
CMC, 449
F-Soft, 450
Java Pathfinder, 449
SATABS, 449
SLAM, 449
SPIN, 449
VeriSoft, 449
ZING, 449
Modellprüfung 26, 106, 156, 178–207
Abstraktionsverfeinerung
von Programmen 425
BDD-basierte, 156
CTL, 179–185
existentielle, 200
explizite, 106, 156, 178–188
implizite, 106, 185
LTL, 185–188
SAT-basierte, 156
von Hardware 331–345
von Programmen 422–425
SE-LTL, 473–474
simulative, 27
symbolische, 106, 156, 197–207
BDD-basierte 197–199
SAT-basierte 199–207
TCTL, 211–222
TLM, 476–484
universelle, 200
von Programmen, 422–431
Modulebene 14, 15
Moment
konstantes, 276
lineares, 276
Momentengraph
binärer, *siehe* BMD
Monitor 103, 107, 189, 323
Monitorschaltung 323
MPSoC 2
MTBDD 275
Multi-Processor System-on-Chip 2
Multigraph 530

Nachbedingung
stärkste, 431
Nebenläufigkeit 43
Netzwerk 531
Boolesches, 525
Interpretation 526
Netzwerkkalkül 508
NFA 47
Definition, 47
Normalform, konjunktive 247, 543
Null-Äquivalenzproblem 129

OBDD 535
reduziertes, *siehe* ROBDD
OBMD 277
reduzierter, *siehe* ROBMD
OFDD 538
reduziertes, *siehe* ROFDD
Off-Testfall 137
OIDD 460
reduziert, *siehe* ROIDD
OKFDD 539
reduziertes, *siehe* ROKFDD
On-Testfall 137
open fault 6
Operandenbereich 382
Operator
modellogischer, *siehe* Pfadquantor
temporaler, 73, 76
Finally 76
Globally 76
Next 76
Release 76
Until 76

p-use 410, 412
Pareto-Optimum 9, 18
Partialordnung 524

Partialordnungsreduktion 158, 172–178
Partition
  einer Menge, 523
Partitionsblock 523
path segment simulation 436
Periode 349
Petri-Netz 41, 91
  Anfangsmarkierung, 41
  Beschränktheit, 44, 164
  Definition, 41
  Dynamisierungsvorschrift, 41
  Erreichbarkeitsmenge, 44
  Flussrelation, 41
  Folgemarkierung, 43
  Grundzustand, 157, 165, 167
  Inzidenzmatrix eines, 168
  Kanten, 41
  Kantengewichte, 41
  Konflikt, 43
  Konservativität, 45, 170
  Lebendigkeit, 44
  Marke, 42
  Markierung, 42
  Markierungsfunktion, 42
  Nachbereich eines Knotens, 42
  Netzgraph, 41
  Reversibilität, 45, 156, 165, 167, 171
  Schaltregel, 43
  schwache Lebendigkeit, 44
  Sicherheit, 44, 156
  starke Lebendigkeit, 44, 167
  Stelle, 41
    Kapazität einer 41
  totes, 44
  Transition, 41
    nicht sichtbare 174
    Schalten einer 43
    sichtbare 174
    unabhängige 172
  Verifikation
    aufzählende 157–167
    strukturelle 157, 167–172
  Verklemmungsfreiheit, 44, 164
  Vorbereich eines Knotens, 41
  zeitbehaftetes, 45–47
    Definition 45
    Erreichbarkeitsgraph 208
    Feuerbereich 208
    Latenz 213–214
    Modellprüfung 211–214
    Prüfung des Zeitverhaltens 207–214
    Zustandsklassen 208
  Zustandsgleichung, 168
PEUF 357
Pfad
  definitionsfreier, 412
  kritischer, 346
  längster, 346
  Präfix, 74
  Suffix, 74
Pfadberechnung 135
Pfadberechnungsanpassung 138
Pfadbereich 135
Pfadbereichsanpassung 138
Pfadbereichstest 134–139
Pfadergänzung 138
Pfadgewicht 531
Pfadquantor 78
  existentieller, 78
  universeller, 78
Plattformmodell 17
point to analysis 449
Polynom 117
  sparse recursive representation, 118
  verschwindendes, 291
Polynomfunktion 119
polynomieller Algorithmus 542
Potenzmenge 523
Prädikatenabstraktion 426
Prädikatenintervall 462
Prädikatenlogik 528
  Alphabet, 528
  Definitionen, 528–529
  erster Ordnung, 23, 528
  Formel, 528
    atomare 528
  Terme, 528
Prädikatensymbol 298
  Ordnung eines, 298
  uninterpretiertes, 298
Präfix
  Länge, 90
  Latenz, 90
primärer Ausgang 526
primärer Eingang 526
Problemgraph 348
  iterativer, 349, 442
Produktautomat 143, 187

