Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen

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

genehmigte

Dissertation

von

Armin Biere
aus Villingen-Schwenningen

Tag der mündlichen Prüfung:	30. Januar 1997
Erstgutachter:	Prof. Dr. rer. nat. P. Deussen
Zweitgutachter:	Prof. Dr. phil. nat. Dr. rer. nat. h. c. G. Krüger

Danksagung

Das Gelingen dieser Arbeit war nur möglich durch die nicht nachlassende und tatkräftige Unterstützung meiner Betreuer, meiner Kollegen und meiner Familie. Ihnen möchte ich an dieser Stelle aufs herzlichste danken. Vor allem für meine Frau hatte ich während der Entstehung sehr wenig Zeit. Ihr und meinen Großeltern, die mich seit meiner Kindheit auf diesem Weg begleitet haben, ist diese Arbeit gewidmet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Modellprüfung ist eine Technik zur Verifikation von in der Informatik auftretenden Systemen. Die zu verifizierenen Eigenschaften werden dabei in einer formalen Sprache (Logik) spezifiziert. In dieser Arbeit wird dafür der μ-Kalkül verwendet, in den sich viele andere Logiken übersetzen lassen. Für diese weniger ausdrucksstarken Logiken gibt es sehr leistungsstarke Modellprüfer.

These dieser Arbeit ist es, daß es möglich ist, einen Modellprüfer für den μ-Kalkül zu konstruieren, der genauso effizient ist, wie Modellprüfer für eingeschränktere Logiken.

1.1 Motivation

Der Untersuchungsbericht /ESA-CNES, 1996/ beginnt mit den Worten

> Am 4. Juni 1996 endete der Jungfernflug der Ariane 5 in einer Katastrophe. Nach etwa 40 Sekunden in einer Höhe von 3700 Metern verließ die Rakete ihre Flugbahn, brach auseinander und explodierte. . . .

Wie aus der Presse zu entnehmen war, hat dieser Fehlschlag die ESA (European Space Agency) viele Millionen ECU gekostet. Deshalb hat sie einer unabhängigen Kommision von Wissenschaftlern und Ingenieuren den Auftrag erteilt, die Ursache für den Absturz zu ermitteln. Aus deren Abschlußbericht stammt obiges Zitat. Die Kommision kam zu dem Schluß, daß der Absturz durch einen Softwarefehler verursacht wurde, der trotz ausgiebiger Tests und anderer Maßnahmen zur Qualitätssicherung nicht gefunden wurde. Die tieferliegenden Gründe für den Fehlschlag dieser Maßnahmen erklärt die Kommision durch (aus dem Englischen)

> . . . (die Enwickler) waren der Ansicht, daß Software solange als korrekt gilt, bis ein Fehler gefunden wird. . . . Die Kommision ist der gegenteiligen Ansicht, daß Software solange als fehlerhaft betrachtet werden sollte, bis die derzeitig besten akzeptierten praktischen Methoden zeigen, daß sie korrekt ist.

Neben der Verwendung der Verifikation zur bestmöglichen Fehlervermeidung, sollte Verifikation zur Reduzierung von Testkosten eingesetzt werden. Der Anteil der Kosten für Tests läßt sich je nach Projekt zwischen 40% und 70% einordnen (siehe /Kit, 1995/ und /Fairley, 1985/), so daß hier ein immenses Einsparungspotential gegeben ist. Erfahrungen im industriellen Umfeld, insbesondere mit der Modellprüfung (z. B. in /Beer et al., 1994/, /Eiríksson und McMillan, 1995/), bestätigen diese Ansicht.

1.2 Hintergrund

1.2.1 Modellprüfung

Die *Modellprüfung* (engl. model checking) ist eine Technik zur Verifikation von in der Informatik auftretenden Systemen. Anwendungsgebiete sind zum Beispiel die Verifikation von Schaltwerken oder Protokollen. Die Systeme werden durch Zustände und Zustandsübergangsrelation modelliert und die zu verifizierende Spezifikation in einer formalen Sprache (z. B. temporaler Aussagenlogik) formuliert.

In der eigentlichen Verifikation ordnet man jeder Formel die Menge der Zustände des fest gewählten Systems zu, die die Formel erfüllen. Diese Zustandsmengen werden nun rekursiv über den syntaktischen Aufbau der Formeln *explizit* ausgerechnet. Schließlich läßt sich dann die Frage, ob ein bestimmtes System eine Spezifikation erfüllt, dadurch beantworten, daß man überprüft, ob der Startzustand des Systems in der Zustandsmenge liegt, die für die Spezifikation berechnet wurde.

Diese Methode wurde anfangs der 80er Jahre von E. Clarke, E. A. Emerson und A. P. Sistla an der CMU entwickelt /Clarke et al., 1986/. Sie litt darunter, daß durch die explizite Repräsentation der Zustände, die eine Formel erfüllen, ihr Einsatz auf endliche Systeme mit einer kleinen Zustandszahl beschränkt blieb. Erst durch die Verwendung von ROBDDs (reduzierte geordnete binäre Entscheidungsdiagramme) zur Repräsentation der Zustandsmengen gelang es, größere Systeme mit mehr als 10^{20} Zuständen zu behandeln. So gibt es mehrere Fallstudien, die belegen, *daß die Modellprüfung in industriellen Projekten zur Kostenreduktion in der Testphase und zur Qualitätssicherung einen großen Beitrag leisten kann* (s. letzter Abschnitt der Motivation).

1.2.2 μ-Kalkül

Die Kombination der Modellprüfung mit den in /Bryant, 1986/ vorgestellten ROBDDs zur kompakten Repräsentation von Schaltnetzen (booleschen Funktionen) läßt sich zurückführen auf Arbeiten um 1990, wie /Burch et al., 1990/ und /Coudert et al., 1989/. Schon zu diesem Zeitpunkt wurde der μ-Kalkül als Eingabesprache für einen Modellprüfer vorgeschlagen. Dieser kann als universelle Eingabesprache sowohl zur Beschreibung des Modells als auch der Spezifikation verwendet werden und erlaubt darüber hinaus eine ROBDD-basierte Modellprüfung. Der Vorteil der Verwendung des μ-Kalküls ist die *weitaus höhere Ausdrucksmächtigkeit* gegenüber eingeschränkteren Logiken wie FairCTL beim SMV-System und die damit verbundene *Flexibilität* bei Verwendung mehrerer Spezifikationssprachen.

Dieses Konzept wurde nur unzureichend in die Praxis umgesetzt. Die aufsehenerregenden Erfolge der Modellprüfung wurden fast alle mit dem von K. L. McMillan entwickelten SMV-System /McMillan, 1993b/ erzielt, welches als Spezifikationssprache nur FairCTL erlaubt.

1.3 Resultate

1.3.1 Optimierungen

In dieser Arbeit wird gezeigt, daß sich Modellprüfung des μ-Kalkül trotz der weitaus höheren Ausdrucksmächtigkeit genauso effizient in der Praxis einsetzen läßt wie Modellprüfung für spezielle Spezifikationssprachen. Hierzu werden verschiedene Optimierungen der Modellprüfung vorgestellt.

Automatische Variablenallokation

Das Hauptergebnis dabei ist eine heuristische Methode zur Allokation (Variablenordnung) von ROBDD-Variablen für Variablen des μ-Kalküls, die unter Verwendung spezieller ROBDD-Algorithmen (schnelle Substitutionen) Aussagen über die Größe der beteiligten ROBDDs gestattet. Für den vollen μ-Kalkül ist dies die erste automatische Methode dieser Art und verwendet einen Ansatz, der auf Randbedingungen (engl. „constraints") beruht. Die „Randbedingungen" haben Vektoren als Grundbereich und beschreiben, wie mehrere solche Vektoren geordnet und ineinander verschränkt sind.

Übertragung von Optimierungen

Weiter wird gezeigt, daß sich Methoden zur Optimierung von CTL-Modellprüfung auf den μ-Kalkül übertragen lassen. Besonders wichtig in diesem Zusammenhang sind solche Techniken, die es erlauben, die Übergangsrelation nicht explizit ausrechnen zu müssen. Dadurch können weitaus größere Systeme der Verifikation durch die Modellprüfung zugänglich gemacht werden.

Damit wurde gezeigt, daß ein μ-Kalkül-Modellprüfer für Eigenschaften, die sich auch in CTL (FairCTL) beschreiben lassen, genauso effizient sein kann, wie spezielle CTL (FairCTL) Modellprüfer. Zweitens erhält man so Optimierungen der Modellprüfung für andere Spezifikationssprachen.

1.3.2 Semantik

Um diese Optimierungen formal begründen zu können, wurde ein spezieller μ-Kalkül definiert, der eine Version des μ-Kalküls von Park darstellt. Dafür wird eine Standardsemantik definiert, die schrittweise in eine Semantik überführt wird, die BDDs als Datenstrukur benützt. Dies ist die erste formale Herleitung der Modellprüfung im μ-Kalkül mit BDDs.

1.3.3 BDDs

BDDs als Termalgebra

Die BDDs selbst werden dabei als freie Termalgebra modulo eines Termersetzungssystem interpretiert. Dies liefert einen algebraischen Zugang zu BDDs und schlägt eine Brücke zu Methoden der Implementierung von kanonischen Termersetzungssystemen.

Abstrakte Maschine zur Implementierung von BDDs

Dieses Kapitel stellt die wichtigsten Techniken für die Modellprüfung im Zusammenhang mit BDDs vor. In der Literatur ist keine einheitliche Darstellung von BDDs zu finden. Deshalb wurde eine abstrakte Maschine für die Abarbeitung von Algorithmen auf BDDs entwickelt und alle wesentlichen BDD-Algorithmen formuliert. Auf dieser Ebene konnten die meisten Algorithmen leicht zugänglich beschrieben und formal verifiziert werden. Darunter findet sich auch ein großer Anteil von Algorithmen, der bislang nur in Implementierungen als Programmtext dokumentiert war.

Das Kapitel schließt mit dem vom Autor entwickelten ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus, der die meisten Spezialalgorithmen für die Modellprüfung verallgemeinert.

1.3.4 Der Modellprüfer μcke

Evaluation

Diese Ergebnisse stehen im unmittelbaren Zusammenhang mit dem vom Autor entworfenen und implementierten Modellprüfer μcke, der zur Evaluation der theoretischen Ergebnisse benutzt wird. Ein detaillierter Vergleich der Leistungsfähigkeit der μcke mit der anderer Modellprüfer wird durchgeführt. Dieser zeigt, daß die μcke bei einer weitaus mächtigeren und flexibleren Eingabesprache mindestens genauso effizient ist wie Spezialmodellprüfer (SMV). Gegenüber anderen Modellprüfern für den μ-Kalkül erweist sie sich als weitaus leistungsfähiger und bietet zusätzlich den Vorteil einer automatischen Variablenallokation.

Architektur

Des weiteren zeichnet sich die Architektur der μcke gegenüber anderen Modellprüfern durch Wiederverwendbarkeit und leichte Erweiterbarkeit aus.

Leitfaden

Im zweiten Kapitel wird der vom Autor entwickelte μ-Kalkül eingeführt. Das dritte Kapitel ist der Behandlung von BDDs gewidmet. Das anschließende Kapitel beschäftigt sich mit Allokationen. Im letzten Kapitel des Hauptteils der Arbeit wird näher auf den Modellprüfer μcke eingegangen. Einige weitere Grundlagen, Definitionen und Schreibweisen sind im Anhang zusammengefaßt. Dort findet man auch noch ergänzende Fakten zum μ-Kalkül und einen Abschnitt mit Quelltexten. Dieser wird gefolgt vom Verzeichnis für die Abbildungen und Algorithmen. Danach folgt das Literaturverzeichnis und schließlich ein Index, in dem alle wichtigen Stichworte und Symbole aufgeführt sind.

Kapitel 2

μ-Kalkül

Der μ-Kalkül ist eine Logik, in der Prädikate *rekursiv* definiert werden können. Für die Verifikation, dem Hauptanwendungsgebiet der Modellprüfung, werden die zu verifizierenden Eigenschaften und Systeme als Formeln des μ-Kalküls beschrieben. Die Verifikation an sich bedeutet nun die Berechnung der Semantik einer μ-Kalkülformel. Im Gegensatz zur reinen Aussagenlogik müssen hier zusätzlich die rekursiv definierten Prädikate berechnet werden.

2.1 Übersicht

Dieses Kapitel dient der Darstellung der Syntax und Semantik des in dieser Arbeit verwendeten μ-Kalküls $\mathbb{B}_\mu^V$. Neben einer Standardsemantik, die auch in der Literatur Verwendung findet, wird eine zweite Semantik (Semantik II) vorgestellt, die eine μ-Kalkülformel nicht nur zu wahr oder falsch ($\mathbb{B}$) evaluiert, sondern einen booleschen Ausdruck aus $\mathbb{B}^W$ dafür berechnet.

In Abb. 2.2 ist der Zusammenhang zwischen den hier betrachteten Logiken und Semantiken erläutert. Ziel dieses Kapitels ist es, einen formalen Rahmen für die mittleren Bestandteile dieser Abbildung aufzustellen, was dann in Abschnitt 3.7 vertieft wird. Damit erhält man eine formale Herleitung der symbolischen Modellprüfung des μ-Kalküls.

2.2 Vergleich mit PROLOG

2.2.1 Formulierung eines Beispiels in PROLOG

Der μ-Kalkül ist eine Logik, in der Prädikate *rekursiv* definiert werden können. Dies geschieht in ähnlicher Weise, wie das z. B. bei PROLOG-Programmen der Fall ist. Gegeben seien folgende Fakten eines PROLOG-Programms:

```
vater(ferdinand,cosimo).
vater(cosimo,leopold).
vater(cosimo,margarete).
```

die einen Auschnitt aus der Stammtafel der Medici darstellen (vgl. Abb. 2.1). Das zweistellige Prädikat `vorfahre`, das die Beziehung zwischen Vorfahren und Nachfahren ausdrückt, läßt sich rekusiv durch die folgenden zwei Klauseln definieren

```
vorfahre(V,N) :- vater(V,N).
vorfahre(V,N) :- vater(V,X), vorfahre(X,N).
```

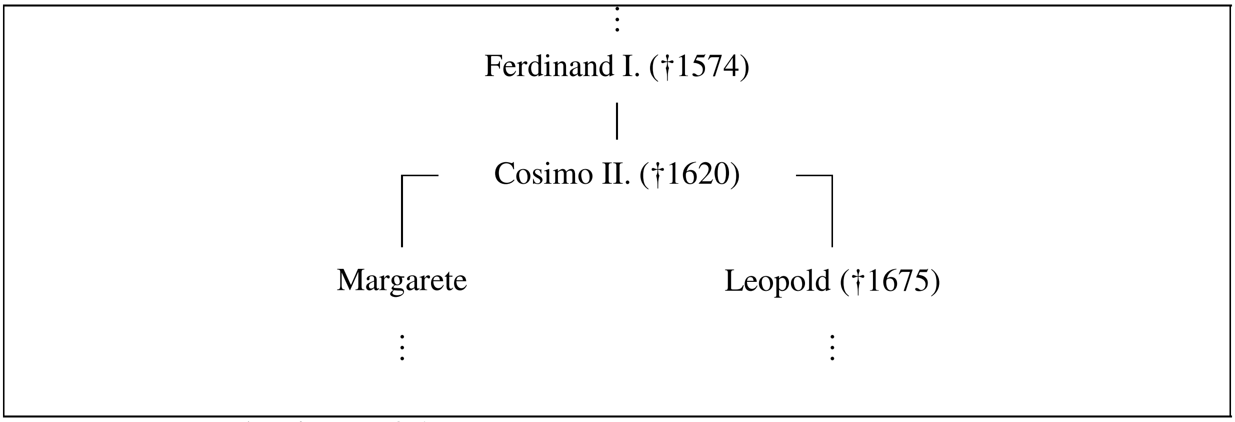


Abbildung 2.1: Ausschnitt aus der Stammtafel der Medici.

Die erste Klausel drückt den Basisfall aus: jede Vater-Sohn Beziehung ist eine Vorfahre-Nachfahre Beziehung. Die zweite Klausel beschreibt den rekursiven Fall: Ist schon bekannt, daß `X` ein Vorfahre von `N` ist, und es ist `V` der Vater von `X`, dann muß auch `V` ein Vorfahre von `N` sein. Die Semantik von PROLOG besagt, daß die Anfrage

```
?- vorfahre(ferdinand, margarete).
```

wahr ist, da nach einmaliger Anwendung der zweiten Klausel (Resolution mit der zweiten Klausel) unter Substitution von `X` durch `cosimo` die erste Klausel zu einem Abbruch der Rekursion führt.

2.2.2 Semantik der PROLOG-Version

Die Menge der Paare von Familienmitgliedern der Medici, für die das Prädikat `vorfahre` wahr ist, kann aber auch auf andere Weise berechnet werden, nämlich rückwärts- statt vorwärtsgerichtet („bottom-up" statt „top-down"). So ist auch die Standardversion der formalen Semantik von PROLOG in /Schmitt, 1992b/ oder /Lloyd, 1987/ definiert. Hierzu sei

$$M := \{\texttt{cosimo}, \texttt{leopold}, \texttt{ferdinand}\}$$

die Menge der betrachteten Medici (das Herbrand-Universum). Obige Klauseln definieren dann für `vorfahre` einen Prädikattransformer Ψ, mit

$$\Psi : \mathbb{P}(M \times M) \to \mathbb{P}(M \times M)$$

der eine „Belegung" ($\subseteq M \times M$) für `vorfahre` in eine neue Belegung überführt, indem jede Klausel von `vorfahre` genau einmal angewendet und dabei für `vorfahre` die gegebene Belegung eingesetzt wird. Will man z. B. Ψ auf

$$N := \{(\texttt{cosimo}, \texttt{leopold})\}$$

anwenden, so sammelt die Anwendung der ersten Klausel die Fakten auf

$$N_1 := \{(\texttt{cosimo}, \texttt{leopold}), (\texttt{cosimo}, \texttt{margarete}), (\texttt{ferdinand}, \texttt{cosimo})\}$$

Die zweite Klausel läßt sich nur für $\texttt{V} = \texttt{ferdinand}$, $\texttt{N} = \texttt{leopold}$ und $\texttt{X} = \texttt{cosimo}$ anwenden, da erfüllende Belegungen für `vorfahre` aus N stammen sollen, und man erhält

$$N_2 := \{(\texttt{ferdinand}, \texttt{leopold})\}$$

Insgesamt gilt für die neue Belegung N'

$$N' = \Psi(N) = N_1 \cup N_2$$

Mit dieser informellen Definition von Ψ läßt sich die Interpretation $I(\texttt{vorfahre})$ des Prädikates `vorfahre` bezüglich dem obigen Programm definieren als

$$I(\texttt{vorfahre}) = \bigcup_{i=0}^{\infty} \Psi^i(\emptyset)$$

In $\Psi^i(\emptyset)$ liegen die Paare von Vorfahren und Nachfahren der Medici, die in der entsprechenden Linie der Stammtafel höchstens i Vater-Sohn (bzw. Vater-Tocher) Beziehungen entfernt sind. Die Vereinigung all dieser Mengen für $i \in \mathbb{N}$ (bei einem endlichem Universum, wie in diesem Beispiel, reicht eine *endliche* Teilmenge von $\mathbb{N}$ aus) ergibt dann genau die Menge von Paaren von Vor- und Nachfahren. Dabei besitzt Ψ die Eigenschaft, eine monotone Abbildung auf dem Teilmengenverband $\mathbb{P}(M \times M)$ zu sein

$$N_a \subseteq N_b \quad \Rightarrow \quad \Psi(N_a) \subseteq \Psi(N_b), \qquad \text{für } N_a, N_b \subseteq M \times M,$$

so daß sich mit den Fixpunktsätzen von Tarski (siehe Abschnitt A.4) $I(\texttt{vorfahre})$ als *kleinster Fixpunkt* von Ψ erweist. Der größte Fixpunkt von Ψ ist ganz $M \times M$ und ist somit sicherlich nicht die Semantik von `vorfahre`, da ja sonst alle Medici Vor- und Nachfahre voneinander wären.

2.2.3 Formulierung im μ-Kalkül

Dieses Beispiel macht deutlich, wie rekursive Definitionen und Fixpunkte mit in PROLOG beschriebenen Prädikaten zusammenhängen. Für den μ-Kalkül ist die Semantik auf ähnliche Weise definiert. Um die Parallele zu ziehen, wird obiges PROLOG-Programm in den μ-Kalkül übersetzt, wobei direkt die Eingabesprache des Modellprüfers μcke verwendet wird, da in der abstrakten Version des μ-Kalküls, wie sie in Abschnitt 2.6 vorgestellt wird, als Konstanten nur Vektoren über $\{0, 1\}$ erlaubt sind. Sonst müßte man noch zusätzlich eine Kodierung angeben. Zunächst wird die Menge der Medici als Aufzählungstyp (s. Abschnitt B.3.1) definiert

```
enum Medici { leopold, cosimo, margarete, ferdinand };
```

Das Vater-Prädikat „`vater`" wird als einfache boolesche Funktion definiert

```
bool vater(Medici V, Medici N)
  V = ferdinand & N = cosimo     |
  V = cosimo    & N = leopold    |
  V = cosimo    & N = margarete  ;
```

Diese Definition von „`vater`" besteht aus der Angabe des Wertebereichs „`bool`", den formalen Parametern „`V`" und „`N`" samt ihrem Typ „`Medici`", gefolgt vom Rumpf, in dem jede Zeile genau einer Klausel von `vater` des PROLOG-Programms entspricht. Dabei steht „`|`" für das boolesche „Oder" und „`&`" für das boolesche „Und". Man beachte, daß die einzelnen Zeilen durch ein logisches „Oder" verbunden sind, was genau der PROLOG-Semantik entspricht, die in diesem Fall besagt, daß verschiedene Klauseln zu ein und demselben Prädikat als Alternativen zu verstehen sind.

Die Definition des Vorfaren-Prädikates „`vorfahre`" wird als kleinster Fixpunkt definiert („`mu`" für das griechische μ)

```
mu bool vorfahre(Medici V, Medici N)
  vater(V,N) |
  (exists Medici X. vater(V,X) & vorfahre(X,N));
```

Die erste Zeile des Rumpfes ist die Übersetzung der ersten Klausel für `vorfahre`. Die zweite Zeile entspricht auch der zweiten Klausel. Hier muß eine neue Variable „`X`“ eingeführt werden. Im PROLOG-Programm ist diese Variable auf die gesamte Klausel gesehen allquantifiziert. Wenn man jedoch den Allquantor in die Prämisse hineinzieht (eine PROLOG-Klausel wird als Implikation von rechts nach links gelesen), so wird dieser zum Existenzquantor („`exists`“). Ein Quantor in der μcke-Eingabesprache hat auch eine Liste von formalen Parametern („`X`“), deren Typen mit angegeben werden müssen („`Medici`“). Die formalen Parameter werden vom Rumpf des Quantors durch einen Punkt „`.`“ getrennt. Die Semantik dieses μ-Kalkül Ausdruckes ist natürlich dieselbe, wie die des PROLOG-Programms und kann wie oben besprochen als Fixpunkt berechnet werden.

Zum Schluß dieses Beispieles soll noch die Übersetzung der PROLOG-Anfrage

```
?- vorfahre(V,N).
```

betrachtet werden. Hier sind die Variablen existentiell quantifiziert, so daß

```
exists Medici V, Medici N. vorfahre(V,N);
```

die Übersetzung ist (In PROLOG wird bei einer erfolgreichen Anfrage eine Antwortsubstitution zurückgeliefert. Bei der μcke muß man dies extra noch angeben durch Voranstellen des Schlüsselwortes „`#witness`“).

Neben den kleinsten Fixpunkten kann man auch größte („`nu`“ für das griechische ν) definieren. Weiterhin können Fixpunkte geschachtelt werden, was mehrfach rekursiv definierten Prädikaten entspricht (ein Prädikat X wird definiert über ein Prädikat Y, das wiederum von X abhängt). Beispiele für solche μ-Kalkül-Ausdrücke werden im Anwendungskapitel 5 gegeben.

$$
\begin{tikzcd}
\mathbb{B}_\mu^V \arrow[rr, "\langle\!\langle \cdot \rangle\!\rangle"] \arrow[dr, "\|\cdot\|"'] \arrow[ddrr, bend right, "[\![\cdot]\!]"'] & & \mathbb{B}^W \arrow[dl, shift right, "\|\cdot\|"'] \arrow[dd, "[\![\cdot]\!]"] \\
& \mathsf{BDD} \arrow[ur, shift right, "\langle\!\langle \cdot \rangle\!\rangle"'] \arrow[dr, "[\![\cdot]\!]"'] & \\
& & \mathbb{B}
\end{tikzcd}
$$

Abbildung 2.2: Zusammenhang zwischen den betrachteten Logiken.

2.3 Motivation

Bis einschließlich Abschnitt 3.7 wird im wesentlichen die Kommutativität des Diagrammes aus Abb. 2.2 gezeigt, wobei in diesem Abschnitt nur die „äußeren" Teile der Abbildung 2.2 betrachtet werden. Das sind einmal die Logiken

- die einfache Boolesche Algebra $\mathbb{B} = \{0, 1\}$,
- boolesche Ausdrücke $\mathbb{B}^W$ über einer Variablenmenge W und
- schließlich der μ-Kalkül $\mathbb{B}_\mu^V$ über einer Variablenmenge V.

Die Variablen W der booleschen Ausdrücke $\mathbb{B}^W$ sind, wie nicht anders zu erwarten, normale aussagenlogische Variable, die Werte aus $\mathbb{B}$ denotieren. Beim μ-Kalkül $\mathbb{B}_\mu^V$ stehen die Variablen aus V für boolesche Vektoren.

In Abb. 2.2 ist weiterhin dargestellt, über welche Funktionen die einzelnen Logiken zusammenhängen. Dabei sind die Bezeichner für die Funktionen mit derselben Logik als Bildbereich überladen. Je nach Herkunft des Argumentes bei der Anwendung einer solchen Funktion läßt sich dann bestimmen, welche Funktion genau gemeint ist.

Die weitere Vorgehensweise ist die folgende. Zunächst wird die boolesche Algebra $\mathbb{B}$ samt zugehöriger Operationen definiert. Dann folgt die Syntax von booleschen Ausdrücken $\mathbb{B}^W$ und deren Semantik, die durch eine Abbildung

$$[\![\cdot]\!]\rho_W : \mathbb{B}^W \to \mathbb{B}$$

unter einer Variablenbelegung $\rho_W : W \to \mathbb{B}$ gegeben ist. Danach wird die Syntax von $\mathbb{B}_\mu^V$ eingeführt. Die Standardsemantik dieser Logik ist wiederum eine Abbildung

$$[\![\cdot]\!]\rho : \mathbb{B}_\mu^V \to \mathbb{B}$$

unter einer Belegung ρ.

Der Hauptgegenstand dieser Arbeit ist die Modellprüfung, welche sich in dem vorgestellten formalen Rahmen als die Berechnung der Semantik von Formeln aus dem μ-Kalkül $\mathbb{B}_\mu^V$ definiert.

$$\boxed{\text{Modellprüfung = Berechnung der Semantik von Formeln des } \mu\text{-Kalküls}}$$

Da, wie sich zeigen wird, die Berechnung der Standardsemantik auf direktem Wege (Abb. 2.2) für praktisch relevante Formeln zu teuer ist, muß ein Umweg über BDD gegangen werden via $\|\cdot\|\colon \mathbb{B}_\mu^V \to \mathsf{BDD}$. Um diese Abbildung formal zu konstruieren, wird in diesem Abschnitt gezeigt, wie sich die Standardsemantik über $\mathbb{B}^W$ berechnen läßt. Durch die in Kapitel 3 gezeigte Korrespondenz zwischen BDD und $\mathbb{B}^W$ erhält man dann formal diese Abbildung.

Eine formale Herleitung ist aus mehreren Gründen wünschenswert. Als erstes liefert sie eine theoretische Rechtfertigung für die Verwendung von BDDs. Der wichtigste Grund aber ist, daß sich die in späteren Kapiteln betrachteten Optimierungsprobleme bei der Modellprüfung ohne eine formale Grundlage nur unzureichend behandeln lassen. Weiter erhält man so einen Modellprüfungsalgorithmus und zu guter Letzt bildet diese Vorgehensweise einen formalen Rahmen, in dem Erweiterungen von $\mathbb{B}_\mu^V$, wie unendliche Domänen, untersucht werden können.

2.4 Die boolesche Algebra $\mathbb{B}$

In diesem Abschnit wird die einfache boolesche Algebra

$$\mathbb{B} := (\{0,1\}, \neg, \wedge)$$

eingeführt. Diese besteht aus den Konstanten 0 und 1, die den Wahrheitswerten „falsch“ und „wahr“ entsprechen (engl. „false“ und „true“). Es gibt formal zwei Operationen ‚$\neg$‘, welche der logischen Negation entspricht, und ‚$\wedge$‘, welche das logische „Und“ darstellt, mit den folgenden Wertetabellen

$\wedge$	0	1
0	0	0
1	0	1

	$\neg$
0	1
1	0

Des weiteren werden später (auch für die anderen Logiken) weitere boolesche Operationen betrachtet, die alle auf diese beiden zurückgeführt werden können. Darunter fallen das logische „Oder“, dargestellt als ‚$\vee$‘, die Implikation ‚$\rightarrow$‘, die logische Äquivalenz ‚$\leftrightarrow$‘ und die dreistellige Operation ‚ite‘. Ihre Definitionen lauten

$$\begin{array}{rcl} a \vee b & := & \neg(\neg a \wedge \neg b) \\ a \rightarrow b & := & \neg a \vee b \\ a \leftrightarrow b & := & (a \rightarrow b) \wedge (b \rightarrow a) \\ \mathrm{ite}(a,b,c) & := & a\,b \,\vee\, \overline{a}\,c, \end{array}$$

wobei $a,b,c \in \mathbb{B}$ beliebige *Terme* aus der booleschen Algebra $\mathbb{B}$ (also nur metasprachliche Platzhalter bzw. Variablen) sind. Als Abkürzung wird in der Definition von „ite“ die Juxtaposition für „$\wedge$“, und „Überstreichen“ für „$\neg$“ verwendet. Dies ergibt kompaktere und leichter zu interpretierende Formeln.

Zur weiteren Schreibweise sei erwähnt, daß hier wie im Rest der Arbeit die üblichen Operatorpräzedenzen verwendet werden. Das heißt ‚$\neg$‘ bindet am stärksten. Dann folgen in dieser Reihenfolge ‚$\wedge$‘ und ‚$\vee$‘. Die Operationen ‚$\rightarrow$‘ und ‚$\leftrightarrow$‘ binden schwächer als ‚$\neg$‘, werden ansonsten aber immer geklammert verwendet.

2.5 Boolesche Ausdrücke $\mathbb{B}^W$

Boolesche Ausdrücke $\mathbb{B}^W$ bilden eine Erweiterung von $\mathbb{B}$. Zusätzlich zu den Konstanten 0 und 1 sind jetzt noch Variablen als „atomare“ Terme erlaubt.

2.5.1 Syntax und Semantik

Die Menge $W = \{x, y, z, \dots\}$ besteht aus den (abzählbar vielen) Variablen in $\mathbb{B}^W$. Sie ist beliebig aber fest gewählt. Die boolesche Algebra $\mathbb{B}^W$ ergibt sich dann zu

$$\mathbb{B}^W := (0, 1, \underbrace{x, y, z, \dots}_{W}, \neg, \wedge)$$

Um die Semantik zu definieren, braucht man den Begriff der Variablenbelegung. Dieser erklärt, wofür die Variablen stehen: Jede Variable $x \in W$ steht für 0 oder 1 also ein Element aus $\mathbb{B}$.

Definition 2.1 (Variablenbelegung bez. $\mathbb{B}^W$) *Eine Abbildung*

$$\rho_W : W \to \mathbb{B}$$

nenne man eine Variablenbelegung ρ_W *für die Variablen W aus* $\mathbb{B}^W$.

Damit läßt sich nach üblichem Schema die Semantik rekursiv definieren als

---

Definition 2.2 (Semantik von $\mathbb{B}^W$)
Die Semantik $[\![\cdot]\!]\rho_W$ *für eine Variablenbelegung* ρ_W *ist eine Abbildung*

$$[\![\cdot]\!]\rho_W : \mathbb{B}^W \to \mathbb{B}$$

und wird rekursiv über die Termstruktur definiert als

$$\begin{aligned} [\![0]\!]\rho_W &:= 0 \\ [\![1]\!]\rho_W &:= 1 \\ [\![x]\!]\rho_W &:= \rho_W(x), \quad x \in W \end{aligned} \qquad \begin{aligned} [\![\neg t]\!]\rho_W &:= \neg [\![t]\!]\rho_W, & t \in \mathbb{B}^W \\ [\![s \wedge t]\!]\rho_W &:= [\![s]\!]\rho_W \wedge [\![t]\!]\rho_W, & s, t \in \mathbb{B}^W \end{aligned}$$

Durch diese Definition (2.2) wird die Semantik ein Homomorphismus bez. den Operationen ‚$\neg$‘ und ‚$\wedge$‘, und die semantische Äquivalenz wird als „Kern“ dieses Homomorphismus definiert (s. Def. 2.3). Man beachte hierbei, daß die Symbole „$\wedge$“ und „$\neg$“ in zweierlei Bedeutung verwendet werden. Dies wird im folgenden noch öfters der Fall sein. Aus dem Zusammenhang sollte aber immer klar werden, zu welcher Algebra das entsprechende Symbol gehört.

Definition 2.3 (Semantische Äquivalenz) *Für* $s, t \in \mathbb{B}^W$ *sei*

$$s \equiv t \quad :\Leftrightarrow \quad ([\![s]\!]\rho_W = [\![t]\!]\rho_W, \quad \textit{für alle Variablenbelegungen } \rho_W)$$

falls $s \equiv t$, *so heiße s (semantisch) äquivalent zu t. Alle zu* 1 *äquivalenten booleschen Ausdrücke s heißen* Tautologien. *Falls s nicht äquivalent zu* 0 *ist, so nenne man s* erfüllbar. *Für ein solches s gibt es eine Variablenbelegung* ρ_W *mit* $[\![s]\!]\rho_W$. *Diese bezeichne man dann als eine* erfüllende Belegung *für s.*

In dieser Definition wurde für die (metasprachliche) Aussage „$[\![s]\!]\rho_W = 1$“ kurz „$[\![s]\!]\rho$“ geschrieben. Und so steht „es gilt nicht $[\![s]\!]\rho$“ für „es gilt $[\![s]\!]\rho_W = 0$“. In den folgenden Beweisen wird dann ebenso

$$[\![s]\!] \;\Leftrightarrow\; \cdots \qquad \text{statt} \qquad [\![s]\!] = 1 \;\Leftrightarrow\; \cdots$$

verwendet. Für den Beweis der Normalformeigenschaft (s. Satz 3.16) von BDDs wird der Begriff der Menge der Variablen benötigt, von denen ein boolescher Ausdruck *semantisch* abhängt.

Auf booleschen Ausdrücken werden Substitutionen definiert, die Variablen durch boolesche Ausdrücke ersetzen. Hier werden die Substitutionen als partielle Funktionen definiert. Schreibweisen und Definitionen zu partiellen Funktionen sind im Anhang Abschnitt A.3 aufgeführt.

Definition 2.4 (Substitutionen in $\mathbb{B}^W$)
Eine Substitution f (von Variablen) ist eine partielle Funktion

$$f\colon W \to \mathbb{B}^W$$

Die totale homomorphe Fortsetzung *h von f sei definiert als*

$$h(0) := 0 \qquad h(1) := 1 \qquad h(x) := \begin{cases} f(x) & f(x)\!\downarrow \\ x & f(x)\!\uparrow \end{cases}$$
$$h(\neg s) := \neg h(s) \qquad h(s \wedge t) := h(s) \wedge h(t)$$

Diese bezeichne man als Abkürzung auch mit f.

So gelte z. B. nach den Konventionen aus Abschnitt A.3

$$(\neg x \vee y)\{x \mapsto y, y \mapsto 0\} \quad = \quad \neg y \vee 0$$

Die Klammern sind notwendig, da $\{\cdots\}$ noch stärker als $\neg$ bindet. Auch wird aus Abschnitt A.3 die Schreibweise für die Verknüpfungen von Belegungen (oder Substitutionen) verwendet. Dieses Beispiel zeigt auch, daß Substitutionen auf Terme aus $\mathbb{B}^W$ immer *parallel* auszuführen sind.

Definition 2.5 (Abhängige Variablen) *Zu $s \in \mathbb{B}^W$ sei*

$$\mathsf{rel}\colon \mathbb{B}^W \to \mathbb{P}(W), \qquad \mathsf{rel}(s) := \{x \in W \mid s\{x \mapsto 0\} \not\equiv s\{x \mapsto 1\}\}$$

die Menge der relevanten *Variablen in s.*

Die Variablen eines booleschen Ausdrucks, von denen er *syntaktisch* abhängt, werden definiert als die Menge der Variablen, die in ihm vorkommen. Später werden Quantoren eingeführt, die Variablen binden. In diesem Sinne sind die Variablen, von denen ein Term syntaktisch abhängt, *frei* in diesem Term, weshalb für den Namen des Operators „free“ gewählt wurde.

Definition 2.6 (Variablen eines booleschen Ausdruck)

$$\mathsf{free}\colon \mathbb{B}^W \to \mathbb{P}(W)$$

wird rekursiv definiert durch

$$\mathsf{free}(0) := \mathsf{free}(1) := \emptyset, \quad \mathsf{free}(\neg s) := \mathsf{free}(s),$$
$$\mathsf{free}(s \wedge t) := \mathsf{free}(s) \cup \mathsf{free}(t), \quad \mathsf{free}(x) := \{x\}, \quad x \in W$$

Als weitere Abkürzung werden Quantoren eingeführt. Die gebundenen Variablen stammen dabei aus W, so daß für sie wie üblich nur $\{0,1\}$-wertige Interpretationen erlaubt sind.

Definition 2.7 (Quantoren in $\mathbb{B}^W$) *Für $x \in W$, $s \in \mathbb{B}^W$ definiere*

$$\exists x.\, s \;:=\; s\{x \mapsto 0\} \vee s\{x \mapsto 1\} \qquad \forall x.\, s \;:=\; s\{x \mapsto 0\} \wedge s\{x \mapsto 1\} \tag{2.1}$$

und für $E = \{x_1, \ldots, x_n\} \subseteq W$ mit $n \in \mathbb{N} \backslash \{0\}$

$$\exists E.\, s \;:=\; \exists x_1, \ldots, x_n.\, s \;:=\; \exists x_1.\, \exists x_2.\, \ldots \exists x_n.\, s \tag{2.2}$$

$$\forall E.\, s \;:=\; \forall x_1, \ldots, x_n.\, s \;:=\; \forall x_1.\, \forall x_2.\, \ldots \exists x_n.\, s \tag{2.3}$$

wobei $x_1, \ldots, x_n$ eine feste Aufzählung von E darstellt.

Mit den oben eingeführten Schreibweisen (vgl. auch Seite 170) hat man dann

$$\exists E.\, s \;\equiv\; \bigvee_{a \in \mathbb{B}^n} s\{(x_1, \ldots, x_n) \mapsto a\} \tag{2.4}$$

$$\forall E.\, s \;\equiv\; \bigwedge_{a \in \mathbb{B}^n} s\{(x_1, \ldots, x_n) \mapsto a\} \tag{2.5}$$

Für diese Arbeit macht es wenig Sinn, eine erweiterte Algebra von „quantifizierten" booleschen Ausdrücken einzuführen, wie das z. B. in /Garey und Johnson, 1979/ auf Seite 171 untersucht wird. So folgt schon aus obiger Definition

$$\mathsf{free}(\exists E.\, s) = \mathsf{free}(\forall E.\, s) = \mathsf{free}(s) \backslash E,$$

was der üblichen Definition von „freien" Variablen entspricht. Die Konsequenz aus dieser Vorgehensweise ist, daß bei der Definition der „freien" Variablen im μ-Kalkül eine zusätzliche Definition notwendig ist, da dort die Quantoren *syntaktische* Konstrukte sind. Dies macht aber bei weitem nicht den Vorteil wett, den man bei der hier getroffenen Wahl hat, daß auf die Definition einer weiteren booleschen Algebra verzichtet werden kann.

Das Ergebnis der Substitution einer Variablen in einem booleschen Ausdruck durch eine Konstante wird als Kofaktor bezeichnet. Eine wesentliche Eigenschaft von den später behandelten geordneten BDDs (OBDDs) ist, daß sich ihre Kofaktoren bez. bestimmten Variablen sehr einfach berechnen lassen.

Definition 2.8 (Kofaktor) *Für $s \in \mathbb{B}^W$ und $x \in W$ heiße*

$$s\{x \mapsto 1\} \quad \textit{bzw.} \quad s\{x \mapsto 0\}$$

der Kofaktor *von s bez. x bzw. $\bar{x}$.*

2.5.2 Fakten

Die in diesem Unterabschnitt vorgestellten Sätze werden für den Beweis der Wohldefiniertheit der Semantik von $\mathbb{B}^W$ (siehe Satz 2.38) und den Beweis der Normalformeigenschaft von BDDs (siehe Satz 3.16) benötigt.

Der erste Satz ist eine Formulierung der Tatsache, daß die semantische Äquivalenz bez. den booleschen Operationen eine *Kongruenz* ist.

Satz 2.9 (Verträglichkeit von „$\equiv$“) *Für $s, s', t, t' \in \mathbb{B}^W$ gilt*

$$s \equiv s' \quad \Rightarrow \quad \neg s \equiv \neg s' \tag{2.6}$$
$$s \equiv s',\ t \equiv t' \quad \Rightarrow \quad s \wedge t \equiv s' \wedge t' \tag{2.7}$$

Beweis: Sei ρ_W eine Variablenbelegung und es gelte $s \equiv s'$ und $t \equiv t'$. Man erhält

$$[\![\neg s]\!]\rho_W = \neg[\![s]\!]\rho_W = \neg[\![s']\!]\rho_W = [\![\neg s']\!]\rho_W$$

und

$$[\![s \wedge t]\!]\rho_W = [\![s]\!]\rho_W \wedge [\![t]\!]\rho_W = [\![s']\!]\rho_W \wedge [\![t']\!]\rho_W = [\![s' \wedge t']\!]\rho_W$$

❑

Man darf also modulo „$\equiv$“ jeden Unterterm durch einen äquivalenten ersetzen. Dies gilt natürlich auch für die oben definierten Abkürzungen wie Quantoren. Dieses Ersetzen wird nicht formalisiert, da man sonst auch „Ersetzungsstellen“ formalisieren müßte, was für diese Arbeit bestimmt nicht notwendig ist.

Satz 2.10 *Für $s \in \mathbb{B}^W$ gilt*

$$[\![s]\!]\rho_W = [\![s]\!]\rho'_W$$

für Variablenbelegungen ρ_W, ρ'_W, mit $\rho_W(x) = \rho'_W(x)$ für alle $x \in \mathsf{free}(s)$.

Beweis: Zunächst sei $s = x$ mit $x \in W$. Nach Voraussetzung folgt unmittelbar

$$[\![x]\!]\rho_W = \rho_W(x) \stackrel{!}{=} \rho'_W(x) = [\![s]\!]\rho'_W$$

Der Rest folgt durch Induktion entlang der Termstruktur. ❑

In Abschnitt A.3 findet man die Definition für die Juxtaposition $\rho_W\gamma$ zweier Belegungen ρ_W und γ (als partielle Funktionen) wie sie im nächsten Lemma auftritt.

Lemma 2.11 *Für eine Variablenbelegung ρ_W, $t \in \mathbb{B}^W$ und $\gamma\colon W \to \mathbb{B}$ gilt*

$$[\![t\gamma]\!]\rho_W = [\![t]\!]\rho_W\gamma$$

wobei γ „innen“ als Substitution und „außen“ als Belegung betrachtet wird.

Beweis: Der Beweis wird durch eine Induktion über den Termaufbau von t geführt. Im Basisfall sei t zunächst eine Konstante. Da eine Konstante durch eine Substitution immer auf sich selbst abgebildet wird, und auch die Semantik diese zur entsprechenden Konstante in $\mathbb{B}$ auswertet, steht auf beiden Seiten das gleiche. Nun sei $t = x$ mit $x \in W$ gegeben. Die linke Seite lautet dann

$$[\![x\gamma]\!]\rho_W = [\![h(x)]\!]\rho_W = \rho_W(h(x)) = \begin{cases} \rho_W(\gamma(x)), & \text{falls } \gamma(x)\downarrow \\ \rho_W(x)\quad, & \text{falls } \gamma(x)\uparrow \end{cases} = \cdots$$

Dabei sei h wie in Def. 2.4 die totale Fortsetzung von γ auf ganz $\mathbb{B}^W$ und somit auch auf W. Mit den Schreibweisen für die Juxtaposition von partiellen Substitutionen aus Abschnitt A.3 erhält man weiter

$$\cdots = (\rho_W\gamma)(x) = [\![x]\!]\rho_W\gamma$$

Der eigentliche induktive Schritt ist Standard. ❑

Der nächste Satz zeigt eine Möglichkeit auf, wie sich ein boolescher Ausdruck für eine Variable x auf kanonische Weise in zwei Teilterme zerlegen läßt, in denen die Variable x nicht mehr vorkommt, und deren Kombination mittels einer Fallunterscheidung nach x, äquivalent zum ursprünglichen Term ist. Die Teilterme sind dabei die Kofaktoren aus Def. 2.8. Für die später eingeführten (geordneten) BDDs wird sich dies von herausragender Bedeutung erweisen.

Satz 2.12 (Shannonsches Expansionstheorem)

$$s \quad \equiv \quad \overline{x}\, s\{x \mapsto 0\} \,\vee\, x\, s\{x \mapsto 1\} \tag{2.8}$$

für alle $s \in \mathbb{B}^W$, $x \in W$.

Beweis: Sei ρ_W eine Variablenbelegung. Im Falle $\rho_W(x) = 0$ ergibt Lemma 2.11

$$[\![s]\!]\rho_W = [\![s]\!]\rho_W\{x \mapsto 0\} = [\![s\{x \mapsto 0\}]\!]\rho_W$$

Damit erhält man

$$\begin{aligned}[\![\overline{x}\, s\{x \mapsto 0\} \,\vee\, x\, s\{x \mapsto 1\}]\!]\rho_W &= \overbrace{\neg\rho_W(x)}^{=1}\,[\![s\{x \mapsto 0\}]\!]\rho_W \,\vee\, \overbrace{\rho_W(x)}^{=0}\,[\![s\{x \mapsto 1\}]\!]\rho_W \\ &= [\![s\{x \mapsto 0\}]\!]\rho_W \;=\; [\![s]\!]\rho_W\end{aligned}$$

Der Fall $\rho_W(x) = 1$ folgt analog. ❑

Ein weiteres einfaches Lemma, das sich wiederum mit der Semantik von „ite“ auseinandersetzt ist

Lemma 2.13 (Negation von ite**)** *Für* $a, b, c \in \mathbb{B}^W$ *ist* $\quad \neg\mathrm{ite}(a, b, c) \;\equiv\; \mathrm{ite}(a, \neg b, \neg c)$

Beweis: $\neg\mathrm{ite}(a,b,c) \equiv \overline{a\, b \,\vee\, \overline{a}\, c} \equiv (\overline{a} \vee \overline{b})\ (a \vee \overline{c}) \equiv a\, \overline{b} \,\vee\, \overline{a}\, \overline{c} \equiv \mathrm{ite}(a, \neg b, \neg c)$ ❑

Der nächste Satz stellt einen Zusammenhang her zwischen der semantischen Äquivalenz von booleschen Ausdrücken und der Menge von Variablen, von denen sie semantisch abhängen.[1]

Satz 2.14 *Für alle* $s, t \in \mathbb{B}^W$ *gilt* $\qquad s \equiv t \;\Rightarrow\; \mathsf{rel}(s) = \mathsf{rel}(t)$

[1] Dies ergibt mit Lemma 3.17.b ein notwendiges Kriterium für die Normalformeigenschaft von BDDs.

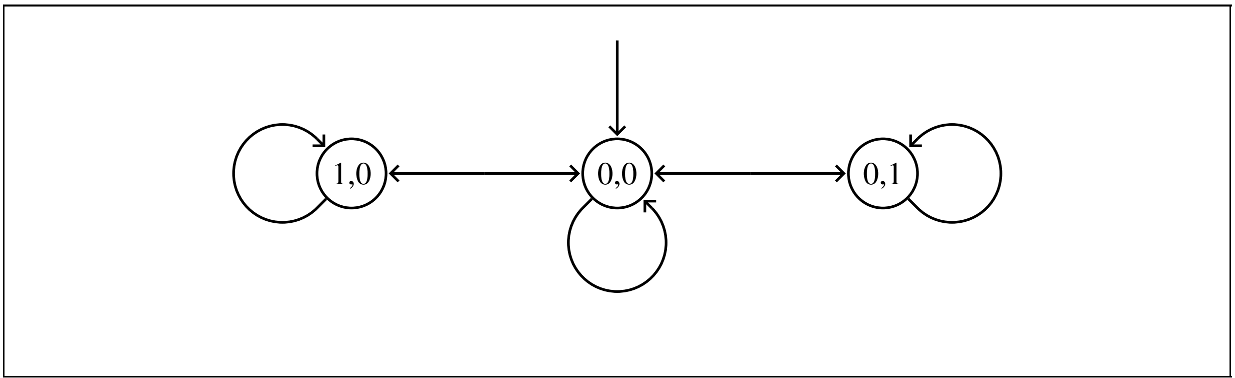


Abbildung 2.3: Einfaches Beispiel für binär kodierte Kripkestrukturen.

Beweis: Seien $s, t \in \mathbb{B}^W$ mit $s \equiv t$ und $x \in \mathsf{rel}(s)$, dann gibt es ρ_W mit

$$[\![s\{x \mapsto 0\}]\!]\rho_W \neq [\![s\{x \mapsto 1\}]\!]\rho_W,$$

so daß

$$\begin{aligned}[\![t\{x \mapsto 0\}]\!]\rho_W &= [\![t]\!]\rho_W\{x \mapsto 0\} = [\![s]\!]\rho_W\{x \mapsto 0\} = [\![s\{x \mapsto 0\}]\!]\rho_W \\ &\neq [\![s\{x \mapsto 1\}]\!]\rho_W = [\![s]\!]\rho_W\{x \mapsto 1\} = [\![t]\!]\rho_W\{x \mapsto 1\} = [\![t\{x \mapsto 1\}]\!]\rho_W\end{aligned}$$

und somit $t\{x \mapsto 0\} \not\equiv t\{x \mapsto 1\}$, was $x \in \mathsf{rel}(t)$ bedeutet (genauso umgekehrt). ❑

2.6 Der μ-Kalkül $\mathbb{B}^V_\mu$

Der μ-Kalkül ist eine Logik, in der verschiedene Verifikationsaufgaben beschrieben und durch Berechnung der Semantik eines μ-Kalkül-Termes auch durchgeführt werden können.

2.6.1 Verifikationsbeispiel

Als Beispiel wird hier die „Verifikation“ eines einfaches Zustandübergangssystem betrachtet. Genauer wird eine Sicherheitseigenschaft nachgewiesen. Das System selbst (als Kripkestruktur vgl. /Emerson, 1990/) ist in Abb. 2.3 gezeigt. Es besteht aus dem Startzustand $(0,0)$ und zwei weiteren Zuständen $(0,1)$ und $(1,0)$. Im folgenden sind die betrachteten Zustandsübergangssysteme immer binär kodiert zu verstehen, wobei das Problem der Binärkodierung nicht weiter erörtert werden soll (siehe hierzu z. B. /Theobald und Meinel, 1996/).

Ein wichtiges Konzept ist das der „erreichbaren Zustände“ (s. Kapitel 5). Hier sind das die drei oben erwähnten Zustände. In der Binärkodierung gibt es aber noch den Zustand $(1,1)$, der gerade *nicht* erreichbar ist. Dieser könnte bei entsprechender Interpretation gerade ein Fehlverhalten des Systems beschreiben. In diesem Sinne ist die Bestimmung der erreichbaren Zustände und der Test, ob in dieser Menge ein Zustand liegt, der zu einem Fehlverhalten des Systems führt, Aufgabe der Verifikation.

So kann man obige Kripkestruktur interpretieren als eine abstrakte Version eines Systems, das aus zwei Prozessen (einem linken und einem rechten) mit gemeinsamen kritischen Abschnitt besteht.[2] Wenn sich ein Prozeß im kritischen Abschnitt befindet, so ist die entsprechende

[2]Es soll in diesem Abschnitt nicht darauf eingegangen werden, wie man die Kripkestruktur aus einer konkreten Beschreibung der Prozesse gewinnt.

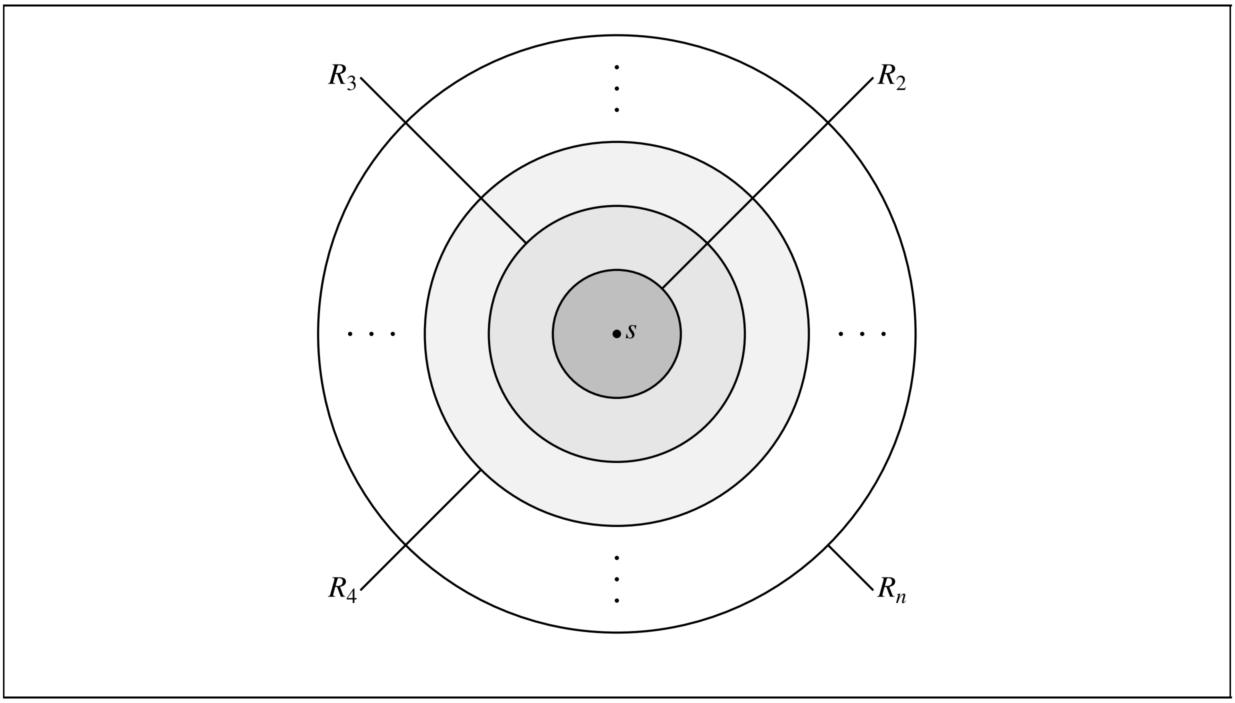


Abbildung 2.4: Erreichbarkeitsanalyse durch Breitensuche ($R_1 = \{s\}$).

Variable in der Kripkestrukur auf 1 gesetzt, sonst auf 0. Die zu verifizierende Sicherheitseigenschaft des Systems lautet

„*Die zwei Prozesse befinden sich nie gleichzeitig im kritischen Abschnitt*“

Dies ist in der abstrakten Kripkestruktur äquivalent zu

$$\text{„\textit{Der Zustand} } (1,1) \text{ \textit{ist nicht erreichbar.}“} \tag{2.9}$$

Für dieses einfache System ist der Nachweis dieser Aussage trivial, da das Gesamtsystem leicht zu überblicken ist. Im allg. ist das System aber nur an Hand der einzelnen Übergänge gegeben und man muß beginnend mit dem Startzustand sukzessive alle erreichbaren Zustände, von denen es sehr viele (10^{20} oder mehr) geben kann, auf die Verletzung einer Sicherheitseigenschaft überprüfen.

Für die naive Durchführung der Erreichbarkeitsanalyse hat man klassischerweise zwei Möglichkeiten, nämlich *Tiefensuche* oder *Breitensuche*. Für nur einen Startzustand s, wie in diesem Beispiel, kann man die Breitensuche wie in Abb. 2.4 graphisch veranschaulichen. In dem betrachteten Beispiel gilt

$$\begin{aligned}
R_0 &= \{\} \\
R_1 &= \{s\} = \{(0,0)\} \\
R_2 &= \{(0,0),(0,1),(1,0)\} \\
R_3 &= \{(0,0),(0,1),(1,0)\} \\
&\vdots
\end{aligned}$$

Das bedeutet, daß die Breitensuche nach zwei Iterationen alle erreichbaren Zustände gefunden hat. Die Tiefensuche läßt sich am besten durch Angabe des Suchbaumes veranschaulichen, wie er in Abb. 2.5 für das betrachtete Beispiel aufgezeigt ist. Dieser stellt einen *spannenden Baum* für die Übergangsrelation aus Abb. 2.3 dar.

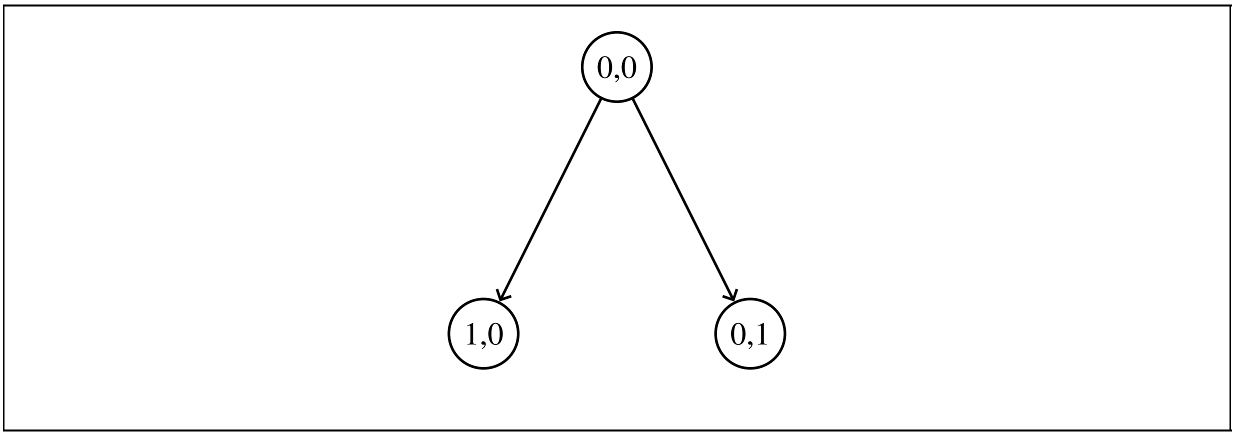


Abbildung 2.5: Erreichbarkeitsanalyse durch Tiefensuche.

Die in dieser Arbeit betrachtete *globale* Modellprüfung entspricht der Breitensuche (siehe /Clarke et al., 1986/ und /McMillan, 1993a/). Dagegen ist die *lokale* Modellprüfung im wesentlichen eine Tiefensuche (siehe /Cleaveland, 1990/ und /Bhat und Cleaveland, 1996/). Für einen Vergleich beider Methoden siehe /Kick, 1996/.

Anstatt die (Breiten-)Suche explizit durchzuführen, wird die Übergangsrelation und die Sicherheitseigenschaft in $\mathbb{B}_\mu^V$ formuliert. So reduziert sich die Verifikation zu einer Auswertung einer Formel des μ-Kalküls. Die Vorgehensweise bei der Auswertung der Formel, zurückprojiziert auf den Anwendungsbereich, stellt sich aber wiederum als Breitensuche heraus. Hierzu werden vier Prädikate über Zuständen benötigt.

- $S(s)$ ist wahr gdw. s ein Startzustand ist ($s = (0,0)$)
- $T(s,t)$ ist wahr gdw. t ein Nachfolger von s ist
- $E(s)$ ist wahr gdw. s die Sicherheitseigenschaft verletzt ($s = (1,1)$)
- $R(s)$ ist wahr gdw. s erreichbar ist

Die ersten drei Prädikate sind direkt in $\mathbb{B}_\mu^V$ zu definieren als

$$\begin{aligned} S(s) &\,.\, \neg s[0] \wedge \neg s[1] \\ T(s,t) &\,.\, \neg s[0] \wedge \neg t[0] \vee \neg s[1] \wedge \neg t[1] \\ E(s) &\,.\, s[0] \wedge s[1] \end{aligned}$$

Der Punkt „." wird dabei als „ist definiert als" gelesen. Die Definition für T kann man so interpretieren, daß ein Prozeß sich beliebig verhalten kann, wenn der andere Prozeß sich nicht im kritischen Abschnitt befindet und diesen auch nicht im nächsten Zustand betritt. Statt dieser etwas verwirrenden Version könnte man natürlich auch alle Übergänge einzeln auflisten. Es soll hier nur beispielhaft gezeigt werden, daß die Beschreibung auch eines solch einfachen Übergangssystem schon unübersichtlich werden kann.[3] Was die erreichbaren Zustände angeht, muß man auf eine rekursive Definition zurückgreifen:

[3] Hier müßte eigentlich diskutiert werden, wie man Systeme so konstruieren kann, daß solche unübersichtlichten Beschreibungen nicht entstehen. Dies würde aber den Rahmen der Arbeit sprengen. Darüber hinaus vertritt der Autor die Auffassung, daß in vielen „verteilten Systemen" diese Komplexität inhärent mit der Problemstellung verbunden ist.

> Die Menge der erreichbaren Zustände R ist die kleinste Menge, in der die Startzustände S liegen, und die abgeschlossen ist gegenüber Nachfolgerbildung.

Man beachte, daß hier eine Menge mit einem Prädikat identifiziert wird. Eine äquivalente Formulierung lautet

> Die Menge der erreichbaren Zustände R ist die kleinste Menge, so daß für jedes s aus R gilt, daß s in S liegt oder es einen Vorgänger t von s gibt (es gilt $T(t,s)$), der in in R liegt.

In $\mathbb{B}_{\mu}^{V}$ wird das wie folgt formuliert.

$$\mu R(s)\,.\,S(s) \;\vee\; \exists\, t\,.\,T(t,s) \wedge R(t)$$

Hier steht das „μ" dafür, daß R das (als Menge) kleinste Prädikat sein soll, das die rechte Seite erfüllt. Nun kann man daran gehen, die eigentlich zu verifizierende Aussage zu formulieren (vgl. (2.9) auf S. 17):

$$\forall s.R(s) \rightarrow \neg E(s) \tag{2.10}$$

Die Verifikation der Sicherheitseigenschaft bedeutet nun die Auswertung dieses Terms. Das (abstrakte) System erfüllt diese Eigenschaft genau dann, wenn der Term zu wahr evaluiert. Der Rest des Abschnitts beschäftigt sich mit der Syntax dieser Terme und zwei Methoden (Semantiken) für deren Auswertung.

2.6.2 Vergleich mit anderen μ-Kalkülen

In den folgenden Abschnitten wird der μ-Kalkül $\mathbb{B}_{\mu}^{V}$ vorgestellt. Er ist eine Abwandlung des μ-Kalküls aus /Park, 1976/, der auch in /Burch et al., 1990/ als Grundlage herangezogen wurde. Die Hauptunterschiede zu diesem μ-Kalkül sind

- keine beliebigen Domänen,
- dafür aber ein einfaches Typsystem
- keine λ-Abstraktionen (und relationale Terme) aber
- dafür definierende Gleichungen für Prädikate.

Im einzelnen bedeutet dies, daß als Grundbereich nur die Menge von booleschen Vektoren und nicht beliebige Domänen herangezogen werden. Dies dient nur dem Zwecke der einfachen Behandlung der Übersetzung in BDDs und der Probleme bei Allokationen (Variablenordnung) von BDD-Variablen. Um für die Allokationen sinnvolle Aussagen machen zu können, werden Typen eingeführt. Dies macht $\mathbb{B}_{\mu}^{V}$ zu einer „sortierten" Logik mit einfachen, fest interpretierten Sorten. Um den Formalisierungsaufwand gering zu halten, beschränkt man sich sogar auf boolesche Vektoren, die natürlich erlauben beliebige endliche Domänen zu kodieren. Dennoch ist die Definition der Syntax (Sorten und Typisierung) wie der Semantik so allgemein gehalten, daß sie sich mit geringfügigen Einschränkungen auch auf den unendlichen Fall übertragen läßt.

Der zweite Unterschied ist durch die Beobachtung begründet, daß die „relationalen Terme", die in der anderen Version des μ-Kalküls auftreten, nur schwer zu verstehen sind und sich durch ein einfacheres Namenskonzept ersetzen lassen. Dies bedeutet, daß für Prädikate definierende

Gleichungen eingeführt werden, wie es von funktionalen Programmiersprachen bekannt ist (siehe z. B. /Jones, 1987/). Für *nicht so allgemeine* Versionen des μ-Kalküls wurde in /Dicky, 1986/ und /Cleaveland und Steffen, 1993/ ähnlich verfahren.

Der μ-Kalkül von Park aus /Park, 1976/ liegt in seiner Ausdrucksstärke zwischen Prädikatenlogik erster und zweiter Stufe (siehe eben diese Arbeit). Aus diesem Grund kann es keine vollständigen Axiomensysteme, noch Entscheidungsprozeduren geben. Dasselbe gilt für $\mathbb{B}_\mu^V$. Diese Problematik ist für den Gegenstand der vorliegenden Arbeit aber unwichtig, da die wesentliche Einschränkung vorgenommen wird, daß die Modelle, für die der Wahrheitswert einer Formel berechnet werden soll, bekannt und (zumeist) endlich sind. In diesem Fall reicht es aus, alle endliche Modelle kodieren zu können, was durch die Wahl von booleschen Vektoren beliebig großer Stelligkeit als Domäne gewährleistet ist. So werden alle Bestandteile der betrachteten Modelle (in der einfachen Version sind das genau alle Prädikate) syntaktisch charakterisierbar. In diesem Sinne wird der μ-Kalkül, der an sich eine Logik darstellt, bis auf die fehlenden Beweisregeln erst zum Kalkül.

Weiter sei auf den Abschnitt B.2 verwiesen, in dem Übersetzungen verschiedener Logiken und formaler Beschreibungssprachen in den μ-Kalkül beschrieben sind. Neben den Beispielen aus Kapitel 5 findet man dort zusätzliche Hinweise wie Modellprüfung an sich und für den μ-Kalkül praktisch eingesetzt werden kann.

2.6.3 Sorten, Signaturen und Belegungen

In diesem Unterabschnitt wird das Typsystem (die Sorten) von $\mathbb{B}_\mu^V$ vorgestellt. Es folgt die Definition von Signaturen und Belegungen von Variablen.[4]

Definition 2.15 (Sorten bzw. Typen) *Die Menge*

$$S := \{\mathbb{B}^n \mid n \in \mathbb{N}\backslash\{0\}\}$$

heiße die Menge der Sorten oder Typen.

Eine Sorte $s = \mathbb{B}^n \in S$ ist also die Menge aller n-stelligen Vektoren über $\mathbb{B}$ – oder kurz die Menge aller *booleschen Vektoren* der Stelligkeit n – für ein n aus $\mathbb{N}\backslash\{0\}$. Weiter werden Variablen und Prädikate über diesen Sorten betrachtet. Dabei steht eine einfache Variable für ein Element ihres Typs und ein Prädikat für eine Teilmenge des Kreuzproduktes der Typen der Argumente, auf die es angewendet wird. Aus diesem Grund wird der Typ eines Prädikates genau als dieses Kreuzprodukt definiert. Dies führt zu etwas technischem Aufwand bei der Definition der Menge aller möglichen Belegungen für Prädikate (vgl. Bsp. nach folgender Def.).

Definition 2.16 (Signatur $\mathcal{S}$ für $\mathbb{B}_\mu^V$) *Eine Signatur besteht aus*

$$\begin{array}{ll} V, & \textit{der Menge der (endlich vielen) Individuenvariablen,} \\ P, & \textit{der Menge der (endlich vielen) Prädikatsvariablen,} \end{array}$$

einer Typisierung τ_V der Individuenvariablen

$$\tau_V : V \to S$$

und einer Typisierung τ_P der Prädikatsvariablen

$$\tau_P : P \to S^+, \qquad \textit{wobei } S^+ := \bigcup_{i=1}^{\infty} S^i \textit{ und } S^i := \{s_1 \times \cdots \times s_i \mid s_k \in S \textit{ für } 1 \leq k \leq i\}$$

Dabei definiere $|u| := n$ für ein $u \in V$, falls $\tau_V(u) = \mathbb{B}^n$, und ebenso $|X| := n$ für ein $X \in P$, falls $\tau_P(X) \in S^n$, und bezeichne n auch als die Länge von u bzw. X. Weiter sei

$$\tau_V(\underline{u}) := \tau_V(u_1) \times \cdots \times \tau_V(u_n)$$

für $\underline{u} = (u_1, \ldots, u_n) \in V^n$, $n \in \mathbb{N}\backslash\{0\}$.

Als Beispiel sei $X \in P$ und $u \in V$ mit den Typisierungen

$$\tau_V(u) = \mathbb{B}^2 \quad \text{und} \quad \tau_V(X) = \mathbb{B}^2 \times \mathbb{B}$$

Es werden also keine impliziten Typkonversionen (wie z. B. $\tau_V(X) = \mathbb{B}^3$) durchgeführt, was den Mißbrauch der Schreibweise S^i rechtfertigt. Wie bei $\mathbb{B}^W$ wird auch hier die Endlichkeit von V

[4] In Einführungsbüchern zur Prädikatenlogik werden Belegungen erst zusammen mit der Semantik eingeführt. Hier wird deren Definition vorgezogen, da keine abstrakten Sorten verwendet werden, und so deutlich gemacht werden soll, daß der klassische Begriff der Interpretation nicht benötigt wird.

verlangt und aus Gründen, die in der Anmerkung zur Definition 2.28 auf Seite 29 klar werden, fordert man dies auch für P. In dieser Arbeit werden keine Probleme betrachtet, bei denen zwischen verschiedenen Signaturen unterschieden werden muß. Deshalb sei eine Signatur $\mathcal{S}$ zusammen mit V, P und den Typisierungen fest gewählt. Der Einfachheit halber lasse man die Indizes bei τ_V und τ_P weg. Man definiere also

$$\tau\colon P \mathbin{\dot\cup} V \to S^+ \cup S = S^+, \qquad \text{mit } \tau := \tau_V \mathbin{\dot\cup} \tau_P$$

Eine Belegung für die Individuenvariable $u \in V$ ist ein Element des Typs von u. Prädikate stehen für eine Teilmenge ihres Typs. In diesem Sinne ist eine Belegung für ein $X \in P$ eine Teilmenge von $\tau(X)$. Man beachte, daß hier und im Rest der Arbeit der Begriff der Interpretation unter den der Belegung subsumiert wird.

Definition 2.17 (Belegungen in $\mathbb{B}_\mu^V$) *Eine* totale *Abbildung*

$$\rho_V : V \to \bigcup_{s \in S} s, \qquad \rho_V(u) \in \tau(u), \quad \textit{für } u \in V$$

ist eine Belegung der Individuenvariablen V. Eine partielle *Abbildung*

$$\rho_P\colon P \to \bigcup_{w \in S^+} \mathbb{P}(w), \qquad \rho_P(X) \subseteq \tau(X), \quad \textit{für } X \in P$$

heiße eine Belegung der Prädikatsvariablen X.

Bei dieser Definition beachte man, daß die Mengen s und $\mathbb{P}(w)$ *nicht* als Mengen bestehend nur aus einem Element uminterpretiert werden. So gilt

$$S = \bigcup_{s \in S} \{s\} \neq \bigcup_{s \in S} s \quad \text{und} \quad \mathbb{P}(w_1) \cup \mathbb{P}(w_2) = \{\xi \mid \xi \subseteq w_1 \text{ oder } \xi \subseteq w_2\}$$

für $w_1, w_2 \in S^+$, wobei immer disjunkte Vereinigung gemeint ist. Die Bezeichner w, w_1 und w_2 sollen suggerieren, daß es sich dabei um Worte über S handelt. Dies ist aber nur dann korrekt, wenn man Wörter und Kreuzprodukte wie folgt identifiziert

$$w = \mathbb{B}^{i_1} \times \cdots \times \mathbb{B}^{i_n} = s_1 \times \cdots \times s_n \quad \text{identifiziert mit} \quad (s_1, \ldots, s_n) = (\mathbb{B}^{i_1}, \ldots, \mathbb{B}^{i_n}),$$

für geeignet gewählte $i_1, \ldots, i_n \in \mathbb{N}\backslash\{0\}$.

Hier steht der Begriff der Belegung auch für den klassischen Begriff der Interpretation, da man ρ_V und ρ_P wiederum zusammenfaßt zu

$$\rho\colon P \mathbin{\dot\cup} V \to (\bigcup_{s \in S} s) \mathbin{\dot\cup} (\bigcup_{w \in S^+} \mathbb{P}(w)), \qquad \text{mit } \rho := \rho_W \mathbin{\dot\cup} \rho_P$$

Als Beispiel sei $X \in P$, mit $\tau(X) = (\mathbb{B}^2, \mathbb{B}) \in S^2$, und

$$\rho(X) = \{(\,(a,b),(c)\,) \mid c \leftrightarrow (a \to b)\}$$

Damit ist $\rho(X)$ ein Prädikat über einem zweistelligen booleschen Vektor (a,b) und einem einstelligen booleschen Vektor (c). Es ist genau dann wahr, wenn c gleich dem Ergebnis der Auswertung von $a \to b$ ist. Weiter seien $u, v \in V$, mit $\tau(u) = \mathbb{B}^2$ und $\tau(v) = \mathbb{B}$. Nun gelte

$$\rho(u) = (1,0) \in \mathbb{B}^2 \qquad \text{und} \qquad \rho(v) = (0) \in \mathbb{B}^1.$$

so daß $X(u,v)$ unter ρ zu wahr evaluiert wird (vgl. Def. 2.33).

2.6.4 Syntax

Nun kann die eigentliche Syntax von $\mathbb{B}_\mu^V$ eingeführt werden. Dazu sei noch einmal erwähnt, daß die Variablen für boolesche Vektoren stehen. Im folgenden wird definiert, wie syntaktisch die Anwendung einer möglicherweise mehrstelligen Präkdikatsvariablen auf ein Wort von Variablen aufgeschrieben wird. So ein Variablenwort steht für einen Vektor von booleschen Vektoren, weshalb dieses Wort mit $\underline{u}$ gekennzeichnet wird, und *nicht* für ein Vektor von booleschen Werten.

Definition 2.18 (Terme T_μ^V in $\mathbb{B}_\mu^V$) *T_μ^V ist die kleinste Menge mit*

$$\begin{array}{rll}
u[i] \in T_\mu^V & \textit{falls} & u \in V,\ i \in \mathbb{N},\ 0 \le i < |u| \\
u \doteq v \in T_\mu^V & \textit{falls} & u, v \in V,\ |v| = |u| \\
X(\underline{u}) \in T_\mu^V & \textit{falls} & X \in P,\ \underline{u} \in V^n,\ \tau(X) = \tau(\underline{u}) \\
\exists u.\, s \in T_\mu^V & \textit{falls} & s \in T_\mu^V,\ u \in V \\
s \wedge t \in T_\mu^V & \textit{falls} & s, t \in T_\mu^V \\
\neg s \in T_\mu^V & \textit{falls} & s \in T_\mu^V
\end{array}$$

Neben den üblichen Annahmen über die Bindung von Operatoren sei hier noch vereinbart, daß der Existenzquantor am schwächsten bindet. Dies bedeutet, daß sich der Gültigkeitsbereich eines Quantor immer so weit wie möglich über den folgenden Term erstreckt.