Programm
  Boolesches, 427
Programmabhängigkeitsgraph 375
  Definition, 375
Programmanalyse
  statische, 416–422
Programmiersprache 59
  Ausführungsmodell, 59
    control-driven 59
    demand-driven 59
  C, 59
  deklarative, 59
  imperative, 59
Programmverifikation 26, 416–431
Prototyp 100
Prozesskalkül 55
Prozessor
  superskalarer, 313
Pseudo-Boolesche Funktion
  Belegung, 274
PSL 72, 83–88, 188
  Überdeckung, 192
  Boolesche Ebene, 83–84
  einfache Teilmenge, 85
  Monitor, 192–197
  temporale Ebene, 83–87
  Verifikationsebene, 83, 87–88
PSL-Eigenschaft 85
Ptolemy 91

QBF-Solver 430
Qualität 8
Qualitätsanforderung 8
Qualitätsmaß 9, 17
Qualitätsmerkmal 8, 17, 18
  funktionales, 9
  nichtfunktionales, 9
Quellknoten eines polaren Graphen 531

Rücksprungverfahren 545
  konfliktgetriebenes, 549
Rückwärtstraversierung 147
rate-monotonic scheduling *siehe* RMS-Algorithmus
real time calculus *siehe* RTC
Record&Play-Algorithmus 269
Referenzergebnis 107, 132
Referenzmodell 103
Referenzstimulus 132
Referenztestfall 132
Regionen-Äquivalenz 216
Regionenautomat 217
register transfer level *siehe* RTL
Registertransferebene *siehe* RTL
Regressionstest 103, 131–132
regulärer Ausdruck
  sequentiell erweiterter, *siehe* SERE
Rekonvergenzmodell 11
Relation 524
  antisymmetrische, 524
  reflexive, 524
  symmetrische, 524
  transitive, 524
requirement *siehe* Anforderung
Resolution 548
Ressourcenauslastung 439
RMS-Algorithmus 444
ROBDD 150, 237, 535
  Isomorphie, 535
  ITE-Operator, 537
ROBMD 277
Robustheit 5
ROFDD 538
ROIDD 460
ROKFDD 237, 243, 539
round robin *siehe* Round-Robin-Algorithmus
Round-Robin-Algorithmus 446
RSM 453–454
  Distanzvektor, 453
  dynamische Transitionen, 454
  dynamische Zustände, 453
  Indexmenge, 453
  Pfad, 454
  Prädikat, 453
  Semantik, 454
  Zustandsdiagramm
    dynamisches 453
    statisches 453
RTC 508–514
RTL 15

safety *siehe* Gefahrlosigkeit
SAT modulo theories *siehe* SMT
SAT-Problem 542
SAT-Solver 26, 113, 246, 356, 542–551
Schaltbereitschaft 43
  Definition, 43, 46

Schalten 43
Schaltnetz 236
Schaltung
  asynchrone, 41
  kombinatorische, 236, 242
Schaltungsmodell
  iteratives, 259, 331
Schaltungsvereinfachung
  dynamische, 358
Schaltvektor 168
Schaltwerk 236, 242
Schleifeninvariante 26
Schleifenkonstrukt 56
Schleifentransformation
  globale, 380
Schnitt 152
Schnittmenge 523
Schnittpunkt 152, 369
SDF-Graph 53
  Definition als Petri-Netz, 53
  Iteration, 226
  Konsumptionsrate, 53
  Produktionsrate, 53
  Repetitionsvektor, 226
  zeitbehafteter, *siehe* TSDF-Graph
SDF-Modell 91
SE-LTL 472
  Modellprüfung, 473–474
  Semantik, 472
  Syntax, 472
Semaphore 30
Senkeknoten eines polaren Graphen 531
sequential extended regular expression *siehe* SERE
Sequentialität 43
Sequenzautomat 358
SERE 84, 192
  Abzählungswiederholung, 85
  Bereichswiederholung, 85
  Definition, 84
  Fusion, 84
  GOTO-Operator, 85
  Konkatenation, 84
  Längendurchschnitt, 84
Servicefunktion 516
Servicekennlinie 518
  obere, 510
  untere, 510
Shannon-Zerlegung 243, 276, 534
shape analysis 449
Sicherheit 44
sifting 536
Signalverarbeitung 53
Simulation 27, 101–105
  symbolische, 105, 237, 259, 357, 362
  zufällige
    gesteuerte 102, 111–114
Simulator 104–105
  ereignisgesteuerter, 104
  hybrider, 104
  zyklenbasierter, 104
Sleep-Set-Methode 177–178
SMT 26, 365, 552
SMT-Solver 551–556
  indirekte, 552
Software-Entwurfsprozess 14
Spannbaum 530
Spezifikation 15, 16, 37
  ausführbare, 38, 39, 59
  Definition, 37
  eindeutige, 38
  formale, 37
  informale, 37
  korrekte, 38
  vollständige, 38
Spezifikationsfehler 5
Sprache
  einer temporalen Struktur, 467
  eines Büchi-Automaten, 186
Sprachinklusion 106
state/event-LTL *siehe* SE-LTL
Stellen-Transitions-Netz 41
Stelleninvariante 171
Steuerbarkeit 109, 110
Stimulation 101
Strukturmodell 17
Stubborn-Set-Methode 175–177
stuck-at fault *siehe* Haftfehler
stuck-at fault model *siehe* Haftfehlermodell
Subsumtion 548
SUV 102
synchroner Datenflussgraph *siehe* SDF-Graph
Synchronisation 43
synchronous dataflow graph *siehe* SDF-Graph
Synthese 4, 15–18

System
  latenzinsensitives, 345
system under verification *siehe* SUV
Systemabhängigkeitsgraph 376
SystemC 28, 60–67, 451
  als LTS, 468–472
  Ereignis, 66
  Kanal, 62, 63
  Methode, 62
  Modellprüfung, 466–476
  Modul, 60
  Port, 60, 61
  Prozess, 62
  Signal, 62
  Thread, 62
SystemCoDesigner 68
Systemebene 15
SysteMoC 60, 68–72
  Aktion, 68
  Anfangsmarke, 69
  Konsumptionsrate, 68
  Modellprüfung, 452–466
  Produktionsrate, 68
  Repräsentation des Zustandsraums, 452–464
  Wächterfunktion, 68
  Zustand, 453
Systemsynthese 15