Terme aus T_μ^V entsprechen bis auf „$u[i]$“ atomaren Aussagen in der üblichen Definition von Prädikatenlogik erster Stufe. Der Variablencharakter der Prädikatsvariablen tritt hier noch nicht auf, so daß man nicht von zweiter Stufe sprechen kann. Terme im Sinne der Prädikatenlogik gibt es nicht, wenn man nicht „$u[i]$“ als solchen ansieht, der es ermöglicht, auf einzelne Komponenten eines Vektors zuzugreifen, damit funktionale Vollständigkeit (die Definierbarkeit aller Prädikate) erreicht wird. Hier ist mit funktionaler Vollständigkeit wie in /Schmitt, 1992a/ gemeint, daß für jede boolesche Funktion über boolesche Vektoren ein μ-Kalkül-Term existiert, dessen Semantik mit der booleschen Funktion übereinstimmt.

Wie die Typisierung der Variablen nahelegt, werden in dieser Arbeit nur endliche Domänen betrachtet, die auch schon durch die Typisierung festgelegt sind. In diesem Sinne ist der betrachtete μ-Kalkül sogar nur so ausdrucksmächtig wie die Aussagenlogik.

Wie in Abschnitt 2.4 werden die üblichen booleschen Operatoren wie ‚$\vee$‘, ‚$\rightarrow$‘ und ‚$\leftrightarrow$‘ als Abkürzungen über ‚$\wedge$‘ und ‚$\neg$‘ definiert. Das gleiche geschieht wie in der Prädikatenlogik üblich mit dem Allquantor

$$\forall u.\, s \quad := \quad \neg\exists u. \neg s$$

Des öfteren wird auch über eine Menge von Variablen quantifiziert. Diese Menge wird dann auch als Vektor $\underline{u} \in V^{|\underline{u}|}$ geschrieben und man verwendet die Abkürzung

$$\exists \underline{u}.\, s \quad := \quad \exists u_1.\, \exists u_2. \ldots \exists u_n.\, s$$

und natürlich analog für den Allquantor. Es sei hier schon erwähnt, daß ‚$u \doteq v$‘ eigentlich auch als Abkürzung hätte definiert werden können, nämlich durch

$$u \doteq v \quad = \quad \bigwedge_{0 \le i < |u|} (u[i] \leftrightarrow v[i])$$

Jedoch soll dieses syntaktische Konstrukt beispielhaft für Erweiterungen von $\mathbb{B}_\mu^V$ stehen, für die spezielle Überlegungen bei Übersetzungen angestellt werden sollten. Dies ist bei der Übersetzung in BDDs besonders von Bedeutung, wie Abschnit 4.3.1 zeigt.

Die Quantoren sind in $\mathbb{B}_\mu^V$ *syntaktische* Konstrukte und nicht nur eine abkürzende Schreibweise wie bei den Quantoren aus Abschnitt 2.5. Deshalb kann hier nicht die Definition von „free" aus jenem Abschnitt übernommen werden, sondern es wird neu definiert (aber mit demselben Symbol gekennzeichnet):

Definition 2.19 (Freie Variablen) free *ist eine Abbildung* $\mathsf{free}\colon T_\mu^V \to \mathbb{P}(V \cup P)$ *mit*

$$\begin{array}{llll} \mathsf{free}(u[i]) & := \{u\} & \mathsf{free}(\exists u.\, s) & := \mathsf{free}(s) \backslash \{u\} \\ \mathsf{free}(u \doteq v) & := \{u, v\} & \mathsf{free}(s \wedge t) & := \mathsf{free}(s) \cup \mathsf{free}(t) \\ \mathsf{free}(X(\underline{u})) & := \underline{u} \cup \{X\} & \mathsf{free}(\neg s) & := \mathsf{free}(s) \end{array}$$

Dabei wird $\underline{u} = (u_1, \ldots, u_{|u|})$ *mit der Menge* $\{u_1, \ldots, u_{|u|}\}$ *identifiziert.*

Für weitere Definitionen wird die übliche Untertermordnung benötigt. Dies ist hier noch einmal extra aufgeführt, um genau die Menge der Unterterme eines Termes zu fixieren.

Definition 2.20 (Untertermordnung) *Auf der Menge* T_μ^V *wird mit „$\leq$" (bzw. „$<$") die* Untertermordnung *bezeichnet. Dabei sei „$<$" definiert als die kleinste transitive Relation, mit*

$$\begin{array}{rlrl} u[i] > t & :\Leftrightarrow t = u & \exists u.\, s > t & :\Leftrightarrow t = s \text{ oder } t < s \\ u \doteq v > t & :\Leftrightarrow t \in \{u, v\} & r \wedge s > t & :\Leftrightarrow t \in \{r, s\},\ t < r \text{ oder } t < s \\ X(\underline{u}) > t & :\Leftrightarrow t = u_i,\ \text{für } 1 \leq i \leq |u| & \neg s > t & :\Leftrightarrow t = s \text{ oder } t < s \end{array}$$

„$\leq$" ist die reflexive Hülle von „$<$".

Ähnlich wie bei PROLOG werden Terme (Anfragen) bezüglich einer Definition von Prädikatsvariablen (PROLOG-Programm) ausgewertet. Nur nach Wahl einer Definition kann die Semantik eines Termes bestimmt werden. Eine Definition von Prädikatsvariablen ist dabei syntaktisch gegeben wie das auch im Beispiel auf S. 18 zu Abb. 2.3 vorgeführt wurde. Zu jedem Prädikat muß eine definierende Gleichung angegeben werden. Diese besteht aus den Bestandteilen

1. formale Parameter,
2. Rumpf und
3. Fixpunktart,

was in folgender Definition formalisiert wird.

Definition 2.21 (Definition von Prädikatsvariablen) *Eine Definition* $\delta = (\pi, \beta, \sigma)$ *besteht aus*

$$\pi\colon P \to V^*, \qquad \beta\colon P \to T_\mu^V, \qquad \sigma\colon P \to \{\varepsilon, \mu, \nu\}$$

Dabei gelten die Einschränkungen

$$\tau(\pi(X)) = \tau(X) \qquad \textit{und} \qquad (\mathsf{free}(\beta(X)) \cap V) \subseteq \pi(X),$$

wobei die Komponenten von $\pi(X)$ paarweise verschieden sind. Für $X \in P$ seien $\pi(X)$ die formalen Parameter, *$\beta(X)$ der* Rumpf *und $\sigma(X)$ die* Fixpunktart *von X.*

Als Eselsbrücke merke man sich β für Rumpf (engl. **b**ody) und π für **P**arameter. Aus der Literatur stammt der Bezeichner σ für die Fixpunktart. Hier wird nur zusätzlich noch ε als Wert zugelassen, was ein konstantes (nicht rekursives) Prädikat kennzeichnen soll. Statt explizit δ anzugeben, schreibe man auch wie oben z. B.

$$\mu X(\underbrace{u,v,w}_{\pi(X)}).\ \underbrace{u \vee X(w,u,v)}_{\beta(X)}$$

Das ‚μ' entspricht dabei $\sigma(X)$. Falls $\sigma(X) = \varepsilon$, so wird in dieser Schreibweise ε einfach weggelassen. Falls nicht anders angegeben entstammen großgeschriebene Variablen aus P und kleingeschriebene aus V. In diesem Beispiel gelte $\tau(u) = \tau(v) = \tau(w) = \mathbb{B}$.

Die Eingabesprache des Modellprüfers μcke ist im wesentlichen eine Erweiterung dieser Schreibweise um Typen. Obiges Beispiel lautet darin

```
mu bool X(bool u, bool v, bool w) u | X(w,u,v);
```

Ein kleiner Unterschied besteht darin, daß hier Prädikate als boolesche Funktionen definiert werden, was durch die Typisierung `bool` vor `X` geschieht, und der Punkt „." weggelassen wird. Als Vorgriff auf die Semantik sei hier schon verraten, daß

$$X(u[0],v[0],w[0]) \quad \equiv \quad u[0] \vee v[0] \vee w[0] \tag{2.11}$$

Mehrfach geschachtelte Fixpunkte lassen sich wie folgt darstellen

$$\begin{array}{ll} T(s,t)\,. & \ldots \\ H(s)\ \ . & \ldots \\ \mu\, Y(s) & .\, H(s) \wedge F(s)\ \vee\ \exists t.\, T(s,t) \wedge Y(t) \\ \nu\, F(s) & .\, \exists t.\, T(s,t) \wedge Y(t) \end{array} \tag{2.12}$$

Hier sind H und T als konstante Prädikate definiert. Die Rümpfe von H und T sind nicht weiter angegeben. Die Prädikate Y und F sind rekursiv definiert. Y als der kleinste Fixpunkt und F als der größte Fixpunkt. Diese Definition kann man ansehen als die Charakterisierung der Menge der „fairen" Zustände F eines Zustandübergangssystems mit Übergangsrelation T unter der „Fairneß" H. Genauere Erläuterungen zur Interpretation findet man in Abschnitt B.2.2.

In einem klassischen μ-Kalkül würde man sich F als den äußeren Fixpunkt und Y als den inneren Fixpunkt vorstellen. In $\mathbb{B}_\mu^V$ ist diese Unterscheidung hinfällig. Dies bringt den Nachteil mit sich, daß man bei Angabe der Semantik von $\mathbb{B}_\mu^V$ nun über allgemeine Graphen- statt Baumstrukturen argumentieren muß. Die verwendete Graphentheorie beschränkt sich aber im wesentlichen auf den Begriff der *stark zusammenhängenden Komponente* (kurz SCC für engl. *strongly connected component*). Dafür hat man unmittelbar den Vorteil, gemeinsame Teilausdrücke in der Syntax nur einmal hinschreiben zu müssen.

Der wichtigste Grund für die Verwendung von definierenden Gleichungen (und somit der Graphenstruktur) ist die natürlichere Schreibweise, die man daraus gewinnt, daß keine λ-Abstraktionen und relationale Terme wie in /Park, 1976/ nötig sind. Dies ist besonders wichtig, wenn man bedenkt, daß ein Modellprüfer nicht nur von Spezialisten zu bedienen sein sollte. Zumindest sollte dessen Bedienung leicht zu erlernen sein. Die Mehrzahl der Anwender

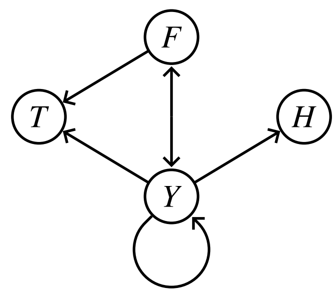


Abbildung 2.6: Abhängigkeitsrelation für zweites Beispiel (Gleichung (2.12)).

wird wohl eher eine moderne Programmiersprache beherrschen, als sich im λ-Kalkül auszukennen, und das hier verwendete Konzept der definierenden Gleichungen findet sich in fast jeder modernen Programmiersprache, die es gestattet, (rekursive) Prozeduren bzw. Funktionen oder Methoden zu definieren.[5]

Schließlich sei noch erwähnt, daß sich Untersuchungen zur denotationalen Semantik von Programmiersprachen wie in /Winskel, 1993/ nicht direkt für die Beschreibung der Semantik von $\mathbb{B}_\mu^V$ verwenden lassen, da in $\mathbb{B}_\mu^V$ auch größte Fixpunkte verwendet werden. Dadurch läßt sich die Existenz der Fixpunkte und somit die Wohldefiniertheit der Semantik nicht übertragen. Weiterer zusätzlicher Aufwand muß für Quantoren und die Negation getrieben werden.

Die oben genannte Graphenstruktur besteht aus den Abhängigkeiten der Prädikatsvariablen, die wie folgt syntaktisch aus der fest gewählten Definition von Prädikatsvariablen bestimmt werden.

Definition 2.22 (Abhängigkeitsrelation) *Die Relation*

$$X \to Y \quad :\Leftrightarrow \quad Y \in \mathsf{free}(\beta(X)), \qquad \text{für } X, Y \in P$$

heiße Abhängigkeitsrelation der Prädikatsvariablen.

Im Beispiel aus Gleichung (2.12) hätte man (vgl. Abb. 2.6)

$$F \to Y, \quad F \to T, \quad Y \to Y, \quad Y \to F, \quad Y \to H, \quad Y \to T$$

Betrachtet man den Graphen mit den Knoten P und Kanten aus „$\to$“, so ergibt dies den Begriff der stark zusammenhängenden Komponente:

Definition 2.23 (Stark zusammenhängende Komponente) *Für $X \in P$ sei*

$$\mathsf{scc}(X) := \{Y \in P \mid X \to^+ Y \text{ und } Y \to^+ X\},$$

wobei $\to^+$ für die transitive Hülle von $\to$ steht.

$$\mathsf{dep}(X) := \{Y \in P \mid X \to^* Y\}$$

[5] Im Modellprüfer μcke wurde dies sogar so weit getrieben, daß die verwendete Eingabesprache sehr stark an die weit verbreitete Programmiersprache C (bzw. C++) angelehnt wurde.

$\cup$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

	$\neg$
$\bot$	$\bot$
0	1
1	0
$\top$	$\top$

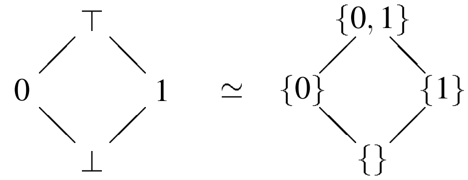


Abbildung 2.7: Der boolesche Verband L_4

$\mathsf{dep}(X)$ *bezeichnet die Menge der Prädikatsvariablen, von denen X rekursiv abhängt. Für Terme* $t \in T_\mu^V$ *definiere*

$$\mathsf{dep}(t) := \bigcup_{X(\underline{u}) \leq t} \mathsf{dep}(X)$$

Man beachte bei dieser Definitionen, daß „$\to^*$" reflexiv ist.

Üblicherweise (/Manber, 1989/) schlägt man einen Knoten seiner stark zusammenhängenden Komponente hinzu, was hier ausdrücklich nicht der Fall ist. Dies dient zur Unterscheidung konstanter Prädikate von solchen, die rekursiv nur von sich selbst abhängen. Unter der üblichen Definition wäre in beiden Fällen die SCC die gleiche. Für obiges Beispiel bestehen die SCCs aus (vgl. auch Abb. 2.6)

$$\mathsf{scc}(F) = \mathsf{scc}(Y) = \{F, Y\}, \qquad \mathsf{scc}(H) = \mathsf{scc}(T) = \emptyset$$

Beim ersten Beispiel (aus 2.11) gilt $\mathsf{scc}(X) = \{X\}$. Man wird nun sinnvollerweise fordern, daß konstante Prädikate, also Prädikate X mit $\sigma(X) = \varepsilon$, eine leere SCC haben. Sie sollen ja gerade nicht von sich selbst abhängen. Dies ist ein erstes Kriterium für die Wohlgeformtheit einer Definition für Prädikatsvariablen (siehe Definition 2.31). Ein zweites betrifft die Monotonie, damit sichergestellt ist, daß die entsprechenden Fixpunkte existieren. In den üblichen μ-Kalkülen stellt man das durch eine Forderung der Art

„Im Rumpf von ‚$\mu X \cdots . r$' stehen nur geradzahlig viele Negationen vor einem X"

sicher. Dies wird in der Literatur auch mit „formale Monotonie" von X bezeichnet. In $\mathbb{B}_\mu^V$ kann sich die Abhängigkeit eines Prädikates aber über mehrere Rümpfe erstrecken, so daß statt dessen *Monotonieindikatoren* betrachtet werden. Der Vorteil der Monotonieindikatoren besteht darin, daß sich dadurch auch beliebige Verbände $[w \to \mathbb{D}]$ mit $\mathbb{D}$ vollständiger Verband anstatt den hier betrachteten $\mathbb{P}(w) \simeq [w \to \mathbb{B}]$ für $w \in S^+$ behandeln lassen. Sie „zählen" auf abstraktere Weise, wieviel Negationen vor einer Prädikatsvariable vorkommen. Eine Negation entspricht dabei dem Übergang von monotonem zu antimontonem Verhalten. Durch eine Einteilung in antimonotone und monotone Operationen können so die folgenden Betrachtungen auf allgemeine Verbände übertragen werden.

Bei den üblichen Formulierungen des μ-Kalküls geschieht das Zählen „modulo 2". Es wird einfach nur festgestellt, ob eine gerade oder ungerade Anzahl von Negationen vor einer Prädikatsvariablen steht.[6] An Stelle des Verbandes $\{\text{gerade}, \text{ungerade}\}$ tritt hier

Definition 2.24 (L_4) *Der boolesche Verband L_4 besteht aus den Elementen*

$$\top \quad ,0 \quad ,1 \quad ,\bot$$

[6] Die einzige dem Autor bekannte Arbeit, in der dies auch exakt durchgeführt wurde, ist /Arnold, 1994/.

und sei definiert durch den Isomorphismus ψ mit

$$\psi\colon L_4 \to \mathbb{P}(\{0,1\}), \qquad \psi(\top) := \{0,1\},\ \psi(0) := \{0\},\ \psi(1) := \{1\},\ \psi(\bot) := \{\}$$

vgl. mit Abbildung 2.7.

Die Ordnung auf L_4, die der Teilmengenordnung auf $\mathbb{P}(\{0,1\})$ entspricht, sei mit $\leq$ bezeichnet. Mit L_4 ist auch $[T_\mu^V \to L_4]$ ein Verband, dessen Operationen kanonisch (punktweise) aus L_4 gewonnen werden und auch mit denselben Symbolen bezeichnet werden.

Definition 2.25 *Die Menge $[T_\mu^V \to L_4]$ wird durch punktweise Erweiterung zum vollständigen Verband mit der Ordnung*

$$f \leq g \quad :\Leftrightarrow \quad f(s) \leq g(s) \text{ für alle } s \in T_\mu^V$$

für $f, g \in [T_\mu^V \to L_4]$

Die Familie der Monotonieindikatoren besteht genau aus den Elementen aus diesem Verband.

Definition 2.26 (Familie von Monotonieindikatoren) *Die Familie*

$$h_i^X : T_\mu^V \to L_4$$

sei rekursiv definiert für ein festes $X \in P$, $i \in \mathbb{N}$ durch

$$\begin{aligned} h_{i+1}^X(u[i]) &:= \bot \\ h_{i+1}^X(u \doteq v) &:= \bot \\ h_{i+1}^X(Y(\underline{u})) &:= h_i^X(\beta(Y)), \quad Y \neq X \\ h_{i+1}^X(X(\underline{u})) &:= 0 \end{aligned} \qquad \begin{aligned} h_{i+1}^X(\exists u.\, s) &:= h_{i+1}^X(s) \\ h_{i+1}^X(s \wedge t) &:= h_{i+1}^X(s) \cup h_{i+1}^X(t) \\ h_{i+1}^X(\neg s) &:= \neg h_{i+1}^X(s) \end{aligned}$$

mit dem Basisfall $h_0^X(s) := \bot$ für alle $s \in T_\mu^V$.

Die einzelnen Approximationen h_i^X sind nicht so sehr interessant. Wichtiger ist der Grenzfall h^X, der sich nach folgendem Satz bilden läßt.

Satz 2.27 (Monotonie von h_i^X) $\qquad h_i^X \leq h_{i+1}^X$

Beweis: (Induktion über i) Für den Beweis halte X fest und definiere $h_i := h_i^X$. Im Basisfall ist laut Definition $h_0(s) = \bot \leq h_1(s)$ für alle $s \in \mathbb{B}_\mu^V$. Im Induktionsschritt gelte die Aussage für i und muß für $i+1$ gezeigt werden. Dazu führe man eine innere Induktion über den Termaufbau von $r \in \mathbb{B}_\mu^V$ durch. Die Fälle $r = u[i]$, $r = u \doteq v$ und $r = X(\underline{u})$ sind trivial, da sich beim Übergang von i nach $i+1$ nichts ändert. Der Fall $r = \exists u.\, s$ ergibt sich unmittelbar nach der inneren Induktionshypothese. Ebenso liefert diese für $r = s \wedge t$ die Aussage

$$h_{i+1}(r) = h_{i+1}(s \wedge t) = h_{i+1}(s) \cup h_{i+1}(t) \leq h_{i+2}(s) \cup h_{i+2}(t) = h_{i+2}(s \wedge t) = h_{i+2}(r)$$

Dabei gilt die Ungleichung wegen der Monotonie von ‚$\cup$' in L_4 nach zweimaliger Anwendung der inneren Induktionsvoraussetzung. Für $r = \neg s$ folgt die Aussage analog unter Ausnutzung der Antimonotonie von ‚$\neg$' in L_4. Im interessantesten Fall $r = Y(\underline{u})$ mit $Y \neq X$ hat man

$$h_{i+1}(r) = h_{i+1}(Y(\underline{u})) = h_i(\beta(Y)) \leq h_{i+1}(\beta(Y)) = h_{i+2}(Y(\underline{u})) = h_{i+2}(r)$$

Hier gilt ‚$\leq$' nach äußerer Induktionsvoraussetzung. ❑

Im Verband $[T_\mu^V \to L_4]$ konvergiert deshalb die Folge (h_i^X) zu einer oberen Schranke der h_i^X, die sich als deren Vereinigung berechnen läßt.

Definition 2.28 (Monotonieindikator) *Für $X \in P$ sei*

$$h^X \quad := \quad \bigcup_{i=0}^{\infty} h_i^X$$

der Monotonieindikator für X (bez. einer Prädikatsdefinition δ).

Bei endlichem P konvergiert die Vereinigung schon nach endlich vielen Schritten. Im nichtendlichen Fall hätte man bei einer Wohlordnung von ‚$\to$' auf den SCCs und endlichen SCCs immer noch die endliche Konvergenz. Ist eine dieser beiden Forderungen nicht erfüllt, so lassen sich einfach Beispiele generieren, bei denen die Folge (h_i^X) echt aufsteigend ist. Jedoch ist bei der späteren Verwendung der Monotonieindikatoren nur das Ergebnis nach Anwendungen auf einen bestimmten Term interessant, und für jeden Term $t \in T_\mu^V$ gilt, daß die Folge $(h_i^X(t))$ ab einem Index j konstant wird. So lassen sich die folgenden Sätze und Beweise mit den h_i^X anstatt h_X und beliebigem (nicht endlichem) P formulieren, was aber einen erheblichen technischen Mehraufwand erfordert. Dies ist der Grund, warum in Definition 2.16 die Endlichkeit von P vorausgesetzt wurde. Eine einfache Konsequenz aus der Definition ist

Lemma 2.29 (Monotonie konstanter Prädikate) *Für $X, F \in P$ gilt*

$$F \notin \mathsf{dep}(X) \quad \Rightarrow \quad h^F(\beta(X)) = \perp$$

Beweis: Die äußere Induktion laufe über i und die innere über den Termaufbau von t in der folgenden verschärften Induktionshypothese

$$\text{für } X, F \in P,\ i \in \mathbb{N} \text{ gilt} \quad F \notin \mathsf{dep}(X) \wedge t \leq \beta(X) \quad \Rightarrow \quad h_i^F(t) = \perp$$

Im Induktionsanfang ($i = 0$) ist nichts zu zeigen. Im Induktionsschritt von i nach $i+1$ gilt die Aussage für $t = u[j]$ oder $t = u \doteq v$ trivialerweise. Für $t = \exists u.\, s$, $t = s \wedge t$ und $t = \neg s$ folgt die Aussage nach der inneren Induktionsvoraussetzung, mit

$$\perp \cup \perp = \perp \quad \text{und} \quad \neg \perp = \perp$$

Der Fall $t = F(\underline{u})$ kann nicht auftreten, da sonst $F < \beta(X)$ und somit $F \in \mathsf{dep}(X)$. Es bleibt noch der Fall $t = Y(\underline{u})$. Da $F \notin \mathsf{dep}(Y) \subseteq \mathsf{dep}(X)$, kann nun die äußere Induktionshypothese angewendet werden und man erhält $h_i^F(\beta(Y)) = \perp$, womit nach Definition 2.28 auch $h_{i+1}^F(t) = \perp$. ❑

Als erstes Beispiel soll wiederum Gleichung (2.12) herangezogen werden. Es werden nur Approximationen für die Rümpfe von den „rekursiven" Prädikatsvariablen F und Y angegeben und das auch nur für F als Parameter von h. Zunächst gilt

$$h_i^F(\beta(T)) = h_i^F(\beta(H)) = \perp \qquad \text{für } i \in \mathbb{N} \text{ nach Lemma 2.29}$$

für die weiteren Approximationen erhält man

$$\beta(Y) = \underbrace{\overbrace{H(s)\wedge F(s)}^{\substack{\vdots\\ h_1^F(\cdot)=0\\ h_0^F(\cdot)=\perp}} \vee \exists t.\, \overbrace{T(s,t)\wedge Y(t)}^{\substack{\vdots\\ h_2^F(\cdot)=0\\ h_1^F(\cdot)=\perp\\ h_0^F(\cdot)=\perp}}}_{\substack{h_0^F(\cdot)=\perp\\ h_1^F(\cdot)=0\\ \vdots}} \qquad \beta(F) = \overbrace{\exists t.\, T(s,t)\wedge Y(t)}^{\substack{\vdots\\ h_2^F(\cdot)=0\\ h_1^F(\cdot)=\perp\\ h_0^F(\cdot)=\perp}}$$

Lemma 2.30

$$\begin{aligned}
h^X(\exists u.\, s) &\le a &&\Leftrightarrow\quad h^X(s) \le a\\
h^X(X(\underline{u})) &\le 0 &&\\
h^X(Y(\underline{u})) &\le a &&\Leftrightarrow\quad h^X(\beta(Y)) \le a, \quad X \ne Y\\
h^X(\neg s) &\le a &&\Leftrightarrow\quad h^X(s) \le \neg a\\
h^X(s \wedge t) &\le a &&\Leftrightarrow\quad h^X(s) \le a \text{ und } h^X(t) \le a
\end{aligned}$$

Beweis: Kombination der Definitionen 2.26, 2.28 und 2.24. ❑

Definition 2.31 (Wohlgeformte Prädikatsdefinition)
Eine Prädikatsdefinition δ *heiße* wohlgeformt *gdw. für alle* $X \in P$

$$(\mathsf{scc}(X) = \emptyset \Rightarrow \sigma(X) = \varepsilon) \qquad \textit{und} \qquad h^X(\beta(X)) \le 0$$

Diese letzte Forderung entspricht der üblichen „formalen Monotonie" von X.

Es sollen noch drei weitere (mehr theoretische) Beispiele, betrachtet werden. Das erste hat dieselbe Semantik wie das Beispiel aus Gleichung (2.11) von S. 25:

$$\begin{aligned}
&\nu\, A(u,v,w)\,.\ \neg B(w,u,v)\\
&\mu\, B(u,v,w)\,.\ u \vee \neg A(u,v,w)
\end{aligned}$$

Es gilt nämlich mit X aus Gleichung (2.11) auf S. 25 mit Vorgriff auf Definition 2.33

$$A(u,v,w) \quad\equiv\quad B(u,v,w) \quad\equiv\quad X(u,v,w) \quad\equiv\quad u \vee v \vee w$$

Hier erhält man folgende Approximationen

$$\beta(A) = \underbrace{\neg\, \overbrace{B(w,u,v)}^{\begin{array}{c}\vdots\\ h_1^B(\cdot) = 0\\ h_0^B(\cdot) = \bot\end{array}}}_{\begin{array}{c}h_0^B(\cdot) = \bot\\ h_1^B(\cdot) = 1\\ \vdots\end{array}} \qquad \beta(B) = \underbrace{u \vee \neg\, \overbrace{A(u,v,w)}^{\begin{array}{c}\vdots\\ h_2^B(\cdot) = 1\\ h_1^B(\cdot) = \bot\\ h_0^B(\cdot) = \bot\end{array}}}_{\begin{array}{c}h_0^B(\cdot) = \bot\\ h_1^B(\cdot) = \bot\\ h_2^B(\cdot) = 0\\ \vdots\end{array}}$$

Zwei Beispiele für *nicht* wohlgeformte Definitionen sind

$$\nu Z_1(u).\ Z_1(u) \vee \neg Z_1(u) \qquad\qquad \nu Z_2(u).\ \neg Z_2(u)$$

mit den Approximationen

$$\beta(Z_1) = \underbrace{\overbrace{Z_1(u)}^{\begin{array}{c}\vdots\\ h_2^{Z_1}(\cdot) = \top\\ h_1^{Z_1}(\cdot) = 0\\ h_0^{Z_1}(\cdot) = \bot\end{array}} \vee \overbrace{\neg Z_1(u)}^{\begin{array}{c}\vdots\\ h_2^{Z_1}(\cdot) = \top\\ h_1^{Z_1}(\cdot) = 1\\ h_0^{Z_1}(\cdot) = \bot\end{array}}}_{\begin{array}{c}h_0^{Z_1}(\cdot) = \bot\\ h_1^{Z_1}(\cdot) = \top\\ \vdots\end{array}} \qquad \beta(Z_2) = \underbrace{\neg\, \overbrace{Z_2(u)}^{\begin{array}{c}\vdots\\ h_1^{Z_2}(\cdot) = 0\\ h_0^{Z_2}(\cdot) = \bot\end{array}}}_{\begin{array}{c}h_0^{Z_2}(\cdot) = \bot\\ h_1^{Z_2}(\cdot) = 1\\ \vdots\end{array}}$$

und es ist $h^{Z_1}(\beta(Z_1)) = \top \not\leq 0$ bzw. $h^{Z_2}(\beta(Z_2)) = 1 \not\leq 0$.

Definition 2.32 ($\mathbb{B}_\mu^V$) *Die Menge $\mathbb{B}_\mu^V$ ist die Vereinigung aller Mengen T_μ^V bez. einer Signatur $\mathcal{S}$ und einer wohlgeformten Prädikatsdefinition δ.*

2.6.5 Semantik

Ab sofort sei $\mathcal{S}$ und ein wohlgeformtes δ fest gewählt, so daß mit $\mathbb{B}_\mu^V$ immer das entsprechende T_μ^V gemeint ist. Die Semantik $[\![\cdot]\!]\rho$ von $\mathbb{B}_\mu^V$ kann nun definiert werden als

Definition 2.33 (Semantik von $\mathbb{B}_\mu^V$) *Für eine Belegung* ρ *definiere*

$$[\![\cdot]\!]\rho\colon T_\mu^V \to \mathbb{B}$$

(als partielle Funktion) rekursiv über den Termaufbau durch

$$\begin{aligned}
[\![u[i]]\!]\rho &:= \rho(u)(i)\\
[\![u \doteq v]\!]\rho &:= \textstyle\bigwedge_{0\le i<|u|} [\![u[i] \leftrightarrow v[i]]\!]\rho\\
[\![s\wedge t]\!]\rho &:= [\![s]\!]\rho \wedge [\![t]\!]\rho\\
[\![\neg s]\!]\rho &:= \neg[\![s]\!]\rho
\end{aligned}$$

für die einfachen Terme. Für den Existenzquantor sei

$$[\![\exists u.\, s]\!]\rho := \begin{cases} 1 & \textit{falls es ein } a\in\tau(u) \textit{ gibt, mit } [\![s]\!]\rho',\ \textit{wobei } \rho' := \rho\{u\mapsto a\}\\ 0 & \textit{sonst.}\end{cases} \tag{2.13}$$

Für Anwendungen von Prädikatsvariablen auf einen Vektor von Variablen $\underline{u}$ *definiere*

$$[\![X(\underline{u})]\!]\rho \;:\Leftrightarrow\; \rho(\underline{u}) \in \zeta, \tag{2.14}$$

wobei $\zeta := \rho(X)$, *falls* $\sigma(X)\in\{\mu,\nu\}$ *und* $\rho(X)\!\downarrow$. *Für* $\sigma(X)=\varepsilon$ *definiere*

$$\zeta \;:=\; \{\underline{a}\in\tau(X) \mid [\![\beta(X)]\!]\rho' \textit{ mit } \rho' := \rho\{\pi(X)\mapsto\underline{a}\}\ \}$$

Für $\sigma(X)\in\{\mu,\nu\}$ *und* $\rho(X)\!\uparrow$ *sei*

$$\zeta \;:=\; \bigcap\{\xi\subseteq\tau(X)\mid \Psi_\rho^X(\xi)=\xi\},\quad \textit{für } \sigma(x)=\mu \tag{2.15}$$

$$\zeta \;:=\; \bigcup\{\xi\subseteq\tau(X)\mid \Psi_\rho^X(\xi)=\xi\},\quad \textit{für } \sigma(X)=\nu \tag{2.16}$$

mit dem Expansionsoperator Ψ_ρ^X *aus Definition 2.34.*

Für die Rekursion wird folgender Operator verwendet

Definition 2.34 (Expansionsoperator für die Standardsemantik)
Zu $X\in P$, ρ *Variablenbelegung definiere*

$$\Psi_\rho^X\colon\ \mathbb{P}(\tau(X))\to\mathbb{P}(\tau(X)),\qquad \underline{a}\in\Psi_\rho^X(\xi)\quad :\Leftrightarrow\quad [\![\beta(X)]\!]\rho'$$

mit $\rho' := \rho\{X\mapsto\xi,\ \pi(X)\mapsto\underline{a}\}$ *für* $\xi\subseteq\tau(X)$ *und* $\underline{a}\in\tau(X)$.

Man kann für den Existenzquantor mit der Schreibweise aus Abschnitt 2.5 auch etwas kürzer schreiben

$$[\![\exists u.\, s]\!]\rho \;\equiv\; \bigvee_{a\in\tau(u)} [\![s]\!]\rho\{u\mapsto a\} \tag{2.17}$$

Die Semantik von $\mathbb{B}_\mu^V$ aus Definition 2.33 ist noch keine *totale* Funktion. Es könnte nämlich sein, daß die Fixpunkte in den Gleichungen (2.15) und (2.16) überhaupt nicht existieren.

Definition 2.35 (Wohldefiniertheit einer Semantik)
Eine Semantik heiße wohldefiniert *genau dann, wenn sie eine* totale *Funktion ist.*

Für eine *wohldefinierte* Semantik braucht man folgendes Korollar zu Satz 2.38.

Korollar 2.36 (Monotonie des Expansionsoperators) *In 2.36 gilt für $\rho(X)\uparrow$*

$$\Psi_\rho^X(\xi)\subseteq\Psi_\rho^X(\xi'), \qquad \textit{für } \xi\subseteq\xi'\subseteq\tau(X)$$

und somit $[\![t]\!]\rho\downarrow$ für alle $t\in T_\mu^V$.

Beweis: Siehe Gleichung (2.20) in Satz 2.38. ❑

Dies ergibt den wesentlichen Satz dieses Abschnittes:

Satz 2.37 *Die Standardsemantik ist wohldefiniert.*

Damit ergibt sich aus Gleichung (2.15) mit Satz A.4 für eine Variablenbelegung ρ und eine Prädikatsvariable X mit $\sigma(X)=\mu$ (bei endlichem P und endlichen Typen)

$$\begin{aligned}\bigcap\{\xi\in\tau(X)\mid\psi_\rho^X(\xi)=\xi\} &= \bigcap\{\xi\in\tau(X)\mid\psi_\rho^X(\xi)\subseteq\xi\}\\ &= \textstyle\bigcup_{i=0}^{\infty}(\psi_\rho^X)^i(\emptyset)\\ &= (\psi_\rho^X)^n(\emptyset) \text{ mit } (\psi_\rho^X)^n(\emptyset)\geq(\psi_\rho^X)^{n+1}(\emptyset)\end{aligned}\tag{2.18}$$

Hierbei verwendet man wie üblich $(\Psi_\rho^X)^0 := \mathrm{id}$. Die letzte Version ist einem rekursiven Algorithmus schon sehr nahe.

Wie auf Seite 32 festgelegt, wurde ja immer eine wohlgeformte Definition von Prädikatsvariablen vorausgesetzt. Dies ist auch notwendig, wie die Auswertung von

$$\nu Z_2(u).\ \neg Z_2(u)$$

als Beispiel einer *nicht* wohlgeformten (vgl. Seite 31) Definition zeigt:

$$
\begin{array}{rcl}
\xi_0 & := & \emptyset \\
a \in \xi_1 & :\Leftrightarrow & [\![\beta(Z_2)]\!]\rho\{Z_2 \mapsto \xi_0, u \mapsto a\} \Leftrightarrow [\![\neg Z_2(u)]\!]\rho\{Z_2 \mapsto \emptyset, u \mapsto a\} \\
& & \Leftrightarrow \neg[\![Z_2(u)]\!]\rho\{Z_2 \mapsto \emptyset, u \mapsto a\} \\
& & \Leftrightarrow \neg(a \in \emptyset) \\
& & \Leftrightarrow 1 \\
a \in \xi_2 & :\Leftrightarrow & [\![\beta(Z_2)]\!]\rho\{Z_2 \mapsto \xi_1, u \mapsto a\} \Leftrightarrow [\![\neg Z_2(u)]\!]\rho\{Z_2 \mapsto \mathbb{B}, u \mapsto a\} \\
& & \Leftrightarrow \neg[\![Z_2(u)]\!]\rho\{Z_2 \mapsto \mathbb{B}, u \mapsto a\} \\
& & \Leftrightarrow \neg(a \in \mathbb{B}) \\
& & \Leftrightarrow 0 \\
& \vdots &
\end{array}
$$

Somit hat man also

$$(\Psi^X_\rho)^n(\emptyset) = \begin{cases} \emptyset & n \text{ gerade} \\ \mathbb{B} & n \text{ ungerade} \end{cases} \qquad (\Psi^X_\rho)^{n+1}(\emptyset) \neq (\Psi^X_\rho)^n(\emptyset) \qquad n \in \mathbb{N}$$

und $[\![Z_2(v)]\!]\rho$ ist nicht definiert für alle Variablenbelegungen ρ.

Unter der Generalvoraussetzung der Verwendung einer wohlgeformten Definition von Prädikatsvariablen zeigt der nächste Satz, daß der Expansionsoperator monoton ist, und deshalb die Semantik wohldefiniert ist. Die Aussage des Satzes stellt eine Verschärfung der Monotonieaussage dar, die benötigt wird, um die Induktion durchzuführen. So wird in Teil a. die Wohldefiniertheit der Semantik an sich gezeigt. Teil b. und c. verbinden Anforderungen an die Monotononieindikatoren (Monotonie und Antimonotonie) mit der „Monotonie der Semantik“ bez. zweier Belegungen, die sich nur durch ihren Wert für eine Prädikatsvariable X unterscheiden. Dabei stehen die Werte von X unter beiden Belegungen in einer Inklusionsbeziehung.

Satz 2.38 *Für $t \in T^V_\mu$, $X \in P$, Variablenbelegungen ρ, ρ', mit $\rho(x)\downarrow \Leftrightarrow \rho'(x)\downarrow$,*

$$\rho(x)\downarrow \Rightarrow \rho(x) = \rho'(x) \ \textit{für } x \in V \cup P,\ x \neq X \qquad \textit{und} \qquad \rho(X) \subseteq \rho'(X)$$

a. $[\![t]\!]\rho\downarrow$ *und* $[\![t]\!]\rho'\downarrow$

b. *Falls $h^X(t) \leq 0$, so gilt $[\![t]\!]\rho \Rightarrow [\![t]\!]\rho'$*

c. *Falls $h^X(t) \leq 1$, so gilt $[\![t]\!]\rho \Leftarrow [\![t]\!]\rho'$*

Beweis: Der Beweis besteht aus einer zweistufigen Induktion mit äußerem Parameter

$$k(t,\rho) := |\mathsf{dep}(t) \backslash \underbrace{\mathsf{domain}(\rho)}_{=\mathsf{domain}(\rho')}| = k(t,\rho'), \tag{2.19}$$

der die Anzahl Prädikatsvariable angibt, von denen t „abhängt“, und für die ρ (bzw. ρ') undefiniert ist. Der innere Induktionsparameter ist wie üblich der Termaufbau. Für $t = u[i]$, $t = u \doteq v$

folgen die Aussagen a.-c. unmittelbar. Für $t = s \wedge r$ und $t = \neg s$ verwende man die innere Induktionsvoraussetzung und Lemma 2.30. Nun sei $t = \exists u.s$. Aussage a. folgt unmittelbar nach innerer Induktionsvoraussetzung. Sei nun $h^X(t) \leq 0$, so ist nach Lemma 2.30 auch $h^X(s) \leq 0$ und nach innerer Induktionsvoraussetzung ist $[\![s]\!]\rho\{u \mapsto a\}$ definiert für alle $a \in \tau(u)$ und ebenso $[\![s]\!]\rho'\{u \mapsto a\}$. Um nun b. zu zeigen gelte $[\![s]\!]\rho\{u \mapsto a\}$. Wiederum mit der inneren Induktionsvoraussetzung (Teil b.) folgt dann $[\![s]\!]\rho'\{u \mapsto a\}$. Mit der Definition der Semantik des Existenzquantors ergibt sich der Rest. Analog für c.

Der interessanteste Fall ist $t = Y(\underline{u})$. Nun sei $\sigma(Y) \in \{\mu, \nu\}$ und zunächst $\rho(Y)\downarrow$, womit auch $[\![t]\!]\downarrow$, und es folgt a. Für den Beweis von b. sei $h^X(t) \leq 0$. Sowohl für $X = Y$ als auch $X \neq Y$ gilt $\rho(Y) \subseteq \rho'(Y)$, so daß mit $\rho(\underline{u}) = \rho'(\underline{u})$

$$[\![Y(\underline{u})]\!]\rho \;\Leftrightarrow\; \rho(\underline{u}) \in \rho(Y) \;\Rightarrow\; \rho'(\underline{u}) \in \rho'(Y) \;\Leftrightarrow\; [\![Y(\underline{u})]\!]\rho'$$

Für $h^X(t) = 1$ ist $Y \neq X$, da δ wohlgeformt ist. Somit ist sogar $\rho(Y) = \rho'(Y)$ und es gilt oben „$\Leftrightarrow$“ statt nur „$\Rightarrow$“, womit c. folgt.

Für $\sigma(X) \neq \varepsilon$ fehlt noch der „rekursive“ Fall $\rho(Y)\uparrow$. Zunächst soll gezeigt werden, daß die Fixpunkte aus den Gleichungen (2.15) und (2.16) existieren. Dazu genügt es nach Satz A.15 die Monotonie von Ψ_ρ^Y zu zeigen. Sei nun $\underline{a} \in \tau(X)$, $\xi \subseteq \xi' \subseteq \tau(X)$ so gilt

$$\begin{aligned} \underline{a} \in \Psi_\rho^Y(\xi) \quad &\Leftrightarrow \quad [\![\beta(Y)]\!]\rho\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \\ &\Rightarrow \quad [\![\beta(Y)]\!]\rho\{Y \mapsto \xi', \pi(Y) \mapsto \underline{a}\} \\ &\Leftrightarrow \quad \underline{a} \in \Psi_\rho^Y(\xi') \end{aligned} \tag{2.20}$$

„$\Rightarrow$“ folgt nach äußerer Induktionsvoraussetzung, da

$$k(\beta(Y), \rho\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\}) \;<\; k(\beta(Y), \rho),$$

$h^Y(\beta(Y)) \leq 0$ nach der Wohlgeformtheit von δ, und

$$\rho\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \quad \text{bzw.} \quad \rho\{Y \mapsto \xi', \pi(Y) \mapsto \underline{a}\}$$

die Voraussetzung des Lemmas erfüllen. Man beachte, daß auch hier die Induktionsvoraussetzung Teil a. eingeht, um $[\![\beta(Y)]\!]\rho\{\cdots\}\downarrow$ zu sichern. Insgesamt ist damit der Induktionsschritt für Teil a. vollzogen.

Für Teil b. sei $\sigma(Y) = \mu$ und $h^X(\beta(Y)) \leq 0$. Ist $\rho(X)\uparrow$, so gilt $\rho = \rho'$ und somit folgt für alle $\underline{a} \in \tau(Y)$, $\xi \subseteq \tau(Y)$ unmittelbar

$$\psi_\rho^Y(\xi) \;\subseteq\; \psi_{\rho'}^Y(\xi) \tag{2.21}$$

ansonsten ist $\rho(X)\downarrow$, $\rho'(X)\downarrow$, $\rho(X) \subseteq \rho'(X)$ und es gilt ($X \neq Y$)

$$\begin{aligned} \underline{a} \in \Psi_\rho^Y(\xi) \quad &\Leftrightarrow \quad [\![\beta(Y)]\!]\rho\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \quad \Leftrightarrow \quad [\![\beta(Y)]\!]\rho''\{X \mapsto \rho(X)\} \\ &\Rightarrow \quad [\![\beta(Y)]\!]\rho''\{X \mapsto \rho'(X)\} \qquad\quad\;\; \Leftrightarrow \quad [\![\beta(Y)]\!]\rho'\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \\ &\Leftrightarrow \quad \underline{a} \in \Psi_{\rho'}^Y(\xi) \end{aligned}$$

mit

$$\begin{aligned} \rho'' \;&:=\; (\rho \backslash \{X \mapsto \rho(X)\})\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \\ &=\; (\rho' \backslash \{X \mapsto \rho'(X)\})\{Y \mapsto \xi, \pi(Y) \mapsto \underline{a}\} \end{aligned}$$

Wiederum folgt „$\Rightarrow$“ nach äußerer Induktionsvoraussetzung, da

$$Y \notin \mathsf{domain}(\rho) \subsetneq \mathsf{domain}(\rho''\{X \mapsto \rho(X)\}) \ni Y$$

Analog zum Beweis der Monotonie von Ψ^X_ρ erhält man auch die Monotonie von $\psi^X_{\rho'}$. Jetzt gilt nach Satz A.20

$$\zeta := \bigcup_{i=0}^{\infty} (\psi^Y_\rho)^i(\emptyset) \subseteq \bigcup_{i=0}^{\infty} (\psi^Y_{\rho'})^i(\emptyset) =: \zeta'$$

und somit (ähnlich wie in der Analysis durch „Vertauschen der Limites“)

$$[\![Y(\underline{u})]\!]\rho \Leftrightarrow \rho(\underline{u}) \in \zeta \Leftrightarrow \rho'(\underline{u}) \in \zeta \Rightarrow \rho'(\underline{u}) \in \zeta' \Leftrightarrow [\![Y(\underline{u})]\!]\rho'$$

Für $\sigma(Y) = \nu$ oder $h^X(t) \leq 1$ (Teil c.) verfahre man analog. Bei konstantem Y (d. h. $\sigma(Y) = \varepsilon$) gilt mit der Wohlgeformtheit von δ, daß

$$Y \in \mathsf{dep}(Y(\underline{u})), \quad Y \notin \mathsf{dep}(\beta(Y))$$

und somit $k(t, \rho) > k(\beta(Y), \rho\{\pi(Y) \mapsto \rho(\underline{u})\})$. Der Rest folgt analog zu $\sigma(Y) \in \{\mu, \nu\}$. ❑

Als Illustration für die Berechnung der Semantik soll die Verifikation des Beispiels von Seite 18 und Abbildung 2.3 durchgeführt werden. Bei der Berechnung der Semantik der Anwendung der konstanten Prädikatsvariablen T, S und E treten in Gleichung (2.14) aus Definition 2.33 für ζ die folgenden Mengen auf

$$\begin{aligned}
\zeta_T &:= \{(00,00),(00,01),(01,00),(01,01),(00,10),(10,00),(10,10)\} \\
\zeta_S &:= \{00\} \\
\zeta_E &:= \{11\},
\end{aligned}$$

wobei hier für zweistellige boolesche Vektoren einfach Wörter über $\{0,1\}$ der Länge zwei verwendet werden. Bei der Berechnung der Semantik des Quantors aus Gleichung (2.10) auf S. 19 muß ein Fixpunkt ζ_R berechnet werden. Für eine Variablenbelegung mit $\rho(R)\uparrow$ definiere für $i \in \mathbb{N}$

$$\zeta^0 := \emptyset \qquad \zeta^{i+1}_R := (\Psi^R_\rho)^{i+1}(\emptyset) = \Psi^R_\rho(\zeta^i_R)$$

Dies sind nach (2.18) von S. 33 die Approximationen an ζ_R und man erhält (vgl. S. 17)

$$\zeta^0_R = \emptyset, \quad \zeta^1_R = \zeta_S = \{00\}, \quad \zeta_R = \zeta^2_R = \zeta^3_R = \ldots = \{00,01,10\}$$

Nun gilt $\overline{\zeta_R} \cup \overline{\zeta_E} = \mathbb{B}^2$, so daß bei der Berechnung des Allquantors in Gleichung (2.10) auf Seite 19 entweder $\rho(s) \notin \zeta_R$ oder $\rho(s) \notin \zeta_E$ gilt, und so wird bei jeder Wahl von a entsprechend Gleichung (2.13) auf Seite 32 in Definition 2.33 der Rumpf zu wahr evaluiert.

2.7 Eine zweite Semantik für $\mathbb{B}_\mu^V$

Als nächstes wird eine *zweite* Semantik für den μ-Kalkül angegeben. Diese evaluiert einen μ-Kalkül Term zu einem booleschen Ausdruck (vgl. Abb. 2.2 auf Seite 9). Dabei werden die bei der Berechnung der Fixpunkte auftretenden Mengen nicht explizit, sondern durch einen booleschen Ausdruck repräsentiert. Ebenso wird die Semantik eines Quantors nicht durch Abändern einer Belegung der Individuenvariablen definiert, sondern konstruktiv durch Angabe eines booleschen Ausdrucks. Dies bildet die Grundlage des in /Burch et al., 1990, Burch et al., 1992, McMillan, 1993a, Burch et al., 1994/ geprägten Begriffs „Symbolische Modellprüfung“ (auch „symbolic model checking“). Es gab aber noch keine Versuche, diese zweite Semantik formal zu definieren, um dann zu zeigen, daß sie mit der ersten übereinstimmt. Dies ist Aufgabe dieses Abschnittes.

Der Hauptgrund für diese Übersetzung ist die dadurch gegebene Möglichkeit, über den genannten Umweg die Auswertung von Prädikaten mittels BDDs zu realisieren, die eine kompakte Repräsentation von booleschen Ausdrücken darstellen. Erst dadurch kann man hoffen, auch große Zustandssysteme der Modellprüfung zugänglich zu machen.

Für diese Übersetzung muß zunächst ein Zusammenhang zwischen den Variablen V von $\mathbb{B}_\mu^V$, die *Vektoren* von booleschen Werten denotieren, und den Variablen W von $\mathbb{B}^W$, die nur für einfache boolesche Werte stehen, hergestellt werden.

Definition 2.39 (Komponentenabbildung)
Eine Komponentenabbildung *ist eine bijektive, partielle Funktion*

$$\cdot[\cdot] : V \times \mathbb{N} \to W$$

mit $u[i]\!\downarrow$ *gdw.* $0 \le i < |u|$ *für* $u \in V$, $i \in \mathbb{N}$.

Wie W und V sei eine Komponentenabbildung für den Rest der Arbeit fest gewählt. Damit rechtfertigt sich auch die Überladung des Symbols „$\cdot[\cdot]$“ einmal für den (syntaktischen) Term $u[i]$ und zweitens für das Ergebnis einer Komponentenabbildung. Als Abkürzung führe man folgende Schreibweise in $\mathbb{B}^W$ ein:

Definition 2.40 *Für* $u, v \in V$, $a \in \tau(u) = \tau(v)$ *definiere*

$$u \doteqdot v \;:=\; \bigwedge_{i=0}^{|u|-1} u[i] \leftrightarrow v[i] \qquad u \doteqdot a \;:=\; \bigwedge_{i=0}^{|u|-1} u[i] \leftrightarrow a(i)$$

und ebenso für $\underline{u}, \underline{v} \in V^+$ *und* $\underline{a} \in \tau(\underline{u}) = \tau(\underline{v})$ *sei*

$$\underline{u} \doteqdot \underline{v} \;:=\; \bigwedge_{i=1}^{|\underline{u}|} \underline{u}(i) \doteqdot \underline{v}(i) \qquad \underline{u} \doteqdot \underline{a} \;:=\; \bigwedge_{i=1}^{|\underline{u}|} \underline{u}(i) \doteqdot \underline{a}(i)$$

unter Verwendung der ersten beiden Definitionen.

Weiter benötigt man auch hier eine Belegung der Prädikatsvariablen. Die Individuenvariablen werden einfach durch Komponentenabbildung übersetzt, brauchen also keine zusätzliche Belegung. Dies ist auch der Hauptunterschied zwischen der ersten Semantik und der hier vorgestellten. Es wird hier eine symbolische Repräsentation der μ-Kalkülterme bestimmt!

Definition 2.41 (Belegung für Semantik II von $\mathbb{B}^V_\mu$)
Eine Belegung ρ_P *ist eine partielle Funktion*

$$\rho_P\colon P \to \mathbb{B}^W$$

Das Fehlen von Belegungen für Individuenvariablen bedeutet für die zweite Semantik, daß die Quantoren völlig anders behandelt werden müssen.

Definition 2.42 (Semantik II für $\mathbb{B}^V_\mu$) *Die partielle Abbildung*

$$\langle\!\langle\cdot\rangle\!\rangle\rho_P\colon T^V_\mu \to \mathbb{B}^W$$

für eine Belegung ρ_P wird rekursiv definiert durch

$$\begin{aligned} \langle\!\langle u[i]\rangle\!\rangle\rho_P &:= u[i], & \langle\!\langle u \doteq v\rangle\!\rangle\rho_P &:= u \doteq v \\ \langle\!\langle \neg s\rangle\!\rangle\rho_P &:= \neg\langle\!\langle s\rangle\!\rangle\rho_P, & \langle\!\langle s \wedge t\rangle\!\rangle\rho_P &:= \langle\!\langle s\rangle\!\rangle\rho_P \wedge \langle\!\langle t\rangle\!\rangle\rho_P \end{aligned}$$

Nun betrachte den Existenzquantor

$$\langle\!\langle \exists u.\, s\rangle\!\rangle\rho_P := \bigvee_{a\in\tau(u)} (\langle\!\langle s\rangle\!\rangle\rho_P)\{u \mapsto a\} \equiv \exists\, u[0],\ldots,u[|u|-1]\,.\,\langle\!\langle s\rangle\!\rangle\rho_P$$

Für Anwendungen von Prädikatsvariablen definiere

$$\langle\!\langle X(\underline{u})\rangle\!\rangle\rho := s\{\pi(X) \mapsto \underline{u}\},$$

wobei $s := \langle\!\langle\beta(X)\rangle\!\rangle\rho_P$ für $\sigma(X) = \varepsilon$ und $s := \rho_P(X)$ für $\sigma(X) \in \{\nu,\mu\}$ und $\rho_P(X){\downarrow}$. Im Falle $\sigma(X) \in \{\nu,\mu\}$ und $\rho_P(X){\uparrow}$ definiere

$$\begin{aligned} s &:= (\Phi^X_{\rho_P})^n(0),\ n := \min\{i \in \mathbb{N} \mid (\Phi^X_{\rho_P})^{i+1}(0) \equiv (\Phi^X_{\rho_P})^i(0)\} \quad \textit{für } \sigma(X) = \mu \\ s &:= (\Phi^X_{\rho_P})^n(1),\ n := \min\{i \in \mathbb{N} \mid (\Phi^X_{\rho_P})^{i+1}(1) \equiv (\Phi^X_{\rho_P})^i(1)\} \quad \textit{für } \sigma(X) = \nu \end{aligned}$$

und $\Phi^X_{\rho_P}$ aus Definition 2.43.

Man beachte, daß bei der Minimumsbestimmung zwei boolesche Ausdrücke auf ihre Äquivalenz ($\equiv$) getestet werden müssen, was ein NP-vollständiges Problem darstellt.

Definition 2.43 (Expansionoperator für Semantik II)

$$\Phi^X_{\rho_P}\colon \mathbb{B}^W \to \mathbb{B}^W, \qquad \Phi^X_{\rho_P}(s) := \langle\!\langle\beta(X)\rangle\!\rangle\rho_P\{X \mapsto s\}, \quad \textit{für } s \in \mathbb{B}^W$$

Später wird $\mathbb{B}^W$ durch BDD repräsentiert, so daß hierdurch auch ein indirekter Weg zur Verwendung von BDDs zur Evaluation der Semantik von $\mathbb{B}^V_\mu$ geebnet ist. Diese Evaluation ist darüber hinaus genau die Aufgabe der Modellprüfung im μ-Kalkül, so daß hiermit ein wichtiger Baustein zur formalen Behandlung der Modellprüfung im μ-Kalkül mit BDDs gelegt ist.

Der „Umweg“ über $\mathbb{B}^W$ erlaubt auch die Verwendung anderer Repräsentationen für $\mathbb{B}^W$ als BDDs für die Modellprüfung. Auch im Modellprüfer μcke ist diese Trennung zwischen dem eigentlichen Fixpunktevaluator und der BDD-Bibliothek durch die Zwischenschicht der booleschen Ausdrücke realisiert (vgl. Abschnitt 5.2).

Wie bei der ersten Semantik ist man sich hier nicht sicher, daß bei einer wohlgeformten Prädikatsvariablendefinition, die Semantik für alle Terme definiert ist. Auch hier ist es möglich, daß die Fixpunkte nicht existieren. Dies drückt sich darin aus, daß eine der Mengen in 2.42, über die ein Minimum bestimmt wird, leer ist. Dann gäbe es kein i, mit $\Phi^{i+1}(\cdot) \equiv \Phi^i(\cdot)$ (hier steht kein „=“!). Um zu zeigen, daß dieser Fall nie eintreten kann, könnte man wie bei der ersten Semantik verfahren. Da aber auch gezeigt werden soll, daß die beiden Semantiken das „Gleiche“ berechnen, wird gleich diese Aussage gezeigt, bei der die Wohldefiniertheit der zweiten Semantik als Nebenresultat abfällt.

Für den Beweis der Äquivalenz der beiden Semantiken müssen die beiden unterschiedlichen Definitionen von „Belegungen“ miteinander in Beziehung gebrachtet werden.

Definition 2.44 (Zuordnung von Belegungen) *Zu einer Variablenbelegung*

$$\rho\colon V \cup P \to \left(\bigcup_{u\in V} \tau(u)\right) \dot{\cup} \left(\bigcup_{X\in P} \mathbb{P}(\tau(X))\right)$$

bez. der ersten Semantik definiere

$$\rho_W : W \to \mathbb{B}, \qquad \rho_W(x) := \rho(u)(i), \quad \textit{für } x \in W,\ u[i] = x,\ u \in V,\ i \in \mathbb{N},\ 0 \le i < |u|$$

als Belegung für einen booleschen Ausdruck aus $\mathbb{B}^W$ *und mit*

$$\mathsf{Exp}_X^{-1} : \mathbb{P}(\tau(X)) \to \mathbb{B}^W \qquad \mathsf{Exp}_X^{-1}(\zeta) := \bigvee_{\underline{a}\in\zeta} \pi(X) \doteq \underline{a}$$

der (inversen) Expansion einer Menge zu einer Prädikatsvariablen $X \in P$ *definiere*

$$\rho_P : P \to \mathbb{B}^W \qquad \rho_P(X) := \mathsf{Exp}_X^{-1}(\rho(X)) \tag{2.22}$$

für $\pi(X) = \underline{u}$, $\rho(X)\downarrow$ *und* $\rho_P(X)\downarrow :\Leftrightarrow \rho(X)\downarrow$.

Das u und das i in der Definition von ρ_W sind eindeutig bestimmt, da die fest gewählte Komponentenabbildung bijektiv ist. Damit ist auch ρ_W eindeutig definiert. Dasselbe gilt trivialerweise für ρ_P. Hier wurde auch für ρ_P eine Symbolüberladung mit dem Bestandteil von ρ selbst vorgenommen (vgl. 2.6.3). Dies ist aber kein Problem, da ρ_P in beiden Fällen ähnliche Bedeutung aber andere Bildmenge hat, so daß immer nur eine Version in einem Kontext verwendet werden kann.

Satz 2.45 (Äquivalenz der Standardsemantik und Semantik II)

$$[\![\langle\!\langle t\rangle\!\rangle\rho_P]\!]\rho_W \;=\; [\![t]\!]\rho$$

für $t \in T_\mu^V$, *Variablenbelegung* ρ *für* $\mathbb{B}_\mu^V$.

Beweis: Hier werden dieselben Induktionsparameter verwendet wie im Beweis von Satz 2.38 (vgl. Gleichung (2.19)). Weiterhin soll nur der Existenzquantor und die Anwendung eines Prädikates behandelt werden. Die Aussage für die anderen Arten von Termen folgen unmittelbar aus den Definitionen.

Insbesondere muß sichergestellt werden, daß die linke Seite der Aussage des Satzes überhaupt definiert ist. Zunächst sei $t = \exists u.\, s$. Dann gilt

$$\begin{array}{rcll}
[\![\langle\!\langle \exists u.\, s\rangle\!\rangle \rho_P]\!]\rho_W & = & [\![\bigvee_{a\in\tau(u)}(\langle\!\langle s\rangle\!\rangle\rho_P)\{u\mapsto a\}]\!]\rho_W & \\
& = & \bigvee_{a\in\tau(u)}[\![(\langle\!\langle s\rangle\!\rangle\rho_P)\{u\mapsto a\}]\!]\rho_W & \\
& = & \bigvee_{a\in\tau(u)}[\![\langle\!\langle s\rangle\!\rangle\rho_P]\!]\rho_W\{u\mapsto a\} & 2.11 \\
& = & \bigvee_{a\in\tau(u)}[\![\langle\!\langle s\rangle\!\rangle\rho'_P]\!]\rho'_W & \rho' := \rho\{u\mapsto a\} \\
& = & \bigvee_{a\in\tau(u)}[\![s]\!]\rho' & \text{Induktionshypothese} \\
& = & [\![\exists u.\, s]\!]\rho & \text{vgl. (2.17)}
\end{array}$$

Der schwierige Fall ist wiederum $t = X(\underline{u})$. Zunächst sei $\sigma(X) = \varepsilon$.

$$\begin{array}{rcll}
[\![\langle\!\langle X(\underline{u})\rangle\!\rangle\rho_P]\!]\rho_W & = & [\![\langle\!\langle \beta(X)\rangle\!\rangle\rho_P\{\pi(X)\mapsto\underline{u}\}]\!]\rho_W & \\
& = & [\![\langle\!\langle \beta(X)\rangle\!\rangle\rho_P\{\pi(X)\mapsto\underline{u}\}]\!]\rho_W\{\underline{u}\mapsto\rho(\underline{u})\} & \\
& = & [\![(\langle\!\langle \beta(X)\rangle\!\rangle\rho_P\{\pi(X)\mapsto\underline{u}\})\{\underline{u}\mapsto\rho(\underline{u})\}]\!]\rho_W & 2.11 \\
& = & [\![(\langle\!\langle \beta(X)\rangle\!\rangle\rho_P\{\pi(X)\mapsto\rho(\underline{u})\})\{\underline{u}\mapsto\rho(\underline{u})\}]\!]\rho_W & \\
& = & [\![\langle\!\langle \beta(X)\rangle\!\rangle\rho_P]\!]\rho_W\{\pi(X)\mapsto\rho(\underline{u})\} & 2.11 \\
& = & \cdots &
\end{array}$$

mit $\rho' := \rho\{\pi(X)\mapsto\rho(\underline{u})\}$ und nach äußerer Induktionshypothese gilt weiter

$$\cdots \;=\; [\![\langle\!\langle \beta(X)\rangle\!\rangle\rho'_P]\!]\rho'_W \;=\; [\![\beta(X)]\!]\rho' \;=\; [\![\beta(X)]\!]\rho\{\pi(X)\mapsto\rho(\underline{u})\} \;=\; [\![X(\underline{u})]\!]\rho$$

Nun gelte $\sigma(X) = \mu$ und im ersten Fall sei $\rho(X)\!\downarrow$. Dann gilt auch $\rho_P(X)\!\downarrow$ und ist definiert wie in Gleichung (2.22).

$$\begin{array}{rcl}
[\![\langle\!\langle X(\underline{u})\rangle\!\rangle\rho_P]\!]\rho_W & = & [\![\rho_P(X)\{\pi(X)\mapsto\underline{u}\}]\!]\rho_W \\
& = & [\![(\bigvee_{\underline{a}\in\rho(X)}\pi(X)\doteqdot\underline{a})\{\pi(X)\mapsto\underline{u}\}]\!]\rho_W \\
& = & [\![\bigvee_{\underline{a}\in\rho(X)}\underline{u}\doteqdot\underline{a}]\!]\rho_W \\
& = & \bigvee_{\underline{a}\in\rho(X)}[\![\underline{u}\doteqdot\underline{a}]\!]\rho_W \\
& = & \cdots
\end{array}$$

Dies läßt sich auch in $\mathbb{B}$ ausdrücken als

$$\cdots \;=\; \bigvee_{\underline{a}\in\rho(X)}\rho(\underline{u})\doteqdot\underline{a} \;=\; \begin{cases} 1 & \text{falls } \rho(\underline{u})\in\rho(X) \\ 0 & \text{sonst}\end{cases} \;=\; [\![X(\underline{u})]\!]\rho$$

Im zweiten Fall gelte $\sigma(X) = \mu$ und $\rho(X)\!\uparrow$ und es soll folgende Hilfsbehauptung bewiesen werden

$$\mathsf{Exp}_X^{-1}(\Psi_\rho^X(\zeta)) \equiv \Phi_{\rho_P}^X(\mathsf{Exp}_X^{-1}(\zeta)) \tag{2.23}$$

Dazu sei γ eine beliebige Variablenbelegung (bez. $\mathbb{B}^W$)

$$\begin{array}{rcl}
[\![\mathsf{Exp}_X^{-1}(\Psi^X_\rho(\zeta))]\!]\gamma & \Leftrightarrow & [\![\bigvee_{\underline{a}\in\Psi^X_\rho(\zeta)} \pi(X) \doteq \underline{a}]\!]\gamma \\
& \Leftrightarrow & \bigvee_{\underline{a}\in\Psi^X_\rho(\zeta)} [\![\pi(X) \doteq \underline{a}]\!]\gamma \\
& \Leftrightarrow & \bigvee_{\underline{a}\in\Psi^X_\rho(\zeta)} \gamma(\pi(X)) \doteq \underline{a} \\
& \Leftrightarrow & \gamma(\pi(X)) \in \Psi^X_\rho(\zeta) \\
& \Leftrightarrow & [\![\beta(X)]\!]\rho\{X \mapsto \zeta, \pi(X) \to \gamma(\pi(X))\} \\
& \Leftrightarrow & [\![\langle\!\langle\beta(X)\rangle\!\rangle\rho_P\{X \mapsto \mathsf{Exp}_X^{-1}(\zeta)\}]\!]\rho_W\{\pi(X) \to \gamma(\pi(X))\} \\
& \Leftrightarrow & \cdots
\end{array}$$

Dies folgt nach äußerer Induktionshypothese.

$$\cdots \;\Leftrightarrow\; [\![\Phi^X_{\rho_P}(\mathsf{Exp}_X^{-1}(\zeta))]\!]\rho_W\{\pi(X) \to \gamma(\pi(X))\} \;\Leftrightarrow\; \underbrace{[\![\Phi^X_{\rho_P}(\mathsf{Exp}_X^{-1}(\zeta))]\!]}_{\mathsf{free}(\cdot)\subseteq\pi(X)}\gamma$$

Letzteres nach Satz 2.10. Nun zeige durch Induktion über $n \in \mathbb{N}$

$$\mathsf{Exp}_X^{-1}((\Psi^X_\rho)^n(\emptyset)) \equiv (\Phi^X_{\rho_P})^n(0)$$

Im Basisfall erhält man

$$\mathsf{Exp}_X^{-1}((\Psi^X_\rho)^0(\emptyset)) = \mathsf{Exp}_X^{-1}(\emptyset) = (\bigvee_{\underline{a}\in\emptyset} \pi(X) \doteq \underline{a}) = 0 = (\Phi^X_{\rho_P})^0(0)$$

Für den Induktionsschritt sei $\zeta := (\Psi^X_\rho)^n(\emptyset)$ und es gilt

$$\begin{array}{rcll}
\mathsf{Exp}_X^{-1}((\Psi^X_\rho)^{n+1}(\emptyset)) & = & \mathsf{Exp}_X^{-1}(\Psi^X_\rho(\zeta)) & \\
& \equiv & \Phi^X_{\rho_P}(\mathsf{Exp}_X^{-1}(\zeta)) & (2.23) \\
& \equiv & \Phi^X_{\rho_P}((\Phi^X_{\rho_P})^n(0)) & \text{Induktionshypothese} \\
& = & (\Phi^X_{\rho_P})^{n+1}(0) &
\end{array}$$

Mit Gleichung (2.18) auf Seite 33 und Satz 2.38 (Wohldefiniertheit der Standardsemantik) erhält man ein $n \in \mathbb{N}$ mit

$$s := (\Phi^X_{\rho_P})^n(0) \equiv (\Phi^X_{\rho_P})^{n+1}(0) \quad \text{und} \quad \zeta := \bigcap\{\xi \mid \Psi^X_\rho(\xi) = \xi\},$$

so daß

$$\mathsf{Exp}_X^{-1}(\zeta) \equiv s$$

Zusammen mit einer ähnlichen Argumentation wie für $\rho(X)\!\downarrow$ zur Behandlung der Argumente folgt der Rest. ❑

Entsprechend Satz 2.37 erhält man

Korollar 2.46 *Die Semantik II ist wohldefiniert.*

2.8 Zusammenfassung

Der hier vorgestellte μ-Kalkül ist ganz auf die Bedürfnisse dieser Arbeit zugeschnitten. Einmal vereinfacht er den formalen Übergang von Standardsemantik zur BDD-Semantik. Zweitens liefert er für das Namenskonzept der Eingabesprache der μcke eine fundierte Semantik. Schließlich wurden Vektoren integriert, damit in Kapitel 4 über Allokationen für Variablen des μ-Kalküls gesprochen werden kann.

Die zweite Semantik des μ-Kalküls stellt erstmals eine *formale* Definition des Begriffes der „symbolischen Modellprüfung" dar, welcher ja die Grundlage für die Erfolge der Modellprüfung bildet.

2.9 Ausblick

Für Anwendungen, wie die Verifikation von Leistungsmaßen oder metrischer Zeit, wäre es sehr wünschenswert auch unendliche Domänen behandeln zu können. Hier sollte untersucht werden, an welchen Stellen bei der Semantik Berechenbarkeitsprobleme auftauchen. Dasselbe trifft für unendlich viele Prädikatsvariable zu.

Kapitel 3

BDDs – Binäre Entscheidungsdiagramme

BDDs (Binary Decision Diagrams – Binäre Entscheidungsdiagramme) sind eine Datenstruktur zur kompakten Repräsentation von Booleschen Funktionen. Nun kann man jede Menge mit ihrer charakteristischen Funktion identifizieren, so daß durch eine entsprechende Kodierung einer endlichen Menge als Untermenge des $\mathbb{B}^n$ für ein geeignetes $n \in \mathbb{N}$ deren charakteristische Funktion zur Booleschen Funktion wird. Auf diese Weise läßt sich jede endliche Menge als BDD repräsentieren. Mengenoperationen, wie Vereinigung, Schnitt usw., lassen sich durch boolesche Operationen auf BDDs realisieren. Hierfür gibt es sehr effiziente Algorithmen. Die Repräsentation von Zustandsmengen von endlichen Systemen wird in dieser Arbeit das Hauptanwendungsgebiet von BDDs darstellen.

3.1 Übersicht

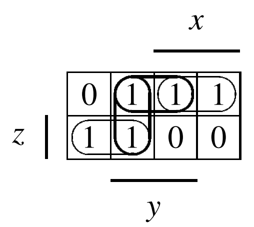
Nach einer Einführung in BDDs (genauer ROBDDs) wird ein neuer Zugang zu BDDs präsentiert, der BDDs als Terme sieht. So kann die Normalformeigenschaft von ROBDSs rein algebraisch bewiesen werden.

Es folgt eine einheitliche und übersichtliche Darstellung der wichtigsten BDD-Algorithmen, die zum Teil nur in Form von Quelltexten von BDD-Bibliotheken dokumentiert waren. Dies wird erreicht durch die Definition einer abstrakten Maschine SCAM, die spezielle Basisoperationen für BDD-Algorithmen zur Verfügung stellt.

Zum Schluß wird der vom Autor entwickelte ${}_{c}\mathsf{ite}_{\exists}$-Algorithmus vorgestellt. Dieser Algorithmus subsumiert die wichtigsten bei der Modellprüfung verwendeten BDD-Algorithmen und verallgemeinert den $\mathsf{collapse}$-Algorithmus des SMV-Systems für beliebige Variablenordnungen. Durch die abstrakte Sichtweise der SCAM konnte auch für den komplexen ${}_{c}\mathsf{ite}_{\exists}$-Algorithmus die Korrektheit formal nachgewiesen werden.

3.2 Einleitung

Boolesche Funktionen sind in der Informatik ein wichtiger abstrakter Datentyp, basiert doch der Entwurf der überwiegenden Anzahl von Computerchips auf der Manipulation von Booleschen Funktionen. Schon früh wurden daher die verschiedensten Repräsentationen dieses abstrakten Datentyps untersucht. Ausgangspunkt unserer Überlegungen bildet die Mathematische Logik. Genauer noch ist es die Aussagenlogik, in der man sich mit Booleschen Ausdrücken beschäftigt. Die Semantik eines Booleschen Ausdrucks läßt sich auffassen als eine Boolesche Funktion, so daß Boolesche Ausdrücke als Repräsentation von Booleschen Funktionen dienen können.



$$\overline{x}y \vee \overline{x}z \vee x\overline{z} \quad \equiv \quad \overline{z}y \vee \overline{x}z \vee x\overline{z}$$

Abbildung 3.1: DNF ist i. allg. keine Normalform (links ein KV-Diagramm).

Oft sind booleschen Funktionen, wie sie in der Informatik auftreten, nicht als Wertetabelle vorgegeben, sondern werden implizit durch eine symbolische Beschreibung charakterisiert, die sich in vielen Fällen eins zu eins in einen Booleschen Ausdruck umsetzen läßt. Es stellt sich also die Frage, wie man vernünftig Boolesche Ausdrücke behandelt.

3.2.1 Normalformen für Boolesche Funktionen

Boolesche Ausdrücke bilden keine Normalform für Boolesche Funktionen. So ist die syntaktische Gleichheit von zwei Booleschen Ausdrücken kein notwendiges Kriterium für die Gleichheit der zugeordneten Booleschen Funktionen: Die beiden Booleschen Ausdrücke

$$(x \leftrightarrow y) \wedge x \quad \text{und} \quad x \wedge y$$

entsprechen zwar der gleichen Booleschen Funktion (sie sind logisch äquivalent), sind aber syntaktisch verschieden.

Eine Normalform erhält man durch Einschränkung der betrachteten Booleschen Ausdrücke, wie zum Beispiel bei der DNF (Disjunktiven Normalform) oder der KNF (Konjunktiven Normalform). Die DNF ist eine Disjunktion über alle Minterme einer Booleschen Funktion. Die Minterme einer Booleschen Funktion f stehen in einer eins zu eins Beziehung zu den Vektoren $\overline{v}$ für die $f(\overline{v}) = 1$. Deshalb ist die DNF für viele Boolesche Funktionen, die wie z. B. die Parität eine wichtige praktische Bedeutung haben, leider exponentiell groß. Eine ähnliche Argumentation gilt für die KNF.

Um die Größe dieser Repräsentation zu verkleinern, wurden die DMF (Disjunktive Minimalform) und KMF (Konjunktive Minimalform) eingeführt (neben diesem Zweck dienen DMF und KMF auch zur Minimierung von Schaltnetzen, was hier aber nicht näher untersucht werden soll). Dabei verliert man aber die Normalformeigenschaft, wie das Beispiel aus Abbildung 3.1 zeigt. Das KV-Diagramm (vgl. /McCalla, 1992/) in dieser Abbildung zeigt, daß beide DMF auf der rechten Seite zur gleichen booleschen Funktion gehören, aber eben verschieden sind. Ebenso bleibt das exponentielle Verhalten erhalten, wofür wiederum die Parität als Beispiel dienen kann.

Da die Normalformen selbst in vielen Fällen exponentiell groß sind (i. allg. in der Anzahl Parameter oder Booleschen Variablen), scheint die Frage nach der Komplexität eines Algorithmus zur Berechnung der Normalform weniger wichtig. Bei der Manipulation von Booleschen Funktionen findet man aber häufig Algorithmen, die Zwischenergebnisse erzeugen, die nicht in Normalform sind, so daß inkrementell immer wieder Normalformen berechnet werden müßten. Hier hätte man gern die Eigenschaft, daß die Normalform in polynomialer Komplexität in der Größe der entstehenden Zwischenergebnisse berechnet werden kann. Bei ROBDDs (der für unsere Zwecke wichtigsten Teilklasse der BDDs) läßt sich dies sogar in linearer Komplexität bewerkstelligen. Diese Betrachtungen machen auch klar, daß es wenig Sinn macht, Normalformen unabhängig von den Algorithmen zu deren Manipulation zu betrachten.