Tableau-Technik 192, 233
Taktdomäne 334
Taktzustand
  globaler, 336
tardiness 440
Tautologie 528
Taylor-Expansions-Diagramm *siehe* TED
Taylor-Reihen-Entwicklung 118
Taylorsches Theorem 124
TCTL 89
  Semantik, 90
  Syntax, 89
  Zeitschranke, 89
TDMA-Ablaufplanung 447
TED 120–129, 283
  Addition, 284–285
  Definition, 120
  Eindeutigkeit, 124
  Eliminationsregel, 121
  geordnetes, 122
  Interpretation, 120
  Kanonizität, 124
  Kompositionsoperatoren, 284–286
  Minimalität, 125
  Multiplikation, 285–286
  normalisiert, 123
  Normalisierung, 122
  Reduktion, 120
  reduziertes, 122
  Verschmelzungsregel, 122
Teilmenge 523
  echte, 523
  nichtnegative, 523
  positive, 523
Teilmengenkonstruktion 191, 196
Temporale Struktur 73–75, 163
  als Büchi-Automat, 186
  attributierte, *siehe* LTS
  Definition, 73
  Pfad, 74, 467
  Sprache der, 467
  zeitbehaftete, *siehe* TTS
Temporallogik 25
Test 5
Testbench 102
Testfall 101
  datenflussorientierter, 391, 400, 410–415
  funktionsorientierter, 391–400
  gerichteter, 102
  kontrollflussorientierter, 391, 400–410
  strukturorientierter, 391, 400–415
  zufälliger, 102
  zustandsorientierter, 396–400
Testfallgenerierung 101–102
  gerichtete, 102
  zufällige, 102
  zur Hardware-Verifikation, 246
  zur Software-Verifikation, 391–415
Testmustergenerierung *siehe* ATPG
Testvektor 253
Testvorschrift 112
Theorembeweis
  automatischer, 25
Theorembeweiser 106–107
Theorie
  Hintergrund-, 26, 365, 551
time division multiple access 447
timed computation tree logic *siehe* TCTL
TLM 28, 476

Definition, 478
Eigenschaftsprüfung, 484–490
Initiatormodul, 478
Modellprüfung, 476–484
Targetmodul, 478
Transaktion, 476–478
Transaktion 20
blockierende, 476
nichtblockierende, 476
Transaktionsebene 452
Transaktionsebenenmodell *siehe* TLM
Transferfunktion 515
Transferzeit 346
Transition
Aktivierbarkeit, 44, 165
Lebendigkeit, 44
neuaktivierte, 46
Schaltbereitschaft, 43
tote, 44
TSDF-Graph 222–232
Aktor, 222
Verzögerungszeit 223
Aktordurchsatz, 226
Anfangsmarkierung, 223
Ausführung, 225
selbstplanende 225
Durchsatz, 226–228
Kanal, 222
Konsumptionsrate, 223
Prüfung des Zeitverhaltens, 222–232
Produktionsrate, 223
Zustand, 223
TTS 89, 213
minimale Latenz, 213
mit exakten Verzögerungen, 89
Turing-Äquivalenz 55
Turing-Maschine 24

Überapproximation 98, 344
Überdeckung
Übergangs-, 397
Anweisungs-, 401
Bedingungs-/Entscheidungs-, 405
Ereignis-, 397
funktionsorientierte, 103
Mehrfach-Bedingungs-, 406
minimale 406
Pfad-, 407
strukturierte 408
strukturorientierte, 103
Zustands-, 397
Zweig-, 402
Überdeckungsmaß 103, 391
Überdeckungstest 401
all c-uses-, 414
all c-uses/some p-uses-, 414–415
all p-uses/some c-uses-, 415
all defs-, 413
all p-uses-, 414
all p-uses/some c-uses-, 415
all uses-, 415
Anweisungs-, 401–402
Bedingungs-, 404–406
einfacher 404–405
Bedingungs-/Entscheidungs-, 405
modifizierter 448
Datenkontext-, 448
defs/uses-, 413–415
Mehrfach-Bedingungs-, 406
einfache 406
Pfad-, 407–410
strukturierter 408–410
required k-tuples, 448
Zweig-, 402–404
Übereinstimmungsproblem 294
Übergangsfunktion 48, 220, 295
erweiterte, 142
Übergangsrelation 47, 49
Überprüfungsstrategien 103
Überdeckung
Zusicherungs-, 391
UIP 550
UML 91
Unabhängigkeitsintervall *siehe* II
Unabhängigkeitsintervallpartition *siehe* IIP
unit propagtion 544
Unterapproximation 342
Unvollständigkeitssatz 24
Ursache-Wirkungs-Analyse 392–396
Ursache-Wirkungs-Graph 393