3.2.2 Algorithmen für Boolesche Funktionen

Der bei weitem schwerwiegendere Nachteil obiger Repräsentationen für boolesche Funktionen ist die Komplexität von Algorithmen zu deren Handhabung. So ist die Zeit- und Platzkomplexität eines Algorithmus zur Invertierung einer Booleschen Funktion, die durch eine KNF, DNF, KMF oder DMF repräsentiert wird, was einer logischen Negation entspricht, im schlimmsten Fall exponentiell, da die Größe der Ausgabe exponentiell größer als die der Eingabe sein kann. Hierzu ein Beispiel in DNF:

$$\neg(xyz) \quad \equiv \quad \overline{x}\overline{y}\overline{z} \vee \overline{x}\overline{y}z \vee \overline{x}y\overline{z} \vee \overline{x}yz \vee x\overline{y}\overline{z} \vee x\overline{y}z \vee xy\overline{z}$$

Auch hier zeigt sich der Vorteil der Verwendung von BDDs (genauer der ROBDDs). Sie gestatten es, alle booleschen Operationen (logisches „und“, „oder“, „nicht“, ...) mit linearer Komplexität in der Größe der Operanden durchzuführen.

3.3 BDDs am Beispiel

In diesem Abschnitt werden BDDs eingeführt, und es wird gezeigt, wie man von einer Wertetabelle für eine Boolesche Funktion zu einem zugehörigen ROBDD gelangt. Es sei gleich darauf hingewiesen, daß diese Vorgehensweise wie oben angedeutet nicht der normale Weg ist, um zu einem BDD zu gelangen.

Als Beispiel betrachten wir die vierstellige Boolesche Funktion f, die durch folgenden Booleschen Ausdruck definiert ist:

$$f(x_0, x_1, x_2, x_3) \quad :\equiv \quad (x_0 \vee (x_1 = x_2)) \wedge (x_0 \neq x_3)$$

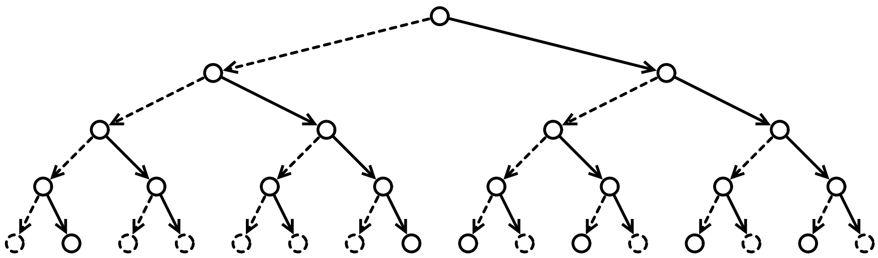
Die entsprechende Wertetabelle lautet

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_0
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_1
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_2
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_3
0	1	0	0	0	0	0	1	1	0	1	0	1	0	1	0	f

Durch Verschmelzen gleicher Tabelleneinträge erhält man

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td>x_0</td></tr>
<tr><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">0</td><td colspan="4">1</td><td>x_1</td></tr>
<tr><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">1</td><td>x_2</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>x_3</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>f</td></tr>
</table>

Um den Wert von f für eine bestimmte Belegungen $\overline{v} = (v_0, v_1, v_2, v_3)$ aus der Tabelle zu ermitteln, kann man nun binäre Suche verwenden. Ist $v_0 = 0$, so sucht man in der linken Hälfte der Tabelle. Für $v_1 = 1$ macht man in der rechten Hälfte der linken Hälfte weiter und so fort, bis man nach viermaliger Fallunterscheidung bei einem einzelnen Wert in der untersten Zeile der Tabelle angelangt ist, der dann den Wert von $f(\overline{v})$ angibt. Die Baumstruktur dieser Fallunterscheidungen kann man als binären Baum darstellen



Die Blätter dieses Baumes entsprechen den Funktionswerten von f. Ein gestricheltes Blatt steht dabei für eine 0 und ein Blatt mit durchgezogener Umrandung für eine 1. Innere Knoten auf einer Ebene entsprechen den Zeilen der Tabelle, sind also genau einer Variablen zugeordnet.

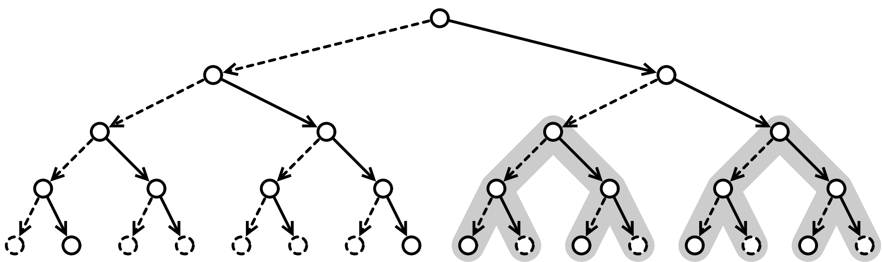
In der Tabelle sind die Werte linear angeordnet. Um bei den folgenden Umstrukturierungen des Baumes diese Ordnung zu erhalten, müßten die Nachfolger eines Knotens unterschiedlich markiert werden. Dies geschieht entsprechend der Fallunterscheidung, die man bei Auswertung des Baumes an einem Knoten treffen würde. Ist der Nachfolger dadurch gegeben, daß man für die zugeordnete Variable eine 0 gewählt hat, dann ist die verbindende Kante gestrichelt. Falls die Variable 1 ist, dann verwendet man eine durchgezogene Kante. Man spricht auch von 0-Nachfolger im ersten bzw. 1-Nachfolger im zweiten Fall.

Dieser Baum ist ein OBDD (Ordered Binary Decision Diagram – geordnetes Binäres Entscheidungsdiagram). Allgemeine BDDs erhält man, wenn man die strikte Zuordnung von Variablen zu Ebenen des Baumes aufhebt und statt dessen jeden Knoten mit einer beliebigen Variable markiert. In diesem Abschnitt werden aber nur OBDDs betrachtet.

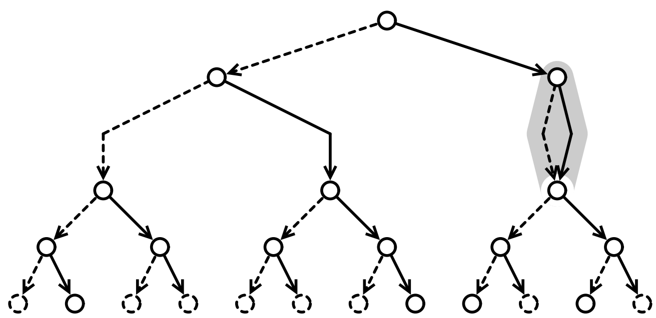
$$\boxed{\text{BDD} + \text{Variablenordnung} = \text{OBDD}}$$

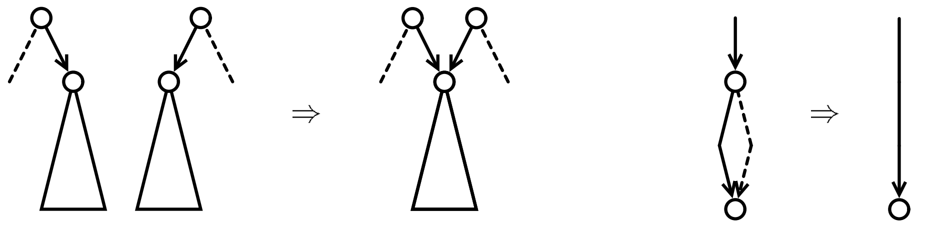
Zunächst hat diese Darstellung einer Booleschen Funktion als Baum keine Vorteile gegenüber einer Darstellung als Wertetabelle. Man muß sogar zusätzlich die Markierungen und inneren Knoten abspeichern. Sieht man sich jedoch die Struktur des Baumes genauer an, so stellt man fest, daß hier sehr viel Redundanz enthalten ist.

Die Hauptidee besteht nun darin, den Baum als gerichteten, azyklischen Graphen (DAG – acyclic directed graph) aufzufassen und isomorphe Teilgraphen nur einmal abzuspeichern.



Ersetzt man die beiden mit grau unterlegten Teilgraphen durch nur eine Kopie, so erhält man den OBDD

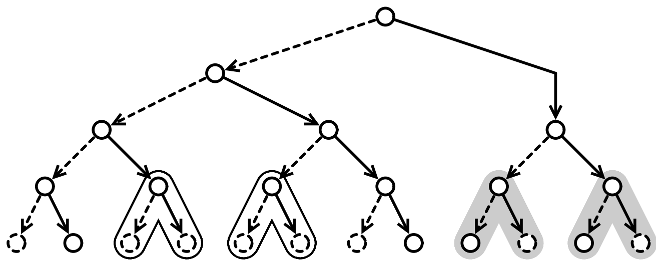




a. Algebraische Reduktion b. Elimination redundanter Knoten

Abbildung 3.2: Operationen zur Normalisierung von BDDs.

Hier ist der Baum an einer Stelle zum echten DAG geworden. Jedoch ist es in diesem Fall wieder möglich, den OBDD in einen Baum zu verwandeln. Der grau gekennzeichnete Knoten hat dieselben Nachfolger. Es ist deshalb unerheblich, welchen Wert die Variable x_1 bei dieser Fallunterscheidung hat, und der Knoten samt den Kanten zu seinen Nachfolgern (der gesamte grau unterlegte Teil des letzten Graphs) kann weggelassen werden. Wenn man noch den 1-Nachfolger des Wurzelknotens entsprechend „umbiegt“, so erhält man



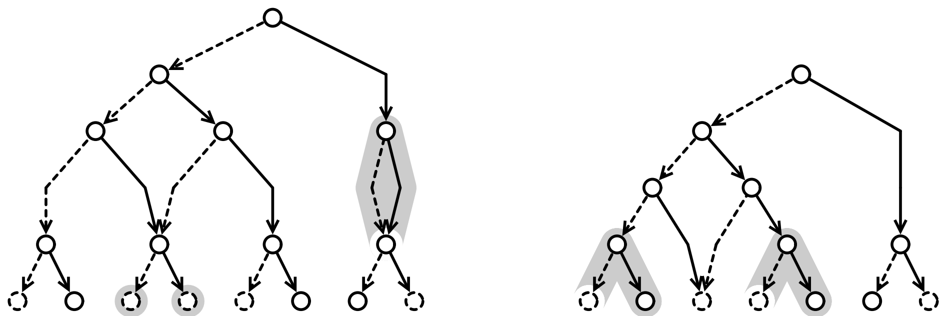
Hier werden zwei Operationen auf BDDs verwendet, die auch als „Algebraische Reduktion“ und „Elimination redundanter Kanten“ bekannt sind (vergl Abb. 3.2). Mit anderen Worten entsprechen diese dem

1. Verschmelzen isomorpher Teilgraphen
2. Löschen von Knoten mit zwei gleichen Nachfolgern

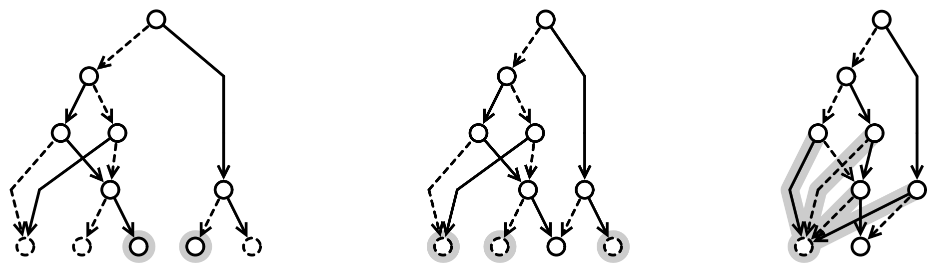
Diese werden nun solange angewandt, bis sich der Graph nicht mehr ändert. Dies liefert einen maximal reduzierten OBDD, der dann ROBDD (reduced OBDD – reduzierter OBDD) genannt wird.

$$\boxed{\text{ROBDD} = \text{maximal reduzierter OBDD}}$$

Durch Zusammenfassen der grau unterlegten bzw. umrandeten Teilgraphen des letzten OBDD erhält man den linken der folgenden OBDDs



Der rechte OBDD geht durch Verschmelzen der Beiden 0-Blätter mit sofortigem Löschen des dadurch entstehenden redundanten Knotens sowie Löschen des redundanten grau unterlegten Teilgraphen hervor. Weitere Anwendungen der Reduktionsoperationen (insbesondere Verschmelzen der Blätter) erzeugt die OBDDs



Der letzte OBDD ist maximal reduziert, da keine der beiden Operationen mehr anwendbar ist. Er ist damit ein ROBDD und bildet die, wie später gezeigt wird, eindeutige Normalform für die Beispielfunktion f unter der gewählten Variablenordnung $x_0 < x_1 < x_2 < x_3$. Für andere Variablenordnungen muß dieser ganze Prozeß wiederholt werden und liefert im allgemeinen einen anderen ROBDD. Für die Variablenordnung $x_1 < x_0 < x_3 < x_2$ erhält man den rechten der beiden folgenden OBDDs

Der Graph links entsteht aus dem ROBDD für f mit der ersten Ordnung durch Löschen aller Kanten, die zum (einzigen) 0-Blatt gehen (diese sind in der ersten Version des ROBDDs grau unterlegt). Dies dient nur der besseren Darstellbarkeit, und entsprechend ist der ROBDD für die zweite Variablenordnung zu interpretieren. Fehlt ein Blatt bei einem OBDD, so erhält man formal den entsprechenden vollständigen OBDD, indem man dieses fehlende Blatt hinzufügt und alle fehlenden Kanten auf dieses Blatt richtet.

Was die Variablenordnung betrifft, zeigt dieses Beispiel auch, daß die Größe eines ROBDDs für eine Boolesche Funktion f abhängig ist von der Variablenordnung. Für die erste Variablenordnung besteht der ROBDD aus sechs inneren Knoten, wogegen der ROBDD für die zweite Variablenordnung aus acht inneren Knoten besteht. Später werden noch weit dramatischere Beispiele folgen, bei dem eine schlechte Wahl der Variablenordnung zu einem exponentiell großen ROBDD führt (in der Anzahl Variablen), obwohl es eine Variablenordnung gibt, für die der ROBDD sogar linear groß bleibt.

3.4 Die Algebra FBDD

In diesem Abschnitt wird eine neue Formalisierung der (RO)BDDs vorgestellt. Sie stellt nicht die Graphenstruktur der (RO)BDDs in den Vordergrund, sondern betrachtet (RO)BDDs als freie Termalgebra modulo eines Termersetzungssystems.

Für den Nachweis der Normalformeigenschaft von (RO)BDDs und die Korrektheit der noch anzugebenden Algorithmen reicht diese abstrakte Sichtweise völlig aus. Das Zusammenlegen (engl. „to share") von gemeinsamen Teiltermen muß erst dann betrachtet werden, wenn man zu günstigen Komplexitätsaussagen gelangen will. Dies ist Thema der *Implementierung* von freien Algebren modulo eines Termersetzungssystems, was in Abschnitt 3.5 behandelt wird.

Diese Vorgehensweise erlaubt einmal eine einfachere (algebraischere) Behandlung der Normalformeigenschaft. Darüber hinaus läßt sich so eine Brücke zwischen Implementierungstechniken für BDDs und für allgemeine kanonische Termersetzungssysteme schlagen.

Definition 3.1 (Algebra FBDD**)** *Die freie Termalgebra*

$$\mathsf{FBDD} = (0, 1, \cdots\!\textcircled{0}\!-, \cdots\!\textcircled{1}\!-, \cdots\!\textcircled{2}\!-, \ldots)$$

mit den zweistelligen Operatoren $\cdots\!\textcircled{i}\!-$ *für* $i \in \mathbb{N}$*, und den beiden Konstanten* 0 *und* 1 *heiße die Menge der* freien binären Entscheidungsdiagramme *(kurz* FBDDs *für engl. „free binary decision diagrams"). Für* $f = g \cdots\!\textcircled{i}\!- h \in \mathsf{FBDD}$ *definiere*

$$\mathsf{lo}(f) := g, \quad \mathsf{hi}(f) := h \quad \textit{und} \quad \mathsf{top}(f) := i$$

Wie in Definition 2.20 für $\mathbb{B}_\mu^V$ wird eine Untertermordnung „$<$" bzw. „$\leq$" definiert.

Definition 3.2 (Untertermordnung auf FBDD**)** *Auf* FBDD *wird mit „$\leq$" (bzw. „$<$") die* Untertermordnung *bezeichnet. Dabei sei „$<$" definiert als die kleinste transitive Relation, mit*

$$g, h, 0, 1 < g \cdots\!\textcircled{i}\!- h, \quad \textit{für } g, h \in \mathsf{FBDD},\ i \in \mathbb{N}$$

„$\leq$" ist die reflexive Hülle von „$<$".

Die Größe eines FBDDs spielt eine wichtige Rolle bei der Beurteilung der Komplexität von Algorithmen für FBDDs (es werden im wesentlichen später nur Algorithmen für die eingeschränkte Klasse der ROBDDs betrachtet, für die die folgende Definition aber auch zutrifft).

Definition 3.3 (Größe eines FBDDs) *Zu* $f \in \mathsf{FBDD}$ *sei*

$$|f| := |\{g \in \mathsf{FBDD} \mid g \leq f\}|$$

definiert als die Größe *von* f.

Die Größe eines FBDDs wird also nicht als die Anzahl Knoten im „Termbaum" definiert, sondern als die Anzahl *verschiedener* Unterterme. Wie sich später in Abschnitt 3.5 herausstellen wird, stimmt diese Größe exakt mit der Größe des Graphen überein, den man als Repräsentation für den Term in der SCAM (siehe Abschnitt 3.5) erhält.

Definition 3.4 (Variablen eines FBDDs) *Es sei* $\mathsf{var}(f) \subseteq \mathbb{N}$ *für* $f \in \mathsf{FBDD}$*, mit*

$$\mathsf{var}(0) := \mathsf{var}(1) := \emptyset, \quad \mathsf{var}(g \cdots\!\textcircled{\scriptsize i}\!\!-\, h) := \mathsf{var}(g) \cup \{i\} \cup \mathsf{var}(h)$$

Für $f_1, \ldots, f_n \in \mathsf{FBDD}$ *definiere als Abkürzung*

$$\mathsf{var}(f_1, \ldots, f_n) \;:=\; \mathsf{var}(f_1) \cup \ldots \cup \mathsf{var}(f_n)$$

Im Vergleich zum vorigen Abschnitt fällt auf, daß hier die Variablen $\mathsf{var}(f)$ eines FBDDs f nicht als Markierungen der inneren Knoten auftreten, sondern einer Operation in der Algebra entsprechen. Wenn im weiteren Verlauf der Arbeit dennoch von den Variablenmarkierungen gesprochen wird, dann ist damit die natürliche Zahl gemeint, die in der Operation auftaucht (z. B. i in $g \cdots\!\textcircled{\scriptsize i}\!\!-\, h$). Somit sei also auch erlaubt, natürliche Zahlen als Bezeichner für Variablen zu verwenden.

Weiter wurden im vorigen Abschnitt nur „geordnete“ BDDs betrachtet. Hierzu wird eine Ordnung auf den FBDDs definiert:

Definition 3.5 (Ordnung auf FBDD**)** *Für* $f, g \in \mathsf{FBDD}$ *sei*

$$f \precneqq g \quad :\Leftrightarrow \quad f \notin \{0,1\}, g \in \{0,1\} \quad \textit{oder} \quad f, g \notin \{0,1\},\ \mathsf{top}(f) < \mathsf{top}(g)$$

und „$\prec$*“ sei die reflexiv-transitive Hülle von „*$\precneqq$*“.*

In dieser Ordnung sind die Konstanten 0 und 1 die größten Elemente. Sonst wird die Ordnung von der „obersten“ Variable bestimmt. Warum die Konstanten *un*natürlicherweise die größten Elemente sind, wird aus der Diskussion zur folgenden Definition klar.

Definition 3.6 (Algebra OBDD**)** $\mathsf{OBDD} \subseteq \mathsf{FBDD}$ *ist die größte Menge, so daß*

$$\textit{für alle } f = g \cdots\!\textcircled{\scriptsize i}\!\!-\, h \in \mathsf{OBDD} \textit{ ist } f \prec g,\ f \prec h \textit{ und } g, h \in \mathsf{OBDD} \tag{3.1}$$

Die Elemente von OBDD *heißen* geordnete binäre Entscheidungsdiagramme *(kurz OBDDs für engl. „ordered binary decision diagrams“).*

Mit anderen Worten ist ein FBDD ein OBDD genau dann, wenn sich die Indizes der Operationen (die Variablen) antimonoton bez. der Untertermordnung verhalten. Oder noch anschaulicher, jeder Unterterm ist eine Konstante oder ist mit einer größeren Variable als all seine Vaterterme beschriftet. Diese Antimonotonie wirkt auf den ersten Augenblick etwas unnatürlich. Andererseits betrachtet man so die Fallunterscheidungen, die durch einen BDD repräsentiert werden in ihrer natürlichen Ordnung: „die kleinste Variable kommt als erstes dran“. Als Folge dieser Festlegung wird im folgenden in Induktionsbeweisen nicht über die Variablen induziert, sondern über den Termaufbau.

Die Menge OBDD ist nur noch eine partielle Algebra, da nicht alle Operationen für alle Elemente definiert sind. Insbesondere kann die Operation $\cdots\!\textcircled{\scriptsize i}\!\!-$ auf kein $f \in \mathsf{OBDD}$ angewendet werden, für das $\mathsf{top}(f) \leq i$, da sonst das Resultat nicht mehr geordnet ist.

Bei der informellen Darstellung der Reduktion von BDDs im vorigen Abschnitt, wurde neben der *algebraischen* Reduktion (vgl. Abb. 3.2.a), die erst bei der Implementierung zum Tragen kommt (siehe Abschnitt 3.5), auch die Entfernung eines Knotens mit gleichen Nachfolgern betrachtet. Diese Operation kann nun als Termersetzungsregel definiert werden.

Definition 3.7 *Auf* FBDD *werde das folgende Termersetzungssystem betrachtet*

$$t \cdots ⓘ - t \quad ::= \quad t \tag{3.2}$$

Die Reduktionsrelation bez. (3.2) *sei mit „$\rightsquigarrow$" bezeichnet. Da das Termersetzungssystem lokal konfluent und noethersch ist, besitzt jedes $f \in$ FBDD eine Normalform $\hat{f}$, mit $f \rightsquigarrow^* \hat{f}$, auf die 3.2 nicht mehr anwendbar ist.*

In /Dershowitz und Jouannaud, 1990, Deussen, 1992/ findet man die Definition der für die Termersetzungssysteme verwendeten Begriffe. Die Reduktion nach Gleichung (3.2) läßt sich auch in OBDD ausführen:

Lemma 3.8 (Reduktion auf OBDD) *Für alle $f \in$ OBDD ist $\hat{f} \in$ OBDD.*

Beweis: Es genügt zu zeigen, daß für $f \rightsquigarrow g$ und $f \in$ OBDD auch $g \in$ OBDD. Um dies formal zeigen zu können, benötigt man eine Formalisierung von Termersetzungssystemen, was hier nicht getan werden soll. Weniger formal argumentiert, kann man feststellen, daß bei einem Ersetzungsschritt mit (3.2) ein Unterterm nur durch einen „größeren" (bez. der Ordnung „$\prec$" auf FBDD) ersetzt werden kann. Die Bedingung (3.1) an Elemente aus OBDD kann dadurch nicht verletzt werden. ❑

Die Normalformen bez. (3.2) bilden die im weiteren betrachtete Menge der ROBDDs.

Definition 3.9 (Menge ROBDD) ROBDD $:= \{f \in$ OBDD $\mid f = \hat{f}\}$. *Dies ist die Menge der* reduzierten geordneten Entscheidungsdiagramme *(kurz ROBDDs für englisch „reduced ordered binary decision diagrams").*

Es sei gleich darauf hingewiesen, daß diese Menge von Entscheidungsdiagrammen am wichtigsten für den Rest der Arbeit ist, so daß hierfür die Abkürzung BDD := ROBDD eingeführt wird, und im folgenden anstatt von reduzierten geordneten BDDs (ROBDDs) nur einfach von BDDs gesprochen wird.

Wie schon mehrfach erwähnt, bilden die ROBDDs eine Normalform für boolesche Ausdrücke (boolesche Funktionen). Um dies formalisieren zu können, muß man für eine Abbildung von booleschen Ausdrücken auf BDDs und umgekehrt sorgen (vergleiche Abb. 2.2). Dazu benötigt man zunächst eine „Allokation" der Variablen. Das ist eine Abbildung der Variablen W der booleschen Ausdrücke auf natürliche Zahlen:

Definition 3.10 (Allokationen für $\mathbb{B}^W$) *Die Menge der Allokationen $\mathcal{A}^W$ für $\mathbb{B}^W$ besteht aus den* partiellen, injektiven *Abbildungen $A\colon W \to \mathbb{N}$.*

Die Menge der Allokationen ist somit überabzählbar. Da aber die betrachteten booleschen Ausdrücke immer nur endlich viele Variable enthalten, ist dies vernachlässigbar. Man könnte $\mathcal{A}^W$ auch auf solche A mit endlichem $\mathsf{domain}(A)$ einschränken. Weiter schreibe im folgenden auch $\mathcal{A} := \mathcal{A}^W$, wenn keine Verwechslungsgefahr besteht.

Diese explizite Verwendung von Allokationen ist in der Literatur selten. Sie treten in ähnlicher Form immer dann auf, wenn formal Aussagen über verschiedene Variablenordnungen getroffen werden sollen, wie z. B. in /Bern et al., 1995/. Ansonsten werden die Knoten in den BDDs immer mit den Variablen direkt markiert. Bei ROBDDs wird eine implizite lineare Ordnung auf den Variablen verwendet.

Definition 3.11 (FBDD Semantik) *Für eine Allokation $A \in \mathcal{A}^W$, definiere*

$$\langle\!\langle \cdot \rangle\!\rangle_A : \mathsf{FBDD} \to \mathbb{B}^W$$

als partielle *Funktion, mit*

$$\mathsf{domain}(\langle\!\langle \cdot \rangle\!\rangle_A) = \{f \in \mathsf{FBDD} \mid \mathsf{var}(f) \subseteq \mathsf{range}(A)\},$$

rekursiv über den Termaufbau

$$\langle\!\langle 0 \rangle\!\rangle_A := 0, \quad \langle\!\langle 1 \rangle\!\rangle_A := 1, \quad \langle\!\langle g \cdots\text{\textcircled{i}}\!-\!h \rangle\!\rangle_A := \neg A^{-1}(i) \wedge \langle\!\langle g \rangle\!\rangle_A \;\vee\; A^{-1}(i) \wedge \langle\!\langle h \rangle\!\rangle_A .$$

Im letzten Fall ist dies gerade „$\mathrm{ite}(A^{-1}(i), \langle\!\langle h \rangle\!\rangle_A, \langle\!\langle g \rangle\!\rangle)$*“.*

Dies ist wohldefiniert, da $\langle\!\langle \cdot \rangle\!\rangle_A$ nur definiert sein soll, wenn A für die Variablen des Argumentes auch definiert ist. In diesem Fall gilt dies auch bei inneren Knoten für die Nachfolger. Wegen der Injektivität von A ist dann $A^{-1}(i)$ immer eindeutig bestimmt. Als erstes Miniresultat erhält man

---

Lemma 3.12 *Für $f = g \cdots\text{\textcircled{i}}\!-\!h \in \mathsf{FBDD}$ mit $i \notin \mathsf{var}(g,h)$ ist*

a. $\langle\!\langle g \rangle\!\rangle_A\{x \mapsto t\} \equiv \langle\!\langle g \rangle\!\rangle_A, \quad$ *für alle* $x \in W \backslash A^{-1}(\mathsf{var}(g)),\ t \in \mathbb{B}^W$

b. $\langle\!\langle f \rangle\!\rangle_A\{A^{-1}(i) \mapsto 0\} \equiv \langle\!\langle g \rangle\!\rangle \quad$ *und* $\quad \langle\!\langle f \rangle\!\rangle_A\{A^{-1}(i) \mapsto 1\} \equiv \langle\!\langle h \rangle\!\rangle$

für eine Allokation $A \in \mathcal{A}$, mit $\mathsf{range}(A) \supseteq \mathsf{var}(f)$.

Beweis:

a. (Induktion über Termaufbau von g) Im Induktionsanfang ist $g \in \{0,1\}$, so daß unmittelbar die Behauptung folgt. Im Induktionsschritt sei $g = g_0 \cdots\text{\textcircled{j}}\!-\!g_1$. Mit der Voraussetzung an g gilt $x \neq j$, $x \notin A^{-1}(\mathsf{var}(g_0))$ und $x \notin A^{-1}(\mathsf{var}(g_1))$. Somit greift die Induktionsvoraussetzung und man erhält

$$\langle\!\langle g_0 \rangle\!\rangle_A\{x \mapsto t\} \equiv \langle\!\langle g_0 \rangle\!\rangle_A \qquad \langle\!\langle g_1 \rangle\!\rangle_A\{x \mapsto t\} \equiv \langle\!\langle g_1 \rangle\!\rangle_A$$

Mit $y := A^{-1}(j) \neq x$ ist nun

$$\begin{aligned}
\langle\!\langle g \rangle\!\rangle_A\{x \mapsto t\} &= (\neg y \wedge \langle\!\langle g_0 \rangle\!\rangle_A \vee y \wedge \langle\!\langle g_1 \rangle\!\rangle_A)\{x \mapsto t\} \\
&= \neg y \wedge \langle\!\langle g_0 \rangle\!\rangle_A\{x \mapsto t\} \vee y \wedge \langle\!\langle g_1 \rangle\!\rangle_A\{x \mapsto t\} \\
&\equiv \neg y \wedge \langle\!\langle g_0 \rangle\!\rangle_A \vee y \wedge \langle\!\langle g_1 \rangle\!\rangle_A \qquad = \langle\!\langle g \rangle\!\rangle_A
\end{aligned}$$

b. Aussage a. trifft auf g und h mit $x := A^{-1}(i)$ und $t \in \{0,1\}$ zu, so daß

$$\begin{array}{rcll} \langle\!\langle f \rangle\!\rangle_A\{x \mapsto t\} & = & (\neg x \wedge \langle\!\langle g \rangle\!\rangle_A \vee x \wedge \langle\!\langle h \rangle\!\rangle_A)\{x \mapsto t\} & \\ & = & \neg t \wedge \langle\!\langle g \rangle\!\rangle_A\{x \mapsto t\} \vee t \wedge \langle\!\langle h \rangle\!\rangle_A\{x \mapsto t\} & \\ & \equiv & \neg t \wedge \langle\!\langle g \rangle\!\rangle_A \vee t \wedge \langle\!\langle h \rangle\!\rangle_A & \text{nach a.} \\ & = & \left\{ \begin{array}{ll} \neg 0 \wedge \langle\!\langle g \rangle\!\rangle_A \vee 0 \wedge \langle\!\langle h \rangle\!\rangle_A, & t = 0 \\ \neg 1 \wedge \langle\!\langle g \rangle\!\rangle_A \vee 1 \wedge \langle\!\langle h \rangle\!\rangle_A, & t = 1 \end{array} \right\} & \equiv \left\{ \begin{array}{ll} \langle\!\langle g \rangle\!\rangle_A, & t = 0 \\ \langle\!\langle h \rangle\!\rangle_A, & t = 1 \end{array} \right. \end{array}$$

❏

Die Konvention, daß der Null-Nachfolger ($\mathsf{lo}(f)$) in der Graphendarstellung zumeist links steht (sozusagen der linke Nachfolger ist), findet man häufig in der Literatur (und auch in Implementierungen!). Die Semantik dargestellt als „ite“ vertauscht aber die Reihenfolge der Argumente, so daß man an dieser Stelle beim Übergang von Graphendarstellung zur Semantik sehr vorsichtig sein muß.[1]

Für die Umkehrung soll nur eine Abbildung $\|\cdot\|_A : \mathbb{B}^W \to$ ROBDD (vgl. Abb. 2.2) betrachtet werden, die rekursiv über der Termstruktur von $\mathbb{B}^W$ definiert wird. Andere in der Literatur zu findende Abbildungen sind nicht so interessant, da sie entweder nur ein Spezialfall der hier betrachteten sind oder aber bei einem Rekursionsschritt schon exponentiellen Aufwand verursachen. Auf diese Problematik wird in dieser Arbeit nicht weiter eingegangen. Statt dessen sei auf /Sieling, 1995/ verwiesen.

Für die rekursive Definition von $\|\cdot\|_A$ braucht man auf seiten von ROBDD entsprechende Operationen $\neg_{\mathsf{BDD}}$ und $\wedge_{\mathsf{BDD}}$ für die booleschen Operationen $\neg$ und $\wedge$. Diese werden in Form von Algorithmen (siehe Abb. 3.3) angegeben, die in einer pseudo-funktionalen Programmiersprache $\mathcal{F}$ geschrieben sind (vgl. Abschnitt 3.5).
Für die Negation kann man auch eine globale (algebraische) Definition geben.

$$\neg_{\mathsf{BDD}} f \;=\; f\{0 \mapsto 1,\, 1 \mapsto 0\},$$

wobei die hier angegebene Substitution als homomorph fortgesetzt auf ganz FBDD zu verstehen ist, ähnlich wie in Definition 2.4. Die fehlende Operation cf ist in Abbildung 3.4 zu finden. Ihre Bedeutung verdeutlicht Abbildung 3.5. Sie wird in den betrachteten Algorithmen nur mit der Vorbedingung verwendet, daß die oberste Variable des BDDs f des ersten Argumentes immer größer oder gleich dem zweiten Argument ist. In diesem Fall zerlegt „cf“ den BDD f in seine Nachfolger, wenn die oberste Variable mit dem zweiten Argument i übereinstimmt. Ansonsten liefert sie das erste Argument unverändert zurück. Das Ergebnis sind die Kofaktoren (s. Def. 2.8) von f bez. der Variablen m.

Diese Operation vereinfacht im Vergleich zu /Bryant, 1986/ die Darstellung der Algorithmen wesentlich. Wenn man sie nicht benutzt, muß man ansonsten z. B. beim „$\wedge_{\mathsf{BDD}}$“ die drei Fälle aus Abbildung 3.6 explizit bearbeiten. Bei BDD-Algorithmen mit drei Argumenten, wie dem $\mathsf{ite}_{\mathsf{BDD}}$ werden daraus sogar 7 Fallunterscheidungen! Die Idee, eine zusätzliche Operation hier zu benutzen, stammt aus der BDD-Bibliothek von Long (/Long, 1994/). Dort ist es das C-Makro `BDD_COFACTOR`. Sonstige Erwähnungen in der Literatur konnten nicht gefunden werden. In allen anderen dem Autor bekannten Implementierungen

[1] In einer vom Autor implementierten BDD-Bibliothek, die als Alternative zu anderen Bibliotheken vom Modellprüfer μcke verwendet werden kann, wurde dieser Fehler an zwei unabhängigen Stellen begangen, was jedesmal über 10 Mannstunden Fehlersuche bedeutete.

$\boxed{\neg_{\mathsf{BDD}}\, f}$ ::=

if $f = 0$ **then** 1
else if $f = 1$ **then** 0
else
$l \cdots (m) — r$
where
$m = \mathsf{top}(f)$
$l \;= \boxed{\neg_{\mathsf{BDD}}\, \mathsf{lo}(f)}$
$r \;= \boxed{\neg_{\mathsf{BDD}}\, \mathsf{hi}(f)}$

a. Negation

$\boxed{f \wedge_{\mathsf{BDD}} g}$ ::=

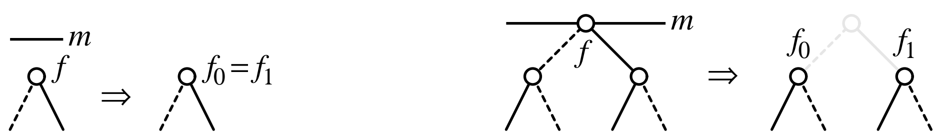
if $f = 0$ **then** 0
else if $f = 1$ **then** g
else
$l \cdots (m) — r$
where
$m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\}$
$l \;= \boxed{f_0 \wedge_{\mathsf{BDD}} g_0}$
$r \;= \boxed{f_1 \wedge_{\mathsf{BDD}} g_1}$
where
$(f_0, f_1) = \mathsf{cf}(f, m)$
$(g_0, g_1) = \mathsf{cf}(g, m)$

b. Logisches „Und“

Abbildung 3.3: $\neg_{\mathsf{BDD}} : \mathsf{ROBDD} \to \mathsf{ROBDD}$ und $\wedge_{\mathsf{BDD}} : \mathsf{ROBDD}^2 \to \mathsf{ROBDD}$

$\boxed{\mathsf{cf}(f,m)}$ ::= **if** $f = 0$ **or** $f = 1$ **or** $\mathsf{top}(f) > m$ **then** (f, f) **else** $(\mathsf{lo}(f), \mathsf{hi}(f))$

Abbildung 3.4: $\mathsf{cf} : \mathsf{ROBDD} \times \mathbb{N} \to \mathsf{ROBDD}^2 \quad (\mathsf{top}(f) \geq m)$



a. $\mathsf{top}(f) > m$ b. $\mathsf{top}(f) = m$

Abbildung 3.5: Die zwei Fälle für $\mathsf{cf}(f,m) = (f_0, f_1)$.

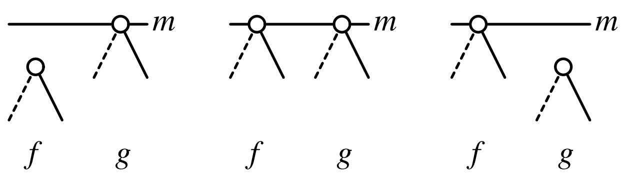


Abbildung 3.6: Fallunterscheidung bei BDD-Algorithmen mit zwei Argumenten.

- /McMillan, 1993b/ bzw. /McMillan, 1993a/,
- dem Tiger System von DEC,
- /Janssen, 1993/ bzw. /Janssen, 1996a/ und
- /Brace et al., 1990/

gibt es keine entsprechende Funktion, obwohl natürlich ähnliche Überlegungen dazu führen, daß nicht alle Fallunterscheidungen durchgeführt werden müssen.

Zur Vereinfachung trägt auch die Verwendung von Termen statt Graphen bei. Man kann hier Terme als Grunddatentyp verwenden und muß diese nicht durch Pascal-Verbunde wie in /Bryant, 1986/ darstellen. Dies erlaubt auch eine einfachere formale Verifikation dieser Algorithmen, was aber erst in Abschnitt 3.8.4 geschehen soll, da die hier vorgestellten Algorithmen ein Spezialfall der dort behandelten sind. Hier soll nur folgendes festgehalten werden:

Satz 3.13 (Korrektheit der Algorithmen „$\neg_{\mathsf{BDD}}$“ und „$\wedge_{\mathsf{BDD}}$“)
Sei $f,g \in \mathsf{ROBDD}$ und $A \in \mathcal{A}$ mit $\mathsf{range}(A) \supseteq \mathsf{var}(f,g)$, so gilt

$$\langle\!\langle \neg_{\mathsf{BDD}} f \rangle\!\rangle_A \equiv \neg \langle\!\langle f \rangle\!\rangle_A \qquad \langle\!\langle f \wedge_{\mathsf{BDD}} g \rangle\!\rangle_A \equiv \langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A$$

und $f \wedge_{\mathsf{BDD}} g,\ \neg_{\mathsf{BDD}} f \in \mathsf{ROBDD}$.

Beweis: Dieser Satz ist eigentlich ein Korollar zu Satz 3.19. Hier wird nur eine verkürzte Begründung gegeben. Zunächst terminieren die Algorithmen. Der Algorithmus für $\neg_{\mathsf{BDD}}$ trivialerweise und der Algorithmus für $\wedge_{\mathsf{BDD}}$, da bei jedem Rekursionsschritt durch cf mindestens ein Argument eine größere oberste Variable $\mathsf{top}(\cdot)$ hat, als das entsprechende Argument in der aufrufenden Prozedur und die Menge der Variablen nach oben beschränkt ist. Zwischenresultate werden „on the fly“ reduziert, so daß das Endergebnis auch reduziert ist (näheres entnehmen man Abschnitt 3.5). Weiter sind die obersten Variablen $\mathsf{top}(l)$ und $\mathsf{top}(r)$ der Zwischenergebnisse immer größer als m in der entsprechenden Prozedur oder l oder r sind Konstanten. Damit ist das Ergebnis jedes Aufrufes und somit auch das Gesamtergebnis immer geordnet. Die Äquivalenzaussagen folgen nun mit Gleichung (2.8) von Seite 15 und Lemma 3.12.b. ❑

Unter Verwendung der beiden in Form von Algorithmen vorgestellten Operationen „$\neg_{\mathsf{BDD}}$“ und „$\wedge_{\mathsf{BDD}}$“ ist es nun möglich, einen booleschen Ausdruck konstruktiv in einen ROBDD zu übersetzen (vgl. Abb. 2.2).

Definition 3.14 (Übersetzung von $\mathbb{B}^W$ nach ROBDD)

$$\|\cdot\|_A : \mathbb{B}^W \to \mathsf{ROBDD}$$

ist für $A \in \mathcal{A}$ eine partielle *Funktion, die für alle $s \in \mathbb{B}^W$ definiert ist mit $\mathsf{domain}(A) \supseteq \mathsf{free}(s)$, und zwar rekursiv durch*

$$\|0\|_A := 0, \qquad \|1\|_A := 1, \qquad \|x\|_A := 0 \cdots (i) - 1, \quad x \in W,\ A(x) = i$$

$$\|\neg s\|_A := \neg_{\mathsf{BDD}} \|s\|_A, \qquad \|s \wedge t\|_A := \|s\|_A \wedge_{\mathsf{BDD}} \|t\|_A$$

Nun kann man sich fragen, wie sich die Semantik eines aus einem booleschen Ausdruck gewonnenen BDD zum ursprünglichen booleschen Ausdruck verhält. Wie zu erwarten, sind beide booleschen Ausdrücke semantisch äquivalent, was der nächste Satz zeigt.

Satz 3.15 $\forall s \in \mathbb{B}^W,\ A \in \mathcal{A},\ \mathsf{domain}(A) \supseteq \mathsf{free}(s).\qquad \langle\!\langle \|s\|_A \rangle\!\rangle_A \equiv s$

Beweis: (durch Induktion über Termaufbau von s) Für $s \in \{0,1\}$ ist nichts zu zeigen. Sei nun $s = x \in W$. Dann gilt für $i := A(x)$

$$\langle\!\langle \|x\|_A \rangle\!\rangle_A = \langle\!\langle 0 \cdots (i) - 1 \rangle\!\rangle_A = \neg x \wedge \langle\!\langle 0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle 1 \rangle\!\rangle_A = \neg x \wedge 0 \vee x \wedge 1 \equiv x$$

Nun betrachte $\neg s$. Es gilt

$$\begin{array}{rcll} \langle\!\langle \|\neg s\|_A \rangle\!\rangle_A & = & \langle\!\langle \neg_{\mathsf{BDD}} \|s\|_A \rangle\!\rangle_A & \text{Definition} \\ & \equiv & \neg \langle\!\langle \|s\|_A \rangle\!\rangle_A & 3.13 \\ & \equiv & \neg s & \text{Induktionshypothese} + (2.6) \end{array}$$

und für $s \wedge t$ ist

$$\begin{array}{rcll} \langle\!\langle \|s \wedge t\|_A \rangle\!\rangle_A & = & \langle\!\langle \|s\|_A \wedge_{\mathsf{BDD}} \|t\| \rangle\!\rangle_A & \text{Definition} \\ & \equiv & \langle\!\langle \|s\|_A \rangle\!\rangle_A \wedge \langle\!\langle \|t\|_A \rangle\!\rangle_A & 3.13 \\ & \equiv & s \wedge t & \text{Induktionshypothese} + (2.7) \end{array}$$

❑

Mit Lemma 3.17 erhält man so die fundamentale Eigenschaft von ROBDDs als Korollar

Korollar 3.16 (Normalformeigenschaft von ROBDD**)** *Für alle* $s,t \in \mathbb{B}^W$ *gilt*

$$s \equiv t \quad \Rightarrow \quad \|s\|_A = \|t\|_A$$

für alle $A \in \mathcal{A}$, *mit* $\mathsf{domain}(A) \supseteq \mathsf{free}(s) \cup \mathsf{free}(t)$.

Beweis: Für $s \equiv t$ ist nach Satz 3.15

$$\langle\!\langle \|s\|_A \rangle\!\rangle_A \equiv s \equiv t \equiv \langle\!\langle \|t\|_A \rangle\!\rangle_A$$

Lemma 3.17.a liefert den Rest. ❑

Lemma 3.17 *Für alle* $f, g \in \mathsf{ROBDD}$ *(*$f = \hat{f}$ *und* $g = \hat{g}$*)*

a. $\langle\!\langle f \rangle\!\rangle_A \equiv \langle\!\langle g \rangle\!\rangle_A \ \Rightarrow\ f = g$

b. $f = f_0 \cdots (i) - f_1 \ \Rightarrow\ A^{-1}(i) \in \mathsf{rel}(\langle\!\langle f \rangle\!\rangle_A)$

Beweis: (durch simultane Induktion über Termaufbau von f und g) Für $f, g \in \{0, 1\}$ ist nichts zu zeigen. O.B.d.A. Sei nun $f = f_0 \cdots (i) — f_1$. Für b. nehme man $x := A^{-1}(i) \notin \mathsf{rel}(\langle\!\langle f \rangle\!\rangle_A)$ an. Dann gilt zunächst

$$\langle\!\langle f \rangle\!\rangle_A \{x \mapsto 0\} \equiv \langle\!\langle f \rangle\!\rangle_A \{x \mapsto 1\}$$

Die Definition von „$\langle\!\langle \cdot \rangle\!\rangle_A$“ eingesetzt, ergibt

$$\langle\!\langle f_0 \rangle\!\rangle_A \equiv \neg 0 \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee 0 \wedge \langle\!\langle f_1 \rangle\!\rangle_A \equiv \neg 1 \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee 1 \wedge \langle\!\langle f_1 \rangle\!\rangle_A \equiv \langle\!\langle f_1 \rangle\!\rangle_A$$

Die Induktionshypothese Teil a. ergibt nun $f_1 = f_0$ im Widerspruch zu $f = \hat{f}$. Dies zeigt den Induktionsschritt für Teil b. Für den Beweis des Induktionsschrittes für Teil a. gelte $\langle\!\langle f \rangle\!\rangle_A \equiv \langle\!\langle g \rangle\!\rangle_A$. Ist $g \in \{0, 1\}$, dann gilt $\mathsf{rel}(\langle\!\langle g \rangle\!\rangle_A) = \emptyset$. Das ergibt mit dem Bewiesenen $\mathsf{rel}(\langle\!\langle g \rangle\!\rangle_A) \neq \mathsf{rel}(\langle\!\langle f \rangle\!\rangle_A)$ im Widerspruch zu Satz 2.14 und der Voraussetzung. Sei also nun weiter $g = g_0 \cdots (j) — g_1$ und $y := A^{-1}(j)$. Um zu zeigen, daß $i = j$, nehme (o.B.d.A. $i \leq j$) $i < j$ an. Da g geordnet ist und somit $x \notin A^{-1}(\mathsf{var}(g))$, folgt

$$x \notin \mathsf{rel}(\langle\!\langle g \rangle\!\rangle_A) \subseteq \mathsf{free}(\langle\!\langle g \rangle\!\rangle_A) = A^{-1}(\mathsf{var}(g))$$

Die Induktionshypothese Teil b. liefert aber $x \in \mathsf{rel}(\langle\!\langle f \rangle\!\rangle_A)$, was mit der Voraussetzung einen Widerspruch zu Satz 2.14 ergibt. Damit ist also $x = y$, und die Definition von „$\langle\!\langle \cdot \rangle\!\rangle_A$“ ergibt

$$\neg x \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle f_1 \rangle\!\rangle_A \equiv \langle\!\langle f \rangle\!\rangle_A \equiv \langle\!\langle g \rangle\!\rangle_A \equiv \neg x \wedge \langle\!\langle g_0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle g_1 \rangle\!\rangle_A$$

Nun sei ρ_W eine beliebige Variablenbelegung

$$\begin{array}{rcll}
[\![\langle\!\langle f_0 \rangle\!\rangle_A]\!]\rho_W & = & [\![\neg 0 \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee 0 \wedge \langle\!\langle f_1 \rangle\!\rangle_A]\!]\rho_W & \text{Definition} \\
& = & [\![(\neg x \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle f_1 \rangle\!\rangle_A)\{x \mapsto 0\}]\!]\rho_W & \\
& = & [\![\neg x \wedge \langle\!\langle f_0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle f_1 \rangle\!\rangle_A]\!]\rho_W\{x \mapsto 0\} & 2.11 \\
& = & [\![\langle\!\langle f \rangle\!\rangle_A]\!]\rho'_W & \rho'_W := \rho_W\{x \mapsto 0\} \\
& = & [\![\langle\!\langle g \rangle\!\rangle_A]\!]\rho'_W & \text{Voraussetzung} \\
& = & [\![(\neg x \wedge \langle\!\langle g_0 \rangle\!\rangle_A \vee x \wedge \langle\!\langle g_1 \rangle\!\rangle_A)\{x \mapsto 0\}]\!]\rho_W & 2.11 \\
& = & [\![\neg 0 \wedge \langle\!\langle g_0 \rangle\!\rangle_A \vee 0 \wedge \langle\!\langle g_1 \rangle\!\rangle_A]\!]\rho_W & \\
& = & [\![\langle\!\langle g_0 \rangle\!\rangle_A]\!]\rho_W &
\end{array}$$

Analog zeigt man die Äquivalenz von $\langle\!\langle f_1 \rangle\!\rangle_A$ und $\langle\!\langle g_1 \rangle\!\rangle_A$. Das ergibt mit der Induktionshypothese Teil a. beidesmal $f_0 = g_0$ und $f_1 = g_1$, so daß auch $f = g$. ❑

Aus diesem Lemma erhält man zusätzlich

Korollar 3.18 *Für* $f \in$ ROBDD, $A \in \mathcal{A}$, $\mathsf{range}(A) \supseteq \mathsf{var}(f)$ *gilt*

$$\mathsf{rel}(\langle\!\langle f \rangle\!\rangle_A) = A^{-1}(\mathsf{var}(f)),$$

was bedeutet, daß in einem BDD genau die Variablen enthalten sind, von denen der Wert des entsprechenden booleschen Ausdrucks abhängt. Die Ordnung der BDDs in Lemma 3.17 ist eine notwendige Voraussetzung, wie folgendes Beispiel zeigt (vgl. Abb. 3.7.a):

$$\langle\!\langle (0 \cdots (1) — 1) \cdots (2) — 1 \rangle\!\rangle_A \equiv x_1 \vee x_2 \equiv \langle\!\langle (0 \cdots (2) — 1) \cdots (1) — 1 \rangle\!\rangle_A$$

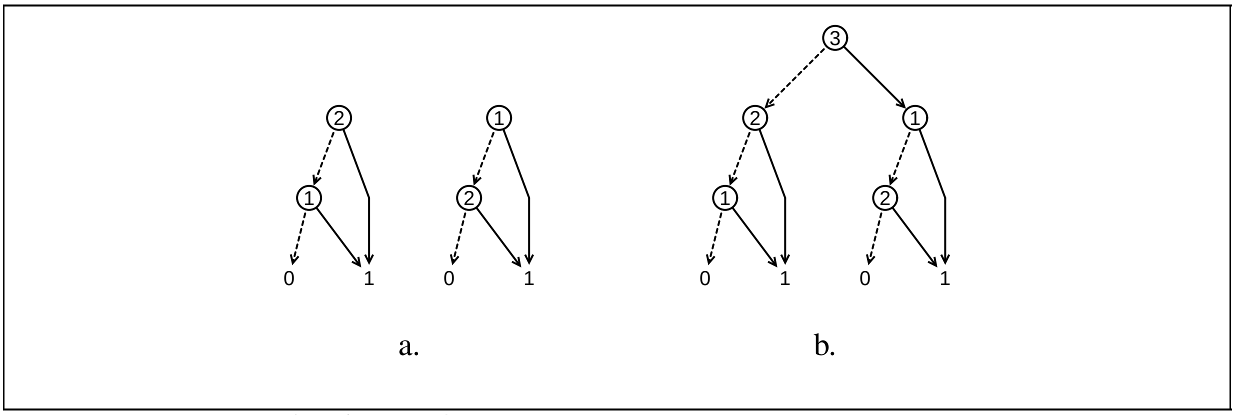


Abbildung 3.7: Beispiele für FBDDs (als Graphen)

wobei $W = \{x_0, x_1, \dots\}$ und $A \in \mathcal{A}$ mit $A(x_i) := i$. Dies falsifiziert Teil a. für nicht geordnete BDDs. Für Teil b. reicht ein leicht modifiziertes Beispiel aus (vgl. Abb. 3.7.b)

$$\langle\!\langle ((0 \cdots 1 - 1) \cdots 2 - 1) \cdots 3 - ((0 \cdots 2 - 1) \cdots 1 - 1) \rangle\!\rangle_A \equiv x_1 \vee x_2$$

Diese Beispiele stammen sogar aus der eingeschränkteren Klasse der „Read-Once-BDDs“, bei denen auf jedem Pfad von der Wurzel zu einer Konstanten jede Variable höchstens einmal auftreten darf.

Programm aus $\mathcal{F}$	$\xrightarrow{\text{Kompilation}}$	Programm aus $\mathcal{F}_{\text{SCAM}}$
Abstraktes Abarbeitungsmodell		SCAM

Abbildung 3.8: Kompilation von $\mathcal{F}$ nach $\mathcal{F}_{\text{SCAM}}$.

3.5 Eine abstrakte Maschine für BDD-Algorithmen

Die Algorithmen aus Abb. 3.3 wurden in einer pseudo-funktionalen Programmiersprache $\mathcal{F}$ beschrieben. In diesem Unterabschnitt sollen die Grundzüge einer abstrakten Maschine SCAM (vgl. Abb. 3.9) für ***S**haring and **C**aching **A**bstract **M**achine* vorgestellt werden, die diese Algorithmen effizient implementiert.

Abstrakte Maschinen können dazu benützt werden, ein unter Umständen kompliziertes Abarbeitungsmodell (die operationale Semantik) einer höheren Programmiersprache auf einem hohem Abstraktionsniveau zu beschreiben (vgl. /Kogge, 1991/). Dabei wird eine Abbildung eines Programmes der höheren Programmiersprache auf ein Programm der abstrakten Maschine angegeben. Diese Abbildung wird auch *Kompilation* genannt. Bei der SCAM spielt $\mathcal{F}$ die Rolle der höheren Programmiersprache und die SCAM selbst versteht nur Befehle der Programmiersprache $\mathcal{F}_{\text{SCAM}}$. Dies wird noch einmal in Abb. 3.8 verdeutlicht.

Die Kompilation von $\mathcal{F}$ wird hier nicht näher betrachtet. Statt dessen stehen die speziellen Befehle von $\mathcal{F}_{\text{SCAM}}$ und Implementierungsfragen der SCAM im Mittelpunkt.

3.5.1 „Worst-Case" Komplexität einer naiven Implementierung

Betrachtet man den Algorithmus für „$\wedge_{\mathsf{BDD}}$" aus Abb. 3.3.b, so stellt man fest, daß die Anzahl rekursiver Aufrufe $\mathtt{nc}(\wedge_{\mathsf{BDD}}, n)$ für $n := |f| + |g|$ bei Berechnung von $\wedge_{\mathsf{BDD}}(f,g)$ auf einer normalen abstrakten Maschine für funktionale Programmiersprachen folgenden Ungleichungen genügt

$$\mathtt{nc}(\wedge_{\mathsf{BDD}}, n) \leq \begin{cases} 1 & \text{für } n \leq 2 \\ 2 \cdot \mathtt{nc}(\wedge_{\mathsf{BDD}}, n-1) & \text{sonst} \end{cases} \tag{3.3}$$

$$\mathtt{nc}(\wedge_{\mathsf{BDD}}, n) \geq \begin{cases} 1 & \text{für } n \leq 2 \\ 2 \cdot \mathtt{nc}(\wedge_{\mathsf{BDD}}, n-2) & \text{sonst} \end{cases} \tag{3.4}$$

Als untere und obere Schranke in „O"-Notation (vgl. Abschnitt A.6) erhält man dann

$$\mathtt{nc}(\wedge_{\mathsf{BDD}}, n) = \Omega(2^{\lfloor (n-1)/2 \rfloor}) \quad \text{und} \quad \mathtt{nc}(\wedge_{\mathsf{BDD}}, n) = \mathrm{O}(2^{n-2})$$

d. h. der Algorithmus verhält sich immer exponentiell in der Größe der Argumente. Auf der anderen Seite stellt man fest, daß es sehr viel weniger Kombinationen von Argumenten gibt, die wirklich bei der Auflösung der Rekursion vorkommen können, nämlich höchstens $|f| \cdot |g| = \mathrm{O}(n^2)$. Außerdem ist der beschriebene Algorithmus „seiteneffekt"-frei, wie es sich für eine saubere funktionale Programmiersprache gehört. Daraus ergibt sich die Konsequenz, daß bei gleichen Argumenten der Algorithmus auch das gleiche Ergebnis liefern muß. Man sollte also deshalb mehrfach vorkommende identische Aufrufe nur einmal ausführen (ähnlich der algebraischen Reduktion selbst!).

3.5.2 Ergebnisspeicher (cache)

Deshalb sieht die abstrakte Maschinen SCAM einen *Ergebnisspeicher* (engl. cache) vor. Sie merkt sich zu jedem rekursiven Aufruf die aktuellen Parameter und bei der Rückkehr (`return`) von einem rekursiven Aufruf wird das Ergebnis zusammen mit den Parametern und dem Namen der aufgerufenen Funktion im Ergebnisspeicher abgelegt. Bevor nun der rekursive Aufruf (`call`) tatsächlich ausgeführt wird, schlägt die SCAM im Ergebnisspeicher nach, ob für diese Funktion mit denselben Parametern nicht schon ein Ergebnis vorhanden ist. Im positiven Fall kann sie unter Verwendung des schon berechneten Ergebnises mit dem nächsten Befehl fortfahren. Ansonsten muß der Rumpf der Definition der aufgerufenen Funktion tatsächlich ausgeführt werden.

Als Optimierung wird verlangt, daß Nachschlagen im Ergebnisspeicher gewisse semantische Eigenschaften der berechneten Funktionen berücksichtigt. Als Beispiel diene die Kommutativität von „$\wedge_{\mathsf{BDD}}$“, die aus der Kommutativität von „$\wedge$“ in $\mathbb{B}^W$ nach Korollar 3.16 folgt.

$$f \wedge_{\mathsf{BDD}} g \;=\; g \wedge_{\mathsf{BDD}} f \tag{3.5}$$

Hier hat man die Situation, daß bei Berechnung von $f \wedge_{\mathsf{BDD}} g$ auch das Ergebnis der Berechnung von $g \wedge_{\mathsf{BDD}} f$ verwendet werden kann, da die Kommutativität gerade aussagt, daß das Ergebnis in beiden Fällen dasselbe ist. Ist also das Ergebnis der Berechnung des zweiten Termes schon im Ergebnisspeicher vorhanden, so sollte beim Nachschlagen für den ersten Term dieses als Resultat geliefert werden. Weitere semantische Eigenschaften, die hier berücksichtigt werden sollten, finden sich bei den später vorgestellten Algorithmen.

Wo diese zusätzliche Funktionalität angesiedelt ist, ob in der Implementierung des Ergebnisspeichers oder in der SCAM, ist prinzipiell egal. Der Einfachheit halber sei die erste Möglichkeit gewählt. Man könnte sogar soweit gehen, daß der Ergebnisspeicher auch die Basisfälle der einzelnen Algorithmen kennt, so daß auf Abfrage des Basisfall in den vorgestellten Algorithmen zum Teil verzichtet werden könnte. Man hätte dann rekursiv beschriebene Algorithmen ohne Rekursionsanfang! Aus Gründen der Übersichtlichkeit wird diese Option hier aber nicht wahrgenommen.

3.5.3 Termhalde (heap)

Die SCAM ist auch für die Generierung und Speicherung von Termen aus FBDD zuständig. Dazu verwendet sie eine Termhalde (engl. heap) kurz Halde, wie das bei abstrakten Maschinen für PROLOG (/Warren, 1983, Kogge, 1991/) oder für funktionale Sprachen (/Jones, 1987/) üblich ist. Programme der SCAM können nur über Referenzen (Zeiger) auf die eigentlichen Termknoten zugreifen. Das Erzeugen und Löschen von Termknoten ist der SCAM selbst vorbehalten.

Was die Halde der SCAM von den Halden dieser abstrakten Maschinen unterscheidet, ist die Zusicherung, daß kein Term zweimal vorhanden ist. Dazu muß die SCAM bei Aufforderung eines SCAM-Programmes, einen neuen Termknoten zu generieren (das wäre der SCAM Befehl `generate`($l \cdots \textcircled{m} - r$)), die Halde durchsuchen, um festzustellen, ob der neu zu generierende Termknoten nicht schon vorhanden ist. Ist dies der Fall, dann wird nur eine Referenz auf den schon vorhandenen Knoten zurückgeliefert. Ansonsten wird tatsächlich die Halde vergrößert und eine Referenz auf einen neu generierten Knoten zurückgeliefert.

Man beachte, daß hier auch die Reduktion der „Elimination redundanter Kanten“ durchgeführt wird (vgl. Abb. 3.2.b). Hierzu wird sofort l (oder äquivalent r) zurückgegeben, falls $l = r$ bei Bearbeitung des Befehls `generate`($l \cdots \textcircled{m} - r$) gilt.

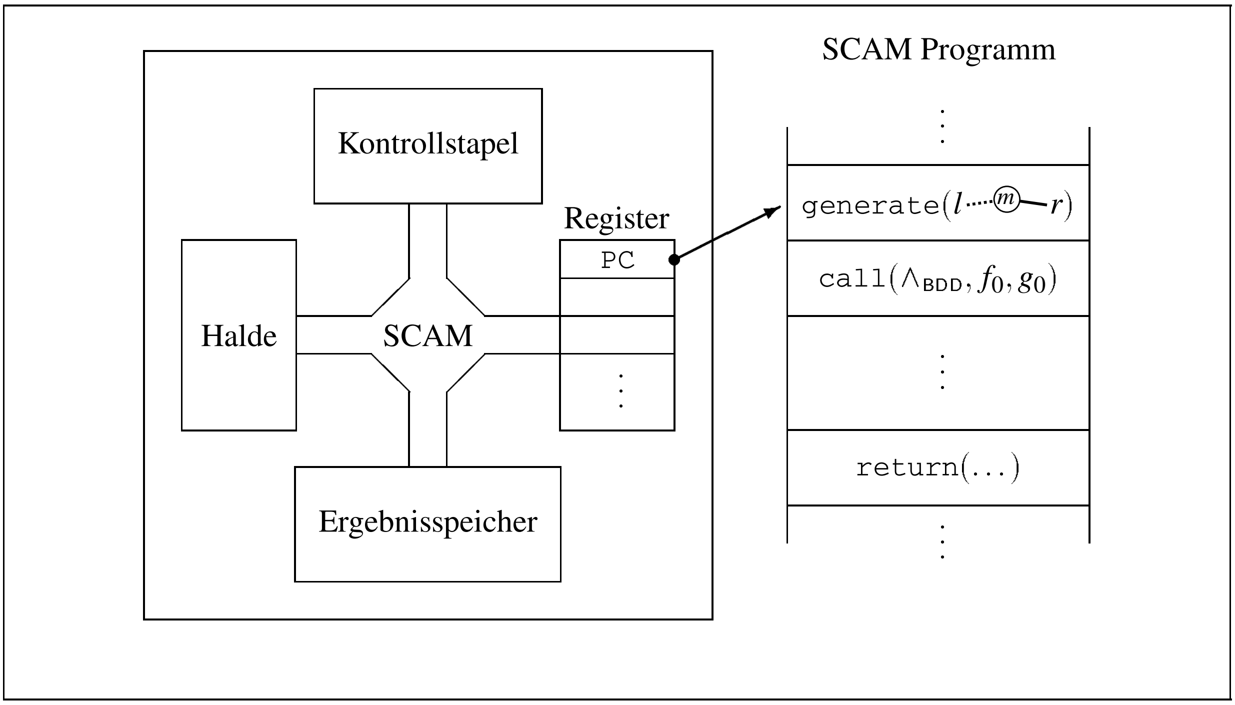


Abbildung 3.9: Architektur der SCAM

3.5.4 Die Architektur der SCAM im Überblick

Neben der Halde und dem Ergebnisspeicher braucht die SCAM noch einen Kontrollstapel für die Verwaltung der Rekursion und einen Registersatz für die lokalen Variablen in einer Funktion, die durch das „**where**" Konstrukt definiert werden können. Die weiteren Befehle der SCAM, z. B. zur Verarbeitung von Kontrollstrukturen, entsprechen denen anderer abstrakter Maschinen (vgl. /Kogge, 1991/) und sollen hier auch nicht weiter behandelt werden. Eine Übersicht über die Architektur der SCAM findet man in Abb. 3.9. Ebenso sind dort die drei wichtigsten speziellen Befehle aus $\mathcal{F}_{\text{SCAM}}$ aufgeführt:

- `generate`($l \cdots \textcircled{m} - r$)
- `call`(...)
- `return`(...)

Der erste wird bei Übersetzung des Ausdrucks „$l \cdots \textcircled{m} - r$" erzeugt (vgl. Abb. 3.3). Der zweite bei Übersetzung eines rekursiven Aufrufes einer Funktion und der dritte beim Rücksprung aus einer Funktion.

3.5.5 Implementierung der SCAM

In diesem Unterabschnitt werden Implementierungsaspekte der SCAM betrachtet. Diese sind natürlich für eine nicht SCAM-basierte Implementierung einer BDD-Bibliothek dennoch von großer Wichtigkeit.

Die SCAM stellt zwei nichttriviale Grundoperationen zur Verfügung. Das ist einmal das Nachschlagen im Ergebnisspeicher und zweitens die Suche nach schon vorhandenen Termknoten in der Halde. Beides sind Suchaufgaben, die sehr häufig anfallen, und auf einer sich ständig ändernden Datenmenge durchgeführt werden. Eine einfache Implementierung, die keine zusätzliche Struktur auf den Daten annimmt, wäre sehr teuer, da ja im Mittel immer die Hälfte

```
public class Object {
   ...
   public int hashCode();
   ...
}
```

Abbildung 3.10: Die Hashfunktion für allgemeine Objekte in JAVA.

des ganzen Ergebnisspeichers bzw. der Halde durchsucht werden müßte. Die schnellste Variante besteht darin, jeweils eine Tabelle für den Ergebnisspeicher und die Halde aufzustellen, die indiziert wird durch die Argumente, nach denen gesucht wird. Bei der Halde wäre dies eine dreidimensionale Tabelle, d. h. eine partielle Funktion

$$\mathtt{tab}_{\mathrm{Halde}}: \mathsf{Ref}(\mathsf{ROBDD}) \times \mathbb{N} \times \mathsf{Ref}(\mathsf{ROBDD}) \rightarrow \mathsf{Ref}(\mathsf{ROBDD})$$

wobei $\mathsf{Ref}(\mathsf{ROBDD})$ die Menge der Referenzen auf Termknoten der Halde darstellt. Würde man diese Tabelle für eine Teilmenge von $\mathbb{N}$ tatsächlich als Vektor (engl. array) aufstellen, so wäre dies wegen des wahrscheinlich hohen Anteils von nicht vorhandenen Knoten viel zu teuer. Nun könnte man wie bei der Speicherung von Zustandsübergangstabellen von prädikativen Parsern (vgl. /Aho et al., 1986/) zu einer „verdichteten“ Speicherung von $\mathtt{tab}_{\mathrm{Halde}}$ übergehen. In der Praxis hat sich stattdessen bei BDD-Bibliotheken die Verwendung von „Hashing“-Techniken als Kompromiß zwischen Suchgeschwindigkeit und Platzverbrauch als die günstigste Lösung erwiesen. Dies ist auch die Vorgehensweise von allen dem Autor bekannten BDD-Bibliotheken (s. Seite 53).

Was den Ergebnisspeicher angeht, gilt eine ähnliche Argumentation. Zusätzlich hat man aber in diesem Fall die Möglichkeit, den Platz für die Speicherung der Hashtabelle frei zu wählen. So kann man in einem Extremfall tatsächlich alle Ergebnisse speichern, oder im anderen Extremfall gar keine. In /Brace et al., 1990/ wurde hierzu folgende Vorgehensweise vorgestellt. Zunächst sollte die Größe der verwendeten Hashtabelle für den Ergebnisspeicher relativ zu der Anzahl existierender Termknoten wachsen (in der erwähnten Arbeit hat sich ein lineares Verhältnis mit Faktor $1/4$ als günstig erwiesen). Zweitens wurde vorgeschlagen, daß man die Kollisionen, die beim „Hashing“ entstehen, nicht auflöst, sondern den Eintrag in der Tabelle durch das neu eingefügte Datum überschreibt („write-through-cache“). Dadurch wächst die Tabelle nur, wenn auch die Anzahl existierender Termknoten wächst.

In der BDD-Bibliothek von /Long, 1994/ wurde dies noch dahingehend erweitert, daß es in der Hashtabelle pro Hashwert immer eine endliche Anzahl von Ausweicheinträgen für Kollisionen gibt (in dem erwähnten System sind das genau 2). Weiter werden die zu speichernden Ergebnisse noch priorisiert. Dafür wird abgeschätzt, ob ihre Berechnung lange dauert oder nicht.[2] Nun werden bei einer Kollision nur noch Einträge mit niedriger oder gleicher Priorität überschrieben.

Ein weiteres kleineres Problem besteht in der Implementierung der Hashfunktion. Am einfachsten tut man sich damit, wenn man für die SCAM eine Implementierungssprache wählt, bei der Zeigerarithmetik erlaubt ist. Für Assembler, C oder C++ trifft das natürlicherweise zu. Für andere Sprachen wie PASCAL ist das schwieriger. Interessanterweise hat diese Problematik aus ähnlichen Überlegungen aber auch schon auf den Entwurf moderner objekorientier Sprachen wie JAVA abgefärbt. So stellt in JAVA (s. /Flanagan, 1996/) die oberste Klasse `Object` die

[2]Zum Beispiel wird Rumpf des Algorithmus $\mathsf{relProd}$ aus Abb. 3.25 eventuell „$\vee_{\mathrm{BDD}}$“ aufgerufen. Das Speichern von Ergebnisen dieser Operation kann nicht soviel Zeit sparen, wie das Speicher von Ergebnis von $\mathsf{relProd}$.

```
class BDDNode {
  int _var;
  BDDIdx _left, _right;
  ...
public:
  int hash()
  {
    return ROL(_left.hash(), 4) ^ _right.hash() ^ (_var << 2);
  }
  ...
};
```

Abbildung 3.11: `BDDNode::hash` aus der BDD-Bibliothek `BDD`.

Methode „`hashCode`“ zur Verfügung (s. Abb. 3.10). Diese Methode liefert zu einem Objekt einen Hashwert, von dem angenommen wird, daß er für verschiedene Objekte auch verschieden ist. Läßt sich dies in einer Implementierungssprache realisieren, so ist man auf der sicheren Seite. Man kann sich natürlich auch darauf zurückziehen, nur Indizes statt Zeiger zu verwenden. Wobei dann die Halde als Vektor (array) implementiert wird, und eine Referenz auf einen Termknoten nun einfach der Index des Termknotens in diesem Vektor ist.

In beiden Fällen stellt sich dennoch die Frage, wie die Hashwerte der einzelnen Referenzen mit dem Index der Variablen (l, r und m bei `generate`($l \cdots m - r$)) auf einen gemeinsamen Hashwert umgerechnet werden. Ähnlich wie das bei der Berechnung von Hashwerten für Zeichenketten der Fall ist (/Aho et al., 1986/), kann hier nur Austesten die im Mittel beste Methode ans Tageslicht fördern. Als Beispiel soll hier in Abb. 3.11 die Hashfunktion der in C++ implementierten BDD-Bibliothek `BDD` (vgl. Seite 65) vorgestellt werden. Die Methode `BDDNode::hash` berechnet den Hashwert eines BDD-Knotens (oben Termknoten genannt). Dazu verwendet sie den Wert der Variable `_var`, den Hashwert des 0-Nachfolgers `_left` und des 1-Nachfolgers `_right`. Nun werden diese drei Werte exklusiv verodert (in C++ ist das „ˆ“) nachdem der Hashwert des 0-Nachfolgers um 4 Bits nach links rotiert und der Wert der Variable um 2 nach links verschoben wurde. Experimente mit anderen Parametern statt 4 und 2 ergaben im Mittel mehr Kollisionen.

In diesem Zusammenhang kann auch geklärt werden, wie man in einer Implementierung sicherstellt, daß der Ergebnisspeicher z. B. die Kommutativität von $\wedge_{\texttt{BDD}}$ berücksichtigt. Dies geschieht einfach dadurch, daß einer der beiden Aufrufe in Gleichung (3.5) von Seite 60 bevorzugt wird. Es wird o.B.d.A. immer derjenige ausgewählt, dessen linkes Argument den kleineren Hashwert hat. Nur dieser ausgewählte wird dann abgespeichert oder gesucht.

3.5.6 Speicherbereinigung

Bei unendlich viel Speicher, was implizit bis jetzt die Voraussetzung war, kann man sich den Luxus leisten, nicht mehr benötigten, aber belegten Speicher nicht mehr wiederzuverwenden. In der Realität hat man aber nicht unbegrenzt viel Speicher und auch gerade die Anwendungen von BDDs erweisen sich als sehr „speicherhungrig“. So wurde in /Eiríksson und McMillan, 1995/ eine Workstation mit 1.5 GByte Hauptspeicher benötigt, um ein Hardwareprotokoll zur „cache coherence“ mit dem SMV-System zu verifizieren.

Auf der anderen Seite ist Speicherbereinigung kein Problem, wenn Objekte nur einmal referenziert werden. In diesem Fall kann der Speicherplatz eines referenzierten Objektes sofort wieder wiederverwendet (engl. „recycled“) werden, wenn das einzige Objekt, das die Referenz besitzt, selbst gelöscht wird. Wenn nun aber manche Objekte von mehreren anderen Objekten referenziert werden, muß man spezielle Algorithmen zur Speicherbereinigung (eng. „garbage collection“) einsetzen.

Hierzu gibt es zwei Varianten (/Knuth, 1973/ und /Jones und Lins, 1996/). Die eine benützt „Zählen von Referenzen“ (engl. „reference counting“) bei der im Objekt selbst die Anzahl der Referenzen auf dieses Objekt gezählt werden. Wird solch ein Zähler auf 0 gesetzt, so wird das Objekt gelöscht und der belegte Speicherplatz kann sofort wiederverwendet werden. Dies verkürzt die Reaktionszeit des Gesamtsystems, was aber für die üblichen Anwendungen von BDDs unwichtig erscheint. Der Nachteil ist, daß dieses Zählen Zeit und Platz für den Zähler kostet.

Die zweite Methode, auch indirekte Methode genannt, die als Spezialfälle den „mark and sweep“ und den „copying garbage collection“ Algorithmus vereinigt, wird in den Speicherallokationsalgorithmus integriert. Wenn der gesamte Speicher belegt ist, dann werden in einer ersten Phase (der „mark“-Phase) alle noch „gültigen“ Objekte markiert. In einer zweiten Phase (der „sweep“- bzw. „copying“-Phase) wird nun der gesamte Speicher nach unmarkierten Speicherplätzen durchsucht. Beim „mark and sweep“ können diese nun wieder dem Freispeicher hinzugefügt werden. Bei einem „copying garbage collector“ werden statt dessen alle markierten Objekte hintereinander in einen zweiten freigehaltenen Speicherbereich kopiert, so daß am Ende dieses kopierten Bereiches wieder neuer Freispeicher vergeben werden kann.

Die meisten[3] Implementierungen von BDD-Bibliotheken verwenden die erste Methode, was wohl auf das Papier /Brace et al., 1990/ zurückzuführen ist. In diesem Papier wurde aber auch schon auf das Problem hingewiesen, daß der Ergebnisspeicher bei sofortiger Wiederverwendung von Speicherplatz falsche Ergebnisse beinhalten kann, weshalb der Speicherplatz von „toten“ Knoten (solche Knoten, deren Referenzenzähler auf 0 gesunken ist), doch nicht sofort frei gegeben werden kann. Man kann den Speicherplatz von toten Knoten erst dann wiederverwenden, wenn man gleichzeitig den Ergebnisspeicher bereinigt, d. h. alle Einträge löscht, bei denen als Argument oder Ergebnis eines Aufrufes ein toter Knoten beteiligt ist. Dies kann man vernünftigerweise nur durch lineares Durchsuchen des Ergebnisspeichers erreichen. Damit ist das Argument der sofortigen Wiederverwendung von nicht mehr benötigten Knoten hinfällig.