V-Modell 13
Validierung 5, 38, 109
Variablendefinition
globale, 412
lokale, 412
Variablenordnung 119

Variablenzugriff
  globaler, 412
  lokaler, 411
Vereinigungsmenge 523
Verfeinerung 17, 18
Verhalten 37
Verhaltensmodell 16
  ausführbares, 59
  formales, 41
Verhaltenssynthese 15
Verifikation 5
Verifikationsaufgabe 21, 95–97
Verifikationsbereich 374
Verifikationsmethode 22, 99–107
  diversifizierende, 130
  formale, 99, 105–107
  hybride, 101, 107
  simulative, 100–105
  unvollständige, 99, 100
  vollständige, 99
Verifikationsmethodik 95
Verifikationsprozess 11–22
Verifikationsvollständigkeit 103–104, 400
Verifikationsziel 95, 97–99
Verklemmung 44
Verschmelzungspunkt 271
Verschmelzungsregel 535, 538
Versetzungsfunktion 462
  inverse, 466
Verzweigung 53
  als Kontrollstruktur, 56
Verzweigungsliteral 544
Verzweigungsstrategie
  DLCS, 545
  DLIS, 545
  MOM, 546
  VSIDS, 546
Vielzweckrechner 1
Volladdierer 29
Vollständigkeitskriterium 400
Vorbedingung
  schwächste, 431
Vorwärtstraversierung 145
VPC 492

Wald 530
Wartezeit 439
WCET 432
WCET-Analyse 432–438
  Architekturmodellierung, 436–438
  Programmpfadanalyse, 433–436
Wertebereich
  abstrakter, 419
    lineare Kongruenzen 422
    nichtrelationaler 419
    relationaler 419, 421
  Intervall-, 419
  Kongruenz-, 419, 421
  konkreter, 419
  Paritäts-, 421
  Vorzeichen-, 419
window permutation 536
worst case execution time *siehe* WCET

X-Diagramm 35
  der Synthese, 16
  für die Verifikation, 20–22

Y-Diagramm 35, 490

Zeitanalyse 345
  auf Systemebene, 490–520
    kompositionale 499–508
    modulare 508–520
  dynamische, 348
  für Hardware, 345
    auf Architekturebene 348–351
    auf Logikebene 345–348
  Hardware, 356
  kompositionale, 499–508
  Schaltung
    synchroner 345–351
  Software, 431–448
  statische, 345
  symbolische, *siehe* SymTA/S
  System
    latenzinsensitives 351–356
  Systemebene
    simulative 492–499
  WCET, 432–436
  Zeitschranke, 491
    obere 491
    untere 491
Zeitbewertungsmodell
  Zeitbewertungsnetzwerk, 512
Zeitschranke 45
Zeitschrittsimulation 259
Zeitstruktur 220

Zeitzone 219
Zerlegung
  momentenbasierte, 275
Zonenautomat 219
Zurückverfolgung 249, 545
  chronologische, 548
  nichtchronologische, 545, 549
Zusicherung 27, 88, 111, 391
Zusicherungssprache 92, 188
Zustand
  ablehnender, 190
  abstrakter, 427
  akzeptierender, 186, 190
Zustandsäquivalenz 142, 143
Zustandsübergang 48
Zustandsautomat
  regulärer, *siehe* RSM
Zustandsdiagramm 48
Zustandsexplosion 32
Zustandsraumexplosion 105, 172
Zustandsraumreduktion
  durch Schleifenabwicklung, 424–425
Zustandsraumtraversierung 144–151
  rückwärtsgerichtete, 147
  symbolische, 148–151
  vorwärtsgerichtete, 145
Zuverlässigkeit 6
Zuweisungsfunktion 462
  inverse, 466
Zweig
  primitiver, 403