Ein weiterer Grund für die fast ausschließliche Verwendung der direkten Methode (Referenzenzählen) besteht darin, daß bei einfachen imperativen Programmiersprachen, wie z. B. C, als Implementierungssprache die Schnittstelle zur BDD-Bibliothek vereinfacht wird. Dies stimmt mit den Betrachtungen in /Jones und Lins, 1996/ überein. Hier muß der Benutzer der Bibliothek nur immer sicherstellen, daß er selbst keine „Speicherlecks“ erzeugt, also Zeiger (Referenzen) auf BDD-Knoten, die nicht mehr benötigt werden, der BDD-Bibliothek als solche mitteilt.

Wenn die indirekte Methode verwendet wird, kann es während der Ausführung einer beliebigen Prozedur der BDD-Bibliothek zum Speicherüberlauf kommen und eine Speicherbereinigung muß gestartet werden. Dazu muß die Speicherbereinigungsroutine alle Zeiger (Referenzen) des Benutzerprogramms kennen, die noch auf einen gültigen BDD-Knoten zeigen. Die Konsequenz ist also, daß der Benutzer der Bibliothek einen *Iterator* (vgl. /Gamma et al., 1995/) über alle gültigen BDD-Knoten zur Verfügung stellen muß, was die Komplexität der Schnittstelle zur Bibliothek unnötigerweise vergrößert. Es genügt natürlich auch ein Behälter (engl. „container“), der einen Iterator der verlangten Art generieren kann (vgl. /Gamma et al., 1995/). Dies bedeutet, daß der Benutzer eine zusätzliche Datenstruktur manipulieren muß, oder eine zusätzli-

[3]Unter den Systemen von Seite 53 benützt nur das SMV System die indirekte Methode (genauer „mark and sweep“).

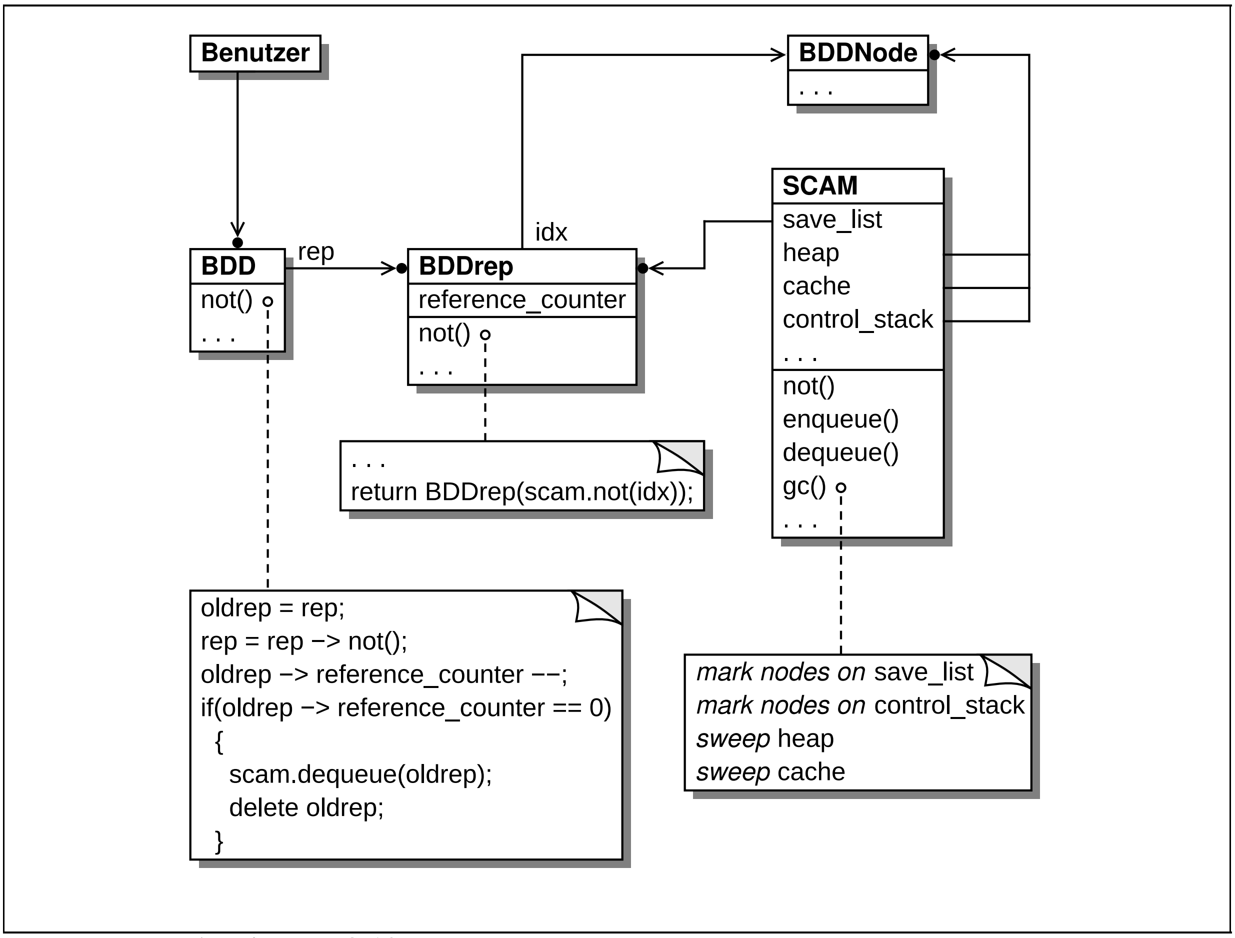


Abbildung 3.12: Schnittstellenschichten der BDD-Bibliothek `BDD`

che Indirektionsstufe vor die BDD-Bibliothek geschaltet wird. Ersteres ist nicht wünschenswert und die zweite Lösung läßt sich z. B. in C nur mit Mühe implementieren.

Das SMV-System benützt die indirekte Methode. Obwohl manche der später vorgestellten Algorithmen in der BDD-Bibliothek des SMV Systems nicht implementiert sind (z. B. compose oder ite_{BDD}), tritt auch hier die Schnittstellenproblematik zutage. Der oben erwähnte Kontainer entspricht im SMV-System einer globalen Liste (`save_list`), in der alle gültigen BDD-Knoten eingetragen werden. In allen Prozeduren, in denen BDDs manipuliert werden, muß deshalb ständig diese Liste auf aktuellen Stand gebracht werden, was einen Mehraufwand bei der Implementierung dieser Prozeduren erforderte. Darüber hinaus können sich so sehr leicht schwer zu erkennende Speicherlecks oder noch schlimmer „Zeiger ins Nirwana“ einschleichen, wenn man eben bei diesem Ein- und Austragen in die `save_list` einen Fehler macht.

Zusätzlich entsteht dabei das Problem, daß im SMV-System, wie bei all den betrachteten BDD-Bibliotheken (vgl. S. 53), die Rekursion der BDD-Algorithmen durch rekursive Aufrufe von C Prozeduren implementiert wird. Dies bedeutet aber, daß die Zeiger, die als lokale Variable auf dem C Kontrollstapel liegen, nicht direkt zugreifbar sind. Diese müssen also beim rekursiven Abstieg immer in die `save_list` ein- und ausgetragen werden, damit eine in einer größeren Rekursionstiefe ausgelöste Speicherbereinigung nicht versehentlich eine in einer höheren Prozedurschachtel später bei der Rückkehr von der Rekursion weiter verwendete Referenz unberücksichtigt läßt.

In der SCAM-basierten, prototypischen[4] Implementierung der BDD-Bibliothek `BDD` wird

[4] Diese vom Autor begonnene Implementierung hat das Ziel, eine Verbesserung gegenüber den besten frei

die Rekursion iterativ aufgelöst, so daß die Zeiger auf dem explizit manipulierten Kontrollstapel (der SCAM – nicht von C) direkt von der Speicherbereinigungsroutine durchlaufen werden können. Gegenüber einer echt rekursiven Implementierung ergab sich dadurch ein Geschwindigkeitsgewinn von 15-30% bei einfachen Beispielen.

Die Überladung der Schnittstelle von `BDD` mit einem vom Benutzer bereitzustellenden Iterator konnte durch die oben erwähnte Einführung einer Indirektionsstufe vermieden werden. Dazu bot sich das C++ Konzept der „handle class“ aus /Coplien, 1992/ (bzw. des „bridge pattern“ aus /Gamma et al., 1995/, siehe auch Abschnitt 5.2.1 und insbesondere Abb. 5.2) an, das seinerseits wiederum Referenzenzähler benützt. Hier werden „externe“ Referenzen gezählt, d. h. Referenzen vom Benutzer der Bibliothek auf BDD-Knoten. Von denen gibt es um mehrere Größenordnungen weniger als interne Referenzen, so daß der Zeitverbrauch beim Bearbeiten dieser Zähler nicht ins Gewicht fällt.

Als Erweiterung werden schließlich noch alle „handles“ in einer Liste (`save_list`) verkettet, damit die Speicherbereinigungsroutine die externen Referenzen ablaufen kann. Dies entspricht der Vorgehensweise des SMV-Systems, wobei aber hier die Referenzen in lokalen Variablen in höheren Rekursionsstufen nicht in die Liste eingetragen werden brauchen. Eine graphische Darstellung dieses Konzeptes in der Diagramm Notation von /Gamma et al., 1995/ findet sich in Abb. 3.12 (siehe auch Abschnitt A.5).

Für den Benutzer der BDD-Bibliothek ist nur die „handle“-Klasse „**BDD**“ sichtbar. Jedem Objekt der Klasse „**BDD**“ ist eine Repräsentation, d. h. eine Instanz (=Objekt) der Klasse „**BDDrep**“ zugeordnet. Aber dieselbe Repräsentation kann von mehreren „handles“ benützt werden.

Eine Operation, z. B. „not()“, auf ein „**BDD**“ Objekt wird auf die dazugehörige Operation der Repräsentation abgebildet, wobei der Referenzenzähler entsprechend angepaßt wird.

Ein Objekt der Klasse „**BDDrep**“ hat eine echte Referenz auf einen BDD-Knoten („**BDDNode**“) auf der Halde („heap“) der „scam“. Weiter sind all dies Objekte der Klasse „**BDDrep**“ in der Liste „save_list“ eingetragen, so daß diese „externen“ Referenzen bei einer Speicherbereinigung („gc“) zugänglich sind. Diese Liste gehört der „singleton“-Klasse „**SCAM**“ (zum Begriff der „singleton“-Klasse siehe /Gamma et al., 1995/) mit einziger Instanz „scam“. Sie könnte in der Klasse „**BDDrep**“ verwaltet werden. Aus Darstellungsgründen ist die Verwaltung in die „**BDD**“-Klasse verlegt worden („scam.dequeue(oldrep)“). Nicht eingezeichnet in dem Diagramm aus Abbildung 3.12 sind die Referenzen von BDD-Knoten auf andere BDD-Knoten, die der Nachfolgerrelation entsprechen.

In den Schnittstellenarchitekturen der anderen BDD-Bibliotheken sind die äußeren und inneren Referenzen vermischt. Sowohl die ausschließliche Verwendung der indirekten Methode zur Speicherbereinigung („mark and sweep“), wie das vom SMV-System betrieben wird, oder ausschließliches Referezenzählen, wie das die übrigen BDD-Bibliotheken auf Seite 53 praktizieren, stellen somit keine optimale Lösung zur Speicherbereinigung dar. Dies gilt natürlich nur für Programmiersprachen *ohne* automatische Speicherbereinigung wie C und C++. In diesem Zusammenhang wäre es interessant zu untersuchen, wie gut eine BDD-Bibliothek abschneiden würde, die ganz in einer Programmiersprache mit automatischer Speicherbereinigung wie JAVA, EIFFEL, MODULA-3 etc. geschrieben ist. Zu hoffen ist, daß man von den effizienten Implementierungen der Speicherbereinigung in diesen Sprachen profitieren kann.

Ein Nachteil der hier vorgestellten SCAM-basierten Vorgehensweise besteht darin, daß die Bibliothek nur in C++ benutzt werden kann.[5] Durch Vorschalten einer weiteren Indirektions-

erhältlichen Bibliotheken wie /Long, 1994/ zu erzielen. Die volle Funktionalität, die für die Anbindung an den Modellprüfer μcke vonnöten wäre, konnte auf Grund dieses ambitionierten Zieles noch nicht erreicht werden.

[5]Es gibt darüber hinaus immer noch keinen vom Hersteller des C++ Kompiler unabhängigen Standard für C++ Bibliotheken, so daß ein Benutzer einer in C++ geschriebenen Bibliothek entweder deren Quellen oder aber

schicht (z. B. einer „C bridge“), kann dies umgangen werden, wobei dann aber natürlich die Benutzerfreundlichkeit auf der Strecke bleibt.

3.5.7 SCAM modulo E – Jungle Evaluation

Die SCAM läßt sich auch für die Implementierung von anderen kanonischen Termersetzungssystemen, die eine Gleichungstheorie E repräsentieren, verwenden. Die Methode der Jungle Evaluation /Hoffmann und Plump, 1988, Plump, 1992/ leistet dasselbe. Dabei wird ein Termersetzungssystem in eine Graphgrammatik /Ehrig, 1979/ übersetzt und man erreicht so, daß gleiche Unterterme modulo E nicht zweimal im Speicher gehalten werden müssen. Aus Anwendersicht ist diese Vorgehensweise bestimmt nicht so effizient wie eine Implementierung auf Basis der SCAM. Andererseits liefert die Implementierung auf Basis von Graphgrammatiken ein theoretisches Werkzeug, um die Korrektheit der SCAM-basierten Vorgehensweise zu sichern. So zeigen diese Arbeiten, daß man (i. allg. nur) bei *kanonischen* Termersetzungssystemen ungestraft gleiche (syntaktisch oder modulo E) Terme zusammenfassen („to share“) darf. Das Termersetzungssystem aus Definition 3.2 erfüllt diese Eigenschaft.

Ebenso sollte untersucht werden, wie sich eine SCAM-basierte Implementierung für PROLOG oder andere logisches und/oder funktionale Programmiersprachen verhält. Das System aus /Gollner, 1996/ ist dafür ein besonders gut geeigneter Kandidat, da für die darin verwendeten Optimierungen ohnehin „Vaterverweise“ benötigt werden, was als eine unabdingbare Voraussetzung für die Implementierung von „logischen Variablen“ (vgl. /Habel und Plump, 1996/) auf Basis der SCAM erscheint.

3.6 BDD-Algorithmen für die Modellprüfung

In diesem Abschnitt werden weitere BDD-Algorithmen vorgestellt. Dabei soll auch auf Laufzeiten eingegangen werden. Hauptziel ist es aber, eine für die Modellprüfung umfassende, einheitliche Darstellung anzugeben. Die Algorithmen selbst sind in Anlehnung an Implementierungen in verschiedenen BDD-Bibliotheken (vgl. Seite 53) konstruiert. In der Literatur sind nur Teilaspekte dieser Algorithmen behandelt. Diese einheitliche Darstellungsweise soll auch als Motivation für den $_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus aus Abschnitt 3.8.4 dienen, der eine Verallgemeinerung der hier vorgestellten Algorithmen darstellt.

3.6.1 Algorithmen für boolesche Operationen

In /Bryant, 1986/ wurde die Frage behandelt, wie man zu einer möglichst kleinen Anzahl von Algorithmen gelangt, die alle booleschen Operationen auf BDDs realisieren. Eine Version der Lösung von Bryant ist der Algorithmus apply aus Abb. 3.13. Dieser hat dieselbe Struktur, wie die Algorithmen aus Abb. 3.3. Zunächst wird der Basisfall abgefragt, nämlich ob die beiden BDDs schon konstant sind. In diesem Fall liefert die boolesche Operation „$\star$“ das gewünschte Ergebnis, das als BDD-Konstante interpretiert wird. Im anderen Fall ist eine Rekursion notwendig. Dazu wird zunächst die oberste Variable „m“ von beiden BDDs bestimmt und entsprechend dieser Variable die Kofaktoren (das sind die Komponenten des Ergebnises von cf, s. Abb. 3.4) gebildet. Danach wird die Prozedur zweimal rekursiv mit den Kofaktoren als Argumente aufgerufen und nach Rückkehr aus der Rekursion ein neuer Knoten mit den Ergebnissen der beiden rekursiven Aufrufe als Nachfolger als Ergebnis zurückgegeben. Man beachte,

denselben Kompiler benötigt, mit der die Bibliothek kompiliert wurde.

$\boxed{\mathsf{apply}(f, g, \star)}$ ::=

if $f \in \{0,1\}$ **and** $g \in \{0,1\}$ **then** $f \star g$
else $l \cdots (m) — r$
where
$m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\}$
$l = \boxed{\mathsf{apply}(f_0, g_0, \star)}$
$r = \boxed{\mathsf{apply}(f_1, g_1, \star)}$
where
$(f_0, f_1) = \mathsf{cf}(f, m)$
$(g_0, g_1) = \mathsf{cf}(g, m)$

Abbildung 3.13: $\mathsf{apply} : \mathsf{ROBDD}^2 \times [\mathbb{B}^2 \to \mathbb{B}] \to \mathsf{ROBDD}$

Beschreibung	Bezeichnung	ite-Darstellung
$\neg a$	Negation	$\neg a$
$a \wedge b$	Und	ite$(a, b, 0)$
$a \vee b$	Oder	ite$(a, 1, b)$
$a \to b$	Implikation	ite$(\neg a, 1, b)$
$a \leftrightarrow b$	Äquivalenz	ite$(a, b, \neg b)$
$\neg(a \wedge b)$	nand	$\neg$ite$(a, b, 0)$
$\neg(a \vee b)$	nor	$\neg$ite$(a, 1, b)$
$\neg(a \to b)$	Anti-Implikation	ite$(b, 0, a)$
$\neg(a \leftrightarrow b)$	Antivalenz	ite$(a, \neg b, b)$

Tabelle 3.1: Interessante binäre boolesche Operationen

daß die Elimination redundanter Kanten aus Abb. 3.2.b nicht behandelt wird, da dies, wie auf Seite 60 dargestellt, die SCAM implizit selbst durchführt.

Wenn man diesen Algorithmus mit dem aus Abb. 3.3.b vergleicht, so stellt man fest, daß bei letzterem eine Optimierung dahingehend vorgenommen wurde, daß schon bei *einem* konstanten Argument die Rekursion beendet wird. Dies läßt sich für alle anderen 16 booleschen Operationen auch bewerkstelligen. Darüber hinaus könnte man in Algorithmus 3.3.b zusätzlich noch die Gleichheit der beiden Argumente f und g abtesten, denn es gilt $f \wedge_{\mathsf{BDD}} f = f$. Von den

$$16 \;=\; 2^{2^2} \;=\; |\{f \in [\mathbb{B}^2 \to \mathbb{B}] \mid f \text{ total}\}|$$

Funktionen sind nur die aus Tabelle 3.1 interessant. Die restlichen stellen die Identität eines Argumentes dar, sind konstant oder gehen durch Vertauschung der Argumente aus einer der in der Tabelle aufgeführten Funktionen hervor.

Die letzte Spalte läßt erkennen, daß sich alle binären booleschen Operationen durch einfache Kombination der Negation und dem ite darstellen lassen. So reicht also auch ein Algorithmus für die Berechnung des „ite“ auf BDDs aus, wenn man zusätzlich noch die Negation einfach berechnen kann. So geht auch /Brace et al., 1990/ vor. Zusätzlich wurden in jener Arbeit noch Negationskanten verwendet, so daß die Negation eines BDDs sich in konstanter Zeit bewerkstelligen läßt. Auf Negationskanten soll in dieser Arbeit aber weiter nicht eingegangen werden.

$\boxed{\mathsf{ite}_{\mathsf{BDD}}(f,g,h)}$ ::=

if $f = 1$ **then** g
else **if** $f = 0$ **then** h
else
$l \cdots (m) — r$
where
$m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g), \mathsf{top}(h)\}$
$l = \boxed{\mathsf{ite}_{\mathsf{BDD}}(f_0, g_0, h_0)}$
$r = \boxed{\mathsf{ite}_{\mathsf{BDD}}(f_1, g_1, h_1)}$
where
$(f_0, f_1) = \mathsf{cf}(f, m)$
$(g_0, g_1) = \mathsf{cf}(g, m)$
$(h_0, h_1) = \mathsf{cf}(h, m)$

Abbildung 3.14: $\mathsf{ite}_{\mathsf{BDD}} : \mathsf{ROBDD}^3 \to \mathsf{ROBDD}$

Die Operation $\mathsf{ite}_{\mathsf{BDD}}$ als Gegenstück zu der Operation ite auf booleschen Ausdrücken $\mathbb{B}^W$ findet man als Algorithmus in Abbildung 3.14. Die Semantik ergibt sich nach folgendem Satz.

Satz 3.19 (Semantik von $\mathsf{ite}_{\mathsf{BDD}}$) *Es gilt*

$$\langle\!\langle \mathsf{ite}_{\mathsf{BDD}}(f,g,h) \rangle\!\rangle_A \;\equiv\; \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A, \langle\!\langle g \rangle\!\rangle_A, \langle\!\langle h \rangle\!\rangle_A)$$

für $f, g, h \in \mathsf{ROBDD}$, $A \in \mathcal{A}$ *mit* $\mathsf{range}(A) \supseteq \mathsf{var}(f,g,h)$.

Beweis: Zunächst soll die partielle Korrektheit gezeigt werden. Dazu zeige man für $m \geq \mathsf{top}(f)$, $(f_0, f_1) := \mathsf{cf}(f, m)$ und $x := A^{-1}(m)$ folgende Hilfsbehauptung

$$\langle\!\langle f \rangle\!\rangle_A \;\equiv\; \mathrm{ite}(x, \langle\!\langle f_1 \rangle\!\rangle_A, \langle\!\langle f_0 \rangle\!\rangle_A) \tag{3.6}$$

Für $m > \mathsf{top}(f)$ bzw. $f \in \{0,1\}$ gilt $f_0 = f_1 = f$, so daß mit $\mathrm{ite}(a,b,b) \equiv b$ die Hilfsbehauptung unmittelbar folgt. Nun sei $m = \mathsf{top}(f)$, womit $f_0 = \mathsf{lo}(f)$ und $f_1 = \mathsf{hi}(f)$. Damit ist Gleichung (3.6) aber gerade die Definition von $\langle\!\langle f \rangle\!\rangle_A$. Mit der Ordnung der Variablen in ROBDD folgt zusätzlich $m \notin \mathsf{var}(f_0, f_1)$.[6]

Nun fahre mit simultaner Induktion über den Termaufbau von f, g und h fort. Für $f \in \{0,1\}$ folgt die partielle Korrektheit mit $\mathrm{ite}(0,a,b) \equiv b$ und $\mathrm{ite}(1,a,b) \equiv a$. Sonst definiere m und die anderen Hilfsvariablen wie im Algorithmus. In einer zweiten Induktion zeigt man leicht

$$\mathsf{var}(\mathsf{ite}_{\mathsf{BDD}}(f,g,h)) \subseteq \mathsf{var}(f,g,h),$$

so daß $m \notin \mathsf{var}(l)$ und $m \notin \mathsf{var}(r)$ und deshalb $l \cdots (m) — r \in \mathsf{ROBDD}$. Im folgenden wird zur Vereinfachung „ $_A$" weggelassen. Nun ergibt sich

$$\mathrm{ite}(\langle\!\langle f \rangle\!\rangle, \langle\!\langle g \rangle\!\rangle, \langle\!\langle h \rangle\!\rangle) \;\equiv\; \ldots$$

[6] Dies ist interessanterweise die einzige Stelle, an der die Ordnung benötigt wird.

$\boxed{f \vee_{\mathsf{BDD}} g}$ ::=

if $f = 1$ **or** $g = 1$ **then** 1
else **if** $f = 0 \vee f = g$ **then** g
else **if** $g = 0$ **then** f
else
  $l \cdots (m) — r$
  where
    $m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\}$
    $l = \boxed{f_0 \vee_{\mathsf{BDD}} g_0}$
    $r = \boxed{f_1 \vee_{\mathsf{BDD}} g_1}$
   where
      $(f_0, f_1) = \mathsf{cf}(f, m)$
      $(g_0, g_1) = \mathsf{cf}(g, m)$

Abbildung 3.15: $\vee_{\mathsf{BDD}} : \mathsf{ROBDD}^2 \to \mathsf{ROBDD}$

nach (mehrfachem) Einsetzen der Definition von „ite“ zu

$$\begin{aligned}
\ldots &\equiv \mathrm{ite}(\ \mathrm{ite}(x, \langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle f_0 \rangle\!\rangle), \mathrm{ite}(x, \langle\!\langle g_1 \rangle\!\rangle, \langle\!\langle g_0 \rangle\!\rangle), \mathrm{ite}(x, \langle\!\langle h_1 \rangle\!\rangle, \langle\!\langle h_0 \rangle\!\rangle)\ ) \\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle f_0 \rangle\!\rangle)\ \mathrm{ite}(x, \langle\!\langle g_1 \rangle\!\rangle, \langle\!\langle g_0 \rangle\!\rangle) \ \vee\ \overline{\mathrm{ite}(x, \langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle f_0 \rangle\!\rangle)}\ \mathrm{ite}(x, \langle\!\langle h_1 \rangle\!\rangle, \langle\!\langle h_0 \rangle\!\rangle)
\end{aligned}$$

Nun verwende man Lemma 2.13, multipliziere aus und fasse wieder nach x zusammen (beidesmal Anwendungen von Distributivgesetzen und der Definition der Semantik von „ite“).

$$\begin{aligned}
\ldots &\equiv \mathrm{ite}(x, \langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle f_0 \rangle\!\rangle)\ \mathrm{ite}(x, \langle\!\langle g_1 \rangle\!\rangle, \langle\!\langle g_0 \rangle\!\rangle) \ \vee\ \mathrm{ite}(x, \overline{\langle\!\langle f_1 \rangle\!\rangle}, \overline{\langle\!\langle f_0 \rangle\!\rangle})\ \mathrm{ite}(x, \langle\!\langle h_1 \rangle\!\rangle, \langle\!\langle h_0 \rangle\!\rangle) \\
&\equiv x\ (\langle\!\langle f_1 \rangle\!\rangle \langle\!\langle g_1 \rangle\!\rangle \vee \overline{\langle\!\langle f_1 \rangle\!\rangle} \langle\!\langle h_1 \rangle\!\rangle) \ \vee\ \overline{x}\ (\langle\!\langle f_0 \rangle\!\rangle \langle\!\langle g_0 \rangle\!\rangle \vee \overline{\langle\!\langle f_0 \rangle\!\rangle} \langle\!\langle h_0 \rangle\!\rangle) \\
&\equiv x\ \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle g_1 \rangle\!\rangle, \langle\!\langle h_1 \rangle\!\rangle) \ \vee\ \overline{x}\ \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle, \langle\!\langle g_0 \rangle\!\rangle, \langle\!\langle h_0 \rangle\!\rangle) \\
&\equiv \mathrm{ite}(\ x, \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle, \langle\!\langle g_1 \rangle\!\rangle, \langle\!\langle h_1 \rangle\!\rangle), \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle, \langle\!\langle g_0 \rangle\!\rangle, \langle\!\langle h_0 \rangle\!\rangle)\ )
\end{aligned}$$

Nach der Induktionsvoraussetzung gilt

$$\langle\!\langle l \rangle\!\rangle_A \equiv \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle_A, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A) \quad \text{und} \quad \langle\!\langle r \rangle\!\rangle_A \equiv \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle_A, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A),$$

so daß man fortfahren kann mit

$$\ldots \equiv \mathrm{ite}(x, \langle\!\langle r \rangle\!\rangle, \langle\!\langle l \rangle\!\rangle) \ \equiv\ \langle\!\langle l \cdots (m) — r \rangle\!\rangle \ \equiv\ \langle\!\langle\, \mathsf{ite}_{\mathsf{BDD}}\,(f, g, h) \rangle\!\rangle$$

womit der Beweis der partiellen Korrektheit beendet ist. Für den Beweis der totalen Korrektheit verwende die Terminierungsfunktion

$$t(f, g, h) \ :=\ \mathsf{max\,var}(f, g, h) - \mathsf{min\,var}(f, g, h) \tag{3.7}$$

Da t offensichtlich positiv ist, muß nur gezeigt werden, daß im rekursiven Fall

$$t(f_0, g_0, h_0),\ t(f_1, g_1, h_1) < t(f, g, h)$$

gilt. Hierzu stellt man fest

$$\mathsf{max\,var}(f, g, h) \geq \mathsf{max\,var}(f_0, g_0, h_0),\ \mathsf{max\,var}(f_1, g_1, h_1),$$

```
BDDsimple* ite(BDDsimple * f, BDDsimple * g, BDDsimple* h)
{
  if(f==False) return inc(h);
  if(f==True) return inc(g);

  if(g==False) return notimplies(h,f);
  if(g==True) return or(f,h);

  if(h==False) return and(f,g);
  if(h==True) return implies(f,g);

  if(f==g) return or(f,h);
  if(f==h) return and(f,g);
  if(g==h) return inc(h);
  ...
}
```

Abbildung 3.16: Basisfall des $\mathsf{ite}_{\mathsf{BDD}}$ in der BDD-Bibliothek `BDDsimple`.

so daß mit

$$\mathsf{var}(f_0, g_0, h_0),\ \mathsf{var}(f_1, g_1, h_1) \subseteq \mathsf{var}(f, g, h) \setminus \mathsf{min}\,\mathsf{var}(f, g, h)$$

die folgende echte Ungleichung folgt.

$$\mathsf{min}\,\mathsf{var}(f, g, h) < \mathsf{min}\,\mathsf{var}(f_0, g_0, h_0),\ \mathsf{min}\,\mathsf{var}(f_1, g_1, h_1),$$

Dies in Gleichung (3.7) eingesetzt, schließt den Beweis. ❑

Die Struktur des Algorithmus ist dieselbe wie bei den schon vorgestellten, jetzt eben nur für drei Argumente. Der Basisfall ist hier nur in seiner einfachsten Form dargestellt. Für das $\vee_{\mathsf{BDD}}$ in Abb. 3.15 wurden dagegen alle möglichen Optimierungen (ohne Beachtung von Negationskanten) angegeben. Die Rekursion kann bei $\vee_{\mathsf{BDD}}$ genau dann abgebrochen werden, wenn ein Argument konstant ist oder beide Argumente dieselben sind. Für das $\mathsf{ite}_{\mathsf{BDD}}$ hat man hier mehrere Möglichkeiten, da man nun drei Argumente auf Gleicheit testen kann. Wenn beim $\mathsf{ite}_{\mathsf{BDD}}$ ein Argument schon konstant ist, kann man i. allg. noch nicht die Rekursion beenden. Statt dessen kann man aber dann immer bei den rekursiven Aufrufen auf das allgemeine $\mathsf{ite}_{\mathsf{BDD}}$ verzichten und dieses durch einen Spezialalgorithmus ersetzen. Für $h = 0$ gilt z. B.

$$\mathsf{ite}_{\mathsf{BDD}}(f, g, h) = \mathsf{ite}_{\mathsf{BDD}}(f, g, 0) = f \wedge_{\mathsf{BDD}} g$$

Führt man solche Optimierungen nicht ein, so hat man bei jedem rekursiven Aufruf den konstanten Mehraufwand, das dritte Argument zu überprüfen. Betrachtet man hier alle Möglichkeiten, so stellt man fest, daß man nur für einen Teil der 16 binären booleschen Operationen (bzw. für die aus Tabelle 3.1) Spezialalgorithmen braucht. Dies zeigt die Abhandlung des Basisfalles des $\mathsf{ite}_{\mathsf{BDD}}$ in der BDD-Bibliothek `BDDsimple`, die in Abbildung 3.16 zu finden ist. Diese Bibliothek verwendet für die Speicherbereinigung Referenzenzählen und so steht „`inc(a)`" für das Erhöhen des Referenzenzählers des BDD-Knoten auf den „`a`" zeigt. Als Eingabe erhält die Funktion `ite` drei Zeiger auf BDD-Knoten und liefert auch einen solchen zurück.

Daraus ergibt sich, daß nur das logische „Und" und „Oder", die Äquivalenz, deren Negation, die Implikation, deren Negation und die eigentliche Negation implementiert werden müssen (die Negation wird für den Basifall der Implikation, der Äquivalenz und deren Negationen benötigt). Dies bedeutet schließlich, daß man bei Verwendung des $\mathsf{ite}_{\mathsf{BDD}}$ alle binären booleschen Operationen darstellen kann. Für eine effiziente Implementierung kann aber auf zusätzliche Implementierungen von fünf weiteren binären booleschen Operationen nicht verzichtet werden.

Bei Verwendung von Negationskanten reduziert sich die Klasse der zu betrachtenden Basisoperationen auf zwei zweistellige z. B. das „Und“ und die „Äquivalenz“. Aber es erhöht sich die Anzahl zu betrachtenden Spezialfälle, insbesondere für die Normalisierung von Einträgen im Ergebnisspeicher (vgl. Seite 63 und 60) beträchtlich.

$\boxed{\mathsf{restrict}(f,i,c)}$ ::=

if $f \in \{0,1\}$ **or** $\mathsf{top}(f) > i$ **then** f
else **if** $\mathsf{top}(f) = i$ **then**
  if c **then** $\mathsf{hi}(f)$ **else** $\mathsf{lo}(f)$
else $l \cdots \textcircled{m} - r$
where
  $m = \mathsf{top}(f)$
  $l \ = \boxed{\mathsf{restrict}(\mathsf{lo}(f),i,c)}$
  $r \ = \boxed{\mathsf{restrict}(\mathsf{hi}(f),i,c)}$

Abbildung 3.17: $\mathsf{restrict}\colon \mathsf{ROBDD} \times \mathbb{N} \times \mathbb{B} \to \mathsf{ROBDD}$

3.6.2 Auswertung von Quantoren und Substitutionen

Mit den bisher vorgestellten Algorithmen lassen sich auf booleschen Ausdrücken alle Operationen, die in Abschnitt 2.5 definiert und in Abschnitt 2.7 verwendet wurden, darstellen. Ausnahme sind Substitutionen und damit Quantoren, die in $\mathbb{B}^W$ als Abkürzungen unter Verwendung von „$\vee$“ und Substitutionen definiert sind.

Um zu Algorithmen für diese Operationen zu gelangen, betrachte man den wichtigen Spezialfall solcher Substitutionen σ, die genau eine Variable durch eine Konstante ersetzen:

$$\sigma = \{x \mapsto c\}, \quad \text{für}\, x \in W,\ c \in \{0,1\}$$

Der entsprechende Algorithmus für BDDs nennt sich $\mathsf{restrict}$ und ist in Abbildung 3.17 dargestellt. Für $\mathsf{restrict}$ gilt das gleich wie für $\neg_{\mathsf{BDD}}$. Man könnte auch durch einfache Tiefensuche und explizites Ablaufen der Graphenstruktur von f das Ergebnis bestimmen. Hier soll aber diese Version bevorzugt werden, da aus ihr weitere Spezialalgorithmen leichter abgeleitet werden können.

Satz 3.20 (Semantik von $\mathsf{restrict}$) *Es gilt*

$$\langle\!\langle \mathsf{restrict}(f,i,c) \rangle\!\rangle_A \equiv \langle\!\langle f \rangle\!\rangle_A \{x \mapsto c\}$$

für $f \in \mathsf{ROBDD}$, $A \in \mathcal{A}$, $x \in W$, $A(x)\!\downarrow$, $i = A(x)$, $c \in \mathbb{B}$ *und* $\mathsf{range}(A) \supseteq \mathsf{var}(f)$.

Beweis: Die Terminierung ist unmittelbar klar. Für die partielle Korrektheit führe eine Induktion über den Termaufbau von f. Der Basisfall $f \in \{0,1\}$ ist trivial. Sei nun f *keine* Konstante und $m := \mathsf{top}(f)$ wie im Algorithmus. Wenn $i = m$ ist, so erhält man aus der Definition der Semantik von „$\langle\!\langle \cdot \rangle\!\rangle$“

$$\langle\!\langle f \rangle\!\rangle_A \{x \mapsto c\} = (x \, \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \vee \overline{x} \, \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A)\{x \mapsto c\} \equiv c \, \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \vee \overline{c} \, \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A$$
$$\equiv \begin{cases} \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A, & \text{falls } c = 1 \\ \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A, & \text{falls } c = 0 \end{cases} = \langle\!\langle \mathsf{restrict}(f,i,c) \rangle\!\rangle_A,$$

da $x \notin \mathsf{var}(\mathsf{lo}(f), \mathsf{hi}(f))$ wegen $f \in \mathsf{ROBDD}$. Für $i \neq m$ und $y := A^{-1}(m) \neq x$ ist

$$\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \{x \mapsto c\} &= (y \,\langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \;\vee\; \bar{y} \,\langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A)\{x \mapsto c\} \\
&\equiv y \,\langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \{x \mapsto c\} \;\vee\; \bar{y} \,\langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A \{x \mapsto c\} \\
&\equiv y \,\langle\!\langle \mathsf{restrict}(\mathsf{hi}(f), i, c) \rangle\!\rangle_A \;\vee\; \bar{y} \,\langle\!\langle \mathsf{restrict}(\mathsf{lo}(f), i, c) \rangle\!\rangle_A \\
&\equiv y\, r \;\vee\; \bar{y}\, l = \langle\!\langle l \cdots (m) - r \rangle\!\rangle_A \\
&= \langle\!\langle \mathsf{restrict}(f, i, c) \rangle\!\rangle_A
\end{aligned}$$

nach Induktionsvoraussetzung. ❑

Ein Beispiel für diesen Algorithmus findet man in Abb. 3.19a.-c. auf Seite 80. Hier sind u und v vierstellige Vektoren, d. h. $u, v \in V$ mit $|u| = |v| = 4$. Der BDD, von dem ausgegangen wird, repräsentiert den booleschen Ausdruck $u \doteq v$ (vgl. Definition 2.40) unter der Allokation A von Seite 121.[7] Den BDD in b. erhält man aus dem aus a., indem bei dem mit $A(u[3])$ gekennzeichneten Knoten der rechte Nachfolger (das ist der 1-Nachfolger) samt dem Knoten selbst entfernt wird. Entsprechend erhält man c. aus a. durch Entfernen des linken Nachfolgers. Dazu beachte man, daß jeweils der resultierende BDD mit dem ursprünglichen BDD bis auf die unterste Variablenebene keine gemeinsamen Teil*terme* hat. Die SCAM muß also den gesamten oberen Teil, obwohl er identisch ist, zweimal auf der Halde halten. Diese Art von Ähnlichkeit zwischen Entscheidungsdiagrammen kann von BDDs nicht ausgenutzt werden. Eine ähnliche Redundanz findet man auch bei BDD-Repräsentationen von Übergansrelationen für Systeme mit „Interleaving"-Semantik (vgl. Abschnitt C.4.5).

Damit lassen sich schon die Quantoren aus Definition 2.7 bewältigen, womit auch die Behandlung der Quantoren in Abschnitt 2.7 durch BDDs möglich wird. Denn Gleichung (2.1) auf Seite 13 läßt sich wegen der Korrespondenz von $\mathbb{B}^W$ und ROBDD auch auf BDDs übertragen und lautet dann

$$\mathsf{exists}(f, i) = \mathsf{restrict}(f, i, 0) \vee_{\mathsf{BDD}} \mathsf{restrict}(f, i, 1) \tag{3.8}$$

für eine dem Quantor in $\mathbb{B}^W$ entsprechende Operation exists auf ROBDD. Durch sukzessive Hintereinanderanwendung dieser Gleichung kann auch über ganze Mengen von BDD-Variablen quantifiziert werden. Dies ist eigentlich die Operation, die bei der BDD-Semantik benötigt wird. Auch hierfür werden im Abschnitt 3.8 Optimierungen angegeben.

Für den nächsten Abschnitt werden zwar nur Substitutionen von Variablen durch Variable benötigt. Dies ist aber ein Spezialfall von Substitutionen von Variablen durch beliebige Terme und läßt sich auch nicht einfacher bewerkstelligen als das allgemeine Verfahren. Zunächst gelten in $\mathbb{B}^W$ folgende Gleichungen

$$\begin{aligned}
s\{x \mapsto t\} \;&\equiv\; \mathrm{ite}(t, s\{x \mapsto 1\}, s\{x \mapsto 0\}) \\
&\equiv\; \exists y.\, s\{x \mapsto y\} \wedge (y \leftrightarrow t),
\end{aligned} \tag{3.9}$$

wobei $y \in W$ eine neue Variable ist, die weder in s noch in t vorkommt. Die letzte Gleichung erlaubt die Reduktion der allgemeinen Substitution auf Substitutionen von Variablen durch Variablen, was aber nach obigem nicht weiter betrachtet werden soll. Damit erhält man für die entsprechende BDD Operation $\mathsf{compose}$

$$\mathsf{compose}(f, i, g) \;=\; \mathsf{ite}_{\mathsf{BDD}}(g, \mathsf{restrict}(f, i, 1), \mathsf{restrict}(f, i, 0)) \tag{3.10}$$

[7] Die dort angegebene Allokation bezieht sich auf μ-Kalkül-Terme. Der Zusammenhang zwischen solchen Allokationen und solchen für $\mathbb{B}^W$ wird erst in Abschnitt 3.7 bzw. 4.2 dargelegt.

Hieraus kann man folgern, daß das „apply“ alleine nicht ausreicht, wenn man nicht auf eine umständliche Implementierung des „$\mathsf{ite}_{\mathsf{BDD}}$“ durch „apply“ zurückgreift. Auch die optimierten Algorithmen für das „compose“ aus Abschnitt 3.8 benützen das „$\mathsf{ite}_{\mathsf{BDD}}$“.

Die vorgestellten Gleichungen erlauben es, Substitutionen und Quantifikationen direkt auf BDDs durchzuführen. Die Berechnung nach diesen Gleichungen ist bei weitem nicht optimal und in Abschnitt 3.8 werden Optimierungen angegeben. Auch erst in diesem Abschnitt findet man Algorithmen für diese Operationen.

Mit „restrict“ und „$\mathsf{ite}_{\mathsf{BDD}}$“ wurden die zwei grundlegenden BDD-Algorithmen angegeben, die für die Berechnung der BDD-Semantik von $\mathbb{B}_\mu^V$ aus dem nächsten Abschnitt benötigt werden.

3.7 Eine BDD Semantik für $\mathbb{B}_\mu^V$

In Abschnitt 2.7 wurde eine Semantik für $\mathbb{B}_\mu^V$ vorgestellt, die für einen μ-Kalkül-Term $t \in T_\mu^V$ einen booleschen Ausdruck berechnet. Da nun BDDs (hier werden soweit nichts anderes gesagt, nur ROBDDs betrachtet) eine Repräsentation für boolesche Ausdrücke darstellen, liegt es nahe, die durch diese Semantik gewonnenen booleschen Ausdrücke als BDDs zu repräsentieren. Auf diese Weise erhält man eine formale Definition der Modellprüfung im μ-Kalkül mit binären Entscheidungsdiagrammen, die in Kapitel 4 für die Behandlung von Allokationen für μ-Kalkül-Terme benötigt wird.

Definition 3.21 (Belegung für BDD-Semantik) *Eine partielle Funktion*

$$\lambda\colon P \to \mathsf{ROBDD}$$

heiße eine Belegung für die BDD-Semantik. Einer Belegung ρ_P für die Semantik II aus Definition 2.41 bzw. einer Belegung für die BDD-Semantik sei

$$\lambda(X) := \|\rho_P(X)\|_A \quad \textit{bzw.} \quad \rho_P(X) := \langle\!\langle \lambda(X) \rangle\!\rangle_A$$

für eine Allokation $A \in \mathcal{A}$ zugeordnet.

Da ROBDDs eine Normalform für boolesche Ausdrücke darstellen (s. Satz 3.16), ist zwei „äquivalenten“ Belegungen für die Semantik II (vgl. Def. B.1) , bei denen die Bilder einer Variable semantisch äquivalent sind, genau eine Belegung für die BDD-Semantik zugeordnet.[8]

Definition 3.22 (BDD-Semantik von $\mathbb{B}_\mu^V$) *Die partielle Funktion*

$${}_A\|\cdot\|\lambda\colon T_\mu^V \to \mathsf{ROBDD}$$

für eine Belegung λ und eine Allokation $A \in \mathcal{A}$ sei die BDD-Semantik *mit*

$${}_A\|t\|\lambda := \|\langle\!\langle t \rangle\!\rangle \rho_P\|_A,$$

wobei ρ_P zugeordnet ist zu λ.

[8] In umgekehrter Richtung besteht nicht einmal ein funktionaler Zusammenhang, was aber im folgenden keine Rolle spielen wird.

Die in dieser Definition auftretende Allokation A sei so gewählt, daß die betrachteten Abbildungen nicht partiell werden. Insbesondere sollte $\mathtt{range}(A)$ immer die Menge der BDD-Variablen der auftretenden BDDs überdecken. O.B.d.A. sei diese Anforderung von allen in diesem Abschnitt vorkommenden Allokationen A erfüllt.

Mit der Wohldefiniertheit der Semantik II nach Korollar 2.46 (unter der Generalvoraussetzung einer wohlgeformten Prädikatsdefinition) ist die BDD-Semantik auch wohldefiniert, also eine *totale* Funktion.

Satz 3.23 *Die BDD-Semantik ist wohldefiniert.*

Folgender Satz zeigt, daß man bei der Berechnung der BDD-Semantik nicht auf die Semantik II oder die Standardsemantik zurückgreifen muß. Sondern man kann ausschließlich mit Operationen auf BDDs auskommen.

Satz 3.24 (Berechnung der BDD-Semantik)
Für $s,t \in T_\mu^V$, $f \in \mathsf{ROBDD}$, $\zeta \in \mathbb{B}^W$, $X \in P$ und $w \in V$,$\underline{u} \in V^n$ gilt

$$\begin{aligned}
{}_A\|\neg t\|\lambda &= \neg_{\mathsf{BDD}}\, {}_A\|t\|\lambda \\
{}_A\|s \wedge t\|\lambda &= {}_A\|s\|\lambda \wedge_{\mathsf{BDD}}\, {}_A\|t\|\lambda \\
{}_A\|\exists w.\, s\|\lambda &= \mathsf{exists}({}_A\|s\|\lambda, A(w)) \\
\|\Phi^X_{\rho_P}(\langle\!\langle f \rangle\!\rangle_A)\|_A &= {}_A\|\beta(X)\|\lambda\{X \mapsto f\} \\
{}_A\|\zeta\{\pi(X) \mapsto \underline{u}\}\|\lambda &= \mathsf{compose}(\|\zeta\|_A, \rho')
\end{aligned}$$

mit

$$\begin{aligned}
\rho' := \{ j \mapsto 0 \cdots \textcircled{i} - 1 \mid{} & \pi(X) = (v_1, \ldots, v_n), \\
& \underline{u} = (u_1, \ldots, u_n), j = A(v_l[k]), i = A(u_l[k]), 0 \le k < |u| \},
\end{aligned}$$

$A(w)$ interpretiert als Teilmenge von $\mathbb{N}$ und ρ_P sei λ zugeordnet.

Beweis: Die ersten beiden Gleichungen folgen unmittelbar aus der Definition von ${}_A\|\cdot\|$ und der Homomorphieeigenschaft von $\langle\!\langle \cdot \rangle\!\rangle_A$ und $\|\cdot\|_A$ bez. „$\wedge$“ und „$\neg$“. Ähnlich argumentiert man für die dritte und letzte Gleichung. Weiter zeigt

$${}_A\|\Phi^X_{\rho_P}(\langle\!\langle f \rangle\!\rangle_A\|\lambda = {}_A\|\beta(X)\rho_P\{X \mapsto \langle\!\langle f \rangle\!\rangle_A\}\|\lambda = {}_A\|\beta(X)\rho_P\|\lambda\{X \mapsto f\}$$

die Gültigkeit der vierten Gleichung. ❑

Damit lassen sich alle Operationen in $\mathbb{B}^W$ zur Berechnung der Semantik II aus Def. 2.42 rekursiv über die Termstruktur direkt mit ROBDDs durchführen. Nur an einer Stelle muß man noch folgende Hilfsbehauptung 3.26 bemühen.

Definition 3.25 (Expansionsoperator für die BDD-Semantik)

$$\Lambda^X_\lambda : \mathsf{ROBDD} \to \mathsf{ROBDD}, \qquad \Lambda^X_\lambda(f) := {}_A\|\beta(X)\|\lambda\{X \mapsto f\}$$

Lemma 3.26 *Seien ρ_P und λ einander zugeordnet. Dann gilt*

$$\|(\Phi^X_{\rho_P})^i(0)\|_A = (\Lambda^X_\lambda)^i(0)$$

Beweis: Der Beweis wird geführt durch Induktion über i. Der Induktionsanfang $i = 0$ ist trivial. Für den Induktionsschritt wähle $f \in \mathsf{ROBDD}$ mit $\langle\!\langle f \rangle\!\rangle_A \equiv (\Phi^X_{\rho_P})^i(0)$. Nach Satz B.2 ist

$$(\Phi^X_{\rho_P})^{i+1}(0) = \Phi^X_{\rho_P}((\Phi^X_{\rho_P})^i(0)) \equiv \Phi^X_{\rho_P}(\langle\!\langle f \rangle\!\rangle_A)$$

Die Bilder unter $\|\cdot\|_A$ dieser drei booleschen Terme sind also alle gleich. Nach Induktionsvoraussetzung gilt nun $f = (\Lambda^X_\lambda)^i(0)$ und mit der vierten Gleichung aus Satz 3.24 erhält man

$$\|(\Phi^X_{\rho_P})^{i+1}(0)\|_A = {}_A\|\beta(X)\|\lambda\{X \mapsto f\} = \Lambda^X_\lambda(f) = (\Lambda^X_\lambda)^{i+1}(0)$$

❏

Damit läßt sich auch das minimale n aus Def. 2.42 mittels Operationen auf BDDs berechnen. Der Test auf Äquivalenz des Ergebnises zweier aufeinanderfolgender Iterationen reduziert sich dabei wegen der Normalformeigenschaft von ROBDDs auf einen Vergleich zweier ROBDDs, was sich in konstanter (je nach Implementierung höchstens in linearer – vgl. /Sieling, 1995/) Zeit durchführen läßt. Für den BDD zum entsprechenden ζ gilt dann nach Lemma 3.26

$$\|\zeta\|_A = (\Lambda^X_\lambda)^n(0),$$

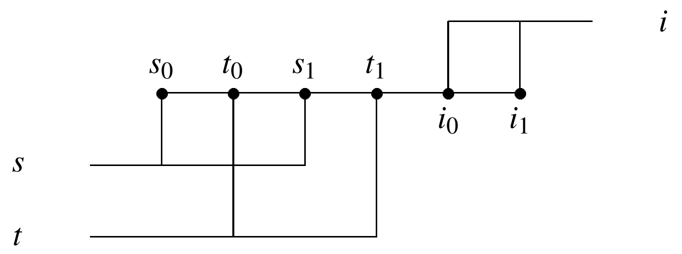
so daß auch hier nur Operationen auf BDDs benötigt werden.

Die BDD-Semantik am Beispiel

Zum Abschluß dieses Abschnittes soll die transitive Hülle der Übergangsrelation des Beispiels aus Abb. 2.3 auf Seite 16 berechnet werden (vgl. Def. A.1). Die (reflexive) transitive Hülle ist die kleinste Relation, die die Identität enthält und die abgeschlossen ist gegenüber dem Relationenprodukt mit der Ausgangsrelation. In $\mathbb{B}_\mu^V$ lautet diese Formulierung (vgl. auch mit Bsp. von S. 7)

$$\mu T^*(s,t) \,.\, s \doteq t \;\vee\; \exists\, i.\, T(s,i) \wedge T^*(i,t),$$

wobei die Definition von T auf Seite 18 zu finden ist. Die Allokation A, die in diesem Beispiel verwendet wird, habe folgende Gestalt.



oder explizit für $j \in \{0,1\}$

$$A(s[j]) = 2j, \quad A(t[j]) = 2j+1, \quad A(i[j]) = 4+j$$

Hiermit läßt sich der BDD ${}_A\|T(s,t)\|$ berechnen, der in Abb. 3.18.a zu finden ist.[9] In der rekursiven Berechnung der BDD-Semantik $T^*(s,t)$, benötigt man aber den BDD ${}_A\|T(s,i)\|$ für $T(s,i)$. Hier muß also noch eine Substitution durchgeführt werden, was den BDD aus Abb. 3.18.b ergibt. Dieser muß nur einmal berechnet werden, da T nicht rekursiv ist ($\mathsf{scc}(T) = \emptyset$). Seien nun

$$f_i := (\Lambda^{T^*})^i(0)$$

die in der Fixpunktberechnung für T^* auftretenden Approximationen an ${}_A\|T^*(s,t)\|$. Nach Definition des Expansionsoperator gilt dabei

$$\begin{aligned}
f_0 &= 0 \\
f_1 &= {}_A\|s \doteq t\| \\
f_2 &= {}_A\|s \doteq t \;\vee\; \exists i.\, T(s,i) \wedge T^*(i,t)\|\{T^* \mapsto f_1\} \\
&= {}_A\|s \doteq t \;\vee\; \exists i.\, T(s,i) \wedge i \doteq t\| \quad = \quad {}_A\|s \doteq t \vee\; T(s,t)\| \\
f_3 &= {}_A\|s \doteq t \;\vee\; \exists i.\, T(s,i) \wedge T^*(i,t)\|\{T^* \mapsto f_2\} \\
&= {}_A\|s \doteq t \;\vee\; \exists i.\, T(s,i) \wedge (i \doteq t \vee T(i,t))\| \\
&\;\;\vdots
\end{aligned}$$

Hierbei beachte man, daß f_i in der $i+1$-ten Iteration nicht direkt verwendet werden kann. Zuvor muß wiederum eine BDD-Substitution

$$\rho \;:=\; \mathsf{id}\{0 \mapsto 0 \cdots \textcircled{4} - 1,\, 2 \mapsto 0 \cdots \textcircled{5} - 1\}$$

durchgeführt werden, die der Substitution von s durch i entspricht. Erst dann kann der BDD für den Rumpf des Quantors durch $\wedge_{\mathsf{BDD}}$ und anschließend für den Quantor selbst bestimmt werden. Die BDDs f_i sind vor und nach der Substitution in Abb. 3.18.c-h. zu finden. Nach drei Iterationen konvergiert das Verfahren. Hierbei muß jeweils das *Relationenprodukt*, d.h. die Kombination von Konjunktion und Quantifikation, des linken BDDs in der unteren Reihe (Abb. 3.18.b) mit den drei weiteren BDDs in der unteren Reihe (Abb. 3.18.d, 3.18.f und 3.18.h) berechnet werden. So ergibt sich der BDD f_3 durch Konjunktion der BDDs aus den Abbildungen 3.18.b und 3.18.f, Quantifikation über die untersten beiden BDD-Variablen und anschließender Disjunktion mit dem BDD f_1 aus Abb. 3.18.d, der ${}_A\|s \doteq t\|$ entspricht.

Dieses Beispiel wird im Kapitel über Allokationsrandbedingungen noch einmal aufgegriffen. Dort wird gezeigt, daß die Allokation A zwangsweise zu einem exponentiellen Aufblähen von $\mathsf{compose}(f_1, \rho)$ führt. Es wird dann eine Methode vorgestellt, Allokationen zu erzeugen, die gewährleisten, daß bei den durchzuführenden Substitutionen, falls möglich, nur gleichgroße BDDs entstehen.

[9] Im weiteren sei die „leere“ Belegung $\lambda = \{\}$ einfach weggelassen.

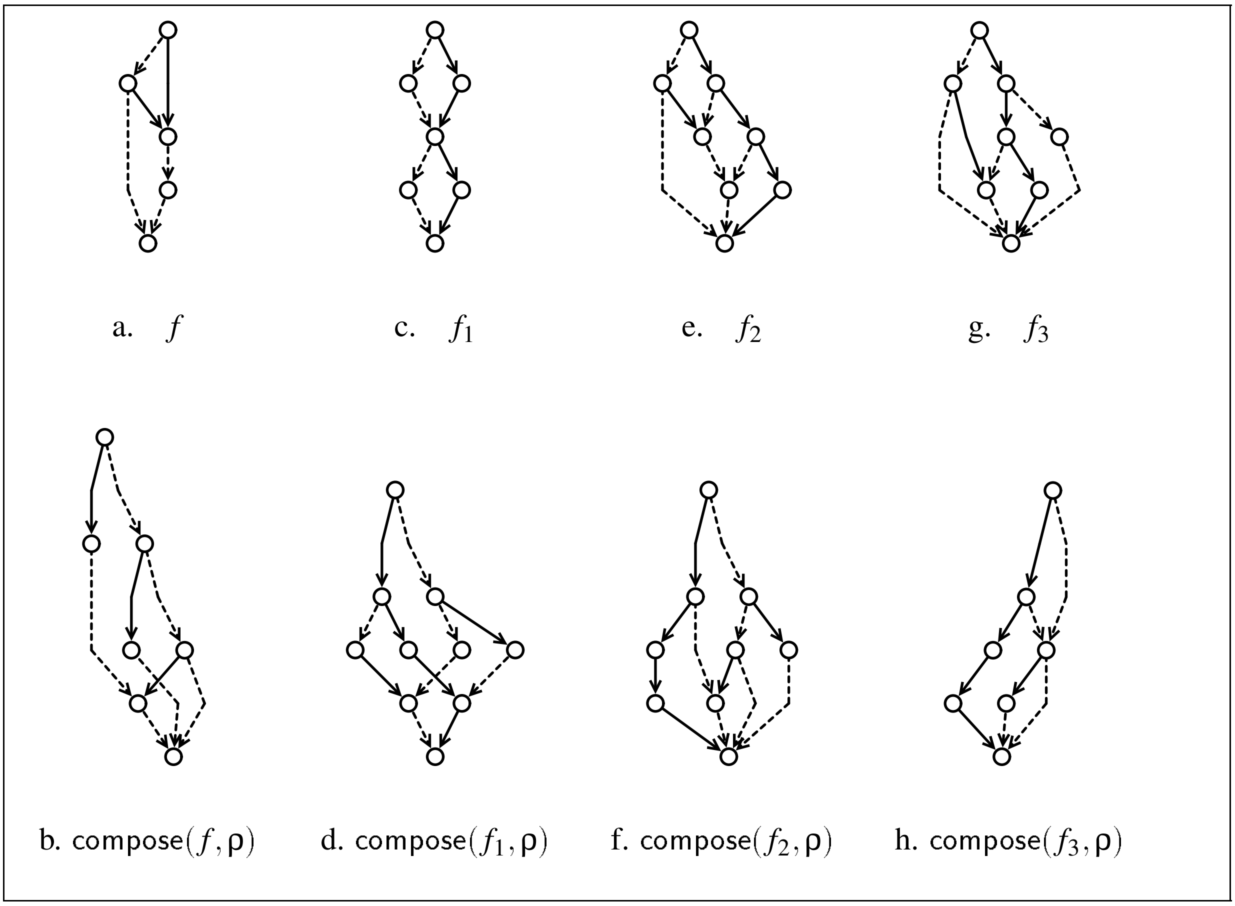


Abbildung 3.18: Berechnung der BDD-Semantik der transitiven Hülle des Beispiels von S. 18 zur *schlechten* Allokation von S. 77.

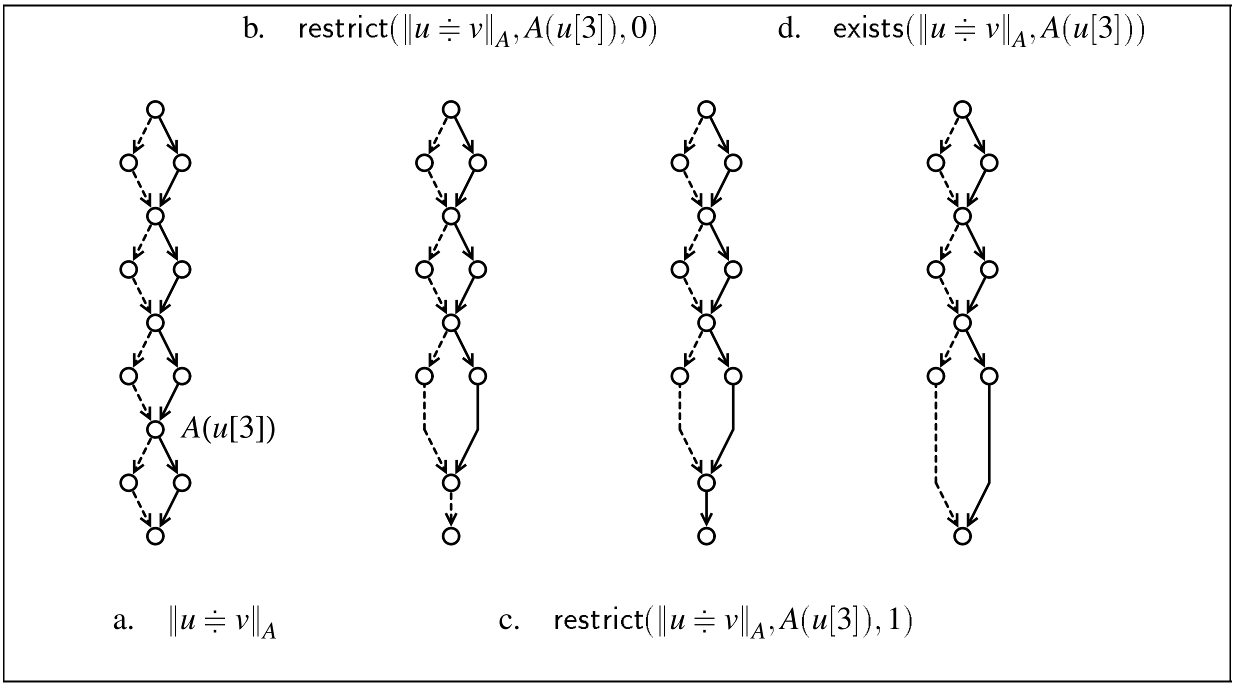


Abbildung 3.19: Berechnung von exists(f, i) als restrict$(f, i, 0) \vee_{\mathsf{BDD}}$ restrict$(f, i, 1)$.

3.8 Optimierte BDD-Algorithmen

3.8.1 Schnelle Algorithmen für Quantoren

Würde man exists nach Gleichung (3.8) auf Seite 74 implementieren, was in Abb 3.19 dargestellt ist, so stellt man fest, daß alle Knoten von f bis auf die „Höhe“ von i zweimal besucht werden müßten, um hinterher wiederum durch beide Zwischenergebnisse von oben nach unten durchlaufend das „Oder“ auszurechnen. Ist i ziemlich groß (liegt weit unten), so bedeutet dies, daß fast ganz f zweimal durchlaufen werden muß. Weiterhin müssen dadurch drei leicht veränderte Kopien von f erzeugt werden. Die ersten zwei entsprechen dem linken und rechten Zwischenergebnis nach Ausführung von restrict und die dritte dem Endergebnis. Erst wenn das Endergebnis nach Abarbeitung von $\vee_{\mathsf{BDD}}$ vorliegt, können die Zwischenergebnisse gelöscht werden. Dies kann also im schlimmsten Fall zu einem dreifachen Platzverbrauch gegenüber dem Endergebnis führen, wenn wie oben i ziemlich weit unten in f liegt. Natürlich wirkt sich das auch auf den Zeitbedarf aus, wie später an einem Beispiel gezeigt wird (vgl. Abb. 3.28).

Deshalb sucht man nach besseren Algorithmen, die diesen Kopieraufwand nicht nötig machen. Die Hauptidee zur Verbesserung dieses Algorithmus besteht darin, zu versuchen die Auswertung von restrict im Sinne der funktionalen Programmierung aufzuschieben, also eine „lazy evaluation“ anzustreben. Für den resultierenden Algorithmus bedeutet dies, daß er eine Verschränkung des Algorithmus des $\vee_{\mathsf{BDD}}$ und des restrict darstellt. Er ist in Abb. 3.20 zu finden.

Für diesen Algorithmus überlegt man sich leicht, daß er höchstens soviele rekursive Aufrufe (des exists^0 und des $\vee_{\mathsf{BDD}}$) benötigt wie eine Berechnung nach Gleichung (3.8) auf Seite 74. Da sowohl restrict als auch das $\vee_{\mathsf{BDD}}$ in der Größe der Argumente linearen Platz und Zeit benötigen und in Gleichung (3.8) nur genau drei solche Anwendungen berechnet werden, bedeutet dies, daß auch exists^0 nur linearen Aufwand erfordert.

Für die Übersetzung des Quantors aus $\mathbb{B}_\mu^V$ bei der BDD-Semantik muß über die Menge von denjenigen BDD-Variablen quantifiziert werden, die den Komponenten der gebundenen Varia-

$\boxed{\mathsf{exists}^0(f,i)}$ ::=

if $f \in \{0,1\}$ **or** $\mathsf{top}(f) > i$ **then** f
else if $\mathsf{top}(f) = i$ **then**
  $\mathsf{lo}(f) \vee_{\mathsf{BDD}} \mathsf{hi}(f)$
else
  $l \cdots m - r$
  where
    $m = \mathsf{top}(f)$
    $l = \boxed{\mathsf{exists}^0(\mathsf{lo}(f), i)}$
    $r = \boxed{\mathsf{exists}^0(\mathsf{hi}(f), i)}$

Abbildung 3.20: $\mathsf{exists}^0 : \mathsf{ROBDD} \times \mathbb{N} \to \mathsf{ROBDD}$

$\boxed{\mathsf{exists}^1(f,E)}$ ::=

if $f \in \{0,1\}$ **or** $\mathsf{var}(f) \cap E = \emptyset$ **then** f
else if $\mathsf{top}(f) \in E$ **then**
  $l \vee_{\mathsf{BDD}} r$
else
  $l \cdots m - r$
where
  $m = \mathsf{top}(f)$
  $l = \boxed{\mathsf{exists}^1(\mathsf{lo}(f), E)}$
  $r = \boxed{\mathsf{exists}^1(\mathsf{hi}(f), E)}$

Abbildung 3.21: $\mathsf{exists}^1 : \mathsf{ROBDD} \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$

ble entsprechen. Hierfür kann man wie bei der einfachen Berechnung des Quantors in Unterabschnitt 3.6.2 sukzessive über die Variablen einzeln quantifizieren. Hierbei entsteht natürlich wiederum nach jeder Quantifikation über eine einzelne BDD-Variable ein Zwischenergebnis, das sofort wieder durchlaufen wird, um danach nicht mehr weiter benötigt zu werden. Deshalb wurde auch hier versucht, eine verbesserte Version von exists^0 zu finden.

Hierfür ist erstaunlicherweise (zumindest in $\mathcal{F}$) nur eine minimale Änderung des Algorithmus von exists^0 aus Abb. 3.20 notwendig. Das Ergebnis ist in Abb. 3.21 zu finden. Hier kann nicht wie beim ursprünglichen Algorithmus die Rekursion beendet werden, wenn die oberste Variable von f in E liegt. Sondern auch aus dem 1- und 0-Nachfolger von f müssen die Variablen aus E „herausquantifiziert" werden.

Für diesen Algorithmus läßt sich die oben durchgeführte einfache Komplexitätsanalyse nicht anwenden, da z. B. E aus der Hälfte der Variablen, die in f vorkommen, bestehen könnte, so daß insgesamt mindestens $\Omega(|f|)$ Aufrufe von $\vee_{\mathsf{BDD}}$ in der Berechnung von $\mathsf{exists}^1(f,E)$ notwendig werden. Hier kann sich das Ergebnis von $\vee_{\mathsf{BDD}}$ zwar jedesmal nur um einen linearen

Faktor vergrößern, was aber insgesamt zu einem exponentiell größeren BDD führen kann. In /McMillan, 1993a/ wird eine Reduktion von 3-SAT (s. /Garey und Johnson, 1979/) für das Problem, das exists^1 löst, angegeben, so daß man hier auch nicht weniger als exponentiellen Aufwand erwarten würde. Man beachte zusätzlich, daß eine Reduktion von QSAT, dem Erfüllbarkeitsproblem für „quantifizierte“ boolesche Formeln (s. wiederum /Garey und Johnson, 1979/) nicht so einfach möglich ist, da das erste Argument von exists^1 schon ein ROBDD sein muß.

Der große Vorteil von exists^1 gegenüber exists^0 ist aber, daß in gutartigen Fällen nicht nur eine lineare Aufwandsverbesserung zu beobachten ist, wie bei der Verwendung von exists^0 statt der Vorgehensweise nach Gleichung (3.8) auf Seite 74 (vgl. Abb. 3.28.

Im SMV System wird der Algorithmus exists^1 aus Abb. 3.21 durch einen kleinen Trick so verändert, daß nur noch ROBDDs als Argumente notwendig sind. Hierzu wird eine Menge $E \subseteq \mathbb{N}$ von Variablen selbst als ROBDD dargestellt und zwar durch folgenden ROBDD bei einer bel. Allokation $A \in \mathcal{A}$, mit $\mathsf{range}(A) \supseteq E$

$$\| \bigwedge_{A(x)\in E} x \|_A \tag{3.11}$$

Der in dieser Gleichung aufgeführte boolesche Ausdruck ist das „Und“ aller booleschen Variablen, die einer BDD Variable aus E entsprechen. Der BDD besteht aus $|E|$ inneren Knoten, deren 0-Nachfolger die Konstante 0 ist, und für alle $i \in E$ gibt es genau einen Knoten, der mit i markiert ist. Den Algorithmus, der einen solchen BDD als zweites Argument erwartet, findet man in Abb. 3.22. Der Vorteil dieser Vorgehensweise besteht darin, daß man sich bei der Implementierung nicht um eine Datenstruktur für Mengen von natürlichen Zahlen kümmern muß.[10] Dies ist insbesondere beim Ergebnisspeicher nützlich. Die Quantifikation über Mengen kann so wie ein normaler binärer Operator behandelt werden. Auf diese Weise lassen sich immer die im folgenden auftretenden BDD-Variablenmengen darstellen. Nur an dieser Stelle wird aber diese Optimierung angewandt. Für die weiteren Algorithmen kann die Anwendung analog erfolgen.

Ein weiterer wichtiger BDD-Algorithmus für die Modellprüfung ist der Algorithmus für das „Relationenprodukt“ (engl. relation product). Aufgabe dabei ist es

$$\mathsf{exists}(f \wedge_{\mathsf{BDD}} g, E) \tag{3.12}$$

zu berechnen (für den Rest der Arbeit sei unter exists ohne weitere Zusätze immer exists^1 verstanden). Der entsprechende μ-Kalkül-Term tritt sehr häufig bei der Modellprüfung auf (vgl. Abschnitt 5) und dessen Auswertung ist meistens (in Varianten) die mit Abstand aufwendigste Operation. Selbst mit dem gleich vorgestellten optimierten Algorithmus ist der Zeitaufwand für die Modellprüfung zum überwiegenden Teil vom Zeitaufwand für diese Operation bestimmt. Dies zeigen die Abbildungen 3.23 und 3.24. Im ersten Beispiel wurde wiederum dieselbe Lebendigkeitseigenschaft des Alternating-Bit-Protokolls überprüft wie in Abb. 3.28. Das zweite Beispiel ist die Verifikation des DME Protokolls, in der einfachen optimierten Version (nur mit Vorwärtsanalyse und keiner inkrementellen Approximation der Übergangsrelation). In diesem Kapitel wurde zur Messung eine nicht optimierte Version der μcke benutzt, wodurch die etwas längeren Zeiten wie bei der ausführlichen Behandlung in Kapitel 5 zustande kommen. Auch die Messung an sich verbraucht natürlich Zeit.

In beiden Beispielen wurde die Gesamtzeit für die Modellprüfung und die Zeit für die Berechnung von Varianten des Relationenproduktes aufgetragen. Zusätzlich findet man noch eine

---

[10] Durch die Darstellung einer Menge von natürlichen Zahlen als ROBDD nach Gleichung (3.11) erhält man sogar eine Normalform!

$\boxed{\mathsf{exists}^2(f,g)}$ ::=

if $f \in \{0,1\}$ **or** $g \in \{0,1\}$ **then** f
else **if** $\mathsf{top}(f) > \mathsf{top}(g)$ **then** $\mathsf{exists}^2(f,\mathsf{hi}(g))$
else
  if $\mathsf{top}(f) = \mathsf{top}(g)$ **then**
    $l \vee_{\mathsf{BDD}} r$
    where
      $l = \boxed{\mathsf{exists}^2(\mathsf{lo}(f),\mathsf{hi}(g))}$
      $r = \boxed{\mathsf{exists}^2(\mathsf{hi}(f),\mathsf{hi}(g))}$
  else
    $l \cdots (m) - r$
    where
      $m = \mathsf{top}(f)$
      $l = \boxed{\mathsf{exists}^2(\mathsf{lo}(f),g)}$
      $r = \boxed{\mathsf{exists}^2(\mathsf{hi}(f),g)}$

Abbildung 3.22: $\mathsf{exists}^2 : \mathsf{ROBDD} \times \mathsf{ROBDD} \rightarrow \mathsf{ROBDD}$

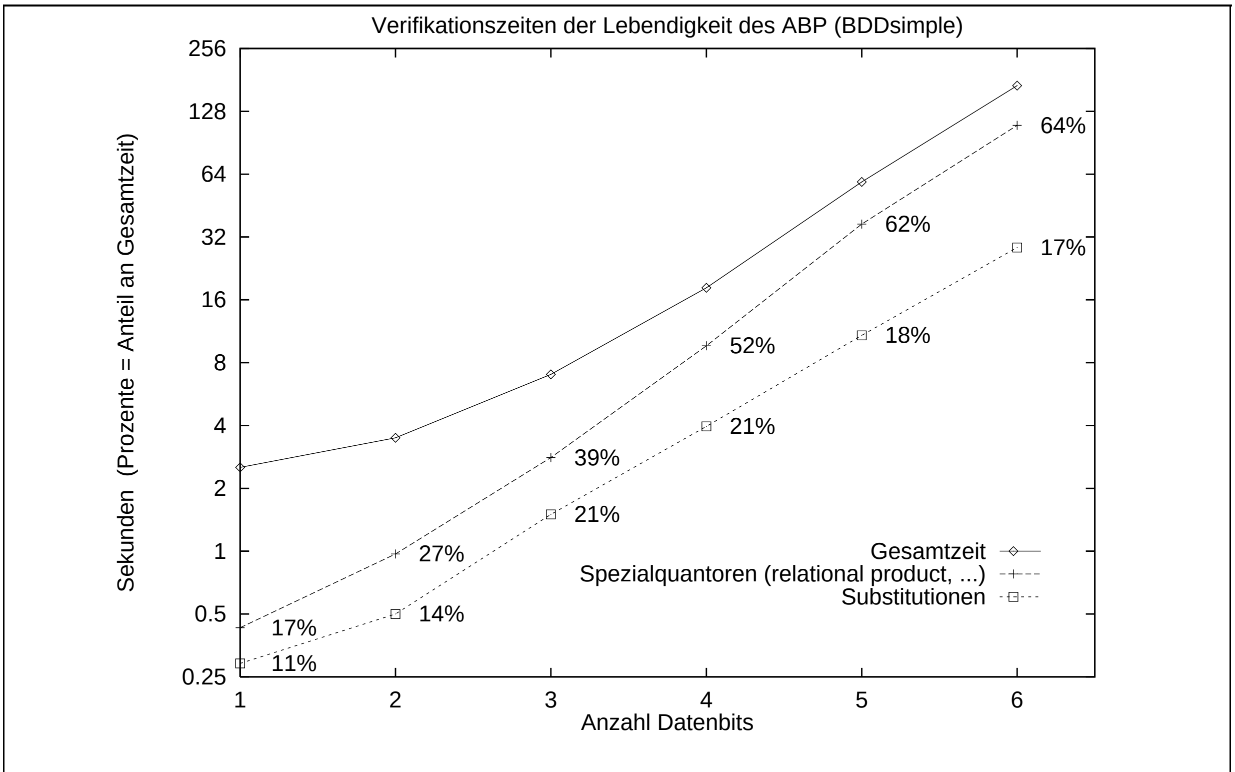


Abbildung 3.23: Aufwand des Relationproduktes und von Substitutionen (ABP)

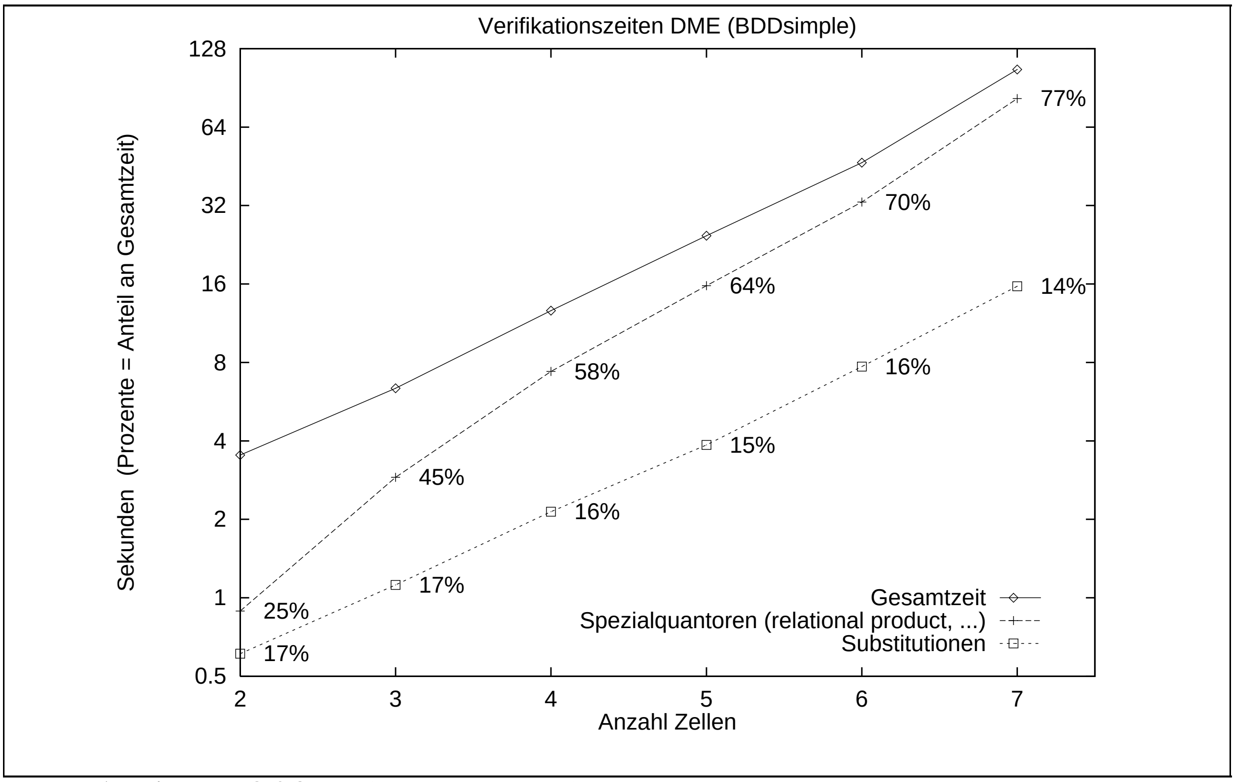


Abbildung 3.24: Aufwand des Relationproduktes und von Substitutionen (DME)

dritte Meßreihe, die die Zeit für die Berechnung von Substitutionen angibt. Die Prozentzahlen geben den jeweiligen Anteil an der Gesamtzeit an.

Wenn man (3.12) einfach direkt ausrechnet, dann hat man nicht nur wie oben beim naiven Ausrechnen des exists den Mehraufwand des mehrmaligen Durchwanderns und Kopierens von beteiligten BDD-Knoten. Zusätzlich entstehen hier Zwischenresultate, die BDD-Variablen enthalten, die im späteren Verlauf der Berechnung „herausquantifiziert" werden, also verschwinden. Dies bedeutet, daß die Zwischenergebnisse wesentlich größer sein können als das Endergebnis.

Um zu einem verbesserten Verfahren zu gelangen, wendet man einen ähnlichen Ansatz wie beim exists^0 an. Man verschränkt die beiden Algorithmen von exists^1 und $\wedge_{\mathsf{BDD}}$ und versucht dabei, die Auswertung des inneren Algorithmus so weit wie möglich hinauszuschieben. Das Resultat dieser Überlegungen findet man in Abbildung 3.25 (vgl. /Burch et al., 1994/).

Auf dieselbe Weise lassen sich Algorithmen für den Allquantor behandeln. Für eine BDD-Bibliothek mit Negationskanten ist dies nicht notwendig, wegen der Dualität von Quantoren:

$$\forall E.\,t \;\equiv\; \neg\exists E.\,\neg t \qquad \forall E.\,s \rightarrow t \;\equiv\; \neg\exists E.\,s \wedge \neg t \tag{3.13}$$

Bei der Modellprüfung tritt der rechte Fall sehr häufig auf, wie in Kapitel 5 erläutert wird. Hier werden die Allgorithmen (z. B. für das „forallImplies") nicht extra aufgeführt, da all diese Spezialquantoren eine Spezialisierung des $\mathsf{ite}_\exists$-Algorithmus aus Abschnitt 3.8.4 darstellen.

Nun sollen die verschiedenen Möglichkeiten zur Berechnung von Quantoren verglichen werden. Hierzu hat man prinzipiell die Möglichkeit, einen Spezialalgorithmus wie relProd zu verwenden oder aber den Quantor durch die früher vorgestellten Algorithmen zu berechnen. Im letzten Fall unterscheidet man noch einmal, ob die Quantifikation über Mengen verwendet wird (exists^1) oder aber sukzessive über die einzelnen Variablen getrennt „quantifiziert" wird.

$$
\begin{array}{l}
\boxed{\mathsf{relProd}(f,g,E)} ::= \\
\quad \textbf{if } f=0 \textbf{ or } g=0 \textbf{ then } 0 \\
\quad \textbf{else} \ \textbf{if } f=1 \textbf{ then } \mathsf{exists}(g,E) \\
\quad \textbf{else} \ \textbf{if } g=1 \textbf{ then } \mathsf{exists}(f,E) \\
\quad \textbf{else} \ \textbf{if } m \in E \textbf{ then} \\
\qquad l \vee_{\mathsf{BDD}} r \\
\quad \textbf{else} \\
\qquad l \cdots (m) — r \\
\quad \textbf{where} \\
\qquad m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\} \\
\qquad l = \boxed{\mathsf{relProd}(f_0, g_0, E)} \\
\qquad r = \boxed{\mathsf{relProd}(f_1, g_1, E)} \\
\qquad \textbf{where} \\
\qquad\quad (f_0, f_1) = \mathsf{cf}(f,m) \\
\qquad\quad (g_0, g_1) = \mathsf{cf}(g,m)
\end{array}
$$

Abbildung 3.25: $\mathsf{relProd}: \mathsf{ROBDD}^2 \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$

Für die Quantifikation über einzelne Variablen hat man nun die Möglichkeit, den Algorithmus exists^0 zu verwenden oder aber Gleichung (3.8) auf Seite 74. Schließlich kann man dabei noch die Reihenfolge der Variablen wählen, über die quantifiziert (max = untere Variablen zuerst, min = obere Variable zuerst).

Abbildung 3.27 zeigt deutlich, daß die Spezialalgorithmen weniger Zwischenresultate erzeugen und deshalb schneller sind (Abb. 3.26). Siehe Abb. 3.28 für weitere Details. Die Kurven für die Zeit unterscheiden sich in einer ersten Näherung durch einen konstanten Faktor. Da für die y-Achse eine logarithmische Skala gewählt wurde, ergibt sich daraus, daß die Beschleunigung jeweils mehr als linear ist.

Die Analyse des Speicherverbrauchs zeigt, daß die Verwendung des Relationenproduktes nur um einen linearen Faktor weniger BDD-Knoten generiert als die explizite Berechnung durch $\wedge_{\mathsf{BDD}}$ und exists^1. Dagegen scheinen die Versionen der sukzessiven Berechnung der Quantifikation über einzelne Variablen um mehr als einen linearen Faktor schlechter.

Dieses Experiment bestätigt ebenso die Beobachtung aus den Erläuterungen zu Abb. 3.19, daß im naiven Fall die Reihenfolge der Variablen, über die quantifiziert wird, so gewählt werden sollte, daß zuerst über die unteren (größeren) Variablen quantifiziert wird. Dieser Effekt spielt sogar eine wichtigere Rolle als die Frage, ob ein Spezialalgorithmus für die Quantifikation über Mengen benutzt wird, oder nach Gleichung (3.8) auf Seite 74 vorgegangen wird.

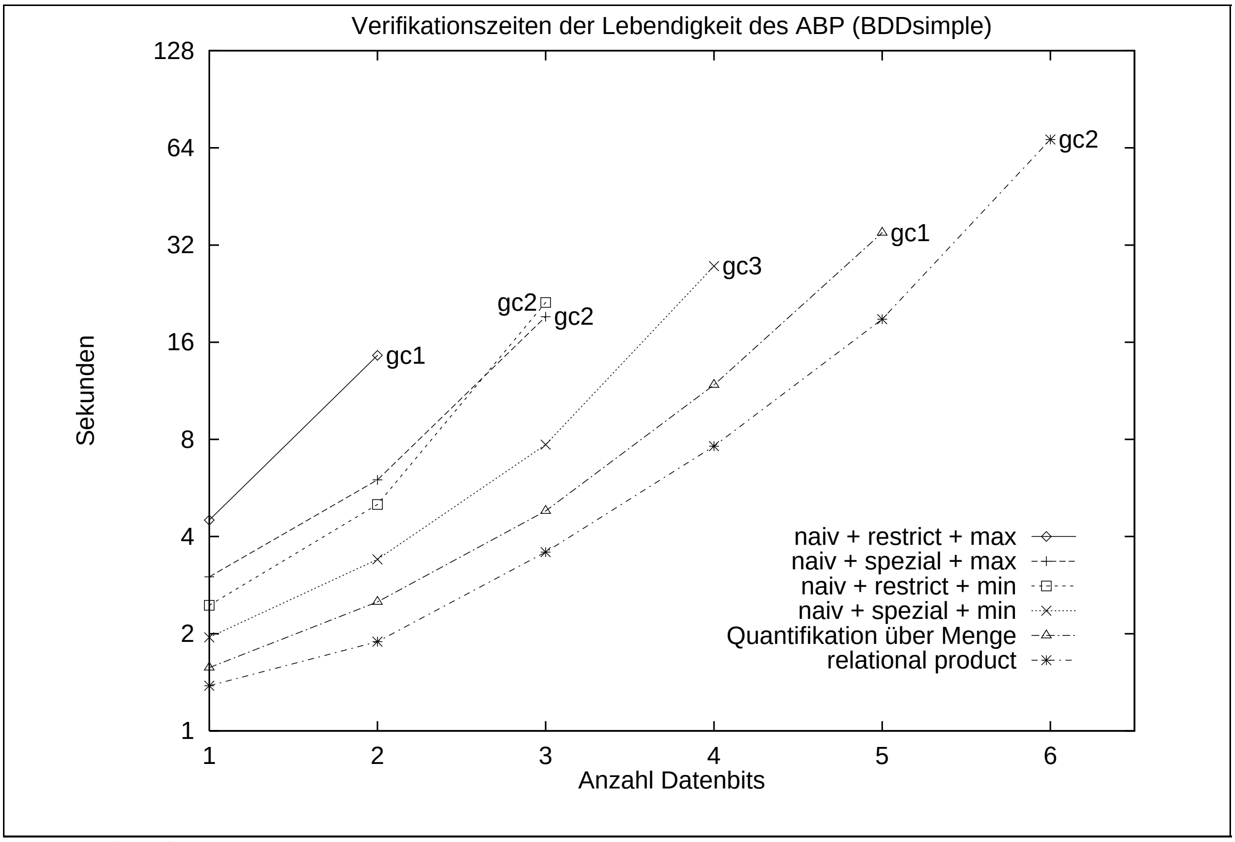


Abbildung 3.26: Vergleich verschiedener BDD-Algorithmen für Quantoren (I).

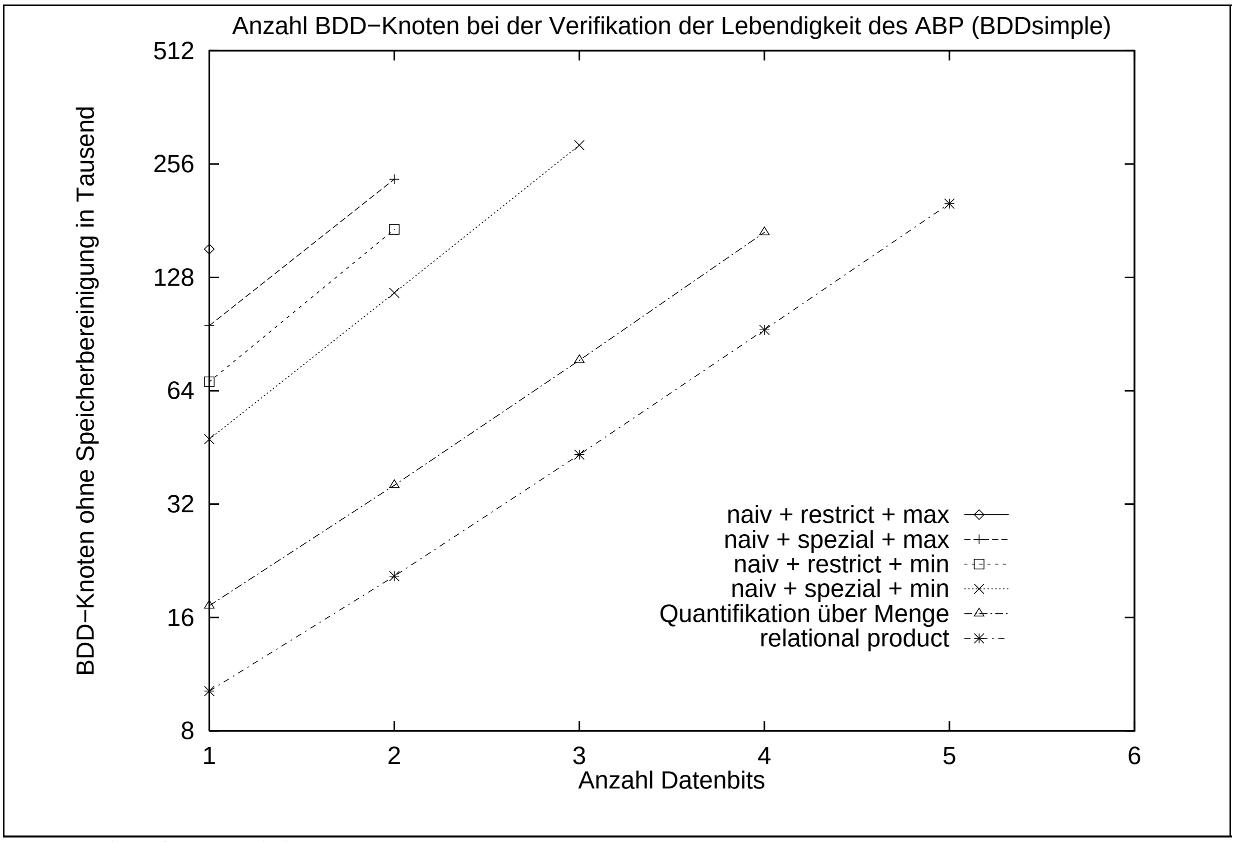


Abbildung 3.27: Vergleich verschiedener BDD-Algorithmen für Quantoren (II).

Datenbits													
	✗		✗		✗		✗						naive Quantifikation
	✗				✗								restrict
	✗		✗										max. Var. zuerst
									✗				spez. Quantifikation
											✗		relational product
	sec	#KN	sec	#KN	sec	#KN	sec	#KN	sec	#KN	sec	#KN	
1	4.5	152	3.0	95	2.5	68	2.0	48	1.6	17	1.4	10	
2	14.6	gc1	6.0	233	5.0	172	3.4	117	2.5	36	1.9	21	
3			19.2	gc2	21.2	gc2	7.7	287	4.8	77	3.6	43	
4							27.5	gc3	11.8	169	7.6	93	
5									34.9	gc1	18.9	201	
6											67.9	gc2	

Hier findet man pro Algorithmus im linken Teil der Spalte die Anzahl Sekunden, die für die Verifikation der Lebendigkeitseigenschaft der ABP benötigt wurde. Auf der rechten ist die Anzahl der BDD-Knoten angegeben, die bei der Berechnung entstanden. Die Messungen wurden bei den verschiedenen Versionen nicht mehr fortgesetzt, wenn eine Speicherbereinigung (engl. garbage collection) notwendig war. In der Tabelle ist aufgeführt, ab welcher Anzahl Datenbits das geschah (z. B. gab es bei der dritten Spalte bei 1-3 Datenbits keine Speicherbereinigung, bei 4 Datenbits waren dann aber schon drei Speicherbereinigungen notwendig). Ab 3000001 aktiven Knoten wurde die Schwelle gesetzt, ab der eine Speicherbereinigung möglich war. Dieser Wert führte bei etwa 25MB Speicherverbrauch zur Speicherbereinigung, was eben gerade noch das Auslagern des Speichers (engl. swappen) bei der verwendeten Maschine Pentium 120Hz, Linux) bei 32MB Hauptspeicher verhinderte. (vgl. Abb. 3.27 und 3.26)

Abbildung 3.28: Vergleich verschiedener BDD-Algorithmen für Quantoren (III).

$$\boxed{\mathsf{compose}^0(f,i,g)} ::=$$

$$
\begin{array}{l}
\textbf{if } f \in \{0,1\} \textbf{ then } f \\
\textbf{else } \textbf{if } \mathsf{top}(f) = i \textbf{ then} \\
\quad \mathsf{ite}_{\mathsf{BDD}}(g, \mathsf{hi}(f), \mathsf{lo}(f)) \\
\textbf{else} \\
\quad l \cdots (m) - r \\
\quad \textbf{where} \\
\qquad m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\} \\
\qquad l = \boxed{\mathsf{compose}^0(f_0, i, g_0)} \\
\qquad r = \boxed{\mathsf{compose}^0(f_1, i, g_1)} \\
\quad \textbf{where} \\
\qquad (f_0, f_1) = \mathsf{cf}(f, m) \\
\qquad (g_0, g_1) = \mathsf{cf}(g, m)
\end{array}
$$

Abbildung 3.29: $\mathsf{compose}^0 : \mathsf{ROBDD} \times \mathbb{N} \times \mathsf{ROBDD} \to \mathsf{ROBDD}$

3.8.2 Schnelle Algorithmen für Substitutionen

Auch für Substitutionen gilt das oben gesagte, daß man Zwischenergebnisse, wo möglich, vermeiden sollte. Man sollte also nicht nach Gleichung (3.10) auf Seite 74 vorgehen, sondern statt dessen Spezialgorithmen verwenden. Mit demselben Ansatz der Verschränkung von Algorithmen erhält man aus Gleichung (3.10) den Algorithmus aus Abb. 3.29, der so auch in der BDD-Bibliothek /Long, 1994/ zu finden ist. Da dieser Algorithmus in der dem Autor bekannten Literatur noch nicht dokumentiert ist, wird hier die Semantik mit Beweis angegeben.

Satz 3.27 (Semantik von $\mathsf{compose}^0$) *Es gilt*

$$\langle\!\langle \mathsf{compose}^0(f,i,g) \rangle\!\rangle_A \equiv \langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\}$$

für $f, g \in \mathsf{ROBDD}$, $x \in W$, $A \in \mathcal{A}$, $\mathsf{range}(A) \supseteq \mathsf{var}(f,g) \cup \{i\}$ und $A(x) = i$.

Beweis: Der Beweis wird geführt durch Induktion über den Termaufbau von f. Für $f \in \{0,1\}$ ist nichts zu zeigen. Nun sei f keine Konstante. Falls $\mathsf{top}(f) = i$, so erhält man

$$
\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\} &= (x \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \vee \overline{x} \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A)\{x \mapsto \langle\!\langle g \rangle\!\rangle_A\} \\
&\equiv \langle\!\langle g \rangle\!\rangle_A \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A \vee \overline{\langle\!\langle g \rangle\!\rangle_A} \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A \\
&\equiv \mathsf{ite}(\langle\!\langle g \rangle\!\rangle_A, \langle\!\langle \mathsf{hi}(f) \rangle\!\rangle_A, \langle\!\langle \mathsf{lo}(f) \rangle\!\rangle_A) \\
&\equiv \langle\!\langle \mathsf{ite}_{\mathsf{BDD}}(g, \mathsf{hi}(f), \mathsf{lo}(f)) \rangle\!\rangle_A \\
&= \langle\!\langle \mathsf{compose}^0(f,i,g) \rangle\!\rangle_A
\end{aligned}
$$

nach Satz 3.19. Sei also $\mathsf{top}(f) \neq i$ und die Hilfsvariablen seien wie im Algorithmus definiert. Nach der Hilfsbehauptung aus Gleichung (3.6) auf Seite 69 und der Variablen y definiert als $y := A^{-1}(m) \neq x$ gilt

$$\langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\} \equiv (y \langle\!\langle f_1 \rangle\!\rangle_A \vee \overline{y} \langle\!\langle f_0 \rangle\!\rangle_A)\{x \mapsto y \langle\!\langle g_1 \rangle\!\rangle_A \vee \overline{y} \langle\!\langle g_0 \rangle\!\rangle_A\}$$

$$\boxed{\mathsf{compose}(f,\rho)} ::=$$

$$\begin{array}{l}
\textbf{if } f \in \{0,1\} \textbf{ then } f \\
\textbf{else} \\
\quad \mathsf{ite}_{\mathsf{BDD}}(\rho(m), r, l) \\
\quad \textbf{where} \\
\qquad m = \mathsf{top}(f) \\
\qquad l = \boxed{\mathsf{compose}(\mathsf{lo}(f),\rho)} \\
\qquad r = \boxed{\mathsf{compose}(\mathsf{hi}(f),\rho)}
\end{array}$$

Abbildung 3.30: $\mathsf{compose}\colon \mathsf{ROBDD} \times [\mathbb{N} \to \mathsf{ROBDD}] \to \mathsf{ROBDD}$

Wenn man die Substitution $\{y \mapsto 1\}$ auf beide Seiten anwendet, erhält man

$$(\langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\})\{y \mapsto 1\} \equiv \langle\!\langle f_1 \rangle\!\rangle_A \{x \mapsto \langle\!\langle g_1 \rangle\!\rangle\} \equiv \langle\!\langle \mathsf{compose}^0(f_1, i, g_1) \rangle\!\rangle_A = \langle\!\langle r \rangle\!\rangle_A$$

und entsprechend für die Substitution $\{y \mapsto 0\}$

$$(\langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\})\{y \mapsto 0\} \equiv \langle\!\langle f_0 \rangle\!\rangle_A \{x \mapsto \langle\!\langle g_0 \rangle\!\rangle\} \equiv \langle\!\langle \mathsf{compose}^0(f_0, i, g_0) \rangle\!\rangle_A = \langle\!\langle l \rangle\!\rangle_A$$

Nach dem Shannonschen Expansionstheorem (Satz 2.12) folgt

$$\langle\!\langle f \rangle\!\rangle_A \{x \mapsto \langle\!\langle g \rangle\!\rangle_A\} \equiv \mathrm{ite}(y, r, l) \equiv \langle\!\langle l \cdots\!\!\!\; m \!\!-\!\!\!- r \rangle\!\rangle_A = \mathsf{compose}^0(f, i, g)$$

Die Terminierung des Algorithmus und $\mathsf{compose}^0(f,i,g) \in \mathsf{ROBDD}$ folgen analog zum Beweis von Satz 3.19. ❑

Dieser Algorithmus läßt sich wie beim Übergang von exists^0 zu exists^1 für beliebige Substitutionen verallgemeinern. Die auf Seite 53 aufgeführten BDD-Bibliotheken verwenden bis auf die vom SMV-System eine Variante des in Abb. 3.30 dargestellten Algorithmus. Dieser erwartet eine „BDD-Substitution“ $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$, die bei einer Allokation A, deren Bild alle auftretenden BDD-Variablen enthält, der folgenden Substitution entspricht.

Definition 3.28 (Assoziierte Substitution) *Zu* $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$ *und*

$$A \in \mathcal{A}, \quad \textit{mit} \quad \mathsf{range}(A) \supseteq \bigcup_{i \in \mathbb{N}} \mathsf{var}(\rho(i))$$

heiße

$$\gamma\colon W \to \mathbb{B}^W \quad \textit{mit} \quad \gamma = \{u \mapsto \langle\!\langle \rho(i) \rangle\!\rangle_A \mid A(u) = i\}$$

die zu ρ assoziierte *Substitution auf* $\mathbb{B}^W$.

Satz 3.29 (Semantik von $\mathsf{compose}$**)** *Sei* $f \in \mathsf{ROBDD}$, $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$ *so gilt*

$$\langle\!\langle \mathsf{compose}(f,\rho) \rangle\!\rangle_A \equiv \langle\!\langle f \rangle\!\rangle_A \gamma$$

für γ *und* A *wie in Definition 3.28.*

Dieser Algorithmus hat den Nachteil, daß im Spezialfall der Substitution von nur einer Variablen (alle anderen werden auf sich selbst abgebildet) wiederum dieses mehrfache Durchlaufen von Zwischenergebnissen auftreten kann. Wenn zum Beispiel eine Variable weit unten im BDD durch einen BDD ersetzt wird, der eine kleine Variable (in der Ordnung weit oberhalb der ersetzten) enthält, so durchläuft beim nachfolgenden rekursiven Aufstieg jedesmal die $\mathsf{ite}_{\mathsf{BDD}}$ Prozedur das Zwischenergebnis bis hinunter zur aktuellen Variable.

In diesem Fall ist der Algorithmus $\mathsf{compose}^0$ vorzuziehen. Hier verhält sich also die Verallgemeinerung schlechter als der spezielle Algorithmus. Leider läßt sich compose nicht durch $\mathsf{compose}^0$ berechnen. Denn compose realisiert eine *parallele* Substitution, was nicht der Hintereinander-Ausführung des letzten Algorithmus entspricht (vgl. auch Definition A.8). So kann man nicht wie in Abbildung 3.28 für exists Laufzeiten für compose mit denen von $\mathsf{compose}^0$ an Hand der Laufzeiten für die Modellprüfung vergleichen, weil die Modellprüfung die parallele Substitution benötigt. Dem Autor ist kein Algorithmus bekannt, der dieses Problem löst.

Auf der anderen Seite kann durch diesen Algorithmus ein BDD zu einer Allokation A in einen BDD einer beliebigen anderen Allokation B umgerechnet werden (Bestimmung des BDDs unter einer anderen „Variablenordnung“). Bei diesem Vorgang kann sich der BDD exponentiell vergrößern, was in Kapitel 4 genauer betrachtet wird. Deshalb muß die „worst case“-Komplexität exponentiell sein.

3.8.3 Weitere Spezialalgorithmen

Der preImg-Operator

Für die Verifikation von hauptsächlich[11] deterministischen Systemen, bei denen die Übergangsrelation eine Funktion darstellt, wurde in /Coudert et al., 1989/ für die Berechnung des Relationenproduktes eine andere Vorgehensweise gewählt. Zunächst wird in diesem Ansatz nicht der BDD f für die Übergangsrelation über den Variablen aus $S = \{s_1, \ldots, s_n\}$ und $S' = \{s'_1, \ldots, s'_n\}$, die den momentanen bzw. Folgezustand kodieren, und den Eingabevariablen I berechnet, sondern nur ein Vektor von BDDs $(f_1, \ldots, f_n)$, wobei f_i von Variablen aus S und I abhängt. Die Hoffnung dabei ist, daß alle f_i zusammen wesentlich kleiner sind als f.

Der Zusammenhang zwischen beiden Varianten der Kodierung von Systemen ist dann in einer vermischten[12] Syntax zwischen BDD und $\mathbb{B}^V_\mu$ gegeben durch

$$f(i,s,s') \;=\; \bigwedge_{i=1}^{n} f_i(i,s) \leftrightarrow s'_i$$

Aufgabe hierbei ist es, unter anderem, den BDD für

$$\mathsf{preImg}(g,[f_1,\ldots,f_n],I) \;:=\; \exists I.\,\exists s'.\,g(s') \wedge f(i,s,s')$$

zu berechnen, ohne tatsächlich den BDD für f berechnen zu müssen. Der Zusammenhang zwischen den f_i und f ergibt zunächst

$$\exists I.\,\exists s'.\,g(s') \wedge \bigwedge_{i=1}^{n} f_i(i,s) \leftrightarrow s'_i$$

[11]Es läßt sich jedes echt nichtdeterministische System durch Einführung weiterer „Orakel“-Eingaben in ein deterministisches überführen.

[12]Für die weitere Darstellung sei es erlaubt, von dem formalen Weg der Trennung zwischen den einzelnen Logiken abzuweichen. Die technischen Schwierigkeiten bei einer exakten Darstellung sind allzu groß.

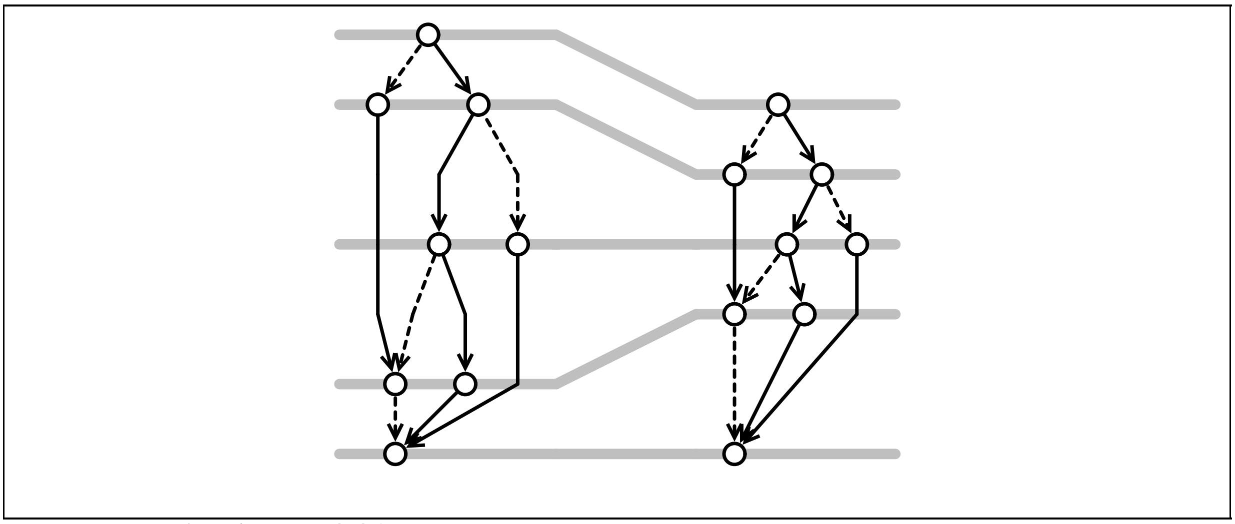


Abbildung 3.31: Veranschaulichung von monotonen Substitutionen.

Mehrmalige Anwendung von Gleichung (3.9) auf Seite 74 ergibt

$$\exists I.\, g(s)\{s_i \mapsto f_i\}$$

Der BDD für diesen Term läßt sich durch den später vorgestellten $\mathsf{compose}_\exists$-Algorithmus berechnen. Das Verfahren von /Coudert et al., 1989/ arbeitet direkt auf der Graphenstruktur und verlangt, daß an den BDD-Knoten Zwischenergebnisse abgelegt werden können.

Monotone Substitutionen und der collapse-Algorithmus

Für manche Substitutionen $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$ gibt es für bestimmte ROBDDs noch einen „direkteren" Algorithmus als der von compose. Die asymptotische Komplexität ändert sich dabei nicht. Dafür kann der Algorithmus als Grundlage für weitere Optimierungen verwendet werden und hat auf jeden Fall eine bessere Konstante. Grundlegend für diesen verbesserten Algorithmus sind die folgenden zwei Definitionen.

Definition 3.30 (Variablensubstitution) *Eine Abbildung* $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$ *heißt* Variablensubstitution *auf* $W' \subseteq W$ *oder bez.* $g \in \mathsf{ROBDD}$ *mit* $W' \supseteq \mathsf{var}(g)$ *genau dann, wenn für alle* $i \in W'$ *ein* $j \in \mathbb{N}$ *existiert mit* $\rho(i) = 0 \cdots j — 1$.

Die Substitutionen, die bei der Auswertung der BDD-Semantik nach Satz. 3.24 auftreten, sind alle von dieser Art. Es ist aber grundsätzlich in $\mathbb{B}^V_\mu$ erlaubt, z. B. wie in dem Term $X(u,u)$, daß zwei Argumente der Anwendung einer Prädikatsvariablen (X) aus derselben Variable (u) bestehen. Solche Substitutionen erfordern den allgemeinen compose-Algorithmus. Sie sind in einem gewissen Sinne nicht *injektiv* und der allgemeine compose-Algorithmus muß verwendet werden. Statt dessen können aber bei einer entsprechenden Wahl der Allokation und, wenn solches mehrfaches Auftreten einer Variable in einer Argumentliste nicht vorkommt, auch *injektive* Substitutionen gewählt werden. Dabei sind insbesondere solche *injektiven* Substitutionen interessant, die zusätzlich noch die Ordnung auf dem OBDD beachten.

$\boxed{\mathsf{fastsub}(f,\rho)}$ ::=

if $f \in \{0,1\}$ **then** f
else
  $l \cdots\textcircled{m}\!-\, r$
  where
    $\rho(\mathsf{top}(f)) = 0 \cdots\textcircled{m}\!-\, 1$
    $l = \boxed{\mathsf{fastsub}(\mathsf{lo}(f),\rho)}$
    $r = \boxed{\mathsf{fastsub}(\mathsf{hi}(f),\rho)}$

Abbildung 3.32: $\mathsf{fastsub}\colon \mathsf{ROBDD} \times [\mathbb{N} \to \mathsf{ROBDD}] \to \mathsf{ROBDD}$, für ρ monotone Substitution.

Definition 3.31 (Monotone Substitution) *Eine Substitution* $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$ *heißt* monotone Substitution *auf* $W' \subseteq W$ *oder bez.* $g \in \mathsf{ROBDD}$ *mit* $W' \supseteq \mathsf{var}(g)$ *genau dann, wenn* ρ *eine Variablensubstitution ist, und für alle* $i, j \in W'$ *aus* $i < j$ *auch* $\mathsf{top}(\rho(i)) < \mathsf{top}(\rho(j))$ *folgt.*

Monotone Substitutionen lassen sich graphisch wie in Abb. 3.31 veranschaulichen. Hierbei erhält man den rechten BDD aus dem linken durch eine monotone Substitution. Wie in diesem Beispiel zeichnet sich eine monotone Substitution dadurch aus, daß sich die grauen Linien nicht schneiden oder auf der rechten Seite zusammenfallen. Nur dann hat der rechte BDD dieselbe Gestalt wie der linke *und* geht aus dem linken durch eine Substitution hervor. Für monotone Substitutionen kann man den Spezialalgorithmus aus Abb. 3.32 verwenden. Seine Semantik erschließt sich aus folgendem Satz. I. allg. würde die Verwendung dieser Methode zur Berechnung von Substitutionen auf ROBDDs die Ordnung zerstören und nicht einmal die Semantik erhalten.

Satz 3.32 (Semantik von $\mathsf{fastsub}$**)** *Es gilt*

$$\mathsf{fastsub}(f,\rho) = \mathsf{compose}(f,\rho),$$

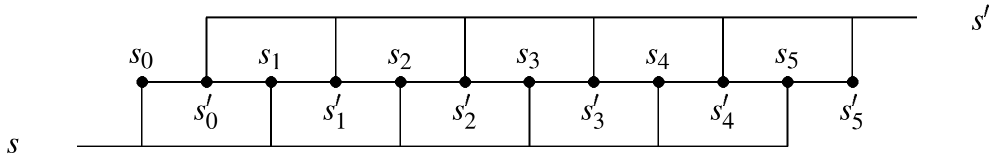
für $f \in \mathsf{ROBDD}$ *und* ρ *eine monotone Substitution.*

Beweis: Der Beweis wird wie üblich über den Termaufbau von f geführt. Der Induktionsanfang $f \in \{0,1\}$ ist trivial. Für $f = f_0 \cdots\textcircled{i}\!-\, f_1$ und $m := \mathsf{top}(\rho(i))$ gilt nach Induktionsvoraussetzung

$$\begin{aligned}
\mathsf{compose}(f,\rho) &= \mathsf{ite}(0 \cdots\textcircled{m}\!-\, 1, \mathsf{compose}(f_1,\rho), \mathsf{compose}(f_0,\rho)) \\
&= \mathsf{ite}(0 \cdots\textcircled{m}\!-\, 1, \mathsf{fastsub}(f_1,\rho), \mathsf{fastsub}(f_0,\rho)) \\
&= \mathsf{fastsub}(f_0,\rho) \cdots\textcircled{m}\!-\, \mathsf{fastsub}(f_1,\rho)) \\
&= \mathsf{fastsub}(f,\rho),
\end{aligned}$$

da ρ monoton ist und deshalb alle Variablen in $\mathsf{fastsub}(f_1,\rho)$ und $\mathsf{fastsub}(f_0,\rho)$ größer als m sind. ❑

Eine Weiterentwicklung dieser Idee ist der collapse-Algorithmus aus dem SMV-System, der bestimmte Voraussetzungen an die Variablen der beteiligten BDDs macht. So werden die Variablen des aktuellen Zustandes (s) mit denen des Folgezustandes (s') verschränkt allokiert (vgl. 4). Die des aktuellen Zustandes werden durch gerade BDD-Variablen (natürliche Zahlen) kodiert und die des Folgezustandes durch ungerade. Wenn $2 \cdot i$ eine BDD-Variable ist, die ein Bit des aktuellen Zustandes kodiert, so erhält man das entsprechende Bit der Kodierung des Folgezustandes als $2 \cdot i + 1$. Für durch 6 Bits kodierte Zustände kann man solch eine Allokation wie folgt visualisieren.



Als Beispiel betrachte man die Übersetzung der CTL-Formel EFp (vgl. B.2.3), die der Prädikatsvariablen X in $\mathbb{B}^V_\mu$ unter der Prädikatsvariablendefinition

$$\mu X(s) \,.\, p(s) \;\vee\; \exists s' \,.\, T(s,s') \wedge X(s')$$

entspricht. Bei Berechnung der Approximationen der BDD-Semantik nach Satz 3.24 werden in einem BDD über den Variablen, die für den momentanen Zustand s allokiert wurden, all diese durch Variablen substituiert, die für den Folgezustand s' allokiert wurden. In der hier getroffenen Festlegung bedeutet dies, man hat den BDD auf den geraden Variablen, braucht ihn aber auf den ungeraden. Durch die besondere Wahl der Allokation ist die benötigte Substitution monoton und man kann den Algorithmus aus Abb. 3.32 verwenden.

Man kann aber auch noch einen Schritt weiter gehen und für den eben beschriebenen Fall des Auftretens einer monotonen Substitution in einem Relationenprodukt einen speziellen Algorithmus angeben. Dies ist der in Abb. 3.33 beschriebene collapse-Algorithmus.

Satz 3.33 (Semantik von collapse**)** *Für* $f, g \in \mathsf{ROBDD}$, $\mathsf{var}(g) \subseteq 2 \cdot \mathbb{N}$ *gilt*

$$\mathsf{collapse}(f,g) = \mathsf{exists}(\mathsf{compose}(g,\rho),E),$$

mit $E := 2 \cdot \mathbb{N} + 1$ *und* $\rho := \{ j \mapsto 0 \cdots i - 1 \mid j \in 2 \cdot \mathbb{N},\ i = j+1 \}$.

Beweis: ρ ist eine monotone Substitution bez. g. Deshalb genügt es nach Satz 3.32 und der Semantik von relProd zu zeigen

$$\mathsf{collapse}(f,g) = \mathsf{relProd}(f,\mathsf{fastsub}(g,\rho),E),$$

was durch Induktion über den Termaufbau von f und g geschieht. Die Fälle $f = 0$ oder $g \in \{0,1\}$ ergeben sich unmittelbar. Im Falle $f = 1$ und $g \neq 0$ beachte man, daß $\mathsf{var}(g) \subseteq E$ und somit $\mathsf{exists}(g,E) = 1$. Seien nun $f,g \notin \{0,1\}$ und die Bezeichner wie im Algorithmus gewählt. Für $m < \mathsf{top}(g)+1$ ist $m-1 < \mathsf{top}(g)$, so daß $\mathsf{cf}(g,m-1) = (g,g)$, und die Induktionsvoraussetzung ergibt den Rest. Sei nun $m = \mathsf{top}(g)+1$ oder $\mathsf{top}(g) = m-1$. Dann gilt mit $g' := \mathsf{fastsub}(g,\rho)$ und $\rho(m-1) = m$

$$g' = g'_0 \cdots m - g'_1, \quad (\mathsf{lo}(g),\mathsf{hi}(g)) = \mathsf{cf}(g,m-1), \quad \text{mit}$$
$$g'_0 := \mathsf{fastsub}(\mathsf{lo}(g),\rho), \quad g'_1 := \mathsf{fastsub}(\mathsf{hi}(g),\rho)$$

$\boxed{\mathsf{collapse}(f,g)}$ ::=

if $f = 0$ **or** $g = 0$ **then** 0
else if $f = 1$ **then** 1
else if $g = 1$ **then** $\mathsf{exists}(f, 2\mathbb{N}+1)$
else if $m \in 2\mathbb{N}+1$ **then**
$\quad l \vee_{\mathsf{BDD}} r$
else
$\quad l \cdots (m) — r$
where
$\quad m = \min\{\mathsf{top}(f), \mathsf{top}(g)+1\}$
$\quad l \; = \boxed{\mathsf{collapse}(f_0, g_0)}$
$\quad r \; = \boxed{\mathsf{collapse}(f_1, g_1)}$
$\quad$**where**
$\quad\quad (f_0, f_1) = \mathsf{cf}(f, m)$
$\quad\quad (g_0, g_1) = \mathsf{cf}(g, m-1)$

Abbildung 3.33: $\mathsf{collapse}\colon \mathsf{ROBDD}^2 \to \mathsf{ROBDD} \quad (\mathsf{var}(g) \subseteq 2\cdot\mathbb{N})$

Sei nun $m \in E = 2\cdot\mathbb{N}+1$. Damit ist

$$\begin{aligned}
\mathsf{relProd}(f, g', E) &= \mathsf{relProd}(f_0, g'_0, E) \;\vee_{\mathsf{BDD}}\; \mathsf{relProd}(f_1, g'_1, E) \\
&= \mathsf{relProd}(f_0, \mathsf{fastsub}(\mathsf{lo}(g), \rho), E) \;\vee_{\mathsf{BDD}} \\
&\quad\; \mathsf{relProd}(f_1, \mathsf{fastsub}(\mathsf{hi}(g), \rho), E) \\
&= \mathsf{collapse}(f_0, \mathsf{lo}(g)) \;\vee_{\mathsf{BDD}}\; \mathsf{collapse}(f_1, \mathsf{hi}(g)) \\
&= \mathsf{collapse}(f_0, g_0) \;\vee_{\mathsf{BDD}}\; \mathsf{collapse}(f_1, g_1) \\
&= \mathsf{collapse}(f, g)
\end{aligned}$$

Analog für $m \notin E$. ❑

Wie aus dem Beweis ersichtlich kann dieser Algorithmus auf allgemeine monotone Substitutionen erweitert werden. So wurde hier nur die „Rückwärtsanalyse“ vorgestellt, die für die Berechnung von einfachen CTL und FCTL Semantiken ausreicht. Für die Berechnung der erreichbaren Zustände wird statt dessen eine „Vorwärtsanalyse“ notwendig. Hier lautet die Prädikatsdefinition wie auf Seite 19

$$\mu R(s)\,.\,S(s) \;\vee\; \exists t\,.\,T(t,s) \wedge R(t)$$

Der prinzipielle Unterschied zum vorigen Beispiel besteht darin, daß die Variable über die quantifiziert wird, dem momentanen Zustand entspricht. Auch hierfür läßt sich eine leicht veränderte Variante des collapse-Algorithmus angeben, die im SMV-System für diesen Fall eingesetzt wird.[13]

[13] Der preImg-Algorithmus ist nur für die „Rückwärtsanalyse“ gedacht. So wurde in /Coudert et al., 1989/ zusätzlich ein Img-Algorithmus vorgeschlagen, der nach einer völlig anderen Methode als der preImg arbeitet und hier nicht weiter betrachtet werden soll.

$$
\begin{array}{l}
\boxed{\mathsf{onsetsize}(f,E)} ::= \\
\quad \textbf{if } f=0 \textbf{ then } 0 \\
\quad \textbf{else if } f=1 \textbf{ then } 2^{|E|} \\
\quad \textbf{else} \\
\quad\quad 2^k \cdot (l+r) \\
\quad\quad \textbf{where} \\
\quad\quad\quad E' = \{e \in E \mid e > \mathsf{top}(f)\} \\
\quad\quad\quad l \ = \boxed{\mathsf{onsetsize}(\mathsf{lo}(f),E')} \\
\quad\quad\quad r \ = \boxed{\mathsf{onsetsize}(\mathsf{hi}(f),E')} \\
\quad\quad\quad k \ = |E \backslash E'| - 1
\end{array}
$$

Abbildung 3.34: $\mathsf{onsetsize}\colon \mathsf{ROBDD} \times \mathbb{P}(\mathbb{N}) \to \mathbb{N}$, wobei $\mathsf{var}(f) \subseteq E$.

Der collapse-Algorithmus vermeidet das Erstellen einer Kopie des zweiten Argumentes. Er kann aber nur angewendet werden, wenn man die Allokation entsprechend wählt. Für nicht monotone Substitutionen liefert er ein falsches Ergebnis. In Abschnitt 3.8.4 wird deshalb ein neuer Algorithmus vorgestellt, der den collapse-Algorithmus als Spezialfall umfaßt und auch für beliebige Substitutionen korrekt arbeitet.

Der onsetsize-Algorithmus

Bei der Erreichbarkeitsanalyse wie beim Beispiel aus Gleichung (2.6.1) auf Seite 19 wird die Menge der erreichbaren Zustände als BDD repräsentiert. Hier ist der Anwender des öfteren auch an der Kardinalität dieser Menge interessiert. Diese Zahl ist gleich der Anzahl erfüllender Belegungen für den entsprechenden booleschen Ausdruck (vgl. Def. 2.3).

Genauer muß man hier nur die Anzahl derjenigen Belegungen zählen, die sich auf einer ausgezeichneten Menge von Variablen unterscheiden und sonst einen beliebigen aber für alle gleichen Wert besitzen. In dem erwähnten Beispiel interessiert man sich also nur für

$$|\{\rho \in [W \to \mathbb{B}] \mid [\![\langle\!\langle R(u)\rangle\!\rangle_A]\!]\rho = 1 \text{ und } \rho(x) = 0 \text{ für alle } x \in W\backslash u\}|,$$

für entsprechend gewähltes $A \in \mathcal{A}$. Hierzu kann man einen Algorithmus verwenden, der als erstes Argument einen BDD f und als zweites eine Menge E von BDD-Variablen als Parameter erhält. Dabei wird $\mathsf{var}(f) \subseteq E$ vorausgesetzt. Dieser Algorithmus ist in Abb. 3.34 aufgezeigt.Seine Semantik ist aus folgendem Satz ersichtlich.

Satz 3.34 (Semantik von onsetsize) *Für $f \in \mathsf{ROBDD}$, $E \subseteq \mathbb{N}$ mit $\mathsf{var}(f) \subseteq E$ gilt*

$$\mathsf{onsetsize}(f,E) = |\{\underline{a} \in \mathbb{B}^n \mid \langle\!\langle f\rangle\!\rangle_A \{\underline{x} \mapsto \underline{a}\} \equiv 1\}|,$$

wobei $A \in \mathcal{A}$, mit $\mathsf{range}(A) \supseteq \mathsf{var}(f) \cup E$ und

$$E = \{e_1, e_2, \ldots, e_n\}, \quad |E| = n \in \mathbb{N}, \quad x_i := A^{-1}(e_i)$$

für $i \in \{1,\ldots,n\}$. Dabei sei $\{\underline{x} \mapsto \underline{a}\}$ wie auf Seite 170 zu verstehen.

Beweis: Zunächst seien die Elemente von E geordnet durch $e_1 < e_2 < \cdots < e_n$ und die Bezeichungen wie im Algorithmus gewählt. Die Fälle $f = 0$ oder $f = 1$ sind unmittelbar klar. Im rekursiven Fall sei $x := A^{-1}(\mathsf{top}(f))$.[14] Nun wähle ein $\underline{a} \in \mathbb{B}^n$ beliebig aus. Jetzt zerlege man $\underline{x}$ in $\underline{y}\, x\, \underline{z} = \underline{x}$, für $\underline{y} \in W^k$, $\underline{z} \in W^{n-k-1}$ und $\underline{a}$ in $\underline{b}\, a\, \underline{c} = \underline{a}$, für $\underline{b} \in \mathbb{B}^k$, $\underline{c} \in \mathbb{B}^{n-k-1}$ und $a \in \mathbb{B}$, so daß $A(\underline{y})$ als Menge gleich $E \backslash (E' \cup \{\mathsf{top}(f)\})$ ist und ebenso $A(\underline{z}) = E'$. Weiter sei zunächst $m := n - k - 1 > 0$. Da $A(\underline{y}) \cap \mathsf{var}(f) = \emptyset$, folgt weiter mit der Definition von $\langle\!\langle \cdot \rangle\!\rangle_A$

$$
\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \{\underline{x} \mapsto \underline{a}\} \equiv 1 &\Leftrightarrow \langle\!\langle f \rangle\!\rangle_A \{\underline{y} \mapsto \underline{b}\}\ \{x \mapsto a\}\ \{\underline{z} \mapsto \underline{c}\} \equiv 1 \\
&\Leftrightarrow \langle\!\langle f \rangle\!\rangle_A \{x \mapsto a\}\ \{\underline{z} \mapsto \underline{c}\} \equiv 1 \\
&\Leftrightarrow (x \,\langle\!\langle\, \mathsf{hi}(f) \rangle\!\rangle_A \ \vee\ \neg x \,\langle\!\langle\, \mathsf{lo}(f) \rangle\!\rangle_A)\{x \mapsto a\}\ \{\underline{z} \mapsto \underline{c}\} \equiv 1 \\
&\Leftrightarrow \begin{cases} \langle\!\langle\, \mathsf{lo}(f) \rangle\!\rangle_A \{\underline{z} \mapsto \underline{c}\} \equiv 1 & \text{falls } a = 0 \\ \langle\!\langle\, \mathsf{hi}(f) \rangle\!\rangle_A \{\underline{z} \mapsto \underline{c}\} \equiv 1 & \text{falls } a = 1 \end{cases}
\end{aligned}
$$

Weil $\underline{a}$ und somit a beliebig sind und das Prefix $\underline{b}$ von $\underline{a}$ keine Rolle spielt, erhält man

$$
\begin{aligned}
&|\{\underline{a} \in \mathbb{B}^n \mid \langle\!\langle f \rangle\!\rangle_A \{\underline{x} \mapsto \underline{a}\} \equiv 1\}| \\
&\quad = |\{\underline{b} \in \mathbb{B}^k\}| \cdot \big(\, |\{\underline{c} \in \mathbb{B}^m \mid \langle\!\langle\, \mathsf{lo}(f) \rangle\!\rangle_A \{\underline{x} \mapsto \underline{a}\} \equiv 1\}| + \\
&\qquad\qquad |\{\underline{c} \in \mathbb{B}^m \mid \langle\!\langle\, \mathsf{hi}(f) \rangle\!\rangle_A \{\underline{x} \mapsto \underline{a}\} \equiv 1\}| \quad \big) = 2^k \cdot (l + r)
\end{aligned}
$$

Im Falle $m = 0$ gilt nach Vorausetzung $\{\mathsf{lo}(f), \mathsf{hi}(f)\} = \{0, 1\}$, womit $l + r = 1$, und analog zum Fall $m > 0$ durch Weglassen der Suffixe $\underline{z}$ und $\underline{c}$ folgt der Rest. ❑

Zur Implementierung ist zu sagen, daß auch hier die Darstellung endlicher Mengen von natürlichen Zahlen durch BDDs wie auf Seite 82 Verwendung findet. Des weiteren sollte man auf Arithmetik mit Fließkommazahlen zurückgreifen, da das Resultat dieses Algorithmus aus sehr großen Zahlen bestehen kann (z. B. über 10^{20}).

Der Constrain- und Restrict-Operator

In /Coudert et al., 1989/ wurden zwei BDD-Operatoren eingeführt, die dazu dienen, einen BDD zu vereinfachen, unter der Annahme der „Gültigkeit" eines weiteren BDDs. Darüber hinaus kann man diese Operatoren dazu benützen, um das Relationenprodukt in eine einfache Quantorenberechnung zu überführen (s. Satz 3.41). Dies bildet die Grundlage des Algorithmus für den schon oben erwähnten Img-Operator aus /Coudert et al., 1989/, der es erlaubt, bei der Erreichbarkeitsanalyse (Vorwärtsanalyse) von deterministischen Systemen auf die Berechnung der globalen Übergangsrelation zu verzichten.

Weitere Anwendung finden diese Algorithmen in den Optimierungen, die in Abschnitt 5.3 vorgestellt werden. Eine neuere Arbeit hierzu ist /McMillan, 1996/. Dort wird eine Technik zur Repräsentation der Übergangsrelation vorgestellt, die implizit funktionale Abhängigkeiten zwischen BDD-Variablen ausnützt, um die Größe der entstehenden BDDs in Grenzen zu halten.

Für den Constrain-Operator findet man in der Literatur auch die Bezeichnung „generalized cofactor". Dies rührt daher, daß er eine Verallgemeinerung des Kofaktors aus Def. 2.8 darstellt, und somit eine Verallgemeinerung des Shannonschen Expansionstheorem erfüllt.

Die Algorithmen für den Constraint-Operator, („$\downarrow$") und den Restrict-Operator („$\Downarrow$") sind in Abbildungen 3.35 und 3.36 dargestellt. In den Algorithmen aus /Coudert et al., 1989/ und

[14] Man verzeihe die Überladung des Symbols „x". So hat „x" mit „$\underline{x}$" nichts zu tun. Ersteres ist eine Variable aus W und letzteres ein *Vektor* von Variablen (ein Element aus W^n).

$\boxed{f \downarrow g}$ ::=

if $g = 0$ **then** 0
else **if** $f \in \{0,1\} \vee g = 1$ **then** f
else
  if $g_0 = 0$ **then** $f_1 \downarrow g_1$
  else **if** $g_1 = 0$ **then** $f_0 \downarrow g_0$
  else
    $l \cdots (m) — r$
    where
      $l = \boxed{f_0 \downarrow g_0}$
      $r = \boxed{f_1 \downarrow g_1}$
  where
    $m \quad = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\}$
    $(f_0, f_1) = \mathsf{cf}(f, m)$
    $(g_0, g_1) = \mathsf{cf}(g, m)$

Abbildung 3.35: $\downarrow : \mathsf{ROBDD}^2 \to \mathsf{ROBDD}$

$\boxed{f \Downarrow g}$ ::=

if $g = 0$ **then** 0
else **if** $f \in \{0,1\}$ **or** $g = 1$ **then** f
else
  if $m = \mathsf{top}(f)$ **then**
    if $g_0 = 0$ **then** $f_1 \Downarrow g_1$
    else **if** $g_1 = 0$ **then** $f_0 \Downarrow g_0$
    else
      $l \cdots (m) — r$
      where
        $l = \boxed{f_0 \Downarrow g_0}$
        $r = \boxed{f_1 \Downarrow g_1}$
  else
    $f \Downarrow (g_0 \vee_{\mathsf{BDD}} g_1)$
  where
    $m \quad = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g)\}$
    $(f_0, f_1) = \mathsf{cf}(f, m)$
    $(g_0, g_1) = \mathsf{cf}(g, m)$

Abbildung 3.36: $\Downarrow : \mathsf{ROBDD}^2 \to \mathsf{ROBDD}$

/Touati et al., 1990/ für diese Operationen ist der Fall $g = 0$ nicht erlaubt. Hierfür kann man aber einfach als Resultat 0 wählen, wie das auch in /McMillan, 1996/ geschehen ist.

Der Algorithmus für „$\Downarrow$“ unterscheidet sich von dem für „$\downarrow$“ in der Behandlung des Falls, daß der BDD des zweiten Argumentes eine kleinere oberste Variable hat als der des ersten. Trifft dies zu, so wird im „$\Downarrow$“-Algorithmus die obere Variable aus dem zweiten BDD „herausquantifiziert“, was durch ein „Oder“ der beiden Nachfolger realisiert wird. Dieser innere Aufruf des Algorithmus für „$\vee_{\mathsf{BDD}}$“ führt dazu, daß man wie beim Algorithmus für „exists^1“ aus Abb. 3.21 keine einfache Komplexitätsanalyse durchführen kann, sondern davon ausgehen muß, daß der Algorithmus im schlimmsten Fall exponentiell ist. Für den „$\downarrow$“-Algorithmus trifft dies nicht zu. Durch Verwendung des Ergebnisspeichers der SCAM ergibt sich hier ein lineares Verhalten bez. der Größe beider Argumente sowohl im Zeit- als auch im Platzverbrauch.

Satz 3.35 *Sei* $f, g \in \mathsf{ROBDD}$*, so gilt*

$$\langle\!\langle f \downarrow g \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \;\equiv\; \langle\!\langle f \Downarrow g \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \;\equiv\; \langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A$$

für $A \in \mathcal{A}$*,* $\mathsf{range}(A) \supseteq \mathsf{var}(f, g)$.

Beweis: Der Beweis erfolgt, wie üblich, über den Termaufbau von f und g. Ist einer der beiden BDDs in $\{0, 1\}$, so folgt die Aussage unmittelbar. Ansonsten wähle die Bezeichner wie in den Algorithmen und $x := A^{-1}(m)$. Ist $g_0 = 0$, so gilt

$$\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A &\equiv (x \,\langle\!\langle f_1 \rangle\!\rangle_A \vee \overline{x}\,\langle\!\langle f_0 \rangle\!\rangle_A) \wedge (x \,\langle\!\langle g_1 \rangle\!\rangle_A \vee \overbrace{\overline{x}\,\langle\!\langle g_0 \rangle\!\rangle_A}^{\equiv 0}) \equiv x \,\langle\!\langle f_1 \rangle\!\rangle_A \,\langle\!\langle g_1 \rangle\!\rangle_A \\
&\equiv x\,(\langle\!\langle f_1 \rangle\!\rangle_A \downarrow \langle\!\langle g_1 \rangle\!\rangle_A) \wedge \langle\!\langle g_1 \rangle\!\rangle_A \equiv \langle\!\langle f_1 \downarrow g_1 \rangle\!\rangle_A \wedge x \langle\!\langle g_1 \rangle\!\rangle_A \\
&\equiv \langle\!\langle f \downarrow g \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A
\end{aligned}$$

Der Fall $g_1 = 0$ wird genauso behandelt. Es gelte also $g_0, g_1 \neq 0$.

$$\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A &\equiv (x \,\langle\!\langle f_1 \rangle\!\rangle_A \vee \overline{x}\,\langle\!\langle f_0 \rangle\!\rangle_A) \wedge (x \,\langle\!\langle g_1 \rangle\!\rangle_A \vee \overline{x}\,\langle\!\langle g_0 \rangle\!\rangle_A) \\
&\equiv x \,\langle\!\langle f_1 \rangle\!\rangle_A \,\langle\!\langle g_1 \rangle\!\rangle_A \vee \overline{x}\,\langle\!\langle f_0 \rangle\!\rangle_A \,\langle\!\langle g_0 \rangle\!\rangle_A \\
&\equiv x\,(\langle\!\langle f_1 \downarrow g_1 \rangle\!\rangle_A \wedge \langle\!\langle g_1 \rangle\!\rangle_A) \vee \overline{x}\,(\langle\!\langle f_0 \downarrow g_0 \rangle\!\rangle_A \wedge \langle\!\langle g_0 \rangle\!\rangle_A) \\
&\equiv x\,r\,\langle\!\langle g_1 \rangle\!\rangle_A \vee \overline{x}\,l\,\langle\!\langle g_0 \rangle\!\rangle_A \equiv (x\,r \vee \overline{x}\,l) \wedge (x\,g_1 \vee \overline{x}\,g_0) \\
&\equiv \langle\!\langle f \downarrow g \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A
\end{aligned}$$

Damit ist der Beweis für „$\downarrow$“ beendet. Der Fall $m = \mathsf{top}(f)$ für „$\Downarrow$“ folgt genauso. Sei also weiterhin $m < \mathsf{top}(f)$. Unter Verwendung der Tautologie $\mathrm{ite}(a, b, c) \to (b \vee c)$ und der Gültigkeit von $d \equiv d \wedge e$ für $a, b, c, d, e \in \mathbb{B}^W$, falls $d \to e$ eine Tautologie ist, folgt

$$\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A &\equiv \langle\!\langle f \rangle\!\rangle_A \wedge \mathrm{ite}(x, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle g_0 \rangle\!\rangle_A) \equiv \langle\!\langle f \rangle\!\rangle_A \wedge (\langle\!\langle g_1 \rangle\!\rangle_A \vee \langle\!\langle g_0 \rangle\!\rangle_A) \wedge \langle\!\langle g \rangle\!\rangle_A \\
&\equiv \langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g_1 \vee_{\mathsf{BDD}} g_0 \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \\
&\equiv \langle\!\langle f \Downarrow (g_1 \vee_{\mathsf{BDD}} g_0) \rangle\!\rangle_A \wedge \langle\!\langle g_1 \vee_{\mathsf{BDD}} g_0 \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \\
&\equiv \langle\!\langle f \Downarrow g \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A
\end{aligned}$$

Letzteres wiederum mit der Tautologie „$\mathrm{ite}(x, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle g_0 \rangle\!\rangle_A) \to \langle\!\langle g_1 \vee_{\mathsf{BDD}} g_0 \rangle\!\rangle_A$“. ❑

Dieser Satz klärt die Bedeutung der beiden Operatoren, nämlich daß sie unter Annahme der Gültigkeit des zweiten Arguments einen BDD erzeugen, der zum ersten äquivalent ist. Weiterhin zeigt die Praxis, daß das Ergebnis meistens auch kleiner als der ursprüngliche BDD ist. Dennoch gibt es auch Beispiele, bei denen sich die Anzahl Knoten vergrößern kann.

Als Korollar erhält man eine Art „bedingte Distributivität" der Vereinfachungsoperatoren über die anderen booleschen Operatoren, welche später bei der Optimierung der inkrementellen Generierung der Übergangsrelation benützt wird.

Korollar 3.36 (Bedingte Distributivität von „$\downarrow$" und „$\Downarrow$") *Für $f,g,h \in \mathsf{ROBDD}$ gilt*

$$
\begin{aligned}
(\neg_{\mathsf{BDD}} f)\downarrow g \wedge_{\mathsf{BDD}} g &= \neg_{\mathsf{BDD}}(f\downarrow g) \wedge_{\mathsf{BDD}} g \\
(\neg_{\mathsf{BDD}} f)\Downarrow g \wedge_{\mathsf{BDD}} g &= \neg_{\mathsf{BDD}}(f\Downarrow g) \wedge_{\mathsf{BDD}} g \\
(f \wedge_{\mathsf{BDD}} g)\downarrow h \wedge_{\mathsf{BDD}} h &= (f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h) \wedge_{\mathsf{BDD}} h \\
(f \wedge_{\mathsf{BDD}} g)\Downarrow h \wedge_{\mathsf{BDD}} h &= (f\Downarrow h \wedge_{\mathsf{BDD}} g\Downarrow h) \wedge_{\mathsf{BDD}} h \\
(f \vee_{\mathsf{BDD}} g)\downarrow h \wedge_{\mathsf{BDD}} h &= (f\downarrow h \vee_{\mathsf{BDD}} g\downarrow h) \wedge_{\mathsf{BDD}} h \\
(f \vee_{\mathsf{BDD}} g)\Downarrow h \wedge_{\mathsf{BDD}} h &= (f\Downarrow h \vee_{\mathsf{BDD}} g\Downarrow h) \wedge_{\mathsf{BDD}} h
\end{aligned}
$$

Hierbei binde „$\downarrow$" bzw. „$\Downarrow$" am stärksten.

Beweis: Sei $\otimes \in \{\downarrow,\Downarrow\}$ und $A \in \mathcal{A}$ eine Allokation mit $\mathsf{range}(A) \supseteq \mathsf{var}(f,g,h)$. Dann folgt mit Satz 3.35 unter Verwendung von „$a \vee \overline{b} \equiv a\,b \vee \overline{b}$" für $a,b \in \mathbb{B}^W$.

$$
\begin{aligned}
\langle\!\langle \neg_{\mathsf{BDD}}(f\otimes g) \wedge_{\mathsf{BDD}} g\rangle\!\rangle_A &\equiv \overline{\langle\!\langle f\otimes g\rangle\!\rangle_A} \wedge \langle\!\langle g\rangle\!\rangle_A &&\equiv \overline{\langle\!\langle f\otimes g\rangle\!\rangle_A \vee \overline{\langle\!\langle g\rangle\!\rangle_A}} \\
&\equiv \overline{\langle\!\langle f\otimes g\rangle\!\rangle_A\, \langle\!\langle g\rangle\!\rangle_A \vee \overline{\langle\!\langle g\rangle\!\rangle_A}} &&\equiv \overline{\langle\!\langle f\rangle\!\rangle_A\, \langle\!\langle g\rangle\!\rangle_A \vee \overline{\langle\!\langle g\rangle\!\rangle_A}} \\
&\equiv \overline{\langle\!\langle f\rangle\!\rangle_A \vee \overline{\langle\!\langle g\rangle\!\rangle_A}} &&\equiv \overline{\langle\!\langle f\rangle\!\rangle_A} \wedge \langle\!\langle g\rangle\!\rangle_A \\
&\equiv \langle\!\langle \neg_{\mathsf{BDD}} f\rangle\!\rangle_A \wedge \langle\!\langle g\rangle\!\rangle_A &&\equiv \langle\!\langle (\neg_{\mathsf{BDD}} f)\otimes g\rangle\!\rangle_A \wedge \langle\!\langle g\rangle\!\rangle_A \\
&\equiv \langle\!\langle (\neg_{\mathsf{BDD}} f)\otimes g \wedge_{\mathsf{BDD}} g\rangle\!\rangle_A,
\end{aligned}
$$

des weiteren für „$\wedge$"

$$
\begin{aligned}
\langle\!\langle (f \wedge_{\mathsf{BDD}} g)\otimes h \wedge_{\mathsf{BDD}} h\rangle\!\rangle_A &\equiv \langle\!\langle (f \vee_{\mathsf{BDD}} g)\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A \equiv \langle\!\langle f \wedge_{\mathsf{BDD}} g\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A \\
&\equiv \langle\!\langle f\rangle\!\rangle_A \wedge \langle\!\langle g\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A \equiv (\langle\!\langle f\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \wedge (\langle\!\langle g\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \\
&\equiv (\langle\!\langle f\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \wedge (\langle\!\langle g\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \\
&\equiv (\langle\!\langle f\otimes h\rangle\!\rangle_A \wedge \langle\!\langle g\otimes h\rangle\!\rangle_A) \wedge \langle\!\langle h\rangle\!\rangle_A
\end{aligned}
$$

und schließlich für „$\vee$"

$$
\begin{aligned}
\langle\!\langle (f \wedge_{\mathsf{BDD}} g)\otimes h \wedge_{\mathsf{BDD}} h\rangle\!\rangle_A &\equiv \langle\!\langle (f \vee_{\mathsf{BDD}} g)\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A \equiv \langle\!\langle f \vee_{\mathsf{BDD}} g\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A \\
&\equiv (\langle\!\langle f\rangle\!\rangle_A \vee \langle\!\langle g\rangle\!\rangle_A) \wedge \langle\!\langle h\rangle\!\rangle_A \equiv (\langle\!\langle f\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \vee (\langle\!\langle g\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \\
&\equiv (\langle\!\langle f\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \vee (\langle\!\langle g\otimes h\rangle\!\rangle_A \wedge \langle\!\langle h\rangle\!\rangle_A) \\
&\equiv (\langle\!\langle f\otimes h\rangle\!\rangle_A \vee \langle\!\langle g\otimes h\rangle\!\rangle_A) \wedge \langle\!\langle h\rangle\!\rangle_A
\end{aligned}
$$

❑

Dieses Korollar läßt sich für den „$\downarrow$"-Operator noch verstärken zu

Satz 3.37 (Distributivität von „$\downarrow$“) *Für $f, g, h \in \mathsf{ROBDD}$ gilt*

$$\begin{aligned}
(\neg_{\mathsf{BDD}} f)\downarrow g &= \neg_{\mathsf{BDD}}(f\downarrow g) && \textit{(hier } g \neq 0\textit{)}\\
(f \wedge_{\mathsf{BDD}} g)\downarrow h &= f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h\\
(f \vee_{\mathsf{BDD}} g)\downarrow h &= f\downarrow h \vee_{\mathsf{BDD}} g\downarrow h
\end{aligned}$$

Beweis: Wähle eine passende Allokation $A \in \mathcal{A}$ mit $\mathsf{range}(A) \supseteq \mathsf{var}(f, g, h)$. Zunächst betrachte die Negation. Für $f \in \{0, 1\}$ oder $g = 1$ ist

$$(\neg_{\mathsf{BDD}} f)\downarrow g = \neg_{\mathsf{BDD}} f = \neg_{\mathsf{BDD}}(f\downarrow g)$$

Seien also g und f keine Konstanten und die Bezeichner wie im Algorithmus und zusätzlich $x := A^{-1}(m)$. Für $g_0 = 0$ ist

$$\neg_{\mathsf{BDD}}(f\downarrow g) = \neg_{\mathsf{BDD}}(f_1\downarrow g_1) = (\neg_{\mathsf{BDD}} f_1)\downarrow g_1,$$

nach Induktionsvoraussetzung. Mit „$\neg\mathrm{ite}(a, b, c) \equiv \mathrm{ite}(a, \neg b, \neg c)$“ erhält man

$$\langle\!\langle \neg_{\mathsf{BDD}} f\rangle\!\rangle_A \equiv \mathrm{ite}(x, \langle\!\langle \neg_{\mathsf{BDD}} f_1\rangle\!\rangle_A, \langle\!\langle \neg_{\mathsf{BDD}} f_0\rangle\!\rangle_A), \tag{3.14}$$

so daß

$$m = \mathsf{min}\{\mathsf{top}(\neg_{\mathsf{BDD}} f), \mathsf{top}(g)\} \quad \text{und} \quad \mathsf{cf}(\neg_{\mathsf{BDD}} f, m) = (\neg_{\mathsf{BDD}} f_0, \neg_{\mathsf{BDD}} f_1) \tag{3.15}$$

Damit ist $(\neg_{\mathsf{BDD}} f)\downarrow g = (\neg_{\mathsf{BDD}} f_1)\downarrow g_1$. Der Fall $g_1 = 0$ folgt analog. Sei also $g_0, g_1 \neq 0$.

$$\begin{aligned}
\langle\!\langle \neg_{\mathsf{BDD}}(f\downarrow g)\rangle\!\rangle_A &\equiv \neg\mathrm{ite}(x, \langle\!\langle f_1\downarrow g_1\rangle\!\rangle_A, \langle\!\langle f_0\downarrow g_0\rangle\!\rangle_A) \equiv \mathrm{ite}(x, \neg\langle\!\langle f_1\downarrow g_1\rangle\!\rangle_A, \neg\langle\!\langle f_0\downarrow g_0\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle \neg_{\mathsf{BDD}}(f_1\downarrow g_1)\rangle\!\rangle_A, \langle\!\langle \neg_{\mathsf{BDD}}(f_0\downarrow g_0)\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle(\neg_{\mathsf{BDD}} f_1)\downarrow g_1)\rangle\!\rangle_A, \langle\!\langle(\neg_{\mathsf{BDD}} f_0)\downarrow g_0)\rangle\!\rangle_A)\\
&\equiv \langle\!\langle(\neg_{\mathsf{BDD}} f)\downarrow g\rangle\!\rangle_A
\end{aligned}$$

wiederum mit den Gleichungen (3.14) und (3.15). Nun betrachte „$\wedge$“. Ist $h = 0$, $g = 0$ oder $f = 0$, so sind beide Seiten 0. Für $h = 1$ sind beide Seiten gleich $f \wedge_{\mathsf{BDD}} g$. Im Falle $g = 1$ sind sie gleich $f\downarrow h$ und für $f = 1$ gleich $g\downarrow h$. Seien also f, g und h keine Konstanten. Definiere

$$m := \{\mathsf{top}(f), \mathsf{top}(g), \mathsf{top}(h)\}, \qquad x := A^{-1}(m)$$

$$(f_0, f_1) := \mathsf{cf}(f, m), \qquad (g_0, g_1) := \mathsf{cf}(g, m), \qquad (h_0, h_1) := \mathsf{cf}(h, m)$$

Fall a: $m < \mathsf{top}(h)$. Hier muß nocheinmal unterschieden werden, ob m in $f \wedge_{\mathsf{BDD}} g$ überhaupt noch vorkommt. Betrachte also den Fall a1: $m < \mathsf{top}(f \wedge_{\mathsf{BDD}} g)$, bei dem m zwar in f oder g aber nicht mehr in $f \wedge_{\mathsf{BDD}} g$ vorkommt. Nach der Semantik von „$\wedge_{\mathsf{BDD}}$“ gilt immer

$$\langle\!\langle f \wedge_{\mathsf{BDD}} g\rangle\!\rangle_A \equiv \mathrm{ite}(x, \langle\!\langle f_1 \wedge_{\mathsf{BDD}} g_1\rangle\!\rangle_A, \langle\!\langle f_0 \wedge_{\mathsf{BDD}} g_0\rangle\!\rangle_A)$$

Da aber $x \notin A^{-1}(\mathsf{var}(f \wedge_{\mathsf{BDD}} g))$, gilt

$$f_1 \wedge_{\mathsf{BDD}} g_1 = f_0 \wedge_{\mathsf{BDD}} g_0 = f \wedge_{\mathsf{BDD}} g \tag{3.16}$$

Weiter hat man (egal ob $\mathsf{top}(g) > m$ oder $\mathsf{top}(f) > m$)

$$\begin{aligned}
\langle\!\langle f\downarrow h \ \wedge_{\mathsf{BDD}}\ g\downarrow h\rangle\!\rangle_A &\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h\rangle\!\rangle_A) \ \wedge\ \mathrm{ite}(x, \langle\!\langle g_1\downarrow h\rangle\!\rangle_A, \langle\!\langle g_0\downarrow h\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h \wedge_{\mathsf{BDD}} g_1\downarrow h\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h \wedge_{\mathsf{BDD}} g_0\downarrow h\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h\rangle\!\rangle_A, \langle\!\langle (f_0 \wedge_{\mathsf{BDD}} g_0)\downarrow h\rangle\!\rangle_A)\\
&\equiv \langle\!\langle (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h\rangle\!\rangle_A,\\
&= \langle\!\langle (f \wedge_{\mathsf{BDD}} g)\downarrow h\rangle\!\rangle_A,
\end{aligned}$$

nach zweimaliger Anwendung der Induktionsvoraussetzung. Für a1: $m = \mathsf{top}(f \wedge_{\mathsf{BDD}} g)$ gilt

$$\mathsf{cf}(f \wedge_{\mathsf{BDD}} g, m) = (f_0 \wedge_{\mathsf{BDD}} g_0, f_1 \wedge_{\mathsf{BDD}} g_1) \tag{3.17}$$

Nun ergibt die Anwendung des Algorithmus

$$\begin{aligned}
\langle\!\langle (f \wedge_{\mathsf{BDD}} g)\downarrow h\rangle\!\rangle_A &\equiv \mathrm{ite}(x, \langle\!\langle (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h\rangle\!\rangle_A, \langle\!\langle (f_0 \wedge_{\mathsf{BDD}} g_0)\downarrow h\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h \wedge_{\mathsf{BDD}} g_1\downarrow h\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h \wedge_{\mathsf{BDD}} g_0\downarrow h\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h\rangle\!\rangle_A) \wedge \mathrm{ite}(x, \langle\!\langle g_1\downarrow h\rangle\!\rangle_A, \langle\!\langle g_0\downarrow h\rangle\!\rangle_A)\\
&\equiv \langle\!\langle f\downarrow h \ \wedge\ g\downarrow h\rangle\!\rangle_A
\end{aligned}$$

Der letzte Schluß gilt auch, wenn $\mathsf{top}(f) \neq m$ oder $\mathsf{top}(g) \neq m$. Nun kommt der komplexere Fall b: $m = \mathsf{top}(h)$, bei dem noch unterschieden werden muß, ob $h_0 = 0$ oder $h_1 = 0$.[15] Betrachte nun zuerst den Fall b1: $m < \mathsf{top}(f \wedge_{\mathsf{BDD}} g)$. Weiterhin sei hier zunächst $h_0 = 0$. Dann hat man

$$f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h = f_1\downarrow h_1 \wedge_{\mathsf{BDD}} g_1\downarrow h_1 = (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h_1 = (f \wedge_{\mathsf{BDD}} g)\downarrow h_1 = (f \wedge_{\mathsf{BDD}} g)\downarrow h$$

nach Anwendung der Induktionsvoraussetzung und Beachtung von Gleichung (3.16). Genauso verfahre man für $h_1 = 0$. Gelte also für den Rest von Fall b1 sowohl $h_1 \neq 0$ als auch $h_0 \neq 0$. Nach dem Algorithmus gilt

$$\begin{aligned}
\langle\!\langle f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h\rangle\!\rangle_A &\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h_0\rangle\!\rangle_A) \wedge \mathrm{ite}(x, \langle\!\langle g_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle g_0\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h_1 \wedge_{\mathsf{BDD}} g_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h_0 \wedge_{\mathsf{BDD}} g_0\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h_1\rangle\!\rangle_A, \langle\!\langle (f_0 \wedge_{\mathsf{BDD}} g_0)\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \langle\!\langle (f \wedge_{\mathsf{BDD}} g)\downarrow h\rangle\!\rangle_A
\end{aligned}$$

wieder mit denselben Argumenten wie im Fall b1 mit $h_0 = 0$. Im letzten Fall b2 gelte $m = \mathsf{top}(f \wedge_{\mathsf{BDD}} g)$. Für $h_0 = 0$ erhält man nun

$$f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h = f_1\downarrow h_1 \wedge_{\mathsf{BDD}} g_1\downarrow h_1 = (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h_1 = (f \wedge_{\mathsf{BDD}} g)\downarrow h$$

mit Beachtung von Gleichung (3.17). Analog argumantiert man für $h_1 = 0$. Gelte also $h_1, h_0 \neq 0$. Der Algorithmus ergibt dann mit Gleichung (3.17)

$$\begin{aligned}
\langle\!\langle (f \wedge_{\mathsf{BDD}} g)\downarrow h\rangle\!\rangle_A &\equiv \mathrm{ite}(x, \langle\!\langle (f_1 \wedge_{\mathsf{BDD}} g_1)\downarrow h_1\rangle\!\rangle_A, \langle\!\langle (f_0 \wedge_{\mathsf{BDD}} g_0)\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h_1 \wedge_{\mathsf{BDD}} g_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h_0 \wedge_{\mathsf{BDD}} g_0\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \mathrm{ite}(x, \langle\!\langle f_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle f_0\downarrow h_0\rangle\!\rangle_A) \wedge_{\mathsf{BDD}} \mathrm{ite}(x, \langle\!\langle g_1\downarrow h_1\rangle\!\rangle_A, \langle\!\langle g_0\downarrow h_0\rangle\!\rangle_A)\\
&\equiv \langle\!\langle f\downarrow h \wedge_{\mathsf{BDD}} g\downarrow h\rangle\!\rangle_A
\end{aligned}$$

[15] Dies war im Fall a nicht nötig, da dort $\mathsf{cf}(h, m) = (h, h)$ gegolten hat.

Nun zum „$\vee_{\mathsf{BDD}}$“. Im Falle $h = 0$ erhält man auf beiden Seiten 0. Sei also $h \neq 0$, so daß sich die Aussagen über „$\neg_{\mathsf{BDD}}$“ und „$\wedge_{\mathsf{BDD}}$“ verwenden lassen.

$$\begin{aligned}(f \vee_{\mathsf{BDD}} g)\downarrow h &= (\neg_{\mathsf{BDD}}(\neg_{\mathsf{BDD}} f \wedge_{\mathsf{BDD}} \neg_{\mathsf{BDD}} g))\downarrow h\\ &= \neg_{\mathsf{BDD}}((\neg_{\mathsf{BDD}} f \wedge_{\mathsf{BDD}} \neg_{\mathsf{BDD}} g)\downarrow h)\\ &= \neg_{\mathsf{BDD}}((\neg_{\mathsf{BDD}} f)\downarrow h \wedge_{\mathsf{BDD}} (\neg_{\mathsf{BDD}} g)\downarrow h)\\ &= \neg_{\mathsf{BDD}}(\neg_{\mathsf{BDD}}(f\downarrow h) \wedge_{\mathsf{BDD}} \neg_{\mathsf{BDD}}(g\downarrow h))\\ &= f\downarrow h \vee_{\mathsf{BDD}} g\downarrow h\end{aligned}$$

nach mehrmaliger Anwendung des DeMorganschen Gesetzes „$\overline{\overline{a} \wedge \overline{b}} \equiv a \vee b$“. ❑

Wegen der funktionalen Vollständigkeit von „$\neg$“ und „$\wedge$“ in $\mathbb{B}^W$ und „$\wedge_{\mathsf{BDD}}$“ in ROBDD) ist dies äquivalent zu (vgl. auch mit Theorem 3 aus /Coudert und Madre, 1990/)

Korollar 3.38 (Proposition 2.1 aus /Touati et al., 1990/)

$$\mathsf{compose}(g, \{i \mapsto f_i\})\downarrow h \;=\; \mathsf{compose}(g, \{i \mapsto f_i\downarrow h\}) \qquad \textit{(für } h \neq 0\textit{)}$$

für $g, h, f_i \in \mathsf{ROBDD}$*,* $i \in \mathbb{N}$.

Damit „distributiert“ der „$\downarrow$“-Operator über alle booleschen Operationen, wenn das zweite Argument ungleich 0 ist. Diese Forderung ist im allg. auch notwendig wie

$$(\neg_{\mathsf{BDD}} 0)\downarrow 0 = 0 \neq 1 = \neg_{\mathsf{BDD}} 0a = \neg_{\mathsf{BDD}}(0\downarrow 0)$$

zeigt. Ein solches Gegenbeispiel läßt sich auf ähnliche Weise auch für die Implikation und andere „nicht monotone“ Operatoren angeben. Ebenso läßt sich Satz 3.37 nicht auf „$\Downarrow$“ übertragen, wie folgendes Beispiel zeigt. Dabei sei $A \in \mathcal{A}$ mit $A(x) := 0$, $A(y) := 1$.

$$\begin{aligned}\|x \otimes y\|_A \Downarrow \|x \leftrightarrow y\|_A &= \begin{cases} \mathsf{ite}_{\mathsf{BDD}}(\|x\|_A, \|y\|_A \Downarrow \|y\|_A, 0) & \text{falls „}\otimes\text{“} = \text{„}\wedge\text{“}\\ \mathsf{ite}_{\mathsf{BDD}}(\|x\|_A, 1, \|y\|_A \Downarrow \|\overline{y}\|_A) & \text{falls „}\otimes\text{“} = \text{„}\vee\text{“}\end{cases}\\ &= \|x\|_A\\ &\neq \|x\|_A \otimes_{\mathsf{BDD}} \|y\|_A\\ &= \|x\|_A \Downarrow \|x \leftrightarrow y\|_A \otimes_{\mathsf{BDD}} \underbrace{\|y\|_A \Downarrow \|x \leftrightarrow y\|_A}_{=\|y\|_A \Downarrow \|y \vee \overline{y}\|_A = \|y\|_A \Downarrow 1}\end{aligned}$$

In /Touati et al., 1990/ wurde etwas weniger gezeigt, da der Algorithmus dort die Einschränkung $h \neq 0$ macht. Für Assoziativitätseigenschaften von „$\downarrow$“ siehe /McMillan, 1996/.

Für die Arbeit /Touati et al., 1990/ war das folgende Korollar zu Satz 3.41 (dort nur für „$\downarrow$“ gegeben) noch wichtiger. Es erlaubt, die Berechnung des Relationenproduktes auf eine Hintereinanderausführung des „$\downarrow$“-Operators und einer normalen Quantifikation zurückzuführen.

Korollar 3.39 (Proposition 2.2 aus /Touati et al., 1990/) *Für* $f, g \in \mathsf{ROBDD}$ *gilt*

$$\mathsf{exists}(f \wedge_{\mathsf{BDD}} g, E) \;=\; \mathsf{exists}(f\downarrow g, E),$$

wobei $\mathsf{var}(g) \subseteq E \subseteq \mathbb{N}$.

Beweis: Für eine passende Allokation $A \in \mathcal{A}$, mit $\mathsf{range}(A) \supseteq \mathsf{var}(f,g) \cup E$, muß für $W_E := \{x_1, \dots, x_n\} := A^{-1}(E)$ folgendes gezeigt werden

$$\exists\, W_E.\, \langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \;\equiv\; \exists\, W_E.\, \langle\!\langle f \downarrow g \rangle\!\rangle_A$$

Für die Richtung von links nach rechts reicht Satz 3.35 und die Tautologie $a \wedge b \to a$ für $a, b \in \mathbb{B}^W$, die auch noch „unter Quantifikation“ gilt. Für die Rückrichtung sei ρ_0 eine Variablenbelegung für $\mathbb{B}^W$ mit

$$[\![\exists\, W_E.\, \langle\!\langle f \downarrow g \rangle\!\rangle_A]\!]\rho_0 \;=\; 1$$

Die Definition der Semantik von $[\![\cdot]\!]$ ergibt die Existenz eines $\underline{a} \in \mathbb{B}^n$, so daß

$$[\![\langle\!\langle f \downarrow g \rangle\!\rangle_A]\!]\rho = 1, \qquad \text{mit } \rho := \rho_0\{\underline{x} \mapsto \underline{a}\}$$

Damit erhält man aus Satz 3.41 eine Belegung ρ' mit

$$\begin{aligned} 1 = [\![\langle\!\langle f \rangle\!\rangle \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho' &= [\![\langle\!\langle f \rangle\!\rangle \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho_0\{\underline{x} \mapsto \underline{a}\}\{\underline{x} \mapsto \rho'(\underline{x})\} \\ &= [\![\langle\!\langle f \rangle\!\rangle \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho_0\{\underline{x} \mapsto \rho'(\underline{x})\} \\ &= [\![\exists\, W_E.\, \langle\!\langle f \rangle\!\rangle \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho_0, \end{aligned}$$

wegen $\rho'(y) = \rho(y)$ für alle $y \in W \backslash A^{-1}(\mathsf{var}(g)) \supseteq W \backslash W_E$. ❑

Durch die Gültigkeit der Sätze 3.35 und 3.41 auch für „$\Downarrow$“ kann mit demselben Beweis eine Erweiterung dieser Resultate aus /Touati et al., 1990, Coudert und Madre, 1990/ gezeigt werden.

Korollar 3.40 *Für $f, g \in \mathsf{ROBDD}$ gilt*

$$\mathsf{exists}(f \wedge_{\mathsf{BDD}} g, E) \;=\; \mathsf{exists}(f \downarrow g, E), \;=\; \mathsf{exists}(f \Downarrow g, E),$$

wobei $\mathsf{var}(g) \subseteq E \subseteq \mathbb{N}$.

Satz 3.41 *Sei $f, g \in \mathsf{ROBDD}$, $A \in \mathcal{A}$, $\mathsf{var}(f) \subseteq \mathsf{range}(A)$, ρ eine Belegung für $\mathbb{B}^W$ und $\otimes \in \{\downarrow, \Downarrow\}$. Falls $[\![\langle\!\langle f \otimes g \rangle\!\rangle_A]\!]\rho = 1$, dann gibt es eine Variablenbelegung ρ', mit*

$$[\![\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho' = 1,$$

wobei $\rho(x) = \rho'(x)$ für alle $x \in W \backslash A^{-1}(\mathsf{var}(g))$.

Beweis: Der Beweis wird geführt durch Induktion über den Termaufbau von f und g. Im Falle $g = 0$ oder $f = 0$ ist die Prämisse nicht erfüllt. Nun ist $g \neq 0$ und es gibt eine erfüllende Belegung ρ' für $\langle\!\langle g \rangle\!\rangle$. Damit erhält man sowohl für $f = 1$ als auch für $g = 1$ (o.B.d.A. gelte $\rho(x) = \rho'(x)$ für alle $x \in W \backslash A^{-1}(\mathsf{var}(g))$)

$$[\![\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A]\!]\rho' = [\![\langle\!\langle f \rangle\!\rangle_A]\!]\rho' \wedge [\![\langle\!\langle g \rangle\!\rangle_A]\!]\rho' = [\![\langle\!\langle f \rangle\!\rangle_A]\!]\rho \wedge 1 = [\![\langle\!\langle f \otimes g \rangle\!\rangle_A]\!]\rho = 1$$

Im folgenden seien f und g keine Konstanten und die Bezeichner wie in den Algorithmen gewählt. Zusätzlich definiere $x := A^{-1}(m)$. Für $g_0 = 0$ ist

$$1 = [\![\langle\!\langle f \otimes g \rangle\!\rangle_A]\!]\rho = [\![\langle\!\langle f_1 \otimes g_1 \rangle\!\rangle_A]\!]\rho$$

Die Induktionsvoraussetzung ergibt dann ein ρ'', mit

$$1 = [\![\langle\!\langle f_1 \rangle\!\rangle_A \wedge \langle\!\langle g_1 \rangle\!\rangle_A]\!]\rho'',$$

das außerhalb der Variablen von g_1 und somit auch außerhalb der Variablen von g mit ρ übereinstimmt. Weiter erhält man aus der Semantik von „$\wedge_{\mathsf{BDD}}$“

$$\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g \rangle\!\rangle_A \equiv \text{ite}(x, \langle\!\langle f_1 \rangle\!\rangle_A \wedge \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle f_0 \rangle\!\rangle_A \wedge 0) \equiv x \wedge \langle\!\langle f_1 \rangle\!\rangle_A \wedge \langle\!\langle g_1 \rangle\!\rangle_A \tag{3.18}$$

Da zwiegenderweise $A(x) \in \mathsf{var}(g)$, kann man ρ definieren als

$$\rho'(y) := \begin{cases} \rho''(y) & \text{falls } y \neq x \\ 1 & \text{falls } y = x \end{cases}$$

Mit Gleichung (3.18) folgt nun die Behauptung. Der Fall $g_1 = 0$ folgt analog unter Verwendung von

$$\rho'(y) := \begin{cases} \rho''(y) & \text{falls } y \neq x \\ 0 & \text{falls } y = x \end{cases}$$

Für den Fall $g_0, g_1 \neq 0$ (und $\otimes \neq \Downarrow$ oder $\mathsf{top}(f) = m$) gilt

$$[\![\langle\!\langle f \otimes g \rangle\!\rangle_A]\!]\rho \; = \; [\![x \,\langle\!\langle f_1 \otimes g_1 \rangle\!\rangle_A]\!]\rho \;\vee\; [\![\overline{x}\,\langle\!\langle f_0 \otimes g_0 \rangle\!\rangle_A]\!]\rho$$

Ist der linke Term der rechten Seite gleich 1, so verfahre man wie bei $g_0 = 0$. Ansonsten muß der rechte gleich 1 sein, und man kann wie bei $g_1 = 0$ vorgehen. Für $\otimes = \Downarrow$ und $m < \mathsf{top}(f)$ gilt weiterhin

$$[\![\langle\!\langle f \Downarrow g \rangle\!\rangle_A]\!]\rho \; = \; [\![\langle\!\langle f \Downarrow (g_0 \vee_{\mathsf{BDD}} g_1) \rangle\!\rangle_A]\!]\rho \; = \; 1$$

Damit erhält man aus der Induktionsbehauptung ein ρ'', mit

$$1 = [\![\langle\!\langle f \rangle\!\rangle_A \;\wedge\; \langle\!\langle g_0 \vee_{\mathsf{BDD}} g_1 \rangle\!\rangle_A]\!]\rho'' = [\![\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g_0 \rangle\!\rangle_A]\!]\rho'' \;\vee\; [\![\langle\!\langle f \rangle\!\rangle_A \wedge \langle\!\langle g_1 \rangle\!\rangle_A]\!]\rho''$$

Je nachdem, ob der linke oder rechte Term gleich 1 ist, definiere ρ' wie zuvor. ❑

3.8.4 Der $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus

In diesem Abschnitt werden die oben vorgestellten Algorithmen verallgemeinert. Dies führt zum $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus, der noch weitere aus der Literatur bekannte Algorithmen subsumiert. Zunächst wird ein weiterer Algorithmus $\mathsf{ite}_\exists$ angegeben, der eine Verallgemeinerung des Algorithmus $\mathsf{relProd}$ aus Abb. 3.25 und somit auch der Algorithmen aus Abb. 3.21 darstellt (vgl. Gleichung (3.13)). Seine Semantik ergibt sich aus der Kombination des Existenzquantors mit $\mathsf{ite}_{\mathsf{BDD}}$:

$$\mathsf{ite}_\exists(f, g, h, E) = \mathsf{exists}(\mathsf{ite}_{\mathsf{BDD}}(f, g, h), E)$$

Der Algorithmus ist in Abb. 3.37 dargestellt und ist eine einfache Erweiterung des Algorithmus aus Abb. 3.25. Entsprechend Tabelle 3.1 können nun weitere Algorithmen für z. B. das „$\mathsf{forallImplies}$“) für den Term aus Gleichung (3.13) auf Seite 84 abgeleitet werden.

Als Zwischenschritt zum Ziel $_{\mathsf{c}}\mathsf{ite}_\exists$ sei der Algorithmus $\mathsf{compose}_\exists$ vorgestellt, den man in Abb. 3.38 zu findet. Seine Semantik läßt sich aus folgender Gleichung ablesen

$$\mathsf{compose}_\exists(f, \rho, E) \; = \; \mathsf{exists}(\mathsf{compose}(f, \rho), E)$$

Sein Aufgabe ist es zusätzlich den Spezialfall von $_{\mathsf{c}}\mathsf{ite}_\exists$ abzufangen, wenn zwei der ersten beiden Argumente von $_{\mathsf{c}}\mathsf{ite}_\exists$ Konstanten sind. Der $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus aus Abb. 3.39 fügt dem noch eine Operation hinzu und es gilt

$\boxed{\mathsf{ite}_\exists(f,g,h,E)}$ ::=

if $f = 0$ **then** $\mathsf{exists}(h,E)$
else **if** $f = 1$ **then** $\mathsf{exists}(g,E)$
else **if** $m \in E$ **then**
  $l \vee_{\mathsf{BDD}} r$
else
  $l \cdots (m) - r$
where
  $m = \mathsf{min}\{\mathsf{top}(f), \mathsf{top}(g), \mathsf{top}(h)\}$
  $l = \boxed{\mathsf{ite}_\exists(f_0, g_0, h_0, E)}$
  $r = \boxed{\mathsf{ite}_\exists(f_1, g_1, h_1, E)}$
  where
    $(f_0, f_1) = \mathsf{cf}(f,m)$
    $(g_0, g_1) = \mathsf{cf}(g,m)$
    $(h_0, h_1) = \mathsf{cf}(h,m)$

Abbildung 3.37: $\mathsf{ite}_\exists : \mathsf{ROBDD}^3 \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$

$\boxed{\mathsf{compose}_\exists(f,\rho,E)}$ ::=

if $f \in \{0,1\}$ **then** f
else
  $\mathsf{ite}_\exists(\rho(m), r, l, E'')$
  where
    $E'' = \mathsf{var}(\rho(m)) \cap E$
    $E' = E \setminus E''$
    $l = \boxed{\mathsf{compose}_\exists(f_0, \rho, E')}$
    $r = \boxed{\mathsf{compose}_\exists(f_1, \rho, E')}$
    where
      $m = \mathsf{top}(f)$
      $(f_0, f_1) = \mathsf{cf}(f,m)$

Abbildung 3.38: $\mathsf{compose}_\exists : \mathsf{ROBDD} \times [\mathbb{N} \to \mathsf{ROBDD}] \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$

$$\boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}(f,g,h,\rho,E)} ::=$$

$$
\begin{array}{l}
\textbf{if } f=0 \textbf{ then } \mathsf{exists}(h,E)\\
\textbf{else if } f=1 \textbf{ then } \mathsf{exists}(g,E)\\
\textbf{else}\\
\quad \textbf{if } m=\mathsf{top}(f) \textbf{ then}\\
\quad\quad \mathsf{ite}_{\exists}(\rho(m),r,l,E'')\\
\quad\quad \textbf{where}\\
\quad\quad\quad \begin{array}{lcl}
E'' & = & \mathsf{var}(\rho(m))\cap E\\
E' & = & E\backslash E''\\
l & = & \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}(f_0,g,h,\rho,E')}\\
r & = & \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}(f_1,g,h,\rho,E')}
\end{array}\\
\quad \textbf{else}\\
\quad\quad \mathsf{ite}_{\exists}(0 \cdots (m) - 1,r,l,E'')\\
\quad\quad \textbf{where}\\
\quad\quad\quad \begin{array}{lcl}
E'' & = & \{m\}\cap E\\
E' & = & E\backslash E''\\
l & = & \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}(f,g_0,h_0,\rho,E')}\\
r & = & \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}(f,g_1,h_1,\rho,E')}
\end{array}\\
\quad \textbf{where}\\
\quad\quad \begin{array}{lcl}
m & = & \mathsf{min}\{\mathsf{top}(f),\mathsf{top}(g),\mathsf{top}(h)\}\\
(f_0,f_1) & = & \mathsf{cf}(f,m)\\
(g_0,g_1) & = & \mathsf{cf}(g,m)\\
(h_0,h_1) & = & \mathsf{cf}(h,m)
\end{array}
\end{array}
$$

Abbildung 3.39: ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$: $\mathsf{ROBDD}^3 \times [\mathbb{N} \to \mathsf{ROBDD}] \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$

Satz 3.42 (Semantik von $_{\mathsf{c}}\mathsf{ite}_\exists$) *Für $f,g,h \in \mathsf{ROBDD}$, $\rho\colon \mathbb{N} \to \mathsf{ROBDD}$, $E \subseteq \mathbb{N}$ gilt*

$$_{\mathsf{c}}\mathsf{ite}_\exists(f,g,h,\rho,E) = \mathsf{exists}(\mathsf{ite}(\mathsf{compose}(f,\rho),g,h),E)$$

Beweis: Für den Beweis verwende man den Satz 3.29 und die Semantik von exists und übertrage die Aussage nach $\mathbb{B}^W$

$$\langle\!\langle {_{\mathsf{c}}\mathsf{ite}_\exists}(f,g,h,\rho,E)\rangle\!\rangle_A \;\equiv\; \exists E_W \,.\, \mathrm{ite}(\langle\!\langle f\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A),$$

wobei $E_W := A^{-1}(E)$, $A \in \mathcal{A}$, γ assoziiert zu ρ und

$$\mathsf{range}(A) \;\supseteq\; E \cup \mathsf{var}(f,g,h) \cup \bigcup_{i\in\mathbb{N}} \mathsf{var}(\rho(i))$$

Der Basisfall $f \in \{0,1\}$ ist trivial, wenn man die Korrektheit von exists annimmt. Für $m = \mathsf{top}(f)$ und $x := A^{-1}(m)$ gilt nach der Hilfsbehauptung aus Gleichung (3.6) auf Seite 69

$$\exists E_W \,.\, \mathrm{ite}(\langle\!\langle f\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A) \equiv \exists E_W \,.\, \mathrm{ite}(\mathrm{ite}(x, \langle\!\langle f_1\rangle\!\rangle_A, \langle\!\langle f_0\rangle\!\rangle_A)\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A) \equiv \ldots$$

Durch Anwendung von Lemma 2.13 erhält man

$$\begin{aligned}
\ldots &\equiv \exists E_W \,.\, \mathrm{ite}(x, \langle\!\langle f_1\rangle\!\rangle_A, \langle\!\langle f_0\rangle\!\rangle_A)\gamma\, g \;\vee\; \mathrm{ite}(x, \overline{\langle\!\langle f_1\rangle\!\rangle_A}, \overline{\langle\!\langle f_0\rangle\!\rangle_A})\gamma\, h \\
&\equiv \exists E_W \,.\, (x\, \langle\!\langle f_1\rangle\!\rangle_A)\gamma\, g \;\vee\; (\overline{x}\, \langle\!\langle f_0\rangle\!\rangle_A)\gamma\, g \;\vee\; (x\, \overline{\langle\!\langle f_1\rangle\!\rangle_A})\gamma\, h \;\vee\; (\overline{x}\, \overline{\langle\!\langle f_0\rangle\!\rangle_A})\gamma\, h \\
&\equiv \exists E_W \,.\, \gamma(x)\; (\langle\!\langle f_1\rangle\!\rangle_A\gamma\, g \;\vee\; \overline{\langle\!\langle f_1\rangle\!\rangle_A\gamma}\, h) \;\vee\; \overline{\gamma(x)}(\langle\!\langle f_0\rangle\!\rangle_A\gamma\, g \;\vee\; \overline{\langle\!\langle f_0\rangle\!\rangle_A\gamma}\, h) \\
&\equiv \exists E_W \,.\, \mathrm{ite}(\gamma(x), \mathrm{ite}(\langle\!\langle f_1\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f_0\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A)) \\
&\equiv \ldots
\end{aligned}$$

Mit den Bezeichnern aus dem Algorithmus sei $E'_W := A^{-1}(E')$ und $E''_W := A^{-1}(E'')$. Nun gilt $\mathsf{free}(\gamma(x)) \cap E'_W = \emptyset$, so daß

$$\begin{aligned}
\ldots &\equiv \exists E''_W \,.\, \exists E'_W \,.\, \mathrm{ite}(\gamma(x), \mathrm{ite}(\langle\!\langle f_1\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f_0\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A)) \\
&\equiv \exists E''_W \,.\, \mathrm{ite}(\gamma(x), \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f_1\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A), \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f_0\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A)) \\
&\equiv \exists E''_W \,.\, \mathrm{ite}(\gamma(x), \langle\!\langle \underbrace{{_{\mathsf{c}}\mathsf{ite}_\exists}(f_1,g,h,\gamma,E')}_{r}\rangle\!\rangle_A, \langle\!\langle \underbrace{{_{\mathsf{c}}\mathsf{ite}_\exists}(f_0,g,h,\gamma,E')}_{l}\rangle\!\rangle_A) \\
&\equiv \ldots
\end{aligned}$$

nach Induktionsvoraussetzung. Mit der Korrektheit von $\mathsf{ite}_\exists$ folgt

$$\ldots \equiv \exists E''_W \,.\, \mathrm{ite}(\langle\!\langle \rho(m)\rangle\!\rangle_A, \langle\!\langle r\rangle\!\rangle_A, \langle\!\langle l\rangle\!\rangle_A) \equiv \langle\!\langle \mathsf{ite}_\exists(\rho(m), r, l, E'')\rangle\!\rangle_A = {_{\mathsf{c}}\mathsf{ite}_\exists}(f,g,h,\rho,E)$$

Nun sei $m < \mathsf{top}(f)$ und E', E'' entsprechend dem zweiten Fall im Algorithmus gewählt. Wiederum definiere $E'_W := A^{-1}(E')$, $E''_W := A^{-1}(E'')$ und $x := A^{-1}(m)$. Jetzt zerlege man g und h nach der Hilfsbehauptung aus Gleichung (3.6) auf Seite 69

$$\begin{aligned}
&\exists E_W \,.\, \mathrm{ite}(\langle\!\langle f\rangle\!\rangle_A\gamma, \langle\!\langle g\rangle\!\rangle_A, \langle\!\langle h\rangle\!\rangle_A) \\
&\quad\equiv \exists E_W \,.\, \mathrm{ite}(\,\langle\!\langle f\rangle\!\rangle_A\gamma, \mathrm{ite}(x, \langle\!\langle g_1\rangle\!\rangle_A, \langle\!\langle g_0\rangle\!\rangle_A), \mathrm{ite}(x, \langle\!\langle h_1\rangle\!\rangle_A, \langle\!\langle h_0\rangle\!\rangle_A)\,) \\
&\quad\equiv \ldots
\end{aligned}$$

Expansion der drei „ite“ ergibt

$$\begin{aligned}
\ldots &\equiv \exists E_W \,.\, \langle\!\langle f \rangle\!\rangle_A \gamma x \,\langle\!\langle g_1 \rangle\!\rangle_A \vee \langle\!\langle f \rangle\!\rangle_A \gamma \overline{x} \,\langle\!\langle g_0 \rangle\!\rangle_A \vee \overline{\langle\!\langle f \rangle\!\rangle_A} \gamma x \,\langle\!\langle h_1 \rangle\!\rangle_A \vee \overline{\langle\!\langle f \rangle\!\rangle_A} \gamma \overline{x} \,\langle\!\langle h_0 \rangle\!\rangle_A \\
&\equiv \exists E_W \,.\, x \,\mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A) \vee \overline{x} \,\mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A) \\
&\equiv \exists E_W \,.\, \mathrm{ite}(x, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A)) \\
&\equiv \exists E''_W \,.\, \exists E'_W \,.\, \mathrm{ite}(x, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A)) \\
&\equiv \exists E''_W \,.\, \mathrm{ite}(x, \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A), \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A)) \\
&\equiv \ldots
\end{aligned}$$

da $x \notin E'_W$. Mit der Induktionsvoraussetzung folgt

$$\begin{aligned}
\ldots &\equiv \exists E''_W \,.\, \mathrm{ite}(\langle\!\langle 0 \cdots m - 1 \rangle\!\rangle_A, \langle\!\langle r \rangle\!\rangle_A, \langle\!\langle l \rangle\!\rangle_A) \equiv \langle\!\langle \mathsf{ite}_\exists(0 \cdots m - 1, r, l, E'') \rangle\!\rangle_A \\
&\equiv \langle\!\langle {}_{\mathsf{c}}\mathsf{ite}_\exists(f, g, h, \rho, E) \rangle\!\rangle_A
\end{aligned}$$

Terminierung folgt analog wie oben, die Terminierung von $\mathsf{ite}_\exists$ vorausgesetzt. Da keine Knoten konstruiert werden, folgt sofort ${}_{\mathsf{c}}\mathsf{ite}_\exists(f, g, h, \rho, E) \in \mathsf{ROBDD}$. ❑

Der Aufwand dieses Algorithmus ist natürlich exponentiell, da der Spezialfall $g = 1$, $h = 0$ und $E = \emptyset$ dem $\mathsf{compose}(f, \rho)$ entspricht und damit jede beliebige Variablenumordnung realisiert werden kann (vgl. Kapitel 4 und Seite 90 und 82). Weitere Spezialisierungen sind

$$\begin{aligned}
{}_{\mathsf{c}}\mathsf{ite}_\exists(f, 1, 0, \rho, \emptyset) &= \mathsf{compose}(f, \rho) \\
{}_{\mathsf{c}}\mathsf{ite}_\exists(f, g, 0, \mathsf{id}, E) &= \mathsf{relProd}(f, g, E) \\
{}_{\mathsf{c}}\mathsf{ite}_\exists(f, g, 0, \rho', 2 \cdot \mathbb{N} + 1) &= \mathsf{collapse}(f, g) \\
{}_{\mathsf{c}}\mathsf{ite}_\exists(g, 1, 0, \{s_i \mapsto f_i\}, I) &= \mathsf{preImg}(g, [f_1, \ldots, f_n], I)
\end{aligned}$$

mit $\rho' := \{ j \mapsto 0 \cdots i - 1 \mid j \in 2 \cdot \mathbb{N},\ i = j + 1 \}$ und natürlich lassen sich deren Spezialisierungen wieder mit ${}_{\mathsf{c}}\mathsf{ite}_\exists$ berechnen. Neu an dieser Auflistung sind auf jeden Fall $\mathsf{collapse}$ und preImg. Im Falle von preImg ist dies ein neuer Algorithmus zu dessen Berechnung, wobei hier schon $\mathsf{compose}_\exists$ ausreicht. Für $\mathsf{collapse}$ sind nun beliebige Mengen und Substitutionen erlaubt.

Man könnte sogar soweit gehen, alle oben vorgestellten Algorithmen in den ${}_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus einzubauen. Damit hätte man *einen* Algorithmus, der die meisten wichtigen BDD-Algorithmen subsumiert. Um dies zu erreichen müßten aber noch sehr viele Fallunterscheidungen berücksichtigt werden, die eine Implementierung unter ausschließlicher Verwendung des ${}_{\mathsf{c}}\mathsf{ite}_\exists$ für alle Spezialagorithmen stark bremsen würde. Man beachte z. B., daß in der vorgestellten Version im ${}_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus selbst keine Knoten erzeugt werden. Würde man also naiverweise einfach die Aufrufe von $\mathsf{ite}_\exists$ und exists durch Aufrufe von ${}_{\mathsf{c}}\mathsf{ite}_\exists$ mit entsprechenden Parametern ersetzen, so würde der entstehende Algorithmus für keine Wahl der Argumente terminieren. Man kann also auch nicht erwarten, daß durch dieses Zusammenführen ein knapper, übersichtlicher Algorithmus entsteht.

In der BDD-Bibliothek `BDDsimple` wurde dieser allgemeine Algorithmus implementiert. Zusätzlich erkennt die μcke, ob eine Substitution in einem Relationenprodukt nicht sofort ausgewertet werden muß, sondern mit dem ${}_{\mathsf{c}}\mathsf{ite}_\exists$ berechnet werden kann. Dies ist zum Beispiel im folgenden μ-Kalkül-Term der Fall, der aus dem Rumpf der Definition eines Prädikates stammt, das für die Menge der erreichbaren Zustände steht (vgl. mit Beispiel auf Seite 19 und 94).

$$\exists t \,.\, T(t, s) \wedge X(t)$$

	nur relProd		allgemeines $_{\mathsf{c}}\mathsf{ite}_\exists$			spezielles $_{\mathsf{c}}\mathsf{ite}_\exists^{\mathsf{var}}$		
	#relProd	sec	#$_{\mathsf{c}}\mathsf{ite}_\exists$	#relProd	sec	#$_{\mathsf{c}}\mathsf{ite}_\exists^{\mathsf{var}}$	#relProd	sec
2	2431	0.28	2828	1300	0.33	1245	1138	0.28
3	8935	0.45	13605	4030	0.73	5396	3381	0.46
4	18120	0.70	32988	7600	1.34	11565	6249	0.74
5	36341	1.10	80910	13640	2.74	25402	10256	1.15
6	59362	1.58	161573	21663	4.93	43564	14645	1.67
7	94529	2.24	294802	32899	8.26	72683	19972	2.40
8	139914	3.30	511046	47609	13.93	108475	25604	3.50
9	205301	4.54	1191601	90816	29.98	167391	32648	4.86
10	293020	6.38	2984880	199980	71.13	242221	39867	6.70

Tabelle 3.2: Vergleich von relProd, $_{\mathsf{c}}\mathsf{ite}_\exists$ und $_{\mathsf{c}}\mathsf{ite}_\exists^{\mathsf{var}}$ an Hand des Schedulers.

Nun nehme man an, daß dieser Term in der Fixpunktiteration von X auftritt, so daß sich bei der Berechnung der BDD-Semantik von X in jeder Iteration der Wert von X ändert und so jedes mal auch der Term $X(t)$ einen anderen Wert bekommt. Der Wert f von $X(t)$ berechnet sich durch eine Substitution ρ aus dem BDD f_X^i für $X(\pi(X))$ der vorigen (i-ten) Iteration. In diesem Beispiel könnte z. B. $\pi(X) = (s)$ gelten, so daß bei einer entsprechenden Allokation A

$$\rho(i) = j, \text{ für } A(s[k]) = i,\ A(t[k]) = j,\ 0 \leq k < |s|$$

Die Substitution vom *einmalig* berechneten BDD für T (hier eigentlich auch $T(\pi(T))$) muß nur einmal ausgeführt werden. Das Resultat g kann für die nächste Iteration gespeichert werden, was die μcke automatisch erkennt. Nach diesen Substitutionen muß das Relationenprodukt berechnet werden. Insgesamt also

$$\mathsf{relProd}(\mathsf{compose}(f,\rho), g, E) = {}_{\mathsf{c}}\mathsf{ite}_\exists(f, g, 0, \rho, E), \quad \text{mit } E := \{A(t[k]) \mid 0 \leq k < |t|\}$$

Man beachte, daß hier *keine* monotone Substitutione vorliegen muß, was die Voraussetzung für die Anwendung des collapse-Algorithmus ist.

3.8.5 Verbesserung für Variablensubstitutionen

Die hier und auch allgemein bei der Berechnung der BDD-Semantik nach Satz 3.24 auftretenden Substitutionen sind Variablensubstitutionen (vgl. Def. 3.30), ohne daß dazu im Algorithmus für $_{\mathsf{c}}\mathsf{ite}_\exists$ ein Sonderfall existiert. Vergleicht man den collapse-Algorithmus mit dem $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus, so stellt man fest, daß beim $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus sogar immer nur genau ein Argument beim rekursiven Aufruf eine größere Variable hat als das entsprechende Argument in der aufrufenden Funktionsinstanz. Beim collapse-Algorithmus können beide Argumente erhöht werden, was weniger Rekursionen erwarten läßt. Dies bestätigte sich auch in der Praxis. In Tabelle 3.2 finden sich Laufzeiten für den Test auf Bisimulation des Schedulers von Milner mit seiner Spezifikation (vgl. Abschnitt C.5 und 5.4.2).[16]

In der Tabelle sind drei Experimente aufgeführt für jeweils zwei bis zehn Agenten. Die erste Messung verwendet explizite Substitutionen und den relProd-Algorithmus. Die zweite

[16] Hier wurde eine Version des Schedulers verwendet, bei der möglichst viele geschachtelte Quantoren auftreten. In diesem Fall, ist der Abstand von Bild zum Urbild bei der Substitution einer BDD-Variable als natürliche Zahl relativ groß, was beim allgemeinen $_{\mathsf{c}}\mathsf{ite}_\exists$-Algorithmus zu vielen zusätzlichen rekursiven Aufrufen führt.

$\boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f,g,h,\rho,E)} ::=$

if $f=0$ **then** $\mathsf{exists}(h,E)$
else if $f=1$ **then** $\mathsf{exists}(g,E)$
else
  if $\rho(\mathsf{top}(f)) \prec g,h$ **then**
    $\mathsf{ite}_{\exists}(\rho(m),r,l,E'')$
    where

$$\begin{aligned}
m &= \mathsf{top}(f)\\
n &= \mathsf{top}(\rho(m))\\
E'' &= \{n\}\cap E\\
E' &= E\backslash E''\\
l &= \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f_0,g_0,h_0,\rho,E')}\\
r &= \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f_1,g_1,h_1,\rho,E')}\\
(f_0,f_1) &= \mathsf{cf}(f,m)\\
(g_0,g_1) &= \mathsf{cf}(g,n)\\
(h_0,h_1) &= \mathsf{cf}(h,n)
\end{aligned}$$

  else
    $\mathsf{ite}_{\exists}(0 \cdots m - 1,r,l,E'')$
    where

$$\begin{aligned}
m &= \mathsf{min}\{\mathsf{top}(g),\mathsf{top}(h)\}\\
E'' &= \{m\}\cap E\\
E' &= E\backslash E''\\
l &= \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f,g_0,h_0,\rho,E')}\\
r &= \boxed{{}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f,g_1,h_1,\rho,E')}\\
(g_0,g_1) &= \mathsf{cf}(g,m)\\
(h_0,h_1) &= \mathsf{cf}(h,m)
\end{aligned}$$

Abbildung 3.40: ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}: \mathsf{ROBDD}^3 \times [\mathbb{N} \to \mathsf{ROBDD}] \times \mathbb{P}(\mathbb{N}) \to \mathsf{ROBDD}$ mit ρ Variablensubstitution.

Messung verwendet den allgemeinen ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus und die dritte den in diesem Unterabschnitt noch vorgestellten speziellen ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$-Algorithmus. Bei den letzten zwei Experimenten wurde auch die Anzahl rekursiver Aufrufe des $\mathsf{relProd}$-Algorithmus angegeben, da dieser als Spezialfall des $\mathsf{ite}_{\exists}$-Algorithmus verwendet wird, welcher wiederum als Spezialfall von ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$ bzw. ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$ auftritt. Die Anzahl rekursiver Aufrufe von $\mathsf{compose}_{\exists}$ und $\mathsf{ite}_{\exists}$ wurde in der Tabelle nicht aufgeführt, sie sind aber auch vernachlässigbar. Die Messungen zeigen, daß der allgemeine ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus nicht mit dem $\mathsf{relProd}$-Algorithmus konkurieren kann. Deshalb braucht man den speziellen ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$-Algorithmus aus Abb. 3.40.

In der BDD-Bibliothek `BDDsimple` wurde statt Verwendung eines weiteren Algorithmus eine zusätzliche Abfrage in ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$ eingebaut, die überprüft, ob $\rho(\mathsf{top}(f))$ eine Variable ist und in diesem Fall den Rumpf von ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$ verwendet. Des weiteren wurde versucht alle Spezialfälle abzudecken. Darunter fallen wie schon erwähnt $\mathsf{compose}_{\exists}$ und $\mathsf{ite}_{\exists}$, aber auch Algorithmen für die Kombination von exists bzw. $\mathsf{compose}$ mit den Algorithmen für die Operationen aus

Abb. 3.1. Darüber hinaus war ein erheblicher Aufwand vonnöten, die auftretenden Mengen (E und E') samt Mengenoperationen effizient auf der Repräsentation durch BDDs zu implementieren (vgl. mit Algorithmus exists^2 von Seite 83). Hier ist noch ein deutlicher Geschwindigkeitszuwachs zu erwarten, wenn man statt BDDs Bitvektoren zur Repräsentation von Mengen von BDD-Variablen verwenden würde. So läßt sich die etwas langsamere Laufzeit von ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$ gegenüber ausschließlicher Verwendung von $\mathsf{relProd}$ erklären. Zusätzlich muß aber erwähnt werden, daß ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$ und ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$ nicht kommutativ bezüglich den ersten beiden Argumenten sind, wenn das dritte Argument 0 ist, so daß die Optimierung von Seite 60 nicht verwendet werden kann, was bei $\mathsf{relProd}$ der Fall ist. Dies gilt auch für den $\mathsf{collapse}$-Algorithmus aus dem SMV-System. Den Quellcode des ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus kombiniert mit ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$-Algorithmus, wie er in der BDD-Bibliothek `BDDsimple` implementiert ist, findet man in Abschnitt C.1.

Satz 3.43 (Semantik von ${}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}$) *Für $f,g,h \in \mathsf{ROBDD}$, ρ Variablensubstitution, $E \subseteq \mathbb{N}$ gilt*

$$\langle\!\langle {}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f,g,h,\rho,E) \rangle\!\rangle_A \;\equiv\; \exists E_W \,.\, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g \rangle\!\rangle_A, \langle\!\langle h \rangle\!\rangle_A),$$

A, γ und E_W wie im Beweis von Satz 3.42.

Beweis: Der Beweis wird geführt durch Induktion über den Termaufbau von f, g und h. Der Induktionsanfang ist klar. Sei also $f \notin \{0,1\}$ und es gelte $\rho(\mathsf{top}(f)) \prec g,h$. Die Bezeichner stammen aus dem Algorithmus. Weiter definiere $x := A^{-1}(m)$ und

$$E''_W := A^{-1}(E''), \quad E'_W := A^{-1}(E')$$

Zunächst ergibt die Voraussetzung

$$\begin{aligned}
\langle\!\langle f \rangle\!\rangle_A \lambda &\equiv \lambda(x)\, \langle\!\langle f_1 \rangle\!\rangle_A \lambda \,\vee\, \overline{\lambda(x)}\, \langle\!\langle f_0 \rangle\!\rangle_A \lambda \equiv \mathrm{ite}(\lambda(x), \langle\!\langle f_1 \rangle\!\rangle_A \lambda, \langle\!\langle f_0 \rangle\!\rangle_A \lambda)\\
\langle\!\langle g \rangle\!\rangle_A &\equiv \lambda(x)\, \langle\!\langle g_1 \rangle\!\rangle_A \,\vee\, \overline{\lambda(x)}\, \langle\!\langle g_0 \rangle\!\rangle_A \equiv \mathrm{ite}(\lambda(x), \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle g_0 \rangle\!\rangle_A)\\
\langle\!\langle h \rangle\!\rangle_A &\equiv \lambda(x)\, \langle\!\langle h_1 \rangle\!\rangle_A \,\vee\, \overline{\lambda(x)}\, \langle\!\langle h_0 \rangle\!\rangle_A \equiv \mathrm{ite}(\lambda(x), \langle\!\langle h_1 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A)
\end{aligned}$$

Damit erhält man mit Lemma 2.13 und „Ausmultiplizieren“

$$\begin{aligned}
&\exists E_W \,.\, \mathrm{ite}(\langle\!\langle f \rangle\!\rangle_A \gamma, \langle\!\langle g \rangle\!\rangle_A, \langle\!\langle h \rangle\!\rangle_A)\\
&\quad\equiv \exists E_W \,.\, \lambda(x)\, (\langle\!\langle f_1 \rangle\!\rangle_A \lambda\, \langle\!\langle g_1 \rangle\!\rangle_A \vee \overline{\langle\!\langle f_1 \rangle\!\rangle_A \lambda}\, \langle\!\langle h_1 \rangle\!\rangle_A) \vee\\
&\qquad\qquad \overline{\lambda(x)}\, (\langle\!\langle f_0 \rangle\!\rangle_A \lambda\, \langle\!\langle g_0 \rangle\!\rangle_A \vee \overline{\langle\!\langle f_0 \rangle\!\rangle_A \lambda}\, \langle\!\langle h_0 \rangle\!\rangle_A)\\
&\quad\equiv \exists E_W \,.\, \mathrm{ite}(\lambda(x), \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle_A \lambda, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle_A \lambda, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A))\\
&\quad\equiv \exists E''_W \,.\, \exists E'_W \,.\, \mathrm{ite}(\lambda(x), \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle_A \lambda, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A), \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle_A \lambda, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A))\\
&\quad\equiv \exists E''_W \,.\, \mathrm{ite}(\lambda(x), \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f_1 \rangle\!\rangle_A \lambda, \langle\!\langle g_1 \rangle\!\rangle_A, \langle\!\langle h_1 \rangle\!\rangle_A),\\
&\qquad\qquad\qquad\qquad \exists E'_W \,.\, \mathrm{ite}(\langle\!\langle f_0 \rangle\!\rangle_A \lambda, \langle\!\langle g_0 \rangle\!\rangle_A, \langle\!\langle h_0 \rangle\!\rangle_A))\\
&\quad\equiv \exists E''_W \,.\, \mathrm{ite}(\lambda(x), \langle\!\langle {}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f_1,g_1,h_1,\rho,E') \rangle\!\rangle_A, \langle\!\langle {}_{\mathsf{c}}\mathsf{ite}_{\exists}^{\mathsf{var}}(f_0,g_0,h_0,\rho,E') \rangle\!\rangle_A)\\
&\quad\equiv \exists E''_W \,.\, \mathrm{ite}(\lambda(x), \langle\!\langle r \rangle\!\rangle_A, \langle\!\langle l \rangle\!\rangle_A)\\
&\quad\equiv \langle\!\langle \mathsf{ite}_{\exists}(\rho(m), r, l, E'') \rangle\!\rangle_A
\end{aligned}$$

nach zweimaliger Anwendung der Induktionsvorausetzung. Der zweite Fall ist derselbe wie der zweite Fall beim Beweis von Satz 3.42. ❑

	Hashtabellengröße: 524287						Hashtabellengröße: 127			
	relProd			$_c\mathsf{ite}_\exists$			relProd		$_c\mathsf{ite}_\exists$	
n	#Knoten	MB	sec	#Knoten	MB	sec	#gc	sec	#gc	sec
8	1182	5	0.3	928	5	0.3	26	0.2	18	0.2
9	2237	5	0.5	1727	5	0.5	45	0.3	32	0.3
10	4318	5	0.7	3296	5	0.7	84	0.6	57	0.6
11	8449	5	1.1	6403	5	1.1	132	1.1	100	1.1
12	16678	6	2.0	12584	6	1.9	261	2.1	181	2.0
13	33101	7	3.9	24911	7	3.7	517	4.3	337	4.0
14	65910	9	7.7	49528	8	7.1	962	8.4	621	7.7
15	131489	11	15.3	98723	10	14.2	1771	17.2	1151	15.8
16	262606	14	30.4	197072	13	27.6	3327	34.5	2055	30.9

Tabelle 3.3: Vergleich von relProd mit $_c\mathsf{ite}_\exists$ an Hand eines n-Bit-Zählers.

Zum Schluß sei noch ein Beispiel vorgestellt, das zeigt, wie der $_c\mathsf{ite}_\exists$-Algorithmus vorteilhaft eingesetzt werden kann. Am eindeutigsten setzt sich der $_c\mathsf{ite}_\exists$ von relProd bei Verifikationsaufgaben ab, bei denen viele Fixpunktiterationen auftreten, da hierbei sehr viele Zwischenresultate erzeugt werden. Ein besonders geeignetes Beispiel ist demnach ein n-Bit-Zähler, bei dem bei einfacher Vorgehensweise (vgl. mit Abschnitt 5.4.2 und C.7) 2^n Approximationen bei der Berechnung der erreichbaren Zustände auftreten.

In Tabelle 3.3 sind für jeden Algorithmus zwei Experimente aufgeführt. Einmal mit einer so großen Hashtabelle mit 524287 Einträgen („unique table size"), daß für bis zu 16 Bits keine Speicherbereinigung („garbage collection") auftritt. Das zweite Experiment verwendet eine solch große Hashtabelle mit 127 Einträgen, daß gerade keine automatische Vergrößerung der Hashtabelle auftritt. Dies bedeutet umgekehrt, daß hier maximal viele Speicherbereinigungen stattfinden. Die dabei auftretende Anzahl BDD-Knoten kann um ein Vielfaches höher als die Anzahl Einträge liegen, da Kollisionen durch *direkte Verkettung* aufgelöst werden (s. /Wirth, 1986/). Beim ersten Experiment ist die Anzahl vorkommender Knoten in der Tabelle aufgeführt. Beim zweiten Experiment war bei 16 Bit die maximale Anzahl Knoten 311. Ansonsten ist hier unter „#gc" die Anzahl benötigter Speicherbereinigungen aufgelistet.

Hier zeigt sich der Effekt, daß die Verwendung von $_c\mathsf{ite}_\exists$ wesentlich weniger Knoten erzeugt. Im ersten Beispiel kann dies direkt gezählt werden und man errechnet eine Ersparnis von 24% erzeugten Knoten. Beim zweiten Experiment spart man sich 38% Speicherbereinigungen. Insgesamt wirkt sich dies in beiden Beispielen in einer um etwa 10% schnelleren Laufzeit aus. Bei allen vom Autor betrachten Beispielen verhielt es sich ähnlich. Immer wurde beim Einsatz von $_c\mathsf{ite}_\exists$ weniger Knoten erzeugt bzw. weniger Speicherbereinigungen benötigt. Die Ersparnis bei der Laufzeit war meist positiv. Eine Verbesserung der Behandlung von Mengen von BDD-Variablen sollte diesen positiven Trend noch verstärken.

3.9 Zusammenfassung

In diesem Kapitel wurde ein neuer Zugang zu BDDs als freie Termalgebra modulo einem kanonischen Termersetzungssystem vorgestellt. Dies gelang durch Einführung einer abstrakten Maschine, mit deren Hilfe sich die für die Modellprüfung wesentlichen Algorithmen auf einer einheitlichen Weise beschreiben und *formal* als korrektnachweisen ließen. Dann wurde untersucht, wie wichtig optimierte BDD-Algorithmen für die Modellprüfung sind. Schließlich wurde noch ausgehend von der Forderung nach effizienter Modellprüfung des gesamten μ-Kalküls ein weiterer Algorithmus (${}_{\mathsf{c}}\mathsf{ite}_{\exists}$) entwickelt, der eine Verallgemeinerung bekannter, optimierter BDD-Algorithmen für die Modellprüfung darstellt.

3.10 Ausblick

Die SCAM-basierte Implementierung von BDD-Bibliotheken verspricht einen Geschwindigkeitsgewinn durch bestmögliche Behandlung von Speicherbereinigung unter Beibehandlung einer sehr effizienten Implementierungssprache wie C und C++. Hier sollte die begonnene Implementierung von `BDD` (vgl. S. 65) zu Ende geführt werden, um dies auch in der Praxis zu bestätigen. Dabei ist auch eine effiziente Behandlung von Mengen von BDD-Variablen einzuplanen, damit der ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus gegenüber der Verwendung von $\mathsf{relProd}$ noch besser abschneidet.

Es ist vorgesehen die Eingabesprache der μcke um beliebige (nicht nur boolesche), endliche Funktionen zu erweitern. Dann läßt sich der ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus direkt mit dem preImg-Operator aus /Coudert und Madre, 1990/ vergleichen.

Schließlich sollte untersucht werden, wie der ${}_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus benützt werden kann, um bei dynamischer Variablenordnung nach /Rudell, 1993/ dennoch für schnelle Berechnung des Relationenproduktes zu sorgen (vgl. S. 162).

Kapitel 4

Allokationsrandbedingungen

Die Generierung von guten Variablenallokationen ist eines der Hauptprobleme bei der Modellprüfung im μ-Kalkül mit BDDs. Hierfür stellt dieses Kapitel ein Verfahren vor.

4.1 Übersicht

Die Größe eines BDDs hängt sehr empfindlich von der gewählten Allokation (Variablenordnung) ab. Dies wird an einigen Beispielen erläutert. Dabei wird klar, daß sich für viele Anwendungsgebiete „gute“ und „schlechte“ Allokationen dadurch unterscheiden, daß sie gewisse Randbedingungen (engl. constraints) erfüllen oder nicht.

Für die Modellprüfung gibt es zusätzliche Anforderungen an Allokationen. Sie sollten sicherstellen, daß BDD-Substitutionen, die bei der iterativen Berechnung von Prädikaten auftreten, schnell durchgeführt werden können.

Diese Randbedingungen werden formalisiert und es wird gezeigt, wie verschiedene Randbedingungen kombiniert werden können. So erhält man einen weitestgehend automatischen Allokationsalgorithmus für den Modellprüfer μcke. In vergleichbaren Systemen /Rauzy, 1995, Enders et al., 1993/ muß der Benutzer die Allokationen per Hand angeben und verliert einen großen Teil des Komforts gegenüber Spezialmodellprüfer wie dem SMV /McMillan, 1993a/, bei dem gute Allokationen, was die Substitutionen betrifft, für die verwendete eingeschränkte Eingabesprache fest gewählt werden können.

4.2 Einführung

Die Normalform eines BDDs ist nur eindeutig bis auf die gewählte Variablenordnung. Dadurch ergibt sich natürlich die Fragestellung, inwieweit verschiedene Variablenordnungen den reduzierten BDD beeinflussen. Der wichtigste Gesichtspunkt, auf den diese Arbeit sich beschränkt, ist die Größe des reduzierten BDDs unter verschiedenen Variablenordnungen.

Aus praktischen Gesichtsgründen stellt sich eher die umgekehrte Frage: Wie kann für eine gegebene Boolesche Funktion eine Variablenordnung bestimmt werden, so daß der reduzierte BDD möglichst wenige Knoten besitzt? Diese Fragestellung wird als Problem der *Variablenallokation* bezeichnet. Die ersten Untersuchungen auf diesem Gebiet sind schon in /Bryant, 1986/ erfolgt. Dort wurde eine exponentielle untere Schranke für die Multiplikation gezeigt. Genauer wurde bewiesen, daß es bei einer festen Variablenordnung unter den BDDs für die $2 \cdot n$ booleschen Funktionen, die die „Ausgabebits“ der Multiplikation zweier n-bit Zahlen darstellen, einer vorhanden sein muß, der in Abhängigkeit von n exponentiell viele Knoten besitzt.

Weitere Funktionen, wie die „*hidden weighted bit function*“, fallen unter dieselbe Klasse von booleschen Funktionen, für die es keine polynomial großen (RO)BDDs gibt (/Bryant, 1991/). Dies führte zu Untersuchungen, die versuchen, das Konzept der BDDs so zu erweitern, daß die schönen Eigenschaften der BDDs, wie Normalform, polynomiale Kompositionsalgorithmen usw. erhalten bleiben, aber die Klasse der polynomial repräsentierbaren booleschen Funktionen größer wird. Eine Übersicht über diese Arbeiten findet man in /Sieling, 1995/.

Boolesche Funktionen, die ein Schaltnetz oder die Übergangsrelation eines Schaltwerkes, Automaten oder reaktiven Systems usw. darstellen, sind meistens nicht abstrakt, sondern durch einen Ausdruck in einer formalen Sprache gegeben. Bei Schaltnetzen ist das üblicherweise eine Beschreibung als Netzliste, bei den anderen Beispielen sind das Zustandübergangstabellen bis hin zu Beschreibungen in den entsprechenden Programmiersprachen. Unter diesen „Programmiersprachen“ sind solche besonders interessant, die eine formale Semantik besitzen, oder die für die betrachteten Anwendungsgebiete speziell entworfen wurden. Im Bereich des Entwurfs von Kontrollern für Regelungs- und Steuerungsaufgaben sind vor allem synchrone Sprachen wie LUSTRE /Halbwachs et al., 1991/ zu nennen. Aber auch der graphische Formalismus STATECHARTS /Harel, 1988/ gewinnt immer mehr an Bedeutung. Diese zwei Beispiele zeichnen sich durch eine exakte Semantik und eine weitestgehende Akzeptanz zumindest im universitären Bereich aus. Letzteres trifft auch im hohen Maße auf die vor allem im Telekommunikationsbereich weit verbreitete Sprache SDL /CCITT, 1992, Hogrefe, 1989/ zu. Bei der Hardware-Verifikation ist dies die Beschreibungssprache VHDL.

Eine Teilaufgabe der Verifikation mit BDDs ist nun die Übersetzung solch implizit definierten booleschen Funktion in eine Darstellung als BDD (in der Schaltnetzverifikation ist dies zumeist die einzige Aufgabe). Dabei muß man i. allg. den (Zustands-)Variablen des formal beschriebenen Systems BDD-Variablen zuordnen. Wiederum heißt diese Zuordnung Variablenallokation.

Komplexitätstheoretische Untersuchungen (vgl. /Meinel und Slobodova, 1994/) zeigen, daß die Bestimmung der optimalen Variablenordnung für eine implizit dargestellte boolesche Funktion ein NP-vollständiges Problem ist. Um also zu „guten“ Variablenallokationen zu kommen, muß man Heuristiken anwenden /Malik et al., 1988/, /Fujita et al., 1988/ und /Fujii et al., 1993/. Diese Arbeiten beschäftigen sich mit Variablenallokationen für Schaltnetze. Die Heuristiken sind relativ einfach, erlauben aber die Verifikation von praktischen Beispielen, was bei zufällig gewählten Ordnungen an der Größe der entstehenden BDDs scheitert. Eine theoretische Rechtfertigung für diese Heuristiken findet man in /Berman, 1991/.

Eine zweite Methode, gute Variablenallokationen zu finden, besteht darin, keine feste Variablenallokationen vorzuschreiben, sondern dynamisch beim Berechnen der BDDs entlang der syntaktischen Struktur des Termes, der „implizit“ die boolesche Funktion beschreibt, die Ordnung an die momentan behandelten BDDs anzupassen. Diese Idee wird in /Rudell, 1993/ näher untersucht. Eine neuere Arbeit ist /Bern et al., 1995/, in der eine Methode zur Umordnung der Variablen in BDDs vorgestellt wird, wobei vermieden wird, daß Zwischenresultate wesentlich größer als der ursprüngliche BDD und das Resultat sind.

Die beiden bisher genannten Ansätze zur Bestimmung guter Variablenordnungen sind im Umfeld der Schaltnetz-Verifikation entstanden. Bei der üblichen Definition von Modellprüfung im Gegensatz zur Schaltnetz-Verifikation müssen einmal Allokationen für Variablen gefunden werden, die Zustände des zu verifizierenden Systems beschreiben, was sich für einfach beschriebene Übergangsrelationen (keine parallelen Prozesse) mit obigen Methoden realisieren läßt. Die Verifikation (z. B. Beweis von Sicherheitseigenschaften über Erreichbarkeitsanalyse) benötigt zusätzlich noch eine Repräsentation der Übergangsrelation als BDD. Die Übergangsrelation des Systems ist dabei eine boolesche Funktion über den Variablen, die den Zustand des

Systems beschreibt, kodiert als Vektor von booleschen Werten, *und* über einer Kopie dieser Variablen, die den Folgezustand beschreibt. Es wurde schon in /Touati et al., 1990/ erwähnt, daß die Allokation für Variablen, die den aktuellen Zustand und Folgezustand beschreiben, „verschränkt" geschehen soll, mit der Begründung, daß dann effiziente Substitutionsalgorithmen für die BDDs verwendet können (vgl. mit collapse-Algorithmus aus Abb. 3.33) Diese Argumentation läßt sich nicht im Rahmen von booleschen Funktionen durchführen, sondern man muß die bei der Modellprüfung auftretenden BDD-Operationen betrachten. Insbesondere die Substitutionen, die bei Berechnung von Fixpunkten auf schon berechneten BDDs ausgeführt werden, stehen hier im Mittelpunkt. Weiter ist klar, daß man nun Variablen behandeln muß, die Vektoren von booleschen Werten beschreiben und nicht nur wie bei der Schaltnetzverifikation boolesche Werte.

Hierzu sei R die Übergangsrelation eines Systems, für die ein BDD generiert wurde. Sie hängt ab vom aktuellen Zustand s und dem Folgezustand s'. Als $\mathbb{B}_{\mu}^{V}$-Ausdruck lautet sie

$$R(s,s')$$

Bei der Erreichbarkeitsanalyse wird die Menge E der erreichbaren Zustände berechnet. Diese läßt sich als Fixpunktiteration darstellen, indem man mit der Menge der Startzustände S beginnt und daraus sukzessive die Folgezustände bestimmt. In $\mathbb{B}_{\mu}^{V}$ kann dies so beschrieben werden

$$\mu E(s).\, S(s) \vee \exists t.\, R(t,s) \wedge E(t)$$

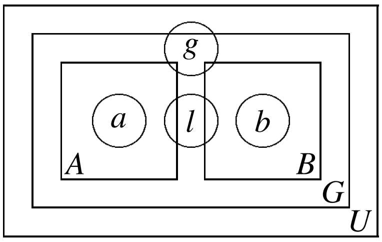
Hier muß man eine Allokation für t finden, so daß bei den Substitutionen bei der Anwendung von R bzw. E im Rumpf des Quantors keine zu großen BDDs entstehen. Es können hier große BDDs entstehen, da jede Substitution eine Umordnung der Variablen bedeutet. Dabei kann ein großer BDD enstehen, falls die Ordnung nach der Substitution einer schlechten Allokation entspricht.

Als formales System, in dem diese Betrachtungen durchgeführt werden können, wird in dieser Arbeit der μ-Kalkül (erweitert um boolesche Vektoren) verwendet. Ein erster Grund ist, daß sich viele Verifikationsprobleme durch Modellprüfung im μ-Kalkül erledigen lassen (vgl. B.2), so daß Algorithmen für die Bestimmung von „guten" Variablenallokationen im μ-Kalkül auch gute Variablenallokationen für eingeschränktere Verifikationsverfahren liefert. Darüber hinaus erlaubt ein Algorithmus für die Generierung guter Allokationen für den μ-Kalkül erst die Verwendung eines allgemeinen μ-Kalkül Modellprüfers, wie das die μcke ist. Hier steht die μcke in Konkurrenz zu spezialisierten Modellprüfern (wie dem SMV /McMillan, 1993a/) mit einer weitaus geringeren Ausdrucksfähigkeit, die aber dank einer genauen Kenntnis der Spezifikationssprache gute Variablenallokationen fest angeben können.

Neben dieser Problematik der Allokation von Variablen des aktuellen Zustandes und des Folgezustandes, müssen bei der Allokation von Variablen bei der Modellprüfung i. allg. auch solche behandelt werden, über die quantifiziert wird. Dies tritt u.a. bei Systemen auf, die rekursiv beschrieben sind, wie das z. B. bei einer Beschreibung durch Prozeßalgebren wie CCS (/Milner, 1989/) der Fall ist. Hier besteht ein System aus Untersystemen, die wiederum aus Untersystemen aufgebaut sind usw. Kombiniert werden diese Untersysteme durch einen (oder mehrere) „Parallele" Verknüpfungsoperatoren, durch die auch die Synchronisation (die Kommunikation) der einzelnen Komponenten definiert ist. Nun muß die Semantik der Synchronisation der einzelnen Komponenten, auf BDD Operationen abgebildet werden. Die Vorgehensweise, wie man hier zu einer globalen Übergangsrelation gelangt, wird dann etwa die folgende sein. Erst bestimmt man die Übergangsrelationen atomarer Untersysteme (in /Enders et al., 1993/ werden diese „elementary transition systems" genannt) unter Verwendung der oben genannten

Heuristiken für die Variablenallokation für Verifikation von Schaltnetzen. Danach geht man rekursiv die Untersysteme nach oben und berechnet die BDDs für die auftretenden Kombinationen von Untersystemen.

Das folgende Beispiel soll diesen rekursiven Schritt erläutern und aufzeigen, wo hier quantifizierte Variablen ins Spiel kommen. Das Gesamtsystem G bestehe aus zwei Untersystemen A und B, die über eine lokale Variable l synchron kommunizieren. Weiterhin gibt es noch eine globale Variable g, über die sich die Untersysteme gemeinsam mit der Umwelt U synchronisieren können. Der Einfachheit halber denotieren die auftretenden Variablen dabei Vektoren von booleschen Werten.



Nun soll die Übergangsrelation von G bestimmt werden. Die Umwelt werde dabei nicht weiter betrachtet. Dazu habe man die Übergangsrelationen R_A und R_B für die Untersysteme A und B irgendwie (z. B. in $\mathbb{B}_\mu^V$) beschrieben und dafür schon BDDs berechnet. Die Übergangsrelationen hängen von den aktuellen Werten des Zustandes des entsprechenden Untersystems, a bei A und b bei B, den Werten der Folgezustände, a' bzw. b', und l und g ab.[1]

$$R_A(g,l,a,a'), \qquad R_B(g,l,b,b')$$

Die Übergangsrelation des Gesamtsystems R_G ergibt sich dann wie folgt

$$R_G(g,a,a',b,b') = \exists l.\, R_A(g,l,a,a') \wedge R_B(g,l,b,b')$$

Der globale Zustand setzt sich aus den Zuständen der Untersysteme zusammen und die lokale Variable l tritt nach außen (zur Umwelt hin) nicht mehr auf. Hat man nun schon syntaxgerichtet den BDD für R_A und R_B berechnet, wobei man auch jeweils eine Allokation für die formalen Parameter angeben mußte, so stellt sich hier die Frage wie allokiert man die quantifizierte Variable l. Im Rumpf des Existenzquantors müssen den Komponenten von l relativ zu den Komponenten von g, a, a', b und b' BDD Variablen zugeordnet werden.

Solche Fragestellungen treten bei der Verifikation von Schaltnetzen nicht auf. Auf der anderen Seite muß jeder Modellprüfer, dessen Eingabesprache es erlaubt, Systeme durch Komposition zu beschreiben, Allokationen für solche Synchronisationsvariablen durchführen, so daß auch hier das im folgenden vorgestellte allgemeine Allokationsverfahren verwendet werden kann.

Eine genauere Analyse, wie man für die Übergangsrelation von durch CCS beschriebenen Agenten gute Variablenallokationen findet, ist in /Enders et al., 1993/ beschrieben. Diese Überlegungen sind nicht nur auf CCS beschränkt, sondern lassen sich auch auf andere kompositionale Systembeschreibungsformalismen übertragen.

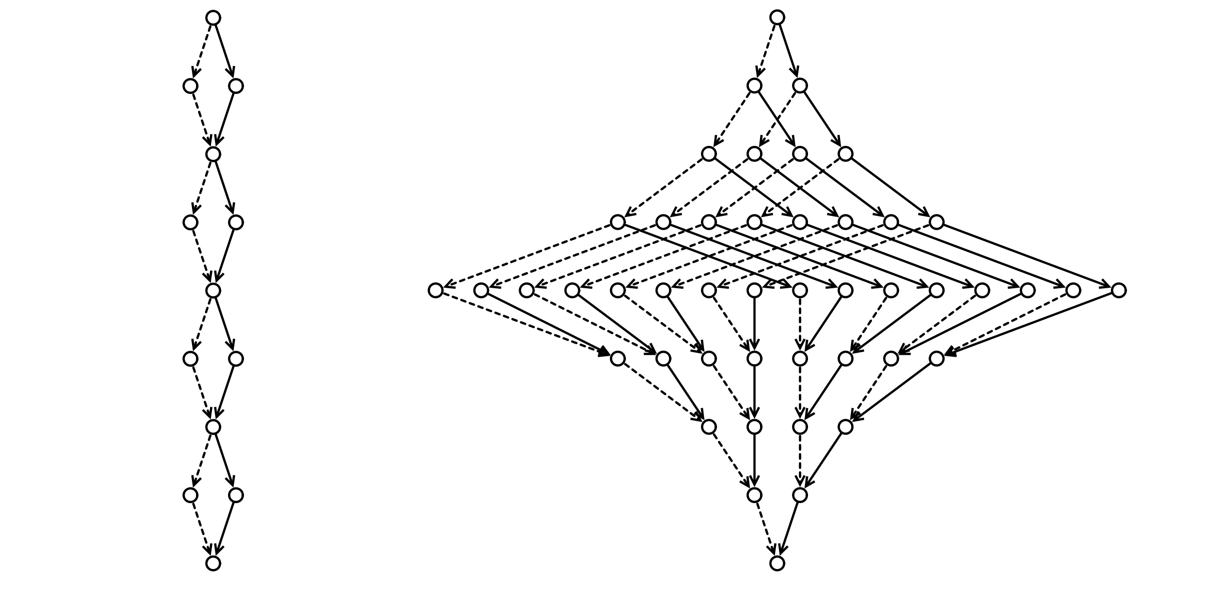
Neben diesen aus Sicht des μ-Kalküls einfachen Verifikationsaufgaben, wie Erreichbarkeitsanalyse und CTL-Modellprüfung (vgl. Abschnitt B.2), gibt es auch Aufgaben, bei denen die Übersetzung in den μ-Kalkül kompliziertere μ-Kalkül Ausdrücke erzeugt. Ein Beispiel hierfür ist die Bisimulation. Hierbei entsteht ein Term (vgl. auch mit Abschnitt B.2), der eine Vielzahl von quantifizierten Variablen enthält. Weiterhin müssen Prädikate definiert werden, die

[1] Die Variablen l und g haben keinen Folgewert. Sie dienen nur zur Synchronisation!

nicht direkt das Modell beschreiben. Ähnliche Effekte treten auf, wenn man *Fairneß* mit integriert oder wenn man den μ-Kalkül als Zwischensprache für CTL*-Modellprüfung benutzt (siehe auch Abschnitt B.2).

Für die quantifizierten Variablen und die formalen Parameter dieser zusätzlichen Prädikate müssen Variablenallokationen gefunden werden. Die Vorgehensweise von /Enders et al., 1993/, /Rauzy, 1995/ ist es, dem Benutzer die Angabe der Allokation aufzubürden. Dies stellt einmal einen unbedarften Benutzer vor gehörige Probleme. Zweitens verhindert es die einfache Änderung von μ-Kalkülausdrücken zur Optimierung der Modellprüfung (man muß ja bei jeder kleinsten Änderung die Variablenordnung wieder konsistent machen) und schließlich erschwert es erheblich den Einsatz eines μ-Kalkül-Modellprüfers als Komponente in einem Verifikations-System, da ja nun die übrigen Werkzeuge, die den Modellprüfer nur benutzen wollen, dazu gezwungen sind, sich intensiv um Variablenallokationen zu kümmern.

Um also die Modellprüfung μ-Kalkül mit BDDs effizient und praxistauglich durchführen zu können, benötigt man eine Methode, die es gestattet, gute Variablenallokationen automatisch zu generieren. Eine solche Methode vorzustellen, ist Ziel dieses Kapitels.



a. gute Allokation b. schlechte Allokation

Abbildung 4.1: Allokationen für den Term $t := u \doteq v$.

4.3 Beispiele für Allokationen

In diesem Abschnitt werden für verschiedene μ-Kalkül-Terme gute und schlechte Allokationen angegeben. Dabei werden drei Begriffe vorgestellt, die beschreiben auf welche Weise Variablen in den betrachteten Termen „allokiert“ sein müssen, um eine gute Allokation zu erhalten. Die Formalisierung dieser drei Begriffe ist Bestandteil des nächsten Abschnittes.

4.3.1 „Interleaving“

Ein oft bei der Modellprüfung im μ-Kalkül vorkommender Ausdruck ist

$$t \quad := \quad u \doteq v,$$

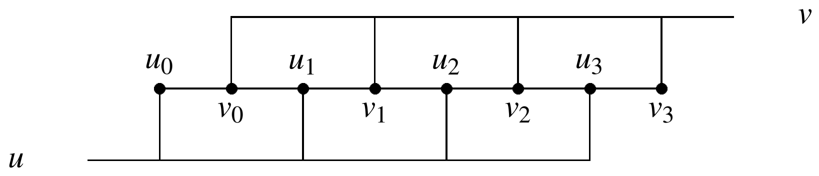
der die Gleichheit zweier boolescher Vektoren darstellt. Er wird zum Beispiel bei der Berechnung der transitiv-reflexiven Hülle oder ähnlichen Operationen gebraucht, die man in abgewandelter Form bei der Berechnung der Bisimulation braucht. Noch dramatischer wird es dann bei Interleaving-Semantiken von parallelen Systemen. Hier ändert sich ja immer nur genau der Zustand *eines* Prozesses. Der Zustand der anderen Prozesse bleibt gleich.

Bei der Modellprüfung mit BDDs muß t nach BDD übersetzt werden. Formal ist dies die Berechnung von $\|t\|_A$ für eine Allokation A. Dabei sollte die Übersetzung gewährleisten, daß der reduzierte BDD möglichst klein ist. Ob das gelingt, hängt sehr von der Wahl der Allokation ab, die man ja bei dieser Abbildung frei wählen konnte. Hier gibt es zwei Extremfälle, die motivieren sollen, warum *gute* Allokationen sehr wichtig sind.

Ein Beispiel für eine *gute Allokation* für die Übersetzung von t (die Länge von u und v sei 4) ist die Allokation A, mit

$$A(u[i]) := 2 \cdot i, \qquad A(v[i]) := 2 \cdot i + 1$$

für $0 \leq i < |u| = |v| = 4$. Hier werden die einzelnen Komponenten von u mit denen von v verschränkt. Man spricht auch von „interleaving“ von u und v. Ebenso könnte man natürlich auch v auf die geraden Zahlen abbilden und u auf die ungeraden, oder auch $A' := c \cdot A + d$ für beliebige ganze Zahlen c, d wäre möglich, und u und v blieben dennoch verschränkt. Die Visualisierung von A findet man im folgenden Diagramm.



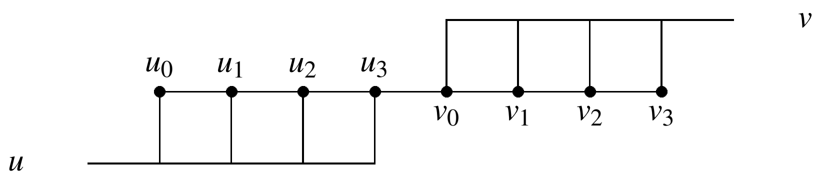
In diesem Diagramm ist ein Auschnitt der positiven Zahlengeraden zu sehen. Die natürlichen Zahlen, auf die eine Komponente einer Variablen u abgebildet wird, ist durch einen Punkt gekennzeichnet. All diese Punkte sind durch senkrechte Linien, mit einer horizontalen Linie verbunden, die mit dem Namen der Variable beschriftet ist. Weiterhin nehme man an, daß der Punkt ganz links der 0 zugeordnet ist, und die einzelnen Punkte genau um 1 auseinander liegen. Um solch ein Diagramm den Ebenen eines BDD zuzuordnen muß man es um 90 Grad im Uhrzeigersinn drehen.

Den BDD von t findet man in Abbildung 4.1.a. Für beliebige u und v, mit $|u| = |v| = n \in \mathbb{N}$, und entsprechender Modifikation von A hat dieser BDD $3 \cdot n$ innere Knoten. Er wächst also linear mit der Länge der Vektoren.

Die *schlechte Allokation B* sei definiert als

$$B(u[i]) := i, \qquad B(v[i]) := |u| + i$$

für $0 \leq i < |u| = |v| = 4$ und hat als Diagramm



Den BDD $\|t\|_B$ findet man in Abbildung 4.1.b. Dieser ist im Gegensatz zur guten Allokation A exponentiell groß in der Länge von u. Letzteres ist die Aussage des nächsten Satzes (ohne Beweis).

Satz 4.1
Für den Term $t := u \doteq v$ aus $\mathbb{B}^V_\mu$, mit $|u| = |v| = n \in \mathbb{N}$, und die Allokation B, wobei

$$B(u[i]) := i, \qquad B(v[i]) := |u| + i,$$

für $0 \leq i < |u| = |v|$, hat $\|t\|_B$ mehr als 2^n innere Knoten.

In /Bryant, 1986/ wurde ein ähnliches Beispiel gegeben bez. dem Term

$$s := x_0 \wedge x_1 \ \vee \ x_2 \wedge x_3 \ \vee \ \cdots \ \vee \ x_{2n} \wedge x_{2n+1}, \quad n \in \mathbb{N}$$

aus $\mathbb{B}^W$. Auch hier muß man diejenigen x_i mit geradzahligem Index i mit den x_j mit ungeradem Index j verschränken. Bei geblockter Allokation erhält man wiederum einen exponentiellen BDD. Für weitere praktisch relevante boolesche Funktionen, wie die Addition zweier n-bit Zahlen verhält es sich ähnlich. Man beachte auch die Erläuterungen zum collapse-Algorithmus auf Seite 93.

a. gute Allokation b. schlechte Allokation

Abbildung 4.2: Allokationen für den Term $f(u) \wedge f(v)$.

4.3.2 „Blöcke“

Nun betrachte den Term

$$t := f(u) \wedge f(v), \qquad |u| = |v| = 3,$$

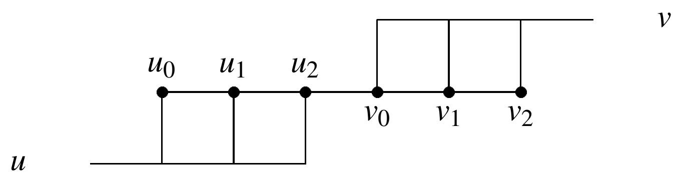
wobei die Funktion f (in $\mathbb{B}_\mu^V$) definiert ist als

$$f(w) := \text{ite}(w[0], w[1], w[2])$$

In Abbildung 4.2 sind die entsprechenden BDDs für die folgenden Allokationen dargestellt. Hier ist die gute Allokation A definiert als

$$A(u[i]) := i, \qquad A(v[i]) := |u| + i, \qquad 0 \leq i < |u| = |v| = 3$$

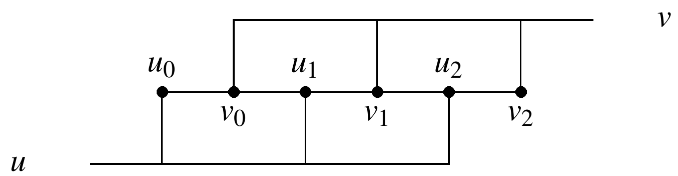
mit folgendem Diagramm



Die schlechte Allokation B ist

$$B(u[i]) := 2 \cdot i, \qquad B(v[i]) := 2 \cdot i + 1, \qquad 0 \leq i < |u| = |v| = 3$$

mit dem Diagramm



Ein prototypisches Vorkommen dieser Art von Termen findet man z. B. bei der Kombination der Übergangsrelationen von relativ unabhängigen Teilsystemen. Der Extremfall tritt dann ein, wenn ein Untersystem A vollkommen unabhängig von einem Untersystem B ist. In diesem

Fall sind die Mengen von Variablen, von denen die Übergangsrelationen (und damit die dafür berechneten BDDs) der einzelnen Systeme abhängen, disjunkt. In dem einführenden Beispiel in diesem Kapitel wäre dies z. B. dann der Fall, wenn R_A nicht von l und R_B nicht von g abhängen würde.

Der Grund dafür, daß hier A einen kleineren BDD liefert als der für B, liegt darin, daß die Anwendungen von f trivialerweise keine gemeinsamen Variablen verwenden, so daß der Wert der booleschen Funktion für $f(u)$ völlig *unabhängig* von dem Wert der booleschen Funktion für $f(v)$ ist. Oder anders ausgedrückt, die Belegung für u ist für den Wert von $f(v)$ völlig bedeutungslos.

Die Knoten in einer Ebene eines BDD entsprechen genau den verschiedenen Möglichkeiten, auf welche Weise nach „Lesen" der Variablen der höheren Ebenen, weitere Variablenbelegungen den Wert der repräsentierten booleschen Funktion noch beeinflussen. Aus diesem Grund sollten zwei Variablen, deren Belegung unmittelbar den Wert von f beeinflussen (wie das z. B. bei den i-ten Komponenten von u und v in $u \doteq v$ der Fall ist) auch möglichst nebeneinander allokiert werden. Sind zwei solche Variablen weit voneinander weg allokiert, dann müssen entsprechend viele BDD-Knoten zwischen den Ebenen dieser beiden Variablen eingeführt werden, um den Wert der höheren Variable zu speichern. Dies erklärt die Güte der Allokationen im letzten Abschnitt.

Aber auch der Umkehrschluß gilt, wie dieses Beispiel zeigt. Fügt man zwischen zwei korrelierten Variablen (z. B. den Komponenten $u[0]$ und $u[1]$ im Beispiel dieses Abschnittes) eine unabhängige Variable ($v[0]$ durch „interleaving" von u mit v) ein, dann führt das nach obiger Argumentation zu zusätzlichen BDD Knoten. Dies bedeutet also, daß die Komponenten unabhängiger Variablen nicht überschneidend, sondern „geblockt" allokiert werden sollten.

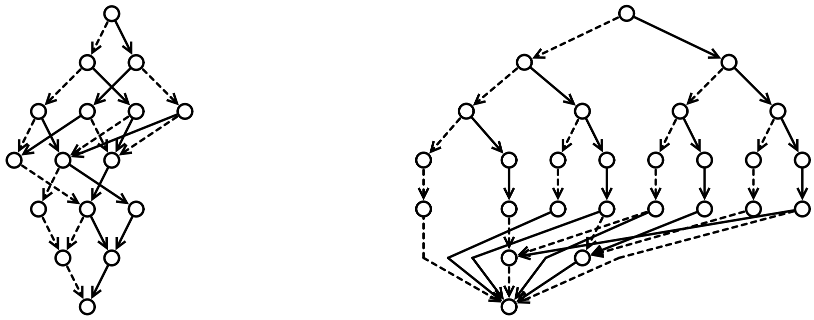
Für CCS wurde in /Enders et al., 1993/ diese Heuristik benutzt, um eine obere Schranke für die Größe von BDDs für die Übergangsrelation von Agenten anzugeben. Diese ist polynomial in der Anzahl Komponenten und exponentiell in der Größe des Komunikationsalphabetes (Anzahl Aktionen). Für eine eingeschränkte Klasse von Agenten konnte zusätzlich auch eine polynomiale Schranke gezeigt.

Hier wird deutlich, daß die beiden Konzepte des „Verschränken" und des „Blocken" komplementär sind. Später bei der formalen Behandlung motiviert dies eine Einschränkung an Randbedingungen für Allokationen, die besagt, daß zwei Variablen nicht gleichzeitig geblockt und verschränkt werden können.

4.3.3 „Ordnung"

Der dritte Bestandteil, der im nächsten Abschnitt formal eingeführten Randbedingungen für Allokationen, ist die Ordnung von μ-Kalkül Variablen, also im allgemeinen Vektoren über booleschen Werten. Diese können ja wie oben dargestellt geblockt oder verschränkt allokiert werden. Im ersten Fall ist relativ klar was das bedeuten soll. Wenn u geblockt *vor* v allokiert werden soll, dann müssen eben alle Komponenten von u vor denen von v allokiert werden. Im zweiten Fall könnte man davon sprechen, daß die erste Komponente vor der zweiten beginnt. Wegen der Monotonie von Allokationen ist dies äquivalent dazu, daß die erste Komponente der „kleineren" Variablen vor der ersten Komponente der „größeren" Variablen allokiert wird.

Hierzu wird ein Beispiel in der Eingabesprache des Modellprüfers μcke angegeben, da in der einfachen Version des μ-Kalkül $\mathbb{B}_\mu^V$ keine Typen oder Fallunterscheidungen („case-statement") vorhanden sind, was die Formulierung eines sinnvollen Beispiel erheblich erschwert. Das Beispiel stellt das Fragment eines sehr einfachen Akkumulators in einem hypothetischen Prozessor dar.



a. gute Allokation b. schlechte Allokation

Abbildung 4.3: Allokationen für die Übergangsrelation eines einfachen Akkumulators.

```
 1  enum N { And, Or, Implies, Not };
 2  bool accumulator(N n, bool old[2], bool new[2])
 3    new[1] = old[1] &
 4    case
 5      n = And:     new[0] <-> (old[0] & old[1]);
 6      n = Or:      new[0] <-> (old[0] | old[1]);
 7      n = Implies: new[0] <-> (old[0] -> old[1]);
 8      n = Not:     new[0] <+> old[0];
 9    esac;

10  bool bad_accumulator(N n, bool old[2], bool new[2])
11    accumulator(n,old,new) {AllocationConstraint new -< n};

12  #size accumulator;     #visualize accumulator;
13  #size bad_accumulator; #visualize bad_accumulator;
```

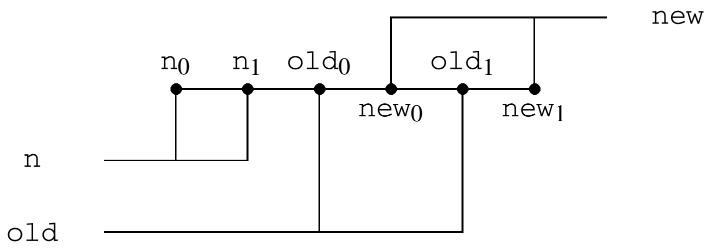
Zum Verständnis einige Erklärungen zur Eingabesprache der μcke. Tiefergehende Erläuterungen findet man im Kapitel 5. In Zeile ‚`1`‘ steht die Definition der Steuerbefehle als Aufzählungstyp, der intern (bei Verwendung von „Booleschen Prädikaten“, vgl. 5) durch zwei Bits kodiert wird, denen zwei BDD Variablen zugeordnet werden. Der hier betrachtete Akkumulator soll nur die booleschen Operatoren für das logische „Und“ und „Oder“, die „Implikation“ und die „Negation“ als Befehle verstehen. Er hat zwei boolesch-wertige Register, die beide als boolescher Vektor der Länge zwei dargestellt werden (`bool old[2]` und `bool new[2]`).

Die Übergangsrelation `accumulator` ist in Wirklichkeit eine Funktion, die einen Befehl `n` auf der Steuerleitung liest, den alten Zustand des Akkumulators `old` hernimmt, um daraus den neuen Zustand `new` zu berechnen. Die Argumente für die durch `n` gegebene boolesche Operation müssen dabei in den beiden Registern `old[0]` und `old[1]` stehen. Das Ergebnis der Operation wird in Register 0 (`new[0]`) abgelegt, der bisherige Inhalt wird also überschrieben. Register 1 (`old[1]` bzw. `new[1]` verändert sich dabei nicht (Zeile ‚`3`‘). Die Fallunterscheidung von Zeile ‚`4`‘ bis Zeile ‚`9`‘ „berechnet“ den neuen Wert von Register 0. Über das Schreiben und Abspeichern von Registern wird hier nichts ausgesagt.

Durch die ASTI-Heurisik (vgl. S. 138) werden `old` und `new` verschränkt allokiert. Ebenso verwendet die μcke die Heuristik, daß wenn sich keine anderen Ordnungs Randbedingungen ergeben, wenn möglich die lineare Anordnung aus der Eingabe beibehalten wird. Dies bedeutet für Allokation der Variablen von `accumulator`, daß `n` vor `old` und `old` vor `new` allokiert wird. So sieht auch die (gute) Allokation *A* aus, wenn die Größe des BDDs für `accumulator` in Zeile ‚`12`‘ ausgegeben werden soll, wofür zuerst der BDD berechnet werden muß.

$$A(\texttt{n}[i]) := i, \quad A(\texttt{old}[i]) := 2 \cdot i + 2, \quad A(\texttt{new}[i]) := 2 \cdot i + 3, \quad \text{für } 0 \leq i < 2$$

Als Diagramm kann man A wie folgt darstellen

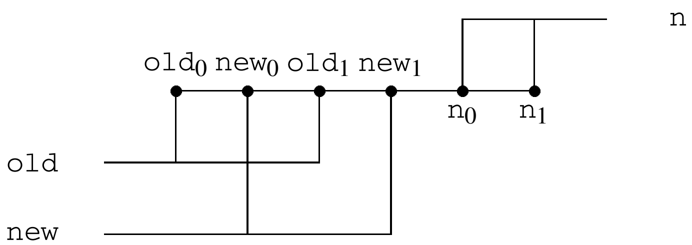


Das Kommando `#visualize` veranlaßt die μcke anschließend, diesen BDD zu visualisieren. Das Ergebnis findet man in Abbildung 4.3.a. Das „Gute“ an dieser Allokation ist, daß die Steuerleitung vor den Daten allokiert ist. Anhand des Inhalts der Steuerleitung wird entschieden, welche Operation der Akkumulator ausführt. Für ein *Entscheidungs*diagramm für die Übergangsrelation des Akkumulators sollte deswegen die Fallunterscheiung nach Art des Operators auch die erste sein.

Bei der schlechten Allokation werden alle Operationen im Prinzip erst ausgeführt werden, bevor entschieden wird, welche denn nun angefordert wurde (ähnlich einem „delayed branch“ in modernen RISC Prozessoren). Dies ergibt die (schlechte) Allokation B

$$B(\texttt{old}[i]) := 2 \cdot i, \quad B(\texttt{new}[i]) := 2 \cdot i + 1, \quad B(\texttt{n}[i]) := i + 4, \quad \text{für } 0 \leq i < 2$$

mit dem Diagramm



Um solch eine schlechte Allokation zu erzeugen, wird `bad_accumulator` in Zeile ‚`10`‘ bis Zeile ‚`11`‘ definiert unter Verwendung von `accumulator`. Zusätzlich wird aber eine Allokations Randbedingung angegeben.

```
 11            ...                 {AllocationConstraint new -< n};
```

Diese besagt, daß die μcke versuchen soll, die Variable `n` *hinter* `new` (und damit auch hinter `old`) zu allokieren. Das Ergebnis findet sich in Abbildung 4.3.b.

Diese Ordnungs-Randbedingungen sind auch sinnvoll für die schon betrachtete Berechnung von BDDs für Übergangsrelationen von Prozeßalgebra-Termen. Hier spielen die Aktionen die Rolle der Steuerleitungen (s. /Enders et al., 1993/). Ähnliche Überlegungen wie beim obigen Beispiel finden sich in /Burch et al., 1994, McMillan, 1996/.

In dieser Arbeit soll nicht versucht werden, diese zum Teil nicht formalen Rechtfertigungen für die verschiedenen Randbedingungen vollständig zu beweisen (Für das „Verschränken“

wurde dies in Satz 4.1 an einem Beispiel durchgeführt). Auch in der Literatur gibt es nur sehr wenige Ergebnisse zu diesem Thema „Bestimmung von minimalen Allokationen für boolesche Funktionen“. Dieses Problem an sich ist ja NP-vollständig (vgl. /Meinel und Slobodova, 1994/). Die schon oft zitierte Arbeit /Enders et al., 1993/ stellt hier eine Ausnahme dar und für einfache Zähler wird diese Thema in /Theobald und Meinel, 1996/ untersucht. Stattdessen wird hier festgestellt, daß es bewährte, aus der Praxis stammende Anforderungen an Allokationen gibt. Darunter fallen auf jeden Fall die drei in diesem Abschnitt vorgestellten, „Verschränken“, „Blocken“ und „Ordnung“.

4.3.4 Kombination von Randbedingungen

In den vorigen Unterabschnitten wurden einige grundlegende Randbedingungen an Allokationen betrachtet. Was aber fehlt, ist eine Methode, Allokationsrandbedingungen zu kombinieren, so daß automatisch Allokationen *generiert* werden können, die die gegebenen Randbedingungen erfüllen. Wie dies geschehen kann, wird in diesem Abschnitt an einem Beispiel vorgestellt. Dem nächsten Abschnitt ist es vorbehalten, dies zu formalisieren und den Allokationsalgorithmus des Modellprüfers μcke vorzustellen.

Als Beispiel betrachte man wie auf Seite 77 die Berechnung der transitiven Hülle T^* einer binären Relation T. Dies spielt in abgewandelter Form als Hülle über ε-Übergänge beim Beweis von Sprachinklusion für ω-Automaten oder in ähnlicher Weise bei Bisimulations-Überprüfungen eine wichtige Rolle (vgl. Kapitel 5). Dazu sei folgende Definition im μ-Kalkül $\mathbb{B}_\mu^V$ gegeben (vgl. S. 77):

$$\mu T^*(s,t)\;.\; s \doteq t \;\vee\; \exists\, i.\, T(s,i) \wedge T^*(i,t),$$

Die grundlegende Idee besteht darin, wie bei der Berechnung der Semantik eine Fixpunktiteration durchzuführen. Jetzt jedoch nicht über dem Verband der Prädikate, sondern über Allokationsrandbedingungen. Am Anfang wird für T^* die kleinste Randbedingung angenommen, die überhaupt keine Einschränkungen macht. Damit geht man in den Rumpf von T^*. Hier findet man in der ersten Iteration die Randbedingung an s und i, die durch $T(s,i)$ gegeben ist. Nun sei keine Heuristik benutzt worden und auch der Benutzer habe keine Randbedingungen zur Allokation der Parameter von T angegeben. Damit besteht auch keine Einschränkung an s und i. Aber der Term $s \doteq t$ erzeugt nach Satz 4.1 die Randbedingung, daß s und t verschränkt werden sollen. Das Ergebnis der ersten Iteration ist also gerade diese letzte Randbedingung.

Mit dieser geht man nun wieder bei der zweiten Iteration in den Rumpf von T^*. Nach Substitution erhält man innerhalb des Quantors die Randbedingung, daß t und i verschränkt sein sollten. Hier stellt sich nun zum erstenmal die Frage, wie die zwei Randbedingungen „s verschränkt mit t“ und „t verschränkt mit i“ kombiniert werden können. Nimmt man nun an (dies die erste Vereinfachung, die man auf der Suche nach einer guten Allokation trifft), daß nur solche Allokationen betrachtet werden, bei denen „Verschränken“ transitiv ist, dann erhält man zusätzlich die Randbedingung, daß auch s mit i verschränkt sein sollte. Eine weitere Iteration bestätigt, daß damit auch schon ein Fixpunkt erreicht ist.

Dieser minimale Fixpunkt läßt sich zu einer maximalen Randbedingung erweitern, indem nacheinander die fehlenden Angaben wiederum durch einen Fixpunktprozeß ergänzt werden. Zum Beispiel kann man festlegen, daß s vor t beginnt. Nach einer weiteren Fixpunktiteration erhält man damit zusätzlich „i beginnt vor t,“. Jetzt bleibt nur noch die Lage von s relativ zu i festzulegen. Wählt man nun zum Beispiel „s beginnt vor i“, so ist man zu einer maximalen Randbedingung gelangt, aus der sich folgende Allokation ablesen läßt:

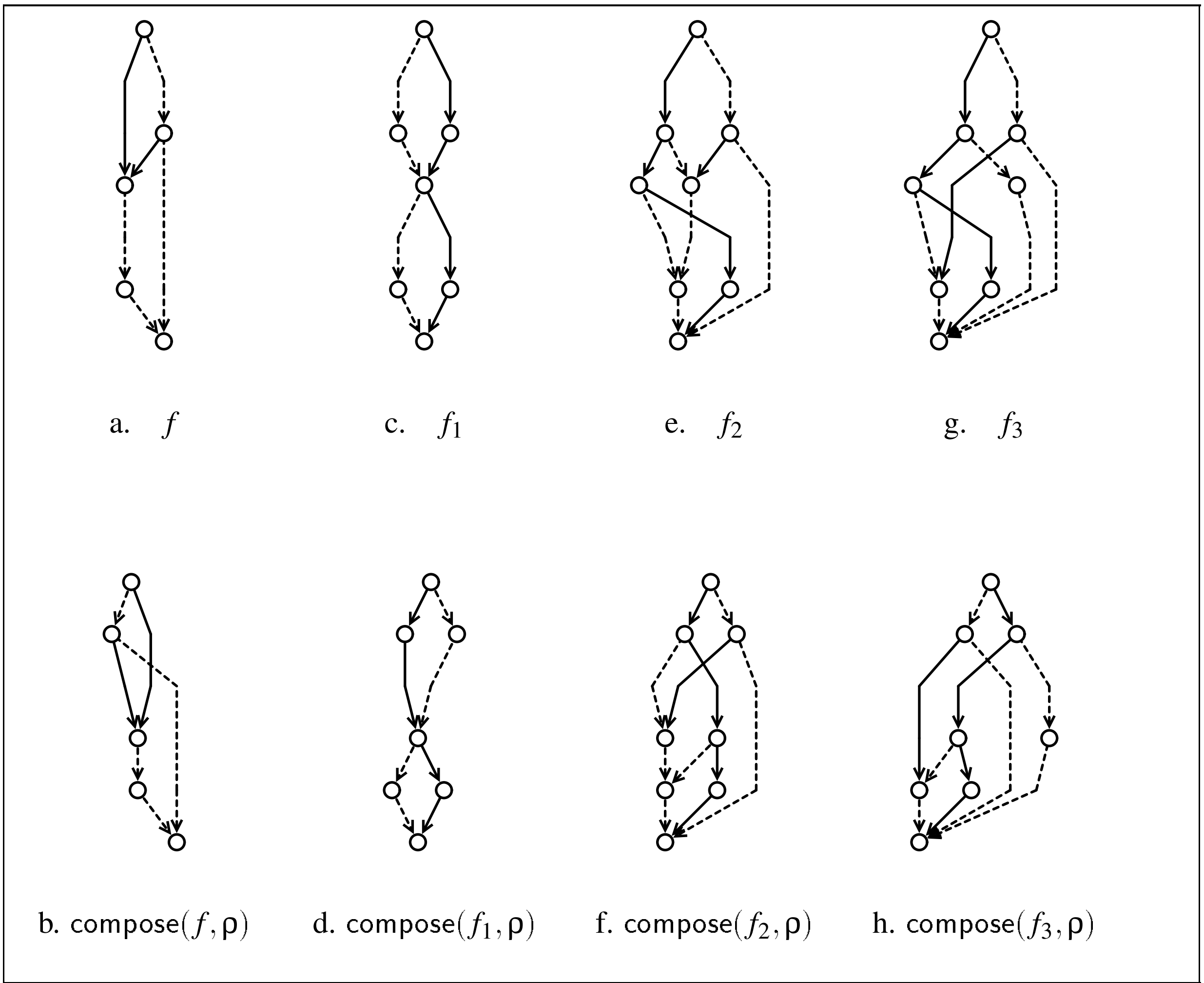
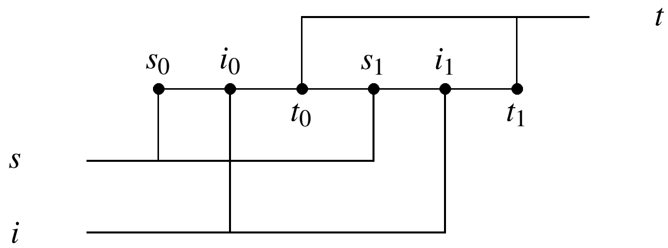


Abbildung 4.4: Berechnung der BDD-Semantik der transitiven Hülle des Beispiels von S. 18 zur *guten* Allokation von S. 127.



Man vergleiche dies mit der Allokation von Seite 77 und der anschließenden Berechnung der BDD-Semantik von T^*. Wie in Abb. 3.18 ist in Abb 4.4 die Berechnung der BDD-Semantik für die gerade bestimmte Allokation dargestellt. Mit dieser Allokation erreicht man, daß bei Berechnung der BDD-Semantik die Substitution, die man in der zweiten Iteration auf den BDD für $s \doteq t$ ausführen muß, keinen exponentiell größeren BDD erzeugt, wie das bei der Allokation von Seite 77 der Fall war. Die auftretenden Substitutionen sind hier alle montonon, so daß sich die Struktur der BDDs und somit auch ihre Größe nicht ändert (vgl. S. 92).

Dieselbe Argumentation trifft zu, wenn man nur die transitive Hülle einer Übergangsrelation berechnet und nicht die reflexiv-transitive, wie das implizit angenommen wurde. Hierbei nimmt man meistens an, daß sich der Zustand bei einem Übergang nur lokal ändert, die Übergangsrelation also „ähnlich“ zur Gleichheit ist. Man würde dann die Definition

$$\mu T^{+}(s,t)\ .\ T(s,t)\ \vee\ \exists i.\, T(s,i) \wedge T^{+}(i,t),$$

verwenden, dabei aber die Randbedingung „ u ist verschränkt mit v “ für die formalen Parameter $(\pi(T))$ u und v von T angeben. Mit einer ähnlichen Analyse würde man dann zu derselben Allokation gelangen.

Zum Schluß sei noch erwähnt, daß nach Festlegung der Allokation für ein Prädikat X, für alle Prädikate in $\mathsf{dep}(X)$ möglicherweise eine Randbedingung erzeugt werden muß. Zum Beispiel bei der obigen Wahl der Allokation für T^* ergibt sich für die Parameter $(\pi(T))$ u und v von T, daß u vor v beginnen muß. Hätte man oben die Festlegung getroffen, daß i vor s beginnt, dann wäre die Ordnung von v und u gerade umgekehrt.

4.4 Formalisierung

Der letzte Abschnitt hat drei wichtige mögliche Randbedingungen an Allokationen aufgezeigt. Für die formale Behandlung werden hier nur diese drei betrachtet. Des weiteren ist zur Vereinfachung nur eine eingeschränkte Klasse von Randbedingungen erlaubt. Das sind solche, die sich in disjunkte Blöcke zerlegen lassen, in denen alle Variablen verschränkt sind. In der Praxis traten noch keine Randbedingungen auf, die allgemeinere Allokationen verlangt hätten.

4.4.1 Verband der Allokationsrandbedingungen $\mathbb{C}$

Zunächst werden Eigenschaften von Allokationen untersucht. Die danach betrachteten Allokationsrandbedingungen aus $\mathbb{C}$ sind Aproximationen an diese Eigenschaften. Diese bilden einen Verband der den Durchschnittshalbverband $\mathbb{C}_1$ beinhaltet, der wiederum als eine Teilmenge von $\mathbb{C}_0$ definiert wird.

Allokationen

Eine Allokation für den μ-Kalkül ist die Hintereinanderausführung einer Allokation für boolesche Variable aus Def. 3.10 nach einer Komponentenabbildung aus Def. 2.39. Dies wurde schon in Abschnitt 3.7 für die Beispiele verwendet und wird hier nur nocheinmal genauer definiert.

Definition 4.2 (Allokationen für $\mathbb{B}_\mu^V$) *Eine Allokation $A \in \mathcal{A}$ interpretiert für $\mathbb{B}_\mu^V$ unter einer fest gewählten Komponentenabbildung ist eine injektive, partielle Funktion*

$$A\colon V \times \mathbb{N} \to \mathbb{N}, \quad A(u,i) := A(u[i])$$

Man schreibe aber auch weiterhin nur $A(u[i])$ statt $A(u,i)$.

Bei fest gewählter Komponentenabbildung sind Allokationen für $\mathbb{B}_\mu^V$ und $\mathbb{B}^W$ wegen der Bijektivität einer Komponententenabbildung eindeutig einander zugeordnet. Des weiteren werden in diesem Abschnitt keine booleschen Variablen mehr benötigt, so daß mit einer Allokation $A \in \mathcal{A}$ immer gleich die Allokation für $\mathbb{B}_\mu^V$ gemeint sein soll.

Soweit wurde nichts neues hinzugefügt. Nur die Schreibweise wurde formal fixiert. Um die Arbeit mit Allokationen zu vereinfachen, kann man die Freiheit der beliebigen Wahl einer Komponentenabbildung ausnützen, und fordert ab sofort, daß alle Allokationen *monoton* sind:

Definition 4.3 (Monotone Allokation) *Eine Allokation $A \in \mathcal{A}$ heißt* monoton *gdw. für alle $x \in V$ und alle $i, j \in \mathbb{N}$ mit $0 \le i \le j < |x|$ auch $A(x[i]) \le A(x[j])$.*

Diese Bedingung an Allokationen wirkt um so natürlicher, wenn man sich klar macht, daß für beliebige endliche Domänen sowieso erst eine boolesche Kodierung erforderlich ist.

Für den Rest des Abschnittes seien alle Allokationen monoton.

Nun kann man daran gehen zu definieren, was es bedeutet, daß in einer Allokation zwei Variablen verschränkt oder geblockt sind, oder eine vor der anderen beginnt.

Definition 4.4 *Für $A \in \mathcal{A}$, $x, y \in V$ definiere für $m := \mathsf{min}\{|x|, |y|\} - 1$*

$$\begin{aligned} x\ ]|_A\ y \quad &:\Leftrightarrow \quad A(x[0]) < A(y[0]) < \cdots < A(x[m]) < A(y[m]) \\ x \lhd_A y \quad &:\Leftrightarrow \quad A(x[|x|-1]) < A(y[0]) \\ x \le_A y \quad &:\Leftrightarrow \quad A(x[0]) \le A(y[0]) \end{aligned}$$

Für die ersten beiden definiere jeweils den symmetrischen Abschluß

$$||_A \quad := \quad]|_A \cup |[_A \qquad\qquad \Box_A \quad := \quad \lhd_A \cup \rhd_A$$

mit $]|_A, \lhd_A, \le_A, ||_A, \Box_A \subseteq V^2$.

Damit steht „ $||_A$ “ für das Verschränken zweier Variablen und „ $\Box_A$ “ für das Blocken. Die Relation „ $\le_A$ “ ordnet Variablen. Das folgende Resultat, das eine erste unmittelbare Folgerung aus der Monotonie von Allokationen und der letzten Definition ist, wird in den weiteren Beweisen immer wieder benötigt.

Lemma 4.5 *Sei A eine Allokation und $x, y \in V$ mit $x\ ||_A\ y$, dann gilt*

$$A(x[i]) < A(y[j]) \quad \textit{für} \quad i < j$$

und $i, j \in \mathbb{N}$, mit $0 \le i < |x| - 1$, $0 \le j < |y|$.

Definition 4.6 *Eine Allokation $A \in \mathcal{A}$ heiße*

$$\begin{aligned} \textit{transitiv} \quad &:\Leftrightarrow \quad ||_A \ \textit{transitiv} \\ \textit{überdeckend} \quad &:\Leftrightarrow \quad ||_A \cup \Box_A = V^2 \\ k\textit{-unverschränkt} \quad &:\Leftrightarrow \quad \textit{für } x\ ||_A\ y \textit{ gilt } |x|, |y| \ge k \end{aligned}$$

für $k \in \mathbb{N}$.

Später wird die Menge der „erlaubten" Allokationen auf transitive, überdeckende und 2-unverschränkte eingeschränkt. Dies ist aber nur für die Vollständigkeit des Verfahrens wichtig. Das heißt, findet das Verfahren keine gute Allokation, dann gibt es keine gute Allokation *mit den genannten Einschränkungen*.

Die einzige wesentliche Einschränkung ist „überdeckend". Sie bedeutet, daß nur solche Allokationen betrachtet werden, bei denen es keine zwei Variablen gibt, die weder verschränkt noch geblockt sind. Neben technischen Gründen läßt sich dies dadurch rechtfertigen, daß in dieser Arbeit nur die obigen drei Arten von Randbedingungen betrachtet werden, wobei „Ordnung" orthogonal zu „Blocken" und „Verschränken" behandelt wird. Für weitere Eigenschaften von Allokationen wäre es besser eine neue Randbedingungsart einzuführen.

Die Forderung der k-Unverschränkheit ist für kleine k ($k = 2,3$) dadurch zu motivieren, daß für so kleine Vektoren der Unterschied zwischen geblockter und verschränkter Allokation nur minimal ist. Zweitens kann man Vektoren der Länge kleiner k einfach als Vektoren der Länge k durch Hinzufügen von Hilfskomponenten interpretieren.[2]

Des weiteren scheint „transitiv" ein sehr starke Einschränkung zu sein. Der nächste Satz zeigt aber, daß 3-unverschränkte Allokationen automatisch transitiv sind.

Satz 4.7 *Eine* 3*-unverschränkte, überdeckende Allokation ist transitiv.*

Beweis: Sei $A \in \mathcal{A}$ eine 3-unverschränkte, überdeckende Allokation und man nehme an, daß A nicht transitiv ist. Dann gibt es Variablen $x, y, z \in V$ mit $x \mid\mid_A y \mid\mid_A z$ und $x \not\mid\mid_A z$. Da A überdeckend ist, erhält man dafür $x \Box_A z$. Damit gilt

$$A(x[2]) \geq A(y[1]) \geq A(z[0]) \qquad A(z[2]) \geq A(y[1]) \geq A(x[0]),$$

was sowohl im Falle $x \lhd_A z$ als auch für $x \rhd_A z$ zu einem Widerspruch zur Definition von „ $\lhd_A$ " führt. ❑

Die nächste Definition dient dazu, in Satz 4.9 den Begriff „überdeckend" noch etwas näher zu erläutern.

Definition 4.8 *Für $A \in \mathcal{A}$, $x \in V$ definiere*

$$\begin{aligned} A(x) &:= \{A(x[0]), \ldots, A(x[|x|-1])\} \\ [A(x)] &:= \{i \in \mathbb{N} \mid \min A(x) \leq i \leq \max A(x)\} \\ &= \{i \in \mathbb{N} \mid A(x[0]) \leq i \leq A(x[|x|-1])\} \end{aligned}$$

(Man beachte, daß A wie oben festgelegt monoton ist).

Nun kann man zeigen, daß die Einschränkung der „Überdeckung" damit äquivalent ist, daß sich die Hüllen über die Bilder der Komponenten zweier Variablen genau dann schneiden, wenn sie verschränkt sind.

[2] Der Allokationsalgorithmus in der μcke behandelt alle Variablen unabhängig von ihrer Länge gleich.

Satz 4.9 *Sei $A \in \mathcal{A}$ eine 2-unverschränkte Allokation, dann gilt*

$$(\textit{ für alle } x,y \in V \textit{ gilt} \qquad [A(x)] \cap [A(y)] \neq \emptyset \quad \Leftrightarrow \quad x \,||_A\, y\ )$$
$$\textit{gdw.}$$
$$||_A \cup \Box_A = V^2$$

Für transitive Allokationen ist $||_A$ eine Äquivalenzrelation. Ist eine Allokation sogar noch 2-unverschränkt, dann besitzt sie die schöne algebraische Eigenschaft, daß $\Box_A$ eine Kongruenzrelation bez. $||_A$ ist. Um das zu zeigen benötigt man die Disjunktheit dieser beiden Relationen.

Lemma 4.10 *Sei $A \in \mathcal{A}$ 2-unverschränkt, dann gilt $||_A \cap \Box_A = \emptyset$.*

Beweis: Angenommen es gäbe $x,y \in V$ mit $x \,||_A\, y$ und $x \Box_A y$. O.B.d.A. ($||_A$ symmetrisch) gelte $x \lhd_A y$, und man erhält den Widerspruch

$$A(y[0]) \leq A(x[1]) \leq A(x[|x|-1]) < A(y[0])$$

❑

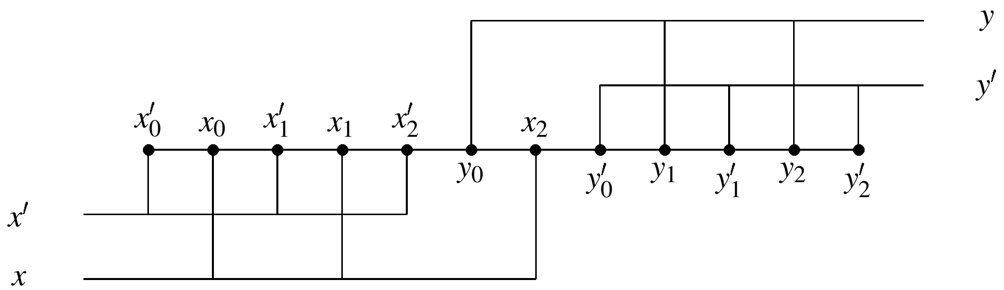
Satz 4.11 *Für eine 2-unverschränkte, überdeckende, transitive Allokation $A \in \mathcal{A}$ gilt*

$$||_A \Box_A ||_A \ \subseteq \ \Box_A,$$

Hierbei ist die Schreibweise aus Abschnitt A.2 verwendet worden.

Beweis: Für $x,y,y',x' \in V$ mit $x \,||_A\, x' \Box_A y' \,||_A\, y$ nehme $x \not\Box_A y$ an. Dann ergibt die Überdeckungseigenschaft von A statt dessen $x \,||_A\, y$. Nun ist $||_A$ transitiv, so daß $x' \,||_A\, y'$ folgt, im Widerspruch zu Lemma 4.10. ❑

Als Beispiel betrachte folgende Allokation, die zwar transitiv, 3-unverschränkt, aber nicht überdeckend ist, und auch obige Kongruenzeigenschaft nicht erfüllt



Es gilt in dieser Allokation A zwar $x \,||_A\, x' \Box_A y' \,||_A\, y$, aber nicht $x \Box_A y$. Die Aussage von Satz 4.9 ist auch nicht gültig, da $A(y[0]) \in [A(x)] \cap [A(y)]$, obwohl $x \not||_A\, y$.

Die Relation „$\leq_A$“ ist eine lineare Ordnung und erfüllt auf den Klassen von „$||_A$“, unter bestimmten Bedingungen an die Allokation, eine Kongruenzeigenschaft.

Satz 4.12 *Für eine 2-unverschränkte, überdeckende, transitive Allokation $A \in \mathcal{A}$ gilt*

$$||_A(\Box_A \cap \leq_A)||_A \;\subseteq\; \leq_A,$$

Beweis: Für $x, y, y', x' \in V$ mit $x \,||_A\, x' \,\Box_A\, y' \,||_A\, y$ und $x' \leq_A y'$ zeigt man

$$A(x[0]) \leq A(x'[1]) \leq A(x'[|x'|-1]) < A(y'[0]) \leq A(y[1]) \tag{4.1}$$

Satz 4.11 liefert $x \,\Box_A\, y$. Im Falle $x \rhd_A y$ erhält man $A(x[0]) \geq A(y[|y|-1]) \geq A(y[1])$ im Widerspruch zur Ungleichung (4.1). Der Fall $x \lhd_A y$ ergibt $A(x[0]) \leq A(x[|x|-1]) \leq A(y[0])$ und man ist fertig. ❑

Randbedingungen

Allokationsrandbedingungen sind Approximationen an eine Allokation. Die Hauptidee besteht nun darin, die drei Relationen $||_A$, $\Box_A$ und $\leq_A$ zu approximieren statt der Allokation selbst. Damit erhält man ein syntaktisches Verfahren, daß nur mit binären Relationen über V arbeitet.

Da hier nun die Ordnung von x und y („x beginnt vor y"), je nachdem ob x und y verschränkt oder geblockt allokiert werden soll, verschieden behandelt werden muß, wird die $\leq_A$ entsprechende Relation in zwei partielle Ordnungen aufgeteilt. Die Allokationsrandbedingungen stammen also aus $\mathbb{C}_0$, definiert in

Definition 4.13 *Für die Elemente $c \in \mathbb{C}_0 := \mathbb{P}(V^2)^4$ führe die Schreibweise ein*

$$(\sim_c^+, \sim_c^-, \leq_c^+, \leq_c^-) \;:=\; c$$

oder einfach $(\sim^+, \sim^-, \leq^+, \leq^-)$, falls eindeutig.

Somit ist eine Randbedingung c aus der *Menge* $\mathbb{C}_0$ ein Quadrupel aus binären Relationen über den Variablen V von $\mathbb{B}_\mu^V$. Eine Allokation „erfüllt" eine solche Randbedingung, wenn die Randbedingung eine Approximation an die Allokation ist.

Definition 4.14 *Eine Allokation $A \in \mathcal{A}$ erfüllt ein $c \in \mathbb{C}_0 \cup \{\top\}$, in Zeichen $A \models c$, gdw. $\sim_c^+ \subseteq ||_A$, $\sim_c^- \subseteq \Box_A$ und $\leq_c^+, \leq_c^- \subseteq \leq_A$.*

Zunächst läßt sich auf $\mathbb{C}_0$ (und damit später auch auf $\mathbb{C}_1$ als Einschränkung von $\mathbb{C}_0$ und auf eine partielle Ordnung einführen.

Definition 4.15 *Auf $\mathbb{C}_0$ definiere die partielle Ordnung $\sqsubseteq$ durch*

$$c \sqsubseteq d \quad :\Leftrightarrow \quad \sim_c^+ \subseteq \sim_d^+, \quad \sim_c^- \subseteq \sim_d^-, \quad \leq_c^+ \subseteq \leq_d^+, \quad \leq_c^- \subseteq \leq_d^-$$

für $c, d \in \mathbb{C}_0$.

Nun kann man die in den obigen Sätzen bewiesenen Eigenschaften von Allokationen ausnützen, um folgendes Lemma zu zeigen.

Lemma 4.16 *Für eine 2-unverschränkte, transitive und überdeckende Allokation $A \in \mathcal{A}$ gibt es ein $e \in \mathbb{C}_0$, so daß $A \models e$ und für alle $e' \in \mathbb{C}_0$ mit $A \models e'$ auch $e \sqsupseteq e'$ gilt.*

Beweis: Wähle $e := (\sim^+, \sim^-, \leq^+, \leq^-)$, mit $\sim^+ := ||_A$, $\sim^- := \Box_A$, $\leq^+ := \leq_A \cap ||_A$ und $\leq^- := \leq_A \cap \Box_A$. Mit Lemma 4.10 und den Sätzen 4.11 und 4.12 zeigt man, daß e die Eigenschaften e., h. und i. erfüllt. Die Gültigkeit der Eigenschaften a.-d., f. und g. folgt unmittelbar, womit $e \in \mathbb{C}_1$. Per definitionem gilt $A \models e$. Für den Rest bemühe man die Definition von „$\models$“ und die Eigenschaften f. und g. ❑

Das nächste Lemma stellt einen Zusammenhang zwischen der „Erfüllbarkeits“-Relation „$\models$“ und „$\sqsubseteq$“. her. Es sagt aus, daß „$\models$“ bez. Abschwächung von Randbedingungen invariant ist.

Lemma 4.17 *Für $c, d \in \mathbb{C}_0$, $A \in \mathcal{A}$ mit $A \models c$ und $c \sqsupseteq d$ gilt $A \models d$.*

Nun kann man für solche Approximationen die Forderung der Überdeckung fallen lassen und braucht auch keine lineare Ordnung anzunehmen. Ansonsten sollen die obigen Sätze und das Lemma auch für die Approximation gelten und die Elemente von $\mathbb{C}_0$ müssen entsprechend eingeschränkt werden.

Definition 4.18 *Die Menge der (erfüllbaren)* Allokationsrandbedingungen $\mathbb{C}_1$ *ist definiert als die größte Menge $\mathbb{C}_1 \subseteq \mathbb{C}_0$ so daß für alle $c \in \mathbb{C}_1$ gilt*

a.	$\sim^+$ *ist eine Äquivalenzrelation*	*e.*	$\sim^+ \cap \sim^- = \emptyset$
b.	$\sim^-$ *ist symmetrisch und irreflexiv*	*f.*	$\leq^+ \subseteq \sim^+$
c.	$\leq^+$ *ist eine partielle Ordnung*	*g.*	$<^- \subseteq \sim^-$
d.	$\leq^-$ *ist eine partielle Ordnung*	*h.*	$\sim^+\sim^-\sim^+ \subseteq \sim^-$
		i.	$\sim^+<^-\sim^+ \subseteq \leq^-$

Hierbei ist für h. und i. die Schreibweise aus Abschnitt A.2 verwendet worden.

Satz 4.19 $(\mathbb{C}_1, \sqsubseteq)$ *ist ein vollständiger Durchschnittshalbverband.*

Beweis: Die Eigenschaften a., c., d., f.-i. und die Symmetrie von $\sim^-$ bleiben unter Schnittbildung erhalten (sie lassen sich als Hornklauseln formulieren). Die Erhaltung der Irreflexivität von $\sim^-$ und die Eigenschaft e. folgen unmittelbar. Damit ist

$$c \sqcap d := (\ \sim_c^+ \cap \sim_d^+\ ,\ \sim_c^- \cap \sim_d^-\ ,\ \leq_c^+ \cap \leq_d^+\ ,\ \leq_c^- \cap \leq_d^-\ ) \in \mathbb{C}_1$$

ein eindeutiges Infimum. Dies und die Vollständigkeit ergibt sich daraus, daß $\mathbb{P}(V^2)$ und somit auch $\mathbb{P}(V^2)^4$ als Mengen bezüglich (komponentenweiser) Mengeninklusion vollständige Verbände sind. ❑

Im weiteren wird es darum gehen, einzelne Randbedingungen zu einer zu vereinigen. Hierzu reicht die mengentheoretische Vereinigung nicht aus. Schon um zwei Äquivalenzrelationen zu vereinigen, muß man eine reflexive-transitive Hülle bilden. Auch die anderen Forderungen müssen durch eine Hüllenbildung gesichert werden. Deshalb stellt sich die Frage, ob überhaupt immer ein Supremum existiert. Da z. B. zwei Randbedingungen, von denen die eine fordert, daß zwei Variablen verschränkt werden, und die andere, daß sie geblockt werden, unvereinbar sind, kann nur durch Hinzunahme einer „unerfüllbaren“ Randbedingung ($\top$) als Maximum aller Randbedingungen gehofft werden, diese Frage mit ja zu beantworten. Dann zeigt aber auch schon Satz A.18, daß man dadurch einen vollständigen Verband erhält.

Definition 4.20 *Sei $\mathbb{C} := \mathbb{C}_1 \dot{\cup} \{\top\}$ die Vervollständigung von $\mathbb{C}_1$ nach Satz A.18.*

Die Erfüllbarkeitsrelation von Randbedingungen wird direkt von $\mathbb{C}_1$ übernommen. Zusätzlich gelte $A \not\models \top$ für alle Allokationen A.

Wie in Satz A.18 wird auch die Ordnung zwischen Elementen aus $\mathbb{C}_1$ übernommen und nur zusätzlich $\top$ als größtes Element definiert. Bei der Vereinigung „$\sqcup$“ zweier Randbedingungen nach Satz A.18

$$c \sqcup d \;:=\; \bigcap\{e \in \mathbb{C}_1 \mid c \sqsubseteq e \text{ und } d \sqsubseteq e\}$$

muß einmal gewährleistet sein, daß jede die Vereinigung erfüllende Allokation auch die einzelnen Randbedingungen erfüllt. Umgekehrt sollte die Vereinigung auch nicht zu einschränkend sein, damit keine Allokationen verloren gehen, die zwar die einzelnen Randbedingungen erfüllen, aber nicht mehr die Vereinigung.

Satz 4.21
Für eine 2-unverschränkte, transitive und überdeckende Allokation $A \in \mathcal{A}$ und $c, d \in \mathbb{C}$ gilt

$$A \models c \sqcup d \qquad \Leftrightarrow \qquad A \models c \quad \textit{und} \quad A \models d$$

(Die Richtung von links nach rechts gilt für beliebige Allokationen).

Beweis: Lemma 4.17 ergibt die Richtung von links nach rechts. Für die Richtung von rechts nach links erhält man aus Lemma 4.16 ein $e \in \mathbb{C}$ mit $e \sqsupseteq c, d$ und $A \models e$. Die Supremumseigenschaft von $c \sqcup d$ ergibt nun $e \sqsupseteq c \sqcup d$ und wiederum Lemma 4.17 liefert den Rest. ❑

Der Beweis zu Satz A.18 liefert nur ein sehr aufwendiges Verfahren zur Berechnung der Vereinigung „$\sqcup$“. Dazu müßten alle möglichen Randbedingungen generiert werden. Von denen werden dann die herausgesucht, die über den beiden zu vereinigenden Randbedingungen liegen. Über all diesen wird nun ein Schnitt gebildet. Die erste Phase dieses Algorithmus ist schon exponentiell. Deshalb wird hier eine direktere Methode vorgestellt, die nur polynomialen Aufwand benötigt.[3]

[3] Diese Komplexitätsaussagen sind natürlich nur bei endlich vielen Variablen sinnvoll.

Satz 4.22 (Berechnung von „$\sqcup$“) *Zu $c,d \in \mathbb{C}_1$ definiere für $i \in \mathbb{N}$*

$$
\begin{array}{lcllcl}
\sim^+ & := & (\sim_c^+ \cup \sim_d^+)^* & \leq^+ & := & (\leq_c^+ \cup \leq_d^+)^* \\
\leq_0^- & := & (\leq_c^- \cup \leq_d^-)^* & \leq_{i+1}^- & := & (\leq_i^- \cup \sim^+ <_i^- \sim^+)^* \\
\leq^- & := & \bigcup_{i=0}^{\infty} \leq_i^-, & \sim^- & := & \sim^+(\sim_c^- \cup \sim_d^- \cup \leq^- \cup \geq^-)\sim^+
\end{array}
$$

und $e := (\sim^+, \sim^-, \leq^+, \leq^-)$ dann ist

$$
c \sqcup d \quad = \quad \begin{cases} e & \text{falls } e \in \mathbb{C}_1 \\ \top & \text{falls } e \notin \mathbb{C}_1 \end{cases}
$$

Beweis: Für den Beweis zeige die Hilfsbehauptung, daß für alle $e' \in \mathbb{C}_1$, mit $e' \sqsupseteq c,d$, folgende Ungleichungen gelten

$$
\begin{array}{c}
\sim_{e'}^+ \ \supseteq \ \sim^+, \qquad \sim_{e'}^- \ \supseteq \ \sim^-, \qquad \leq_{e'}^+ \ \supseteq \ \leq^+, \\
\leq_{e'}^- \ \supseteq \ \leq^- \ \supseteq \ \cdots \ \supseteq \ \leq_1^- \ \supseteq \ \leq_0^- \ \supseteq \ \leq_c^-, \leq_d^-
\end{array}
\tag{4.2}
$$

Die erste Zeile folgt unmittelbar aus der Definition und den Eigenschaften von e' (verwende Zeile zwei). Für die zweite Zeile führe man den Beweis durch Induktion von rechts nach links. Damit erhält man unter der Voraussetzung der Hilfsbehauptung $e' \sqsupseteq e$. Damit ist e ein Supremum für c und d in $\mathbb{C}_1$, falls $e \in \mathbb{C}_1$. Wenn $e \notin \mathbb{C}_1$, dann gilt trotzdem, daß $\sim^-$ symmetrisch ist, daß $\leq^+$ und $\leq^-$ transitiv und reflexiv sind, und, daß e die Eigenschaften a. und f. bis i. erfüllt. D. h. entweder ist $\sim^-$ nicht irreflexiv, $\leq^+$ oder $\leq^-$ ist nicht antisymmetrisch oder e erfüllt Eigenschaft e. nicht. Dies überträgt sich aber für jedes $e' \sqsupseteq e$, so daß es kein $e' \in \mathbb{C}_1$ geben kann mit $e \sqsupseteq c,d$. Dies bedeutet $c \sqcup d = \top$. ❑

Zur Bestimmung des Aufwandes zur Berechnung von $c \sqcup d$ auf diese Weise nehme man an, daß die binären Relationen über V als Bitmatrix gespeichert sind. Vereinigung und Schnitt benötigen dann $O(n^2)$ für $n := |V| < \infty$. Ein Relationenprodukt wird als Matrixmultiplikation implementiert und kann so mit $O(n^3)$ Operationen berechnet werden. Wie im μ-Kalkül (s. S. 77) läßt sich die transitive Hülle (*) als Fixpunkt berechnen. Dabei erhält man eine aufsteigende Folge von Approximationen an die transitive Hülle. Eine solche aufsteigende Folge von Matrizen kann höchstens die Länge n^2 haben. Damit läßt sich für die transitive Hülle zeigen, daß höchstens n^2 mal ein Relationenprodukt und eine Vereinigung durchgeführt werden muß, was $O(n^5)$ als obere Schranke ergibt. Der Aufwand für die Bestimmung des Relationenproduktes übertrifft damit den Aufwand für die anderen Operationen und man erhält $O(n^5)$ als obere Schranke für den Aufwand in einem Iterationsschritt. Mit einer ähnlichen Argumentation zeigt man, daß die aufsteigende Folge in der zweiten Zeile nach höchstens n^2+1 Iterationen fix wird. Dies ergibt für den Gesamtaufwand für diese Art der Berechnung von $c \sqcup d$ eine obere Schranke von $O(n^7)$.

Nun kommt ein wesentlicher Gesichtspunkt dieser Vorgehensweise. Eine Randbedingung sollte erfüllbar sein, d. h. es gibt eine sie erfüllende Allokation, solange die Rangbedingung in $\mathbb{C}_1$ liegt. Damit dies gewährleistet ist, mußten die Einschränkungen a. bis i. getroffen werden.

Satz 4.23 *Sei $|V| < \infty$, so gibt es für alle c eine Allokation $A \in \mathcal{A}$, mit $A \models c$.*

Beweis: Sei mit $[x]_+$ die Äquivalenzklasse von x bez. $\sim^+$ bezeichnet. Wähle Repräsentanten $x_1, \dots, x_n \in V$, mit

$$\bigcup_{i=1}^{n} [x_i]_+ \;=\; V \quad \text{und} \quad x_i \not\sim^+ x_j \text{ für } i \neq j$$

Auf $V/\sim^+$ definiere nun die Relation

$$[x]_+ \leq [y]_+ \quad :\Leftrightarrow \quad x \sim^+ y \quad \text{oder} \quad x \leq^- y$$

Diese Relation ist repräsentantenunabhängig wegen i. und der Transitivität von $\sim^+$. Mit denselben Argumenten und der Transitivität von $\leq^-$ erweist sie sich auch als transitiv und antisymmetrisch, also als partielle Ordnung. Damit kann man nach Satz A.5 o.B.d.A. die x_i so wählen, daß für $[x_i]_+ \leq [x_j]+$ sich $i \leq j$ ergibt. Nun wähle mit demselben Argument $y_{i,j} \in V$ so, daß

$$[x_i]_+ = \{y_{i,1}, \dots, y_{i,a_i}\}, \quad \text{mit} \quad a_i := |[x_i]_+| \quad \text{und} \quad j_1 \leq j_2 \quad \text{für} \quad y_{i,j_1} \leq^+ y_{i,j_2}$$

Mit der Abkürzung $l_i := a_i \cdot \mathsf{max}\{|y| \mid y \sim^+ x_i\}$ definiere $A \in \mathcal{A}$ durch

$$A(y_{i,j}[k]) \quad := \quad (j-1) + k \cdot a_i + \sum_{m=1}^{i-1} l_m$$

für $1 \leq i \leq n$, $1 \leq j \leq a_i$ und $0 \leq k < |y_{i,j}|$. Nun rechnet man nach, daß $||_A = \sim^+$, $\Box_A \supseteq \sim^-$, $\leq_A \supseteq \leq^+$ und $\leq_A \supseteq \leq^-$. ❑

Dieser konstruktive Beweis liefert das Verfahren, das in der μcke verwendet wird, um aus *einer* Randbedingung eine Allokation zu erzeugen. In Worten werden dabei alle Variablen, die verschränkt werden sollen, in „Blöcke“ gepackt. Diese Blöcke werden untereinander nach $\leq^-$ topologisch sortiert und die Variablen innerhalb den Blöcken nach $\leq^+$.

Der Sortieralgorithmus sollte dabei eine initiale Anordnung[4] der Variablen wenn möglich beibehalten. Dies gilt auch für die Blöcke. Hier hat sich folgendes Verfahren bewährt. Jedem Block wird der Mittelwert aller initialen Positionen seiner Variablen zugeordnet und mit diesem Wert als Vergleichswert eine initiale Position (von Blöcken) bestimmt, bevor der Sortieralgorithmus gestartet wird.

4.4.2 Allokationsalgorithmus

Um erläutern zu können, wie nun die Gewinnung von Allokationen realisiert werden kann, fehlen noch zwei Operationen auf Randbedingungen. Das ist die Projektion einer Randbedingung auf eine Variablenmenge und die Substitution von Randbedingungen. Beide lassen sich mengentheoretisch definieren.

Definition 4.24 *Zu $c \in \mathbb{C}_1$, $V_0 \subseteq V$, $\lambda \in [V \to V]$ definiere*

$$\begin{aligned} c|_{V_0} \quad &:= \quad (\,(V_0^2 \cap \sim_c^+) \cup \mathsf{id}\,,\, V_0^2 \cap \sim_c^-\,,\, (V_0^2 \cap \leq_c^+) \cup \mathsf{id}\,,\, (V_0^2 \cap \leq_c^-) \cup \mathsf{id}\,) \\ \lambda(c) \quad &:= \quad (\lambda(\sim_c^+), \lambda(\sim_c^-), \lambda(\leq_c^+), \lambda(\leq_c^-)) \end{aligned}$$

mit $\lambda(R) := \{(\lambda'(x), \lambda'(y)) \mid (x,y) \in R\}$ für $R \subseteq V^2$ und λ' die totale Erweiterung von λ (wie in Def. 2.4). Man verwende auch für Substitutionen die Schreibweise $\lambda(c) = c\{x \mapsto \lambda(x) \mid x \in V\}$.

[4] Zum Beispiel die textuelle Reihenfolge von Variablen in einer Parameterliste.

Offensichtlich ist $c|_{V_0} \in \mathbb{C}_1$. Für Substitutionen kann das Ergebnis nicht mehr in $\mathbb{C}_1$ liegen. In diesem Fall lege einfach fest, daß das Ergebnis $\top$ ist. Damit kann man nun entlang der Termstruktur von Termen aus T_μ^V Allokationsrandbedingungen propagieren. Die Basisrandbedingungen stammen entweder vom Benutzer (vgl. Abschnitt B.3.2), werden für spezielle Terme (zum Beispiel „$\doteq$") oder durch Heuristiken generiert. Als Heuristik hat sich hier die ASTI-Heuristik als sehr nützlich herausgestellt. „ASTI" steht für „allocate same type interleaved". Hierbei wird für zwei Variablen in einer Paramterliste einer Prädikatsvariablendefinition in der Eingabesprache der μcke eine „Verschränkheits"-Randbedingung generiert, wenn die beiden Variablen denselben Typ besitzen.

Bei binären Operatoren verwendet man hierzu die Vereinigung. Für eine Anwendung einer Prädikatsvariablen berechnet man rekursiv eine Randbedingung für den Rumpf und führt eine Substitution durch. Für rekursive Prädikate verwendet man eine Fixpunktiteration wie bei der Standardsemantik von $\mathbb{B}_\mu^V$ aus Abschnitt 2.6.5. Dabei werden aber nur kleinste Fixpunkte generiert (vgl. Seite 27), d. h. man startet mit der Randbedingung $(\mathsf{id}, \emptyset, \mathsf{id}, \mathsf{id})$. Satz 4.21 ergibt dann, daß der Fixpunkt die kleinste Randbedingung ist, die garantiert, daß eine sie erfüllende Allokation, alle Basisrandbedingungen erfüllt, und umgekehrt wenn der Fixpunkt nicht erfüllbar ist, daß dann auch keine Randbedingung existiert, die alle Basisrandbedingungen erfüllt.

Definition 4.25 (Generierung kleinster Randbedingungen)
Sei $\rho \in [P \to \mathbb{C}]$, *mit* $V^{\mathsf{r}}(\rho(X)) \subseteq \pi(X)$ *für alle* $X \in P$ *und* $\rho(X)\downarrow$ *(s. Def. 4.26). Nun definiere* $\mathsf{cs}_\rho \colon T_\mu^V \to \mathbb{C}$ *rekursiv durch*

$$
\begin{aligned}
\mathsf{cs}_\rho(u[i]) &:= \bot = (\mathsf{id}, \emptyset, \mathsf{id}, \mathsf{id}) \\
\mathsf{cs}_\rho(u \doteq v) &:= (\{(x,y),(y,x)\} \cup \mathsf{id}, \emptyset, \mathsf{id}, \mathsf{id}) \\
\mathsf{cs}_\rho(\exists\, u \,.\, s) &:= \mathsf{cs}_\rho(s)|_{V \setminus \{u\}} \\
\mathsf{cs}_\rho(\neg s) &:= \mathsf{cs}_\rho(s) \\
\mathsf{cs}_\rho(s \wedge t) &:= \mathsf{cs}_\rho(s) \sqcup \mathsf{cs}_\rho(t) \\
\mathsf{cs}_\rho(X(\underline{u})) &:= \zeta\{\pi(X) \mapsto \underline{u}\}
\end{aligned}
$$

wobei $\zeta := \mathsf{cs}(\beta(X))$ *für* $\sigma(X) = \varepsilon$ *und*

$$\zeta \quad := \quad \bigcap \{c \in \mathbb{C} \mid \mathsf{cs}_{\rho\{X \mapsto c\}} = c, V^{\mathsf{r}}(c) \subseteq \pi(X)\}$$

für $\sigma(X) \in \{\mu, \nu\}$.

Definition 4.26 (Einschränkende Variablen) *Zu einer Randbedingung* $c \in \mathbb{C}_1$ *sei die Menge* $V^{\mathsf{r}}(c)$ *der einschränkenden Variablen definiert als*

$$V^{\mathsf{r}}(c) \quad := \quad \{x \in V \mid \textit{es gibt } y \in V, \textit{ mit } x \neq y \textit{ und } x(\sim_c^+ \cup \sim_c^- \cup \leq_c^- \cup \leq_c^+)y\}$$

Die Bestimmung des kleinsten Fixpunktes wird genauso als Iteration behandelt wie bei der Standardsemantik auf Seite 32.

Hat man nun nach Definition 4.25 eine kleinste Randbedingung gefunden, die erfüllbar ist, so kann man nach Satz 4.23 eine Allokation generieren, die alle Basisrandbedingungen erfüllt. Ist das Ergebnis nicht erfüllbar, so muß entweder bei einer Substitution oder einer Vereinigung „$\top$" generiert worden sein. Bei dieser Operation versucht dann die μcke durch Variableneinschränkung oder den Schnitt eine schwächere Randbedingung zu generieren, die noch erfüllbar ist und Teile der ursprünglichen enthält. Leider kann diese Strategie zur Nichterminierung der Fixpunktiteration führen. Dies wird dadurch abgefangen, daß die Iteration abgebrochen wird, sobald eine schwächere Randbedinung als in der vorigen Iteration auftritt.

Gelingt es sogar, die Randbedingung sukzessive so zu erweitern, daß diese maximal in $\mathbb{C}_1$ ist, und dennoch ein Fixpunkt für alle rekursiven Prädikate bleibt, dann sind die Anwendungen von allen Prädikatsvariablen ordnungserhaltend. Dies bedeutet, daß die Substitutionen von BDD-Variablen bei der Berechnung der BDD-Semantik für eine diese maximale Randbedingung erfüllende Allokation alle monoton sind (vgl. Def. 3.31).

In der μcke werden nicht alle Ordnungen ausprobiert, sondern nach Satz 4.23 eine Allokation generiert. Zu dieser gehört dann nach Satz 4.16 eine Randbedingung, die sich als maximal in $\mathbb{C}_1$ erweist. In der Praxis war dann in den meisten Fällen die so gewonnene Randbedingung wieder ein Fixpunkt für alle Prädikate. Hier sollten weitere Untersuchungen zeigen, warum dies immer so gut funktioniert hat.

Des weiteren sollten die formalen Parameter von Prädikaten verschieden sein, damit nicht unötig die Kombination von Randbedingungen unerfüllbar wird. Dies kann immer durch α-Konversion erreicht werden. Durch die Forderung an Allokationen, daß sie injektiv sind, folgt daraus aber, daß die Anzahl benötigter BDD-Variablen steigt und die BDDs für verschiedene Prädikate sich nicht überschneiden können. Deshalb wurde beim Allokationsalgorithmus der μcke der Weg begangen, daß pro Prädikat eine eigene Allokation generiert wird. Genauso kann man optional für jeden Quantor eine eigene Allokation generieren lassen. Die Vorgehensweise ist dabei an das hier vorgestellte Verfahren angelehnt, ist aber wesentlich komplexer, so daß eine formale Beschreibung sehr umfangreich wäre.

Beispiel

Betrachte das Beispiel des „Schedulers" von Milner aus Abschnitt C.5. Hier wurde vom Benutzer nur *eine* Randbedingung angegeben, nämlich bei der Definition von `bisimulates`

```
nu bool bisimulates(SchedState p1, SpecState p2) p1 ~+ p2
```

Standardmäßig wurde die ASTI-Heuristik verwendet. Startet man die μcke mit „`mucke -v -v -v -v`", so werden auch die generierten Allokationen ausgegeben. Ein Auschnitt aus der Ausgabe bei diesem Beispiel lautet

```
final allocation constraint for
  'bisimulates::@Forall12::@Exists13'
  interleaving: p1 ~+ p2, p2 ~+ q1, q1 ~+ q2
         block: p1 ~- a, p2 ~- a, a ~- q1, a ~- q2
      ordering: p1 < p2, p2 < q1, a < p1, q1 < q2
allocation for 'bisimulates::@Forall12::@Exists13' is
  a
  p1 p2 q1 q2
```

Dabei wird de Namen `Forall12` mit dem ersten Allquantor im Rumpf von `bisimulates` assoziiert, und `Forall12::Exists13` steht für den Existenzquantor im Rumpf dieses Allquantors. Im Bindungsbereich von `Exists13` liegen die Variablen `p1`, `p2`, `q1`, `q2` und `a`. Für diese erzeugt die μcke die angegebene Allokation. In dieser Schreibweise, stehen die Variablen aus demselben Block in einer Zeile. Die Blöcke sind von oben nach unten geordnet und die Variablen in einem Block von links nach rechts. Man beachte, daß die endgültige Randbedingung (`final ...`) maximal erfüllbar ist, so daß alle bei der Berechnung von `bisimulates` auftretenden Substitutionen monoton sind.

Bei diesem Beispiel hat die obige Allokation zur schnellsten Verifikationszeit geführt. In einem ersten Ansatz wurde die angegebene Randbedingung weggelassen, was in folgender Ausgabe resultierte

```
final allocation constraint for
  'bisimulates::@Forall12::@Exists13'
  interleaving: p1 ~+ q1, p2 ~+ q2
         block: p1 ~- p2, p1 ~- a, . . .
      ordering: p1 < q1, p2 < q2, a < p1, q1 < p2
allocation for 'bisimulates::@Forall12::@Exists13' is
  a
  p1 q1
  p2 q2
```

Auch hier erhält man nur monotone Substitutionen, die BDDs bei der Berechnung der Approximationen für `bisimulates` waren aber größer.

4.5 Zusammenfassung

Dieses Kapitel beschäftigte sich mit einem wichtigen Aspekt bei der Verwendung von BDDs zur Modellprüfung im μ-Kalkül. Nämlich mit der Frage, wie Variablen des μ-Kalküls BDD-Variablen zugeordnet werden. Es wurde motiviert, warum es außerordentlich wichtig ist, zu „guten“ Zuordnungen (Allokationen) zu kommen. Dann wurde erstmals für den *vollen* μ-Kalkül eine automatische Methode vorgestellt, die aus partiellen Anforderungen an eine Variablenallokation eine vollständige Allokation erzeugt. Schließlich wurde gezeigt, wie sich diese Methode in die Praxis umsetzen läßt, so daß unter Verwendung von Heuristiken der Benutzer eines μ-Kalkül Modellprüfers sich weitestgehend um den Aspekt der Variablenallokation nicht zu kümmern braucht.

4.6 Ausblick

Dynamische Variablenumordnung nach /Rudell, 1993/ kann sehr hilfreich sein, um für BDDs bessere Variablenordnungen zu finden. Diese Methode beachtet aber keine Randbedingungen, so daß nicht garantiert werden kann, daß Substitutionen monoton sind. Hier ist nach einem BDD-Algorithmus für dynamische Variablenumordnung gesucht, der Randbedingungen respektiert.

Des weiteren sollte nach neuen Arten von Randbedingungen gesucht und in das Verfahren integriert werden. Auch sollte man untersuchen, wie die Konfliktlösung beim Enstehen einer nicht erfüllbaren Randbedingung durch Prioritäten besser behandelt werden kann. Ein weiteres Ziel ist es, die Allokationsalgorithmen für Schaltnetze aus /Malik et al., 1988, Fujii et al., 1993/ in einen Formalismus zu integrieren.

Kapitel 5

Der Modellprüfer μcke

5.1 Übersicht

Dieses Kapitel besteht aus drei Teilen. Im ersten Teil wird die objektorientierte Architektur der μcke vorgestellt. Danach wird gezeigt, wie sich Optimierungen der Modellprüfung auf den μ-Kalkül übertragen lassen. Zum Schluß wird die Leistungsfähigkeit der μcke mit der anderer Modellprüfer verglichen.

5.2 Architektur

Beim SMV System /McMillan, 1993b/ und dem Modellprüfer /Janssen, 1996b/ ist der Modellprüfungsalgorithmus fest mit der verwendeten BDD Bibliothek gekoppelt. In beiden Fällen ist es deshalb sehr schwierig, andere BDD Bibliotheken zu verwenden oder etwa eine völlig andere Repräsentation von Prädikaten. Ebenso läßt sich so kaum ein dynamischer Wechsel der Repräsentation zur Laufzeit erreichen. Dies ist wichtig, da sich BDD Bibliotheken bez. Effizienz und Funktionalität unterscheiden.

Im SMV System (und auch bezüglich der expliziten Freigabe von nicht mehr benötigten BDDs im System von Janssen) ist es darüber hinaus aus den in Abschnitt 3.5 dargelegten Gründen notwendig, im Modellprüfungsalgorithmus die Speicherverwaltung von BDDs zu handhaben. Deshalb muß bei diesen Systemen bei Änderung oder Erweiterung des Modellprüfungsalgorithmus viel Aufwand getrieben werden, um eine korrekte Speicherverwaltung (keine Zeiger ins „Nirvana“, keine Speicherlecks) zu gewährleisten.

Für die μcke wurde deshalb eine Schichtenstruktur entworfen, in der die Repräsentation von Prädikaten und der eigentliche Modellprüfungsalgorithmus durch mehrere Ebenen voneinander getrennt sind. Für die Schnittstellen wurde ein „Entwurfsmuster“ (engl. design pattern) entwickelt, das die einfache (sogar dynamische) Austauschbarkeit verschiedener Implementierungen gestattet. Als Resultat dieser Bemühungen kann die μcke verschiedene BDD Bibliotheken verwenden, die erst zur Laufzeit geladen werden.

Neben der Trennung von Modellprüfungsalgorithmus und Repräsentation von Prädikaten ist es sehr wichtig, die Implementierung des Modellprüfungsalgorithmus an sich erweiterbar zu halten. Dies ist einmal notwendig, um in Zukunft weitere Domänen, wie statistische Größen oder Ungleichungssysteme zur expliziten Behandlung von Zeit oder reellen Werten, integrieren zu können, was die Ausdrucksmächtigkeit der verwendeten Logik und somit das Anwendungsgebiet erweitert. Zweitens muß es die Implementierung gestatten, auf einfache Weise Optimierungen der Modellprüfung einzubringen, für die es in der Literatur (s. Abschnitt 5.3) eine Vielzahl von Vorschlägen gibt. Um dies, zu gewährleisten, wurde eine objektorientierte Herangehensweise gewählt. Es wurde auch hier ein bekanntes Entwurfsmuster verwendet und um wesentliche Teile erweitert. Damit konnte der Modellprüfungsalgorithmus für die Gegenbeispielgenerierung zu großen Teilen wiederverwendet werden (/Jäger, 1996/), und die Implementierung erwies sich als robust gegenüber der Integration verschiedener Optimierungen wie „frontier set simplification“ und Abhängigkeitsanalyse mit gezieltem Rücksetzen.

5.2.1 Schichtenmodell

Die μcke verwendet als Modellprüfungsalgorithmus einen Algorithmus, der sich unmittelbar aus der Definition 2.42 der Semantik II von $\mathbb{B}_\mu^V$ ableitet. In $\mathbb{B}_\mu^V$ gibt es nur Variablen für boolesche Vektoren. Dagegen erlaubt es die μcke-Eingabesprache, auch Variablen über beliebige endliche Domänen zu verwenden. Deshalb arbeitet der Modellprüfungsalgorithmus nicht auf booleschen Ausdrücken wie es die Semantik II tut, sondern verwendet

Prädikate über Variablen aus beliebigen endlichen Domänen

als Abstraktion von booleschen Ausdrücken, die als Prädikate über boolesche Variablen aufgefaßt werden können.

Alternative Repräsentationen von Prädikaten

Für den Modellprüfungsalgorithmus selbst spielt es keine Rolle, welche konkrete Implementierung für die Repräsentation von Prädikaten gewählt wurde. Es ist sogar möglich die aktuelle Repräsentation zur Laufzeit (auch als *dynamisch* bezeichnet) zu wechseln. Denkbar als alternative Implementierungen sind

Terme Dies ist die „symbolischste“ Variante. Hier wird intern dieselbe Repräsentation von Prädikaten verwendet, wie es die μcke-Eingabesprache erlaubt. Der Nachteil ist natürlich, daß es keine effizienten Algorithmen gibt, die die benötigten Operationen wie Quantifikation und Vergleich auf semantische Äquivalenz realisieren. Dafür sind z. B. Substitutionen unproblematisch.

Boolesche Prädikate Bei endlichen Modellen kann man jeden Typ als Teilmenge eines $\mathbb{B}^n$ für ein geeignetes $n \in \mathbb{N}\backslash\{0\}$ kodieren. So läßt sich ein allgemeines Prädikat durch ein „boolesches Prädikat“ ausdrücken. Für boolesche Prädikate stehen nun wiederum die anderen Implementierungsalternativen zur Verfügung.

Entscheidungsdiagramme Die wichtigste Variante ist die Verwendung von Entscheidungsdiagrammen. Diese können entweder direkt auf beliebigen endlichen Typen definiert werden (/Rauzy, 1995/) oder, falls nur boolesche Prädikate vorliegen, als eine Variante von BDDs implementiert werden. Letzteres stellt eine Realisierung der BDD-Semantik aus Abschnitt 3.7 dar.

Mengenrepräsentation Die am wenigsten „symbolische“ Repräsentation ist die Identifikation eines Prädikats mit seiner *Extension*, d. h. der Menge der erfüllenden Belegungen. Bis auf die fehlende Belegung von freien Variablen und der Behandlung von Quantoren entspricht diese Vorgehensweise der Standardsemantik von $\mathbb{B}_\mu^V$ aus Definition 2.33.

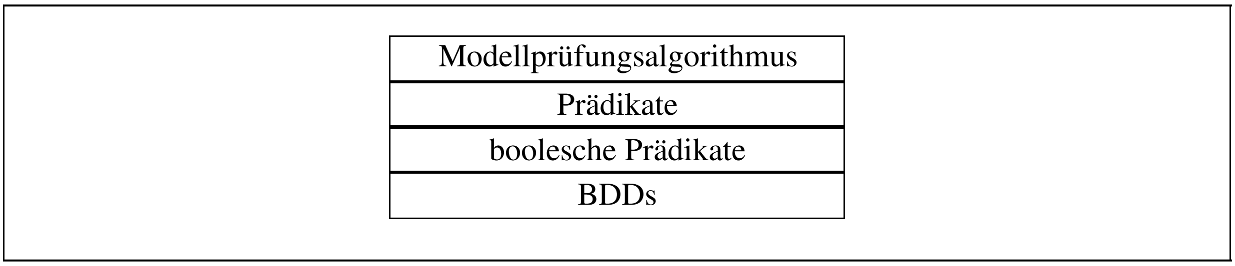


Abbildung 5.1: Vereinfachtes Schichtenmodell bei Verwendung von BDDs.

Für spezielle Aufgabenstellungen lassen sich noch weitere Repräsentationen angeben. So sind insbesondere solche Domänen interessant wie Realzeit bei der Verifikation von hybriden Systemen /Henzinger et al., 1993/, wie statistische Größen bei Betrachtung von Fehlertoleranz- oder Leistungsmerkmalen /Aziz et al., 1996/, oder potentiell unendliche Puffer bei der Verifikation von Telekommunikationsprotokollen /Rinderspacher, 1996, Boigelot und Godefroid, 1996/.

Die in dieser Auflistung verwendeten Implementierungen von Prädikaten haben die Gemeinsamkeit, daß sie für die jeweilige Anwendung genau den passenden Grad von „symbolischer Repräsentation" besitzen, so daß die auszuführende Aufgabe gerade entscheidbar bzw. semi-entscheidbar wird oder effizient (in der Praxis) durchgeführt werden kann. Hier sollten weitere Untersuchungen zeigen, wie die entsprechenden Konzepte mit der Architektur der μcke verbunden werden können.

Der große Vorteil, den man durch die abstrakte Sichtweise auf Prädikate bei der Implementierung der μcke hat, ist die Wiederverwendbarkeit all der Bestandteile des Systems, die sich nur auf diese Schnittstelle abstützen. Das sind zum Beispiel die Abhängigkeits- und Monotonieanalyse und die dazugehörigen Optimierungen. Es ist auch zu erwarten, daß große Teile des Modellprüfungsalgorithmus selbst wiederverwendet werden können.

Übersicht über die Implementierung

Die wichtigste Instanz des Architekturmodells der μcke verwendet auf unterster Ebene BDDs zur Repräsentation von Prädikaten (vgl. Abb. 5.1). Dazwischen findet man boolesche Prädikate, die, wie oben schon erläutert, dazu dienen, Variablen über beliebige Domänen durch boolesche Variablen zu kodieren. Schon dieser Schritt ist vom Modellprüfungsalgorithmus vollkommen unabhängig. Die nächste Schnittstelle zwischen booleschen Prädikaten und BDDs ist genauso allgemein gehalten wie der Übergang zu booleschen Prädikaten.

Neben Lizenzgründen, aber auch um die unterschiedliche Funktionalität oder Effizienz verschiedener BDD-Bibliotheken ausnützen zu können, müssen diese Schnittstellen so implementiert werden, daß nicht nur die oben betrachteten alternativen Repräsentationen einfach zu integrieren sind, sondern, daß sogar erst zur Laufzeit und nicht schon bei der Kompilation die tatsächlich verwendete Repräsentation festgelegt wird.

Darüber hinaus erlaubt die Architektur der μcke sogar das dynamische Wechseln der Repräsentation. Das ist besonders dann von großer Bedeutung, wenn das Konzept des Abspeichern und Ladens von Repräsentationen über mehrere Schichten hinweg realisiert ist. Dann ist es nämlich auch möglich, *externe* Prädikate bei der Verifikation zu verwenden, die nicht in der Eingabesprache der μcke beschrieben sind.

Um dies zu erreichen, wurde das Entwurfsmuster der *Brücke* (engl. „bridge pattern") aus /Gamma et al., 1995/ erweitert um eine „Manager"-Komponente. Die Struktur des ursprünglichen Entwurfsmusters findet man in Abb. 5.2, wobei gegenüber /Gamma et al., 1995/ für „**Abstraktion**" keine Unterklassen betrachtet werden, und die des erweiterten in Abb. 5.3.

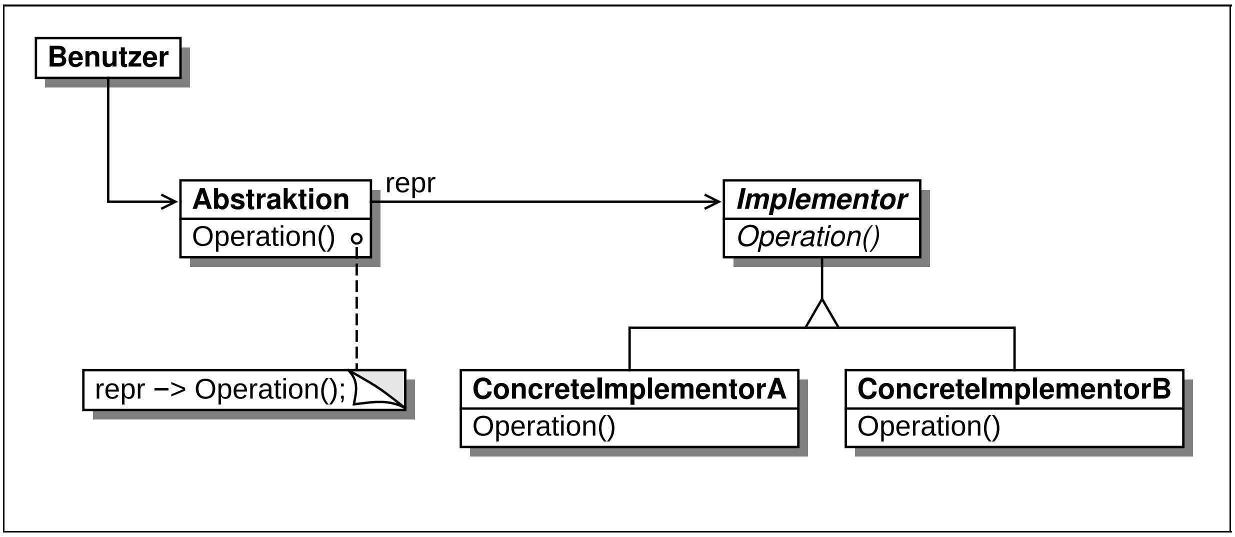


Abbildung 5.2: Das Enwurfsmuster der *Brücke* (engl. bridge pattern).

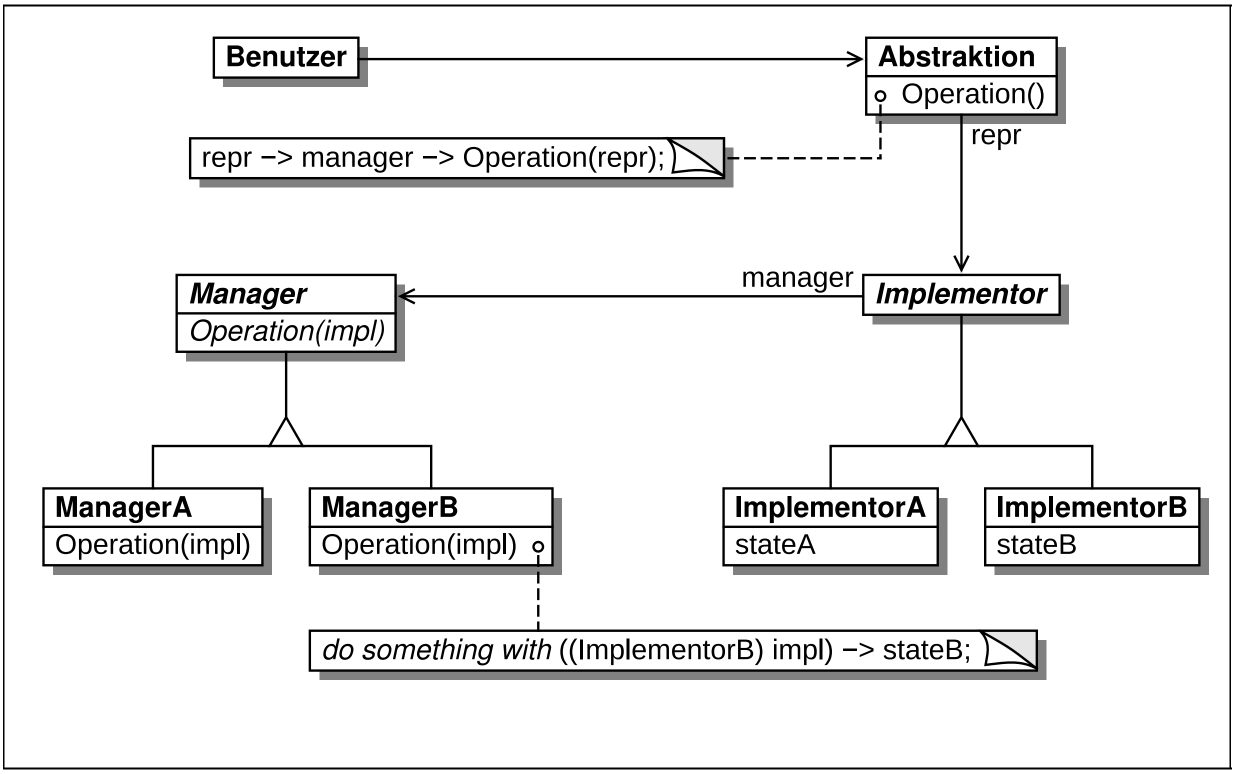


Abbildung 5.3: Eine Erweiterung des Enwurfsmuster der *Brücke*.

Der Hauptunterschied besteht darin, daß die Klasse „**Implementor**“ bei der erweiterten Version im wesentlichen nur noch als Speicher für den Zustand der Repräsentation (z. B. „stateA“) dient und alle Operationen in den „**Manager**“ verlegt worden sind. Neben den Vorteilen des ursprünglichen Entwurfsmusters erreicht man so:

1. Durch einen Verweis auf den classManager in einem „**Implementor**“-Objekt kann die bei einer binären Operation notwendige Typeinschränkung (engl. down cast) sicher durchgeführt werden. Hier dient die „Manager“-Komponente als dynamische Typinformation.

2. Um eine konkrete Repräsentation („**ConcreteImplementorA**“) zu verwenden, muß nur ein entsprechender Manager generiert werden. Für dynamisches Laden bedeutet dies, daß aus der zu ladenden Bibliothek nur eine Funktion explizit importiert und ausgeführt werden muß.

Weiter ist der Manager eine „Abstrakte Fabrik“ (engl. abstract factory) für konkrete Implementoren (s. /Gamma et al., 1995/) und Zusammenlegen (engl. „to share“) von mehreren Implementoren durch eine Abstraktion durch Zählen von Referenzen kann einfach integriert werden (s. Abschnitt 3.5.6 und /Coplien, 1992/). Dieses erweiterte Entwurfsmuster wurde in der μcke für die Implementierung der mittleren zwei Schichten aus Tab. 5.1 benützt, was in Abb. 5.4 ausgeführt ist.

Jeweils für einen „**Implementor**“ und eine „**Abstraktion**“ wurde die Implementierung angegeben. Man kann sogar textuell denselben Programmkode für die „and“-Operation der zwei Abstraktionen „**Predicate**“ und „**Boole**“ verwenden. Außerdem machen diese Zeilen deutlich, daß die Speicherverwaltung durch Zählen von Referenzen vollständig in der Abstraktion erledigt werden kann. Ein Manager muß sich darum nicht kümmern, was es vereinfacht, eine neue konkrete Klasse samt Manager zu implementieren.

Für die Implementierungen von „**Boole**“ sind zwei Varianten angegeben. Einmal „**BDDsimpleBRepr**“, die die BDD-Bibliothek `BDDsimple` von Seite 3.6.1 verwendet, und zweitens „**BDDLongBRepr**“ (s. auch S. 3.4). Für letztere ist auch eine abstrakte Version der Implementierung für die „and“-Operation aufgezeigt. Darin wird eine Typkonversion der Argumente durchgeführt, wenn der Manager eines Argumentes nicht mit dem Manager übereinstimmt, der die Operation ausführt (normalerweise der Manager des ersten Argumentes).[1] Danach wird unter Verwendung des Managers der BDD-Bibliothek („long_bddMan“) die entsprechende Routine „bdd_and“ der BDD-Bibliothek aufgerufen. Man beachte dabei, daß die dynamische Typeinschränkung (down cast) „(BDDLongBRepr)“ korrekt und sicher ist.

5.2.2 Evaluatoren

In diesem Unterabschnitt wird als eine Erweiterung des Besucher-Entwurfsmusters (engl. visitor pattern) aus /Gamma et al., 1995/) das Entwurfsmuster des *Evaluators* vorgestellt. In der μcke wird es dazu benutzt, Algorithmen zu implementieren, die auf der internen Term-Struktur arbeiten. Unter diese Klasse von Algorithmen fällt auch der eigentliche Modellprüfungsalgorithmus, wie er sich aus Definition 2.42 ergibt, als auch die Gegenbeispielgenerierung, die aus /Kick, 1996/ stammt. Das Entwurfsmuster des Evaluators ist immer dann geeignet, wenn man eine höchstmögliche Unabhängikeit von Algorithmen braucht, die auf einer festen Term- oder Graphenstruktur arbeiten, wie zum Beispiel bei Algorithmen in Übersetzern (engl. „compiler“).

Bei der μcke ergaben sich folgende Anforderungen an die Implementierung des Modellprüfers, die mit anderen Techniken nur sehr schwer zu realisieren wären.

[1] In der gegenwärtigen Implementierung der μcke wird hier eine Ausnahme (engl. exception) ausgelöst.

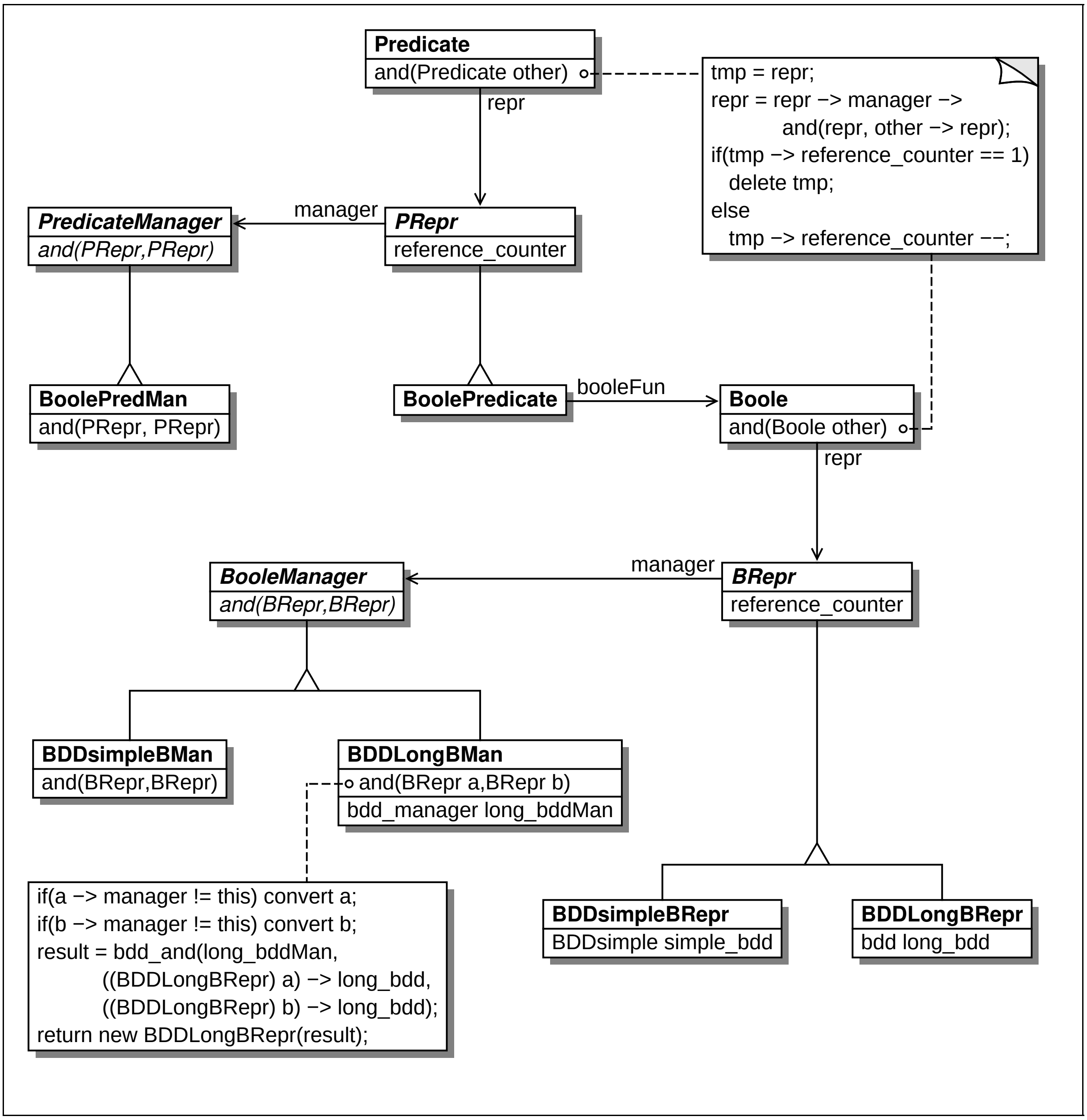


Abbildung 5.4: Struktur der Prädikatsrepräsentation der μcke im Detail.

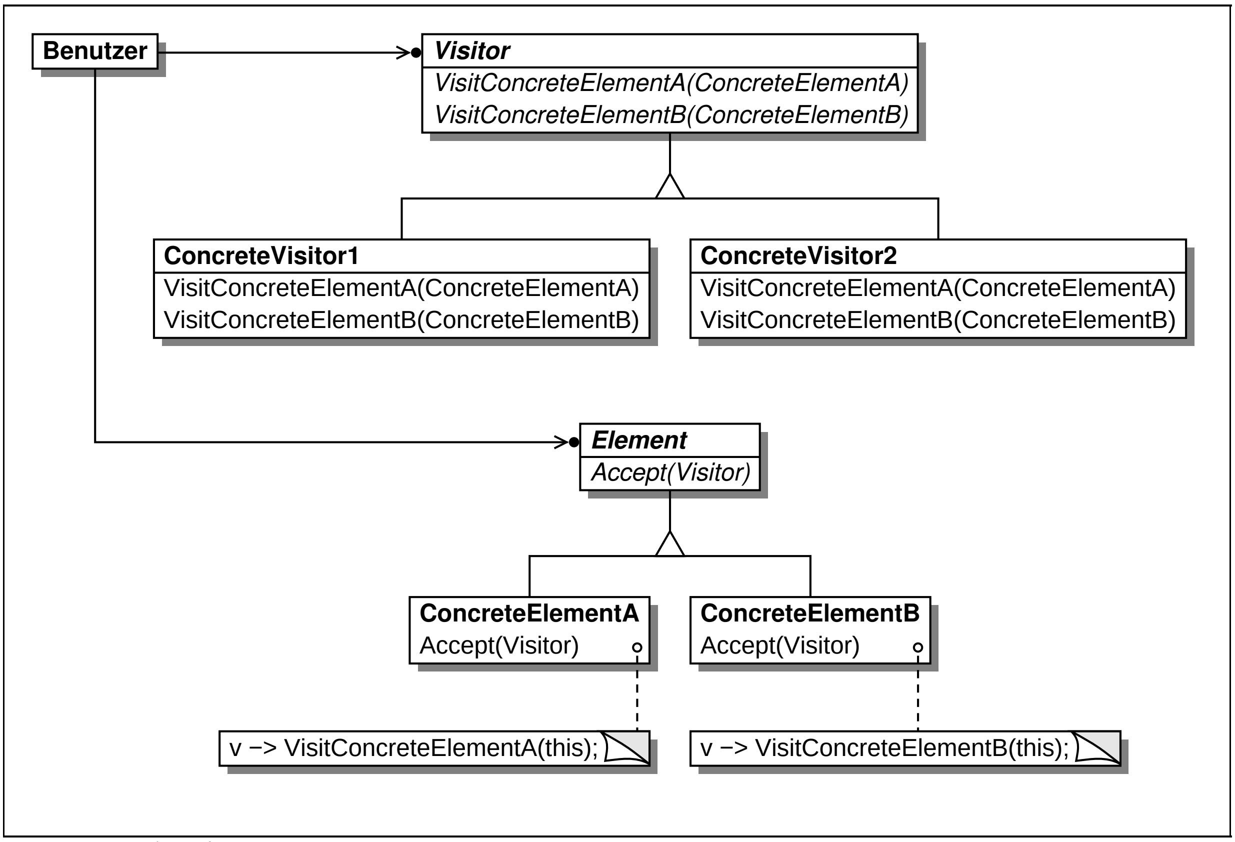


Abbildung 5.5: Das Enwurfsmuster des *Besuchers* (engl. visitor pattern).

1. Speicherlecks auf höheren Schichten sind unvertretbar, wenn sich dahinter riesige BDDs verbergen.

2. Die Optimierung der Abhängigkeitsanalyse erfordert, daß „lokale" Zustände mit einem Termknoten verbunden werden, im Gegensatz zum Entwurfsmuster des Besuchers, bei dem nur ein globaler Zustand im Besucher selbst gespeichert wird.

3. Eine „faule" (lazy) Allokations- und Berechnungsstrategie erfordert prinzipiell, daß mit einem Termknoten eine unbestimmte Anzahl von Zuständen verschiedener Algorithmen verknüpft sein kann.

4. Die parallele Entwicklung der Optimierung der Modellprüfung durch den Autor und die Implementierung der Erzeugung von Gegenbeispielen in /Jäger, 1996/ verlangte eine feste Termstruktur, an der sich nichts änderte. Nach einer parallelen Entwicklungsphase sollte die Integration beider Versionen leicht durchzuführen sein.

Das Entwurfsmuster des Besuchers ist in Abb. 5.5 zu finden. Hier ist zu erkennen, daß die Unterklassen von „**Element**" nichts über die konkreten Besucher wissen müssen. Ebenso sind die Implementierungen der einzelnen Besucher völlig unabhängig. Was aber hier vollkommen fehlt, ist die Möglichkeit einem Element für einen bestimmten Besucher einen Zustand zuzuordnen. Bei der μcke wäre so ein Zustand zum Beispiel ein Prädikat und ein Kompiler würde z. B. gerne in der Kodeerzeugungsphase den erzeugten Kode am entsprechenden Termknoten des Syntaxbaumes ablegen.

Natürlich könnte jeder Besucher das intern bewerkstelligen z. B. durch eine Tabelle in der Zustände mit Termen assoziiert werden. Dies führt aber zur unötigen Duplizierung von Programmkode zur Verwaltung sowohl der Tabelle als auch der Zustände. Eine andere Möglichkeit besteht darin, einen entsprechenden Verweis auf den Zustand im Termknoten selbst abzulegen. Hier muß man aber dafür sorgen, daß diese Verweise wiederum abstrakt sind, damit keine Abhängigkeit zwischen den einzelnen Besuchern oder den Besuchern und den konkreten Elementen entsteht. Diesen zweiten Weg geht auch das Entwurfsmuster des Evaluators, das instantiiert für die μcke in Abb. 5.6 zu finden ist.

Die „**Term**"-Klasse entspricht der abstrakten „**Element**"-Klasse und prototypisch sind die drei Unterklassen „**Definition**", „**Application**" und „**Binary**" aufgeführt. Letztere hat wiederum zwei Unterklassen „**And**" und „**Or**", die für das boolesche „Und" bzw. „Oder" stehen. Die „Accept"-Methode heißt hier nun „eval".

Auf der rechten Seite sind die Evaluatoren zu finden, die den Besuchern entsprechen. Erweitert wurde das Besucher Entwurfsmuster um den mittleren grau unterlegten Teil, bestehend aus „**EvalState**" samt Unterklassen und dem Manager von Evaluatoren „**EtorManager**". Die „stateList" eines Termes besteht aus abstrakten Zuständen, so daß die Implementierung von Term-Methoden nichts von dem konkreten Zustand wissen muß. Ebenso können die einzelnen Evaluatoren unabhängig voneinander definiert werden. Der zu einem bestimmten Evaluator gehörende Zustand wird vom Evaluatoren-Manager „etorManager" durch die „get"-Methode geliefert. Ensprechend gibt es „add" und „remove" zum Hinzufügen oder Löschen eines Zustandes. Der Evaluatoren-Manager übernimmt außerdem die Aufgabe, nach Löschen eines Evaluators dessen noch existierenden Zustände zu löschen. Ansonsten müßte jeder Evaluator eine entsprechende Routine implementieren.

Der Evaluatoren-Manager kann einen Zustand auf der „stateList" eines Termes zu einem Evaluator durch die Registrierungsnummer „regrNum" des Evalutors zuordnen. Diese ist nur durch den Evaluator-Manager manipulierbar. Damit das Löschen aller Zustände nach „Dahinscheiden" eines Evaluators durch den Evaluatoren-Manager durchgeführt werden kann, verwaltet dieser zu jedem Evaluator mit der Registrierungsnummer „regrNum" in „states[regrNum]" eine doppelt verzeigerte Liste aller Zustände des Evaluators.

Weiter zeigen die Ausschnitte aus dem Programmkode für die Methoden von „**Evaluator**" wie sich die Klassenstruktur von „**Term**" auch auf die Evaluatoren übertragen läßt. So muß zum Beispiel der Evaluator „**DepEtor**" zur Abhängigkeitsanalyse von Prädikatsvariablen nur eine Implementierung für die Evaluation von Applikationen definieren. Alle anderen Methoden kann er übernehmen.

Ähnliches gilt für „**CEXEtor**", den Evaluator zur Generierung von Gegenbeispielen und Zeugen. Er kann tatsächlich alle Methoden des Evaluators „**PredEtor**" zur Implementierung der Modellprüfung verwenden. Nur bei der Evaluation von Funktionsdefinitionen ist eine besondere Behandlung notwendig. In der μcke verwendet „**CEXEtor**" sogar alle Methoden, da auch ein großer Teil der „evalDef"-Methode sich als redundant erwies. Das Abspeichern von Prädikaten im „iterationVector" geschieht dann, wie oben schon erwähnt, durch „template methods", die in der Abbildung nicht aufgeführt sind.

All die oben aufgelisteten Anforderungen werden von dem Entwurfsmuster des Evaluators erfüllt: Zustände können Ihren Evaluator nicht überleben. Versehentlich hinterlassene Zustände werden automatisch gelöscht. Die Übergabe von Zuständen an andere Evaluatoren ist dennoch einfach dadurch realisierbar, daß Evaluatoren durch Agregation geschachtelt werden. Die zweite und dritte Anforderung sind per Konstruktion erfüllt.

Zum letzten Punkt ist noch zu sagen, daß durch die Verwendung dieser Technik es gelang, in der Gegenbeispielgenerierung durch Vererbung den gesamten Programmkode der optimierten Modellprüfung wiederzuverwenden (die Gegenbeispielgenerierung führt in einer ersten Phase eine modifizierte Modellprüfung mit „Abspeichern" durch). Bei der optimierten Modellprüfung mußten dabei nur vier „template methods" eingeführt werden (s. /Gamma et al., 1995/), was ei-

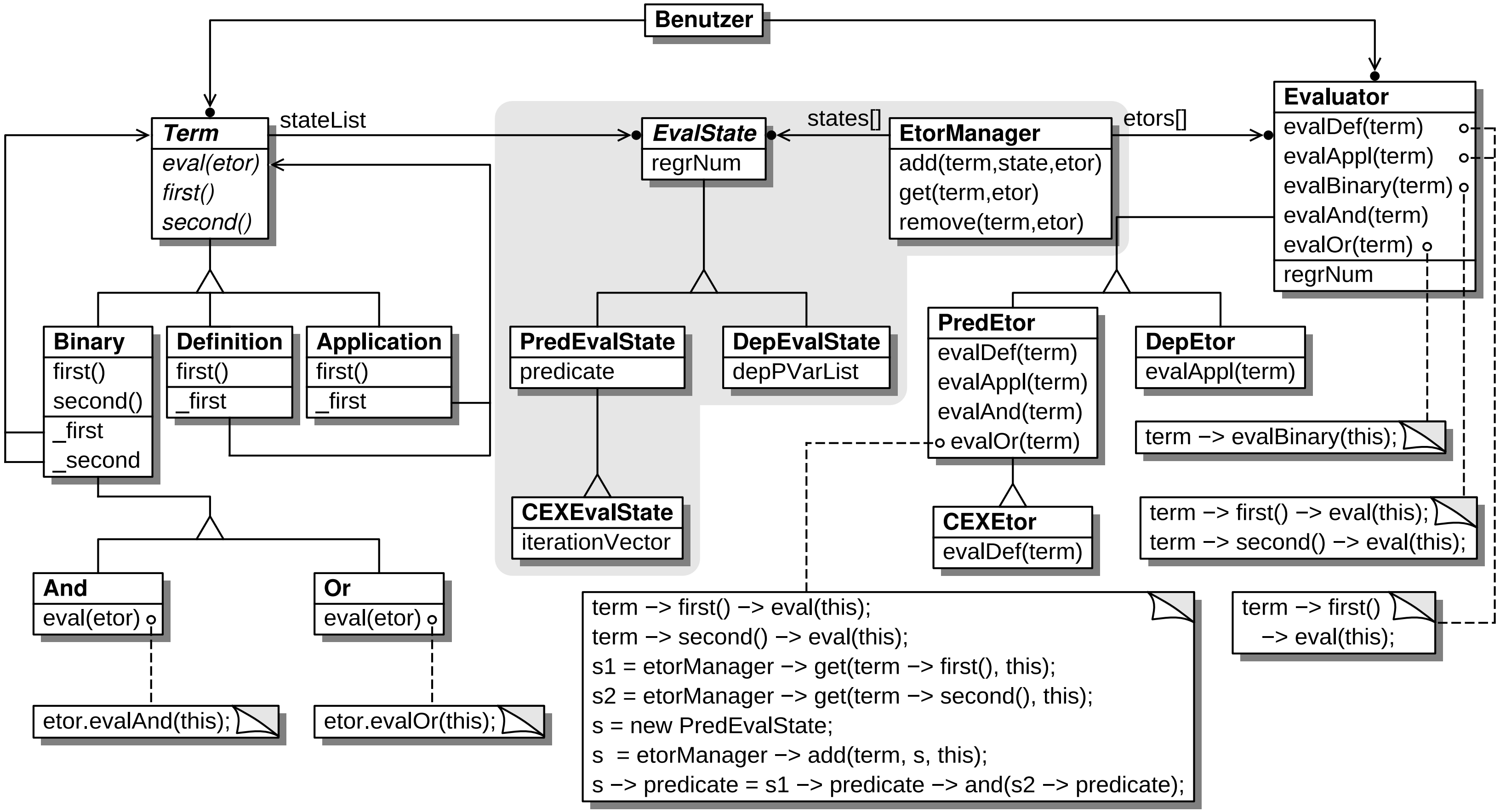


Abbildung 5.6: Das Evaluator Entwurfsmuster in der μcke.

ne Änderung von etwa 10 Zeilen bedeutete, bei rund 1200 Programmzeilen des Moduls für die optimierte Modellprüfung. Dagegen mußte im SMV-System der Gegenbeispielalgorithmus zur Erzeugung von Gegenbeispielen nochmals vollkomen neu geschrieben und um entsprechende „Abspeicherroutinen“ erweitert werden. Bei der μcke konnte nicht nur nach weiteren Optimierungsarbeiten am Modellprüfungsalgorithmus der gesamte Programmkode der Gegenbeispielgenerierung unverändert übernommen werden (sogar *ohne* eine Rekompilation). Sondern die Gegenbeispielgenerierung profitierte sogar automatisch von den Optimierungen.

Die wichtigsten gegenwärtig verwendeten Evaluatoren lauten

„EvaluatorPredicate“ Der eigentliche Algorithmus für die Modellprüfung. In Abb. 5.6 ist er abgekürzt als „**PredEtor**“.

„EvaluatorAllDefined“ Überprüft, ob alle Prädikate, die bei Berechnung eines Termes benötigt werden, nicht nur deklariert sondern auch definiert sind.

„EvaluatorAllocationConstraint“ Dieser Evaluator implementiert den ursprünglichen Allokationsalgorithmus für Variablen, bei dem auch für jeden Quantor eine eigene Allokation erzeugt wird.

„AllocCSMapper“ Assoziiert Terme mit vom Benutzer explizit angegebenen Allokationsrandbedingungen.

„EvaluatorGetScope“ Erstellt eine Liste aller Variablen, die in einem Term durch Quantoren gebunden sind.

„EvaluatorOneAlloc“ Dies ist ein experimenteller Evaluator zur Variablenallokation. Im Gegensatz zu „**EvaluatorAllocationConstraint**“ benützt er „**EvaluatorGetScope**“, um alle Variablen in einem Term auf einmal zu allokieren.

„EvaluatorOneAllocPhase1“ Die „top down“-Phase von „**EvaluatorOneAlloc**“.

„EvaluatorDependencies“ Erzeugt einen Graphen, der jeden Term mit den Definitionen der Prädikatsvariablen („**Definition**“) verbindet, von denen der Term abhängig ist (vgl. mit $\mathsf{dep}(t)$ aus Def. 2.23). Dieser Graph wird nach Berechnung des Prädikates eines Termes wieder gelöscht, da er sehr groß sein kann. Nur das Prädikat selbst wird in einem Ergebnisspeicher für Prädikate gerettet.

„EvaluatorScc“ Berechnet die stark zusammenhängenden Komponenten (SCC) einer Prädikatsdefinition nach Def. 2.23 und speichert sie direkt in den Term-Knoten ab.

„EvaluatorFunDefDependencies“ Berechnet denjenigen Teilgraphen des von „**Evaluator-Dependencies**“ berechneten Graphen, der nur noch aus Repräsentanten der SCCs besteht, und speichert diesen auch im Term-Knoten ab.

„EvaluatorMonotonicity“ Berechnet die Monontonieindikatoren nach Def. 2.28. Damit kann festgestellt werden, ob eine Prädikatsdefinition wohlgeformt ist.

„EvalPredStore“ Evaluator zur Durchführung der modifizierten Modellprüfung mit Abspeichern nach /Kick, 1996/. Er ist eine Unterklasse von „**EvaluatorPredicate**“ und entspricht „**CEXEtor**“ aus Abb. 5.6.

„TNCreationEvaluator“ Generiert Tableaus als Zeugen und verwendet dazu „**EvalPredStore**“.

„EvalPNFState“ Assoziiert zu einem Term eine boolesche Variable, die angibt, ob ein Term bei der Generierung von Gegenbeispielen als negiert zu betrachten ist (s. /Jäger, 1996/).

„EvaluatorResetPredicate“ Implementiert die Optimierung des gezielten Rücksetzens. Dieser Evaluator wird von „**EvaluatorPredicate**“ benutzt, um gezielt rekursiv Prädikate (und damit eventuell riesige BDDs) von Termen zum frühest möglichen Zeitpunkte freizugeben, wenn sie nicht mehr benötigt werden. Er erfüllt damit Teilaufgaben des Evaluatoren-Managers.

„EvaluatorWitExpand“ „pretty printer“ für Zeugen bzw. Gegenbeispiele.

„PrintTermEvaluator“ „pretty printer“ für Terme.

5.3 Optimierungen

In diesem Abschnitt wird gezeigt, daß sich die wesentlichen Optimierungen bei der Modellprüfung als Äquivalenzumformung im μ-Kalkül oder als interne Optimierung im Modellprüfer realisieren lassen. Zusammen mit der automatischen Variablenallokation aus Kapitel 4 ist damit gezeigt, daß Modellprüfung im μ-Kalkül genauso effizient sein kann wie Modellprüfung von speziellen Logiken.

5.3.1 Interne Optimierungen

Interne Optimierungen sind in den Modellprüfer integriert. Im Gegensatz zu externen Optimierungen erfordert ihre Anwendung keine Veränderung der μ-Kalkül-Beschreibung.

Abhängigkeitsanalyse

Die Abhängigkeitsanalyse berechnet die Funktion „$\mathsf{dep}(\cdot)$" aus Definition 2.23 von Seite 27 und wird hauptsächlich zum gezielten Rücksetzen von gespeicherten Repräsentationen verwendet. Zum Beispiel wird bei der Erreichbarkeitsanalyse im einfachsten Fall die Prädikatsvariablendefinition

$$\mu R(u) \,.\, S(u) \;\vee\; \exists\, v \,.\, T(v,u) \wedge R(v) \tag{5.1}$$

verwendet. Bei rekursiver Ausführung des Modellprüfungsalgorithmus wird zunächst die Repräsentation (z. B. ein BDD) von S und T berechnet. Dies sollte nun nicht noch einmal in jeder Iteration von R geschehen, denn S und T hängen nicht von R ab. Die in der ersten Iteration von R berechneten Repräsentationen von S und T werden deshalb gespeichert (vgl. Abschnitt 5.2.2).

Aber nicht nur Repräsentationen für Prädikatsvariablen sollten gespeichert werden. Im obigen Beispiel ist auch die erneute Substitution für T in späteren Iterationen unötig. Man sollte auch das Ergebnis der Substitution aufbewahren. Um dies zu realisieren berechnet die μcke einen Abhängigkeitsgraphen, der im wesentlichen die Funktion dep aus Definition 2.23 darstellt. Bei rekursiver Berechnung einer Operation (Quantifikation, Substitution, Oder, usw.) werden zunächst die Unterresultate bestimmt und die Operation ausgeführt, um das Ergebnis der Operation zu erhalten. Dann wird getestet, ob die Unterterme vom momentanen (innersten) Fixpunkt abhängen. Ist ein Untertem unabhängig, aber der Vaterterm (der gerade zu bearbeitende Term) abhängig, so wird das Unterresultat am Unterterm abgespeichert und kann so für die nächste Iteration wiederverwendet werden. Ansonsten kann es gelöscht werden. Dies erfordert dann aber nach der Berechnung des Rumpfes einer rekursiven Prädikatsvariablen, daß die gespeicherten Repräsentationen aller Terme, die von dieser Variablen abhängig sind, gelöscht werden, bevor eine erneute Iteration durchgeführt wird.

Bei mehrfach rekursiven Fixpunkten (vgl. Übersetzung von FairCTL in Abschnitt B.2 und Formulierung vom ABP in Abschnitt C.4) ist diese Optimierung besonders wichtig, da damit vermieden wird, daß zum Beispiel Substitutionen von Repräsentationen von äußeren Fixpunkten in jeder Iteration eines inneren Fixpunktes erneut durchgeführt werden.

Auch wird die Abhängigkeitsanalyse bei der Gegenbeispielgenerierung benötigt, um die Prädikatsdefinition zu „hierarchisieren". Ansonsten läßt sich der Algorithmus von /Kick, 1996/ nicht anwenden. Für nähere Details siehe /Jäger, 1996/.

Spezialalgorithmen

Die Berechnung eines Relationenproduktes tritt im SMV-System nur für die Berechnung der CTL-Formel EXf (AX$f \equiv \neg EX \neg f$) und für einen Schritt bei der Breitensuche zur Berechnung der erreichbaren Zuständen auf. Hierfür wird dann immer gleich der collapse-Algorithmus aus Abb. 3.33 bzw. eine Variante für die Erreichbarkeitsanalyse verwendet. Für EXf lautet die Übersetzung nach $\mathbb{B}_\mu^V$

$$\exists\, u \,.\, T(u,v) \wedge f(v)$$

Damit hier die μcke z. B. mit dem SMV konkurrieren kann (vgl. Abschnitt 3.8.1), muß dieser Fall erkannt werden, und statt der rekursiven Evaluation von „$\wedge$“ gleich der relProd-Algorithmus aufgerufen werden. Dies kann durch einfache Analyse des Rumpfes beim Aufruf des Modellprüfungsalgorithmus für eine Quantor geschehen.

Wenn wie in Gleichung (5.1) ein Argument des Relationenproduktes die Anwendung eines rekursiven Prädikates ist, und das Relationenprodukt im Rumpf des rekursiven Prädikates auftritt, dann wird bei jeder Iteration eine Substitution notwendig.[2] In diesem Fall verwendet die μcke „lazy substitutions“, was auf Ebene der BDDs durch den $_{\mathsf{c}}\mathsf{ite}_{\exists}$-Algorithmus (s. Abschnitt 3.8.4) realisiert wird. Dadurch wird wie beim collapse-Algorithmus des SMV-Systems in jeder Iteration eine Substitution gespart.[3] Genauso wird mit den häufig auftretenden Termen $\forall\, u \,.\, s \rightarrow t$ und $\forall\, u \,.\, s \vee t$ umgegangen.

„frontier set simplification“

In /Burch et al., 1994/ wurde die einsichtige Feststellung getroffen, daß bei einem Schritt der iterativen Berechnung der erreichbaren Zustände in Breitensuche nur die im letzten Schritt neu hinzugekommenen berücksichtigt werden müssen. Man kann darüber hinaus sogar irgendeine Menge von Zuständen wählen, die zwischen den neu hinzugekommen und allen schon erreichten liegt. Damit eröffnet sich die Möglichkeit, die Algorithmen aus Abschnitt 3.8.3 zu verwenden, um zu kleinen BDDs zu gelangen. In der μcke wurde diese Optimierung für beide Fixpunktarten eingebaut. Für kleinste Fixpunkte ist dies auch im System von Janssen /Janssen, 1996b/ geschehen.

5.3.2 Externe Optimierungen

Unter „Externen Optimierungen“ sollen solche Optimierungen der Modellprüfung verstanden werden, die sich auf Ebene des μ-Kalküls beschreiben lassen. Ihre Anwendung erfordert also ein Verändern der Eingabe in die μcke. Weiterhin sollen hier keine probabilistischen Optimierungen wie der Super-Trace-Algorithmus aus /Holzmann, 1991/ betrachtet werden, die nicht die absolute Korrektheit postulieren, sondern sich mit einer Wahrscheinlichkeitsaussage begnügen. Deshalb bestehen die folgenden Optimierungen alle aus Äquivalenzumformungen im μ-Kalkül, die die Modellprüfung mit BDDs beschleunigen.

[2] Genauer wird getestet, ob der innerste Fixpunkt, bei dem sich der Wert der Anwendung ändert, der Prädikatsvariablen der Anwendung entspricht, was wiederum durch Abhängigkeitsanalyse erledigt werden kann.

[3] Man kann aber auch die μcke zwingen, wann immer möglich „lazy substitutions“ einzusetzen. Dies geschieht mit der Kommandozeilenoption `-attuls` für „always try to use lazy substitutions“.

Vorwärtsanalyse

Die Vorwärtsanalyse bedeutet, daß bei der Verifikation von Zustandssystemen Prädikate über Zustände auf die erreichbaren Zustände eingeschränkt werden. In /McMillan, 1993a/ wurde gezeigt, daß sich durch diese Technik erhebliche Geschwindigkeits- und Platzeinsparungen erreichen lassen.

In Satz B.11 auf Seite 180 wird dies auf den μ-Kalkül übertragen, wodurch sich eine formale Rechtfertigung und ein weitaus breiteres Answendungsgebiet ergibt. Als Beispiel sei in folgender Prädikatsdefinition S als die Menge der Startzustände, T als die Übergangsrelation und F als eine Menge von Zuständen interpretiert, von denen mindestens einer vom Startzustand erreichbar sein soll.

$$\mu X(u)\,.\,F(u)\ \vee\ \exists v\,.\,T(u,v) \wedge X(v)$$

Wenn man noch im μ-Kalkül

$$\forall u\,.\,S(u) \rightarrow X(u)$$

evaluiert, so entspricht das in CTL EFF. Wie im Beispiel zu Satz B.11 erhält man folgende (bez. des obigen zu evaluierenden Term) äquivalente Prädikatsvariablendefinition

$$\mu R(u)\,.\,S(u)\ \vee\ \exists v\,.\,T(v,u) \wedge R(v)$$
$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \wedge X(v))$$

Man beachte auch die Beschreibung des ABP in Abschnitt C.4, bei der die Vorwärtsanalyse schon integriert ist.

In den beiden Beispielen zum Vergleich der μcke mit dem SMV (vgl. Abschnitt 5.4.1) war diese Optimierung sehr wichtig. Ohne sie benötigte die Verifikation des ABP mit dem SMV-System bei nur einem Datenbit schon 8 Sekunden. Für das DME-Beispiel ist die Situation noch viel dramatischer. Hier hatte das SMV-System bei zwei Zellen nach einer Stunde immer noch kein Ergebnis geliefert, obwohl mit Vorwärtsanalyse dieses schon nach 2.4 Sekunden vorlag.

Vereinfachung der Übergangsrelation

Nach Anwendung der Vorwärtsanalyse wurden alle rekursiven Prädikate auf die erreichbaren Zustände eingeschränkt. Deshalb kann bei der Berechnung des Relationenproduktes der Übergangsrelation mit einem rekursiven Prädikat auch die Übergangsrelation auf die erreichbaren Zustände eingeschränkt werden.

Hierzu sei obiges Beispiel fortgeführt und nach Expansion von X nach Satz B.4 und Hineinmultiplizieren von R erhält man die Prädikatsdefinition

$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \wedge R(u) \wedge R(v) \wedge X(v))$$

Im SMV-System wurde dies dazu benützt, nun T global im ersten Argument mit R zu konjugieren. Dadurch erreicht man eine weitere Geschwindigkeitsverbesserung. Nach Satz 3.35 kann man aber auch statt dessen den „$\downarrow$“-Operator oder den „$\Downarrow$“-Operator verwenden. Unter anderen sind dann neben der Konjunktion die folgenden Definitionen möglich

$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \downarrow R(u) \wedge X(v))$$
$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \Downarrow R(u)) \wedge X(v))$$
$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \downarrow R(u) \downarrow R(v) \wedge X(v))$$
$$\mu X(u)\,.\,R(u) \wedge (F(u)\ \vee\ \exists v\,.\,T(u,v) \Downarrow R(u) \Downarrow R(v) \wedge X(v)),$$

wenn man die Vereinfachungsoperatoren linksassoziativ ließt. Insbesondere die letzte Variante brachte in der Messung „μcke II“ in Tab. 5.2 eine erhebliche Geschwindigkeitsersparnis mit sich (in der Beschreibung des ABP in Abschnitt C.4 wird „$\Downarrow$,“ zu `assume`). Im SMV-System wird diese Optimierung nicht verwendet.

Inkrementelle Generierung der Übergangsrelation

Diese Optimierung wird bei der Berechnung der erreichbaren Zustände eingesetzt und ist in /McMillan, 1993a/ näher beschrieben. Sie generiert Approximationen an die Übergangsrelation, und erlaubt es so, auf die vollständige Repräsentation der gesamten Übergangsrelation zu verzichten. Dies ist besonders nützlich, wenn der BDD für die gesamte Übergangsrelation sehr groß werden würde. Als Beispiel sei eine Übergangsrelation als Konjunktion von kleineren Relationen definiert:

$$T(v,u)\,.\,\bigwedge_{i=1}^{n} T_i(v,u)$$

In die Prädikatsdefinition von R eingesetzt ergibt dies

$$\mu R(u)\,.\,S(u)\ \vee\ \exists\, v\,.\,\bigwedge_{i=1}^{n} T_i(v,u) \wedge R(v)$$

und nach Satz 3.35 erhält man

$$\mu R(u)\,.\,S(u)\ \vee\ \exists\, v\,.\,(\bigwedge_{i=1}^{n} T_i(v,u) \Downarrow R(v)) \wedge R(v)$$

Ein konkretes Beispiel hierfür liefert die Beschreibung des DME in Abschnitt C.6 und die entsprechende Messung in Tab. 5.3. Damit ist einmal die Methode des SMV auf den μ-Kalkül übertragen worden, und man hat zweitens einen formalen Korrektheitsbeweis.

Als Variante kann man statt „$\Downarrow$,“ auch „$\downarrow$“ verwenden. Dann spart man sich nach Satz 3.41 die Berechnung des Relationenproduktes. Bei allen vom Autor durchgeführten Beispielen war aber die höhere Ersparnis durch Vereinfachung mit „$\Downarrow$“ nicht zu überbieten. Interessanterweise zeigt Satz 3.35, daß diese Argumentation auch für disjunktiv zusammengesetzte Übergangsrelationen gilt, was im SMV-System nicht implementiert ist.

MBFS – Modified Breadth First Search

Für disjunktiv zerlegbare Übergangsrelationen wurde in /Burch et al., 1994/ die Optimierung der „modified breadth first search“ angegeben. Dabei wird in einer Iteration der Erreichbarkeitsanalyse zunächst für jeden disjunktiv verknüpften Teil der Übergangsrelation alle Zustände berechnet, die nur mit Übergängen aus diesem Disjunkt von der bisher berechneten Menge von Zuständen erreichbar sind. Die Vereinigung all dieser Mengen ergibt dann den Ausgangspunkt für die nächste Iteration. Die Behauptung lautet nun, daß diese Vorgehensweise auch die Menge der erreichbaren Zustände ergibt.

Im μ-Kalkül sei also die Übergangsrelation T gegeben als

$$T(v,u)\,.\,\bigvee_{i=1}^{n} T_i(v,u)$$

In die Definition von R eingesetzt erhält man

$$\mu R(u)\,.\,S(u)\ \vee\ \exists\, v\,.\,(\bigvee_{i=1}^{n} T_i(v,u)) \wedge R(v)$$

Ausmultipliziert und den Quantor in die entstehende Disjunktion hineingezogen ergibt

$$\mu R(u) \,.\, S(u) \;\vee\; \bigvee_{i=1}^{n} (\exists\, v \,.\, T_i(v,u) \wedge R(v))$$

Nun überzeugt man sich, daß diese Definition äquivalent ist zu

$$\mu R_i(u) \,.\, R(u) \;\vee\; \exists\, v \,.\, T_i(v,u) \wedge R_i(v)$$
$$\mu R(u) \,.\, S(u) \;\vee\; \bigvee_{i=1}^{n} R_i(u)$$

Es ist schwierig, diese Umformung in $\mathbb{B}_{\mu}^{V}$ als korrekt nachzuweisen. Dafür kann man aber die entsprechende Aussage im modalen μ-Kalkül leicht zeigen, was hier nicht durchgeführt werden soll.

		SMV 16381		SMV 5227318		μcke A		μcke B	
#bits	#Zustände	MB	sec	MB	sec	MB	sec	MB	sec
1	140	2	0	10	0	2	0	2	0
2	912	2	0	10	0	3	0	3	0
3	9920	2	0	10	0	3	1	3	1
4	139776	2	0	10	1	3	2	3	1
5	2147840	2	1	10	2	3	6	3	2
6	3e7	2	2	10	4	3	16	3	5
7	5e8	3	5	11	7	5	38	4	12
8	8e9	5	17	13	20	8	83	6	27
9	1e11	9	117	17	109	16	185	10	61
10	2e12	21	778	29	418	>30	-	18	133

Tabelle 5.1: Berechnung Erreichbarer Zustände des ABP.

5.4 Effizienzvergleich mit anderen Modellprüfern

Die im letzten Abschnitt vorgestellten Optimierungen werden hier praktisch angewandt und dadurch gezeigt, daß auch in der Praxis ein μ-Kalkül-Modellprüfer mit speziellen Modellprüfern konkurieren kann. Durch die höhere Flexibilität kann sogar erreicht werden, daß die Modellprüfung wesentlich einfacher zu handhaben ist, und es wird ein Beispiel aufgezeigt, bei dem diese Flexibilität sogar in ein Geschwindigkeitsvorteil umgewandelt werden kann.

5.4.1 … mit SMV

Alternating Bit Protokoll

In diesem Beispiel wird das Alternating Bit Protokoll mit expliziten Hin- und Rückkanälen betrachtet. Siehe Abschnitt C.4 für weitere Details. Es werden Sicherheits- und Lebendigkeitseigenschaften verifiziert bei variierenden Größe der übertragenden Daten (Anzahl Datenbits). Für dieses Protokoll ist es unbedingt zu empfehlen, die Vorwärtsanalyse zu verwenden, da der Zustandsraum sehr dünn mit erreichbaren Zuständen besetzt ist. In diesem Fall müssen zunächst die erreichbaren Zustände bestimmt werden. Für Sicherheitseigenschaften ist dies auch schon fast alles, was zu tun ist und so wurde als Beispiel statt der Verifikation einer Sicherheitseigenschaft einfach nur die Größe der Menge der erreichbaren Zustände bestimmt.

Für die μcke wurde für dieses Beispiel die BDD-Bibliothek /Long, 1994/ verwendet.[4] Bei dieser BDD-Bibliothek paßt sich die Größe des Ergebnisspeichers automatisch an die Anzahl existierender BDD-Knoten an (s. S. 62). Beim SMV-System muß diese Größe vom Benutzer gewählt werden. Für kleinere Probleme kann man die Standardeinstellung des SMV-System verwenden, die mögliche 16381 Einträge vorsieht. Dagegen sollte bei großen Problemen auch ein großer Ergebnisspeicher verwendet werden, da sonst die Berechnung zu lange dauert. Für die Messungen auf einem 120MHz-Pentium-System mit 32MB Hauptspeicher wurde deshalb einmal die Standardeinstellung und zweitens ein Ergebnisspeicher der Größe 5227318 gewählt.

Die Zeiten für die Erreichbarkeitsanalyse sind in Tab. 5.1 zu finden. Das SMV-System hat keine „frontier-set-simplification“ für Interleaving Semantik. In diesem Beispiel kommt diese

[4] Der $_{\mathsf{c}}\mathtt{ite}_{\exists}$-Algorithmus brachte hier keinen Vorteil.

		SMV `-c 16381`		SMV `-c 5227318`		μcke I		μcke II	
#bits	#Zustände	MB	sec	MB	sec	MB	sec	MB	sec
1	140	2	0	10	0	3	1	3	1
2	912	2	1	10	1	3	1	3	1
3	9920	2	2	10	4	3	2	3	2
4	139776	2	7	10	13	3	5	3	4
5	2147840	2	24	10	36	4	19	3	8
6	3e7	3	72	11	77	6	61	5	19
7	5e8	4	215	12	202	13	226	7	47
8	8e9	7	789	15	696	25	781	13	99
9	1e11	-	-	23	3555	>30	-	25	231

Tabelle 5.2: Verifikation einer Lebendigkeitseigenschaft des ABP.

in der μcke verwendete (interne) Optimierung voll zum tragen und verbessert die Laufzeiten gegenüber beiden SMV-Varianten. Die Messung der μcke ohne „frontier set simplification" steht in der Spalte mit der Überschrift „μcke A" und mit „frontier set simplification" in der Spalte mit der Überschrift „μcke B".

Als prototypische Lebendigkeitseigenschaft wurde in Tabelle 5.2 überprüft, ob unter fairer Ausführung des Senders, des Empfängers und der beiden Kanäle unter Annahme, daß beide Kanäle nicht irgendwann nur noch Fehler erzeugen, der Sender immer wieder ein Datum vom Benutzer entgegennimmt. Genaueres zur Modellierung diese Sachverhaltes und Diskussion siehe wiederum Abschnitt C.4.

Der Verifikationsaufwand für die μcke ist bei Verwendung des identischen Verfahrens mit Vorwärtsanalyse vergleichbar mit dem des SMV (μcke I). Beim SMV wurden wiederum die zwei Größen 5227318 und 16381 für den Ergebnisspeicher verwendet. Die Verifikation wurde für 9 Datenbits beim SMV bei kleiner Größe des Ergebnisspeichers nach 3 Stunden abgebrochen und bei der μcke I war der Hauptspeicher (32MB) aufgebraucht.

Benützt man bei der μcke die externe Optimierung der Vereinfachung der Übergangsrelation und bei der Berechnung der erreichbaren Zustände die „frontier set simplification", so erreicht man (μcke II) noch bessere Zeiten als der SMV. Der größere Speicherverbrauch läßt sich auf die vielen BDDs zurückführen, die bei den geschachtelten Fixpunkten relativ lange gespeichert bleiben. Der Modellprüfungsalgorithmus im SMV nimmt die Speicherverwaltung von BDDs selbst in die Hand, so daß er die nicht mehr benötigten BDDs baldmöglichst freigeben kann.

DME

Der asynchrone DME-Schaltkreis aus /Martin, 1985/ wurde in /McMillan, 1993a/ als Beispiel für verschiedene Optimierungen im SMV-System herangezogen. Er dient dazu, verteilten, gegenseitigen Ausschluß (distributed mutual exclusion) zu garantieren. Die Optimierung der Modellprüfung, die bei diesem Beispiel besonders gute Effekte erzielt, ist die der inkrementellen Generierung der Übergangsrelation. Diese wurde nach der in Abschnitt 5.3.2 vorgestellten Vorgehensweise auf den μ-Kalkül übertragen und für den DME angewandt.

In Tabelle 5.3 wurde die Optimierung der Vorwärtsanalyse (SMV `-f` bzw. μcke α) und die Optimierung der inkrementellen Generierung der Übergangsrelation (SMV `-f -inc` bzw. μcke β) verwendet. Hier zeigt sich, daß die μcke bei der Vorwärtsanalyse schneller ist als der SMV,

	SMV `-f`		SMV `-f -inc`		μcke α		μcke β	
#Zellen	MB	sec	MB	sec	MB	sec	MB	sec
2	3	2	2	1	8	4	5	1
3	3	6	2	2	9	9	7	4
4	4	15	2	4	10	18	7	9
5	5	76	2	8	11	37	7	22
6	5	233	2	13	12	83	7	46
7	6	859	2	22	16	168	8	91
8	-	-	2	34	17	215	8	156
9	-	-	2	52	18	606	8	260

Tabelle 5.3: Verifikation einer Sicherheitseigenschaft des DME.

		[EFT93]	[DB95]	μcke I	μcke II
#	#Zustände	sec	sec	sec	sec
6	577	21	5	7	6
8	3073	40	13	11	10
10	15361	87	30	16	13
12	73729	145	55	22	17
14	344065	233	95	31	23
16	1572865	348	196	39	29
18	7077889	569	522	55	38
20	31457281	850	?	68	47

Tabelle 5.4: Verifikation des Schedulers von Milner.

da sie zusätzlich die Optimierung der „frontier set simplification“ verwendet. (die Messung beim SMV für 8 und 9 Zellen wurde nach einer Stunde abgebrochen). Für die spezielle Optimierung der inkrementellen Generierung der Übergangsrelation kann die μcke nur schwer konkurieren. Hier ist sie ungefähr Faktor 5 mal langsamer als der SMV. Eine Analyse zeigte, daß dies vom aufwendigen Evaluatorenkonzept (s. Abschnitt 5.2.2) herrührt, da die auftretenden BDDs sehr klein sind und deshalb das Ablaufen der Indirektionsstufen viel stärker ins Gewicht fällt. Bei größeren Beispielen sollten die Rechenzeiten der μcke und die des SMV wieder näher beianander liegen.

Dieses Problem könnte unter Beibehaltung des μ-Kalküls als Eingabesprache vollständig eleminiert werden, wenn man den Modellprüfungsalgorithmus nicht als Interpreter implementiert, sondern als Übersetzer in eine BDD-Manipulationssprache.

5.4.2 ... mit μ-Kalkül Modellprüfern

In der Literatur /Enders et al., 1993, Dsouza und Bloom, 1995/ wird eine vereinfachte Version des „Schedulers“ aus /Milner, 1989/ als Standardbeispiel für die Modellprüfung im μ-Kalkül verwendet. Den Vergleich der Verifikationszeiten für eine variierende Anzahl von Agenten ist in Tabelle 5.4 aufgeführt. Dabei wurde dieselbe Maschine (SUN 4/75) wie in /Enders et al., 1993/ verwendet (Spalte zwei). Die erste Spalte gibt die Anzahl erreichbarer Zustände des Gesamtsystems (ohne „Parallelschaltung“ der Spezifikation) an. Weiterhin findet man in Spalte drei

	Janssen	μcke
n	sec	sec
10	1.4	0.6
12	7.5	2.0
14	36.6	7.7
16	176.1	30.9

a. n-Bit Zähler

	Janssen	μcke
#Zellen	sec	sec
8	0.5	0.2
16	5.1	0.5
24	21.6	2.1
32	60.9	6.9

b. arbiter

Tabelle 5.5: Vergleich der μcke mit dem System von Janssen.

([DB95]) die Messung aus /Dsouza und Bloom, 1995/. Für Vergleiche mit nicht BDD-basierten Ansätzen siehe /Enders et al., 1993/.

Durch die effiziente Implementierung der μcke in C++ und die Verwendung der obigen internen Optimierungen ist die μcke schon schneller als die zitierten Systeme (μcke I). Hier wurde die BDD-Bibliothek `BDDsimple` und der ${}_c\text{ite}_\exists$-Algorithmus eingesetzt.[5] Die „frontier set simplification" brachte keine Verbesserung und wurde deshalb abgeschaltet.

Unter Verwendung der Optimierung der Vorwärtsanalyse (μcke II), die nun auch für andere Formalismen als FairCTL eingesetzt werden kann, erreicht man eine zusätzliche Verbesserung. Den Quelltext zu diesem Beispiel für zwei Agenten findet man in Abschnitt C.5.

Die Messung μcke I bei 20 Agenten benötigte 10 MB Hauptspeicher (mit der BDD-Bibliothek von D. E. Long wären es nur 6.7 MB gewesen, hätte aber 10% länger gedauert) und die Messung II ca. 7.6 MB (bzw. 6.9 MB). Bei den Messungen aus der Literatur wurde kein Speicherplatzverbrauch angegeben.

Das System von Janssen /Janssen, 1996b/ ist ein Modellprüfer des μ-Kalküls, der nur aussagenlogische Variable zuläßt. Vektoren sind nur insofern vorhanden, als daß sie syntaktisch als boolesche Variable interpretiert werden. Man kann einer booleschen Variablen z. B. den *Namen* „`a[2]`" geben. Deshalb gibt es auch keine Allokationskonzepte wie Verschränken oder Blocken. Nur durch das syntaktische Vorkommen wird eine initiale Ordnung vorgegeben.

Diese Ordnung kann während der Modellprüfung dynamisch nach /Rudell, 1993/ verändert werden. Dies kann zwar einerseits die BDDs z. B. der Übergangsrelation verkleinern, hat aber den unerwünschten Nebeneffekt, daß bei Substitutionen im Rumpf von rekursiven Prädikatsvariablen fast nie schnelle Substitutionen verwendet werden können. Dieselbe Vorgehensweise lag schon dem ersten Prototyp der μcke zu Grunde, der in der Arbeit /Melcher, 1995/ verwendet wurde, und im Vergleich zum SMV-System sehr schlecht abschnitt. Dies zeigt auch der folgende Vergleich der neuen μcke mit dem System von Janssen.

Da das System von Janssen kein Typsystem besitzt, wurden nur zwei einfache Beispiele verglichen. Einmal ein binärer n-Bit-Zähler, der bei der hier verwendeten uneffektiven Vorwärtsanalyse genau 2^n Iterationen für die Erreichbarkeitsanalyse benötigt, und zweitens ein weiteres einfaches Protokoll zum gegenseitigen Ausschluß (arbiter), welches bei den Quellen des Systems als Beispiel mitgeliefert wurde. Siehe auch Abschnitt C.7 und Abschnitt C.8.

Der Vergleich in Tab. 5.5 der Erreichbarkeitsanalyse bei beiden Beispielen fällt deutlich zugunsten der μcke aus. Wie schon erwähnt resultiert dies aus dem Allokationskonzept der μcke, wodurch in den Fixpunktiterationen, wenn immer möglich, monotone Substitutionen verwendet werden, so daß sich die BDDs unter der Substitution nicht vergrößern.

[5] Genauere Analysen ließen sich nicht durchführen, da die Systeme dem Autor nicht zur Verfügung standen.

5.5 Zusammenfassung

Es wurde ein Entwurf eines Modellprüfers vorgestellt, der durch Verwendung von objektorientierten Methoden leicht erweiterbar ist. Dazu mußten spezifische Entwurfsmuster entwickelt werden. Es wurde an Hand einer Implementierung gezeigt, daß sich dadurch vielfältige Anforderungen, wie dynamisches Austauschen der zu Grunde liegenden Repräsentation, erfüllen lassen.

Die Implementierung in Form des Modellprüfers μcke erwieß sich als ebenso effizient wie die von Spezialmodellprüfern (z. B. SMV) und bei weitem leistungsfähiger als vergleichbare Systeme. Somit wurde die These dieser Arbeit, daß sich die Modellprüfung im μ-Kalkül effizient durchführen läßt, auch praktisch nachgewiesen.

Die vorgestellten Optimierungen können durch ihre Übertragung auf den μ-Kalkül nun auch für viele Verifikationsaufgaben eingesetzt werden, für die sie ursprünglich nicht gedacht waren. Am Beispiel der Vorwärtsanalyse für die Bisimulation wurde dies vorgeführt (Abschnitt 5.4.2).

5.6 Ausblick

Es gab einen Einsatz der μcke im industrienahen Umfeld /Philipps und Scholz, 1997/. Hier wurde die μcke als Modellprüfer gewählt, weil für die Systembeschreibungssprache „μ-Charts“ der μ-Kalkül benötigt wurde. Diese Arbeiten werden fortgeführt und werden auch umgekehrt Einfluß auf die weitere Entwicklung der μcke haben. So ergab sich in diesem Zusammenhang die Forderung nach varianten Typen und, daß es möglich sein sollte, Repräsentationen von Prädikaten zu speichern. Beide Erweiterungen werden in einer zukünftigen Version der μcke berücksichtigt und sollten durch die objektorientierte Architektur leicht zu integrieren sein.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde gezeigt, daß sich Modellprüfung des μ-Kalküls effizient durchführen läßt. Dies wurde einmal möglich durch die Übertragung wesentlicher Optimierungen der Modellprüfung für eingeschränkte Logiken auf den μ-Kalkül. Hierunter fallen die Vorwärtsanalyse und die „frontier set simplification“. Daneben wurden spezielle Optimierungen vorgestellt, wie Abhängigkeitsanalyse und die automatische Variablenallokation. Insbesondere das Verfahren zur automatischen Variablenallokation ermöglicht es, daß BDD-Algorithmen effizient eingesetzt werden können.

Durch Einführung eines Namenskonzeptes gelang es, den μ-Kalkül anwenderfreundlicher zu gestalten. Hierfür mußte die Wohldefiniertheit der Semantik neu gezeigt werden, was formal durchgeführt wurde. In diesem Zusammenhang wurde auch eine formale Herleitung der symbolischen Modellprüfung bzw. der Modellprüfung mit BDDs aus der Standardsemantik angegeben.

Die Grundlage für den Erfolg der Modellprüfung bilden die BDDs. Hierzu wurden in dieser Arbeit auf einheitliche Weise alle wesentlichen BDD-Algorithmen aufgelistet, die zuvor zum Teil nur in Form von Programmtext vorlagen. Die Korrektheit dieser Algorithmen konnte formal gezeigt werden, was ein Verdienst der abstrakten Beschreibung unter Verwendung der SCAM ist. Schließlich wurde der neue BDD-Algorithmus ${}_{\mathtt{c}}\mathtt{ite}_{\exists}$ vorgestellt, der als Spezialalgorithmus die Effizienz der Modellprüfung des μ-Kalküls steigert.

Der Modellprüfer μcke wurde dazu benutzt, die theoretischen Resultate zu evaluieren. Er erwies sich als durchaus vergleichbar mit höchst effizienten Spezialmodellprüfern wie dem SMV und ist wesentlich leistungsfähiger als die aus der Literatur bekannten Modellprüfer für den μ-Kalkül.

Anhang A

Grundlagen

In diesem Kapitel werden die verwendeten mathematischen Begriffe definiert. Wenn nicht anders angegeben sei dabei /Jacobson, 1985/ die letzte Referenz.

A.1 Natürliche Zahlen und Potenzmengen

In dieser Arbeit werde unter *natürlichen Zahlen* immer die Menge $\mathbb{N} := \{0, 1, 2, 3, \ldots\}$ verstanden. Das heißt die „0“ ist immer in $\mathbb{N}$ schon enthalten. Wenn explizit nur die *positiven* natürlichen Zahlen gemeint sein sollen dann wird $\mathbb{N} \backslash \{0\}$ geschrieben. Für jede beliebige Menge A sei $\mathbb{P}(A) := \{A' \mid A' \subseteq A\}$ definiert als die *Potenzmenge* von A. Diese besitzt als „kleinstes“ Element immer die *leere Menge* geschrieben als $\emptyset$.

A.2 Relationen

Eine Relation *zwischen* einer Menge A und einer Menge B ist eine Teilmenge von $A \times B$, dem kartesischen (direkten) Produkt von A und B. Oft ist in dieser Arbeit von Relationen *über* einer Menge A die Rede. Darunter versteht man eine Relation zwischen A und sich selbst ($\subseteq A^2$). Z.B. ist die *Diagonale* $\Delta = \Delta(A)$ Über einer Menge A definiert als eine Relation Δ über A mit

$$\Delta = \Delta(A) = \{(a, a) \mid a \in A\}$$

Relationen werden mit der Mengeninklusion geordnet und neben den Mengenoperationen, „$\cup$“, „$\cap$“ und „$\backslash$“, werden folgende Operationen eingeführt:

Definition A.1 (Operatoren über Relationen) *Sei $R \subseteq A \times B$, $S \subseteq A^2$ und $T \subseteq B \times C$*

$$
\begin{array}{lll}
R^{-1} & := \{(b,a) \in B \times A \mid (a,b) \in R\} & \textit{Inversion} \\
R \cdot T & := \{(a,c) \in A \times C \mid \exists\, b \in B.\, (a,b) \in R \wedge (b,c) \in T\} & \textit{Produkt} \\
S^{*} & := \{(a,a') \in A^2 \mid \exists\, a_0, \ldots, a_n \in A.\, \forall\, 0 \leq i < n. & \\
 & \qquad a = a_0 \wedge a_n = a' \wedge (a_i, a_{i+1}) \in S\} & \textit{reflexiv-transitive Hülle} \\
S^{+} & := \{(a,a') \in A^2 \mid \exists\, a_0, \ldots, a_n \in A.\, \forall\, 0 \leq i < n. & \\
 & \qquad n > 0, a = a_0 \wedge a_n = a' \wedge (a_i, a_{i+1}) \in S\} & \textit{transitive Hülle}
\end{array}
$$

Statt „$R \cdot T$“ wird auch häufig die Juxtaposition *RT verwendet und $S \cup \Delta(A)$ wird als reflexive Hülle bezeichnet.*

Definition A.2 (Eigenschaften von Relationen) *Eine Relation R über A heiße*

reflexiv	:⇔	$\Delta(A) \subseteq R$	*antisymmetrisch*	:⇔	$(R \cap R^{-1}) \subseteq \Delta(R)$
irreflexiv	:⇔	$\Delta(A) \cap R = \emptyset$	*transitiv*	:⇔	$R^{+} \subseteq R$
symmetrisch	:⇔	$R^{-1} \subseteq R$	*totalgeordnet*	:⇔	$(R \cup R^{-1}) = A^2$

In manchen Büchern werden zur Definition der Eigenschaften nicht die Operationen auf Relationen verwendet, sondern es wird direkt formuliert:

R reflexiv	gdw.	$(a,a) \in R$	, $\forall a \in A$
R irreflexiv	gdw.	$(a,a) \notin R$	, $\forall a \in A$
R symmetrisch	gdw.	$(a,b) \in R \Rightarrow (b,a) \in R$	, $\forall a,b \in A$
R antisymmetrisch	gdw.	$(a,b) \in R \wedge (b,a) \in R \Rightarrow a = b$	, $\forall a,b \in A$
R transitiv	gdw.	$(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$	, $\forall a,b,c \in A$
R totalgeordnet	gdw.	$(a,b) \in R \vee (b,a) \in R$	, $\forall a,b \in A$

Die obige Definition hat den Vorteil, daß sich die Namen der Hüllen Operationen aus Definition A.1 unmittelbar rechtfertigen lassen. Z.B. ist $R^{+} \subseteq R^{+}$, so daß R^{+} sich als transitiv erweißt. Eine häufig benötigte Tatsache ist

Lemma A.3 (Minimalität der Transitiven Hülle)
Für $R \subseteq A^2$ ist R^{+} (bzw. R^{*}) die kleinste (reflexiv-) transitive Relation $\supseteq R$.

Definition A.4 (Äquivalenzrelation und Partielle Ordnung)
Eine transitive Relation $R \subseteq A^2$ heißt

Äquivalenzrelation :⇔ { *(a) R reflexiv*; *(b) R symmetrisch* }

Partielle Ordnung :⇔ { *(a) R reflexiv*; *(b) R antisymmetrisch* }

Totalordnung *(oder lineare Ordnung)* :⇔ { *(a) R partielle Ordnung*; *(b) R totalgeordnet* }

Jede partielle Ordnung läßt sich „Totalordnen“, aber leider nicht eindeutig.

Lemma A.5 (Topologisches Sortieren) *Eine partielle Ordnung R auf einer endlichen Grundmenge läßt sich auf konstruktive Weise zu einer Totalordnung S erweitern.*

Beweis: Der Algorithmus hierzu findet sich in fast allen Einführungsbüchern über Algorithmen wie z. B. /Manber, 1989/. ❑

A.3 Partielle Abbildungen und Substitutionen

In dieser Arbeit werden an vielen Stellen Abbildungen (Funktionen) definiert. Wenn nichts anderes gesagt, sind damit immer totale Funktionen gemeint. Da diese aber ein Spezialfall der partiellen Funktionen darstellen, werden die folgenden Definitionen gleich für partielle Funktionen gegeben.

Definition A.6 (Partielle Funktion) *Seien A, B beliebige Mengen, so bezeichne*

$$[A \to B]$$

die Menge der partiellen Funktionen von A nach B, mit der Schreibweise

$$f\colon A \to B \quad :\Leftrightarrow \quad f \in [A \to B]$$

Für so ein f gelte

$$f \subseteq A \times B \quad \textit{und} \quad \forall x \in A,\ y,z \in B.\quad (x,y) \in f \wedge (x,z) \in f \quad \Rightarrow \quad y = z$$

Wegen dieser Eindeutigkeit schreibe auch $f(x) = z$ statt $(x,z) \in f$. Eine weitere Schreibweise ist $x \mapsto z$, wenn klar ist, daß sich ‚$\mapsto$' auf f bezieht. Wenn es für ein $x \in A$ ein $z \in B$ gibt mit $f(x) = z$, so heiße f für x definiert, geschrieben $f(x)\downarrow$. Falls es ein solches z nicht gibt, dann nenne man f undefiniert für x, geschrieben $f(x)\uparrow$. Daraus leiten sich folgende Begriffe ab:

Definition A.7 (range und domain) *Für eine partielle Funktion $f\colon A \to B$ definiere*

$$\mathsf{range}(f) := \{z \in B \mid \exists x \in A.\, f(x) = z\} \quad \textit{und} \quad \mathsf{domain}(f) := \{x \in A \mid f(x)\downarrow\}$$

Dabei heiße $\mathsf{domain}(f)$ Definitionsmenge *und* $\mathsf{range}(f)$ *die* Bildmenge *von f.*

„Substitution“ ist nur ein anderes Wort für „partielle Funktion“. Es wird immer dann gebraucht, wenn zwei partielle Funktionen auf besondere Art verknüpft werden.

Definition A.8 (Verknüpfungen von partiellen Funktionen)
Für $g \in [A \to B]$, $f \in [B \to C]$ ist die Komposition definiert als

$$f \circ g \in [A \to C] \quad \textit{mit} \quad (f \circ g)(x) = z \quad :\Leftrightarrow \quad \exists y \in B.\, g(x) = y \wedge f(y) = z$$

Für eine Substitution (partielle Funktion) $h \in [A \to B]$ definiere

$$g \cdot h = gh \in [A \to B] \quad \textit{mit} \quad (gh)(x) = z \quad :\Leftrightarrow \quad \begin{cases} g(x) = z, & \textit{falls} \quad h(x)\uparrow \\ h(x) = z, & \textit{falls} \quad h(x)\downarrow \end{cases}$$

Man beachte, daß ‚$\circ$' rechtsassoziativ und ‚$\cdot$' linksassoziativ ist.

Die Juxtaposition von g und h ist nun mehrdeutig, da g und h als Relationen aufgefaßt werden könnten. Da „$\circ$“ aber nur eine andere Schreibweise für das Relationale Produkt darstellt, kann man dieses falls nötig mit „$\circ$“ ausdrücken. Weiter werden in dieser Arbeit partielle Funktionen nicht als Relationen betrachtet, und so sollte keine Verwechslungsgefahr bestehen.

Für Substitutionen wird oft die *Mengenschreibweise* für partielle Funktionen verwendet:

$$\{x \mapsto z \mid f(x) = z\}$$

So bezeichnet z. B. $\{1 \mapsto 2\}$ in $[\mathbb{N} \to \mathbb{N}]$ die partielle Funktion $f\colon \mathbb{N} \to \mathbb{N}$, mit

$$\mathsf{domain}(f) = \{1\}, \quad \mathsf{range}(f) = \{2\}, \quad f(1) = 2.$$

Sei nun $g\colon \mathbb{N} \to \mathbb{N}$ mit $g(i) := i$ $(g = \mathsf{id})$, so gilt

$$(gf)(i) = i, \text{ für } i \neq 1 \quad \text{und} \quad (gf)(1) = 2$$

In diesem Fall kann man sehr viel Schreib- und Definitionsarbeit sparen, wenn man $g\{1 \mapsto 2\}$ statt gf schreibt. Ebenso kann dies für Substitutionen mit größerer Definitionsmenge geschehen, wie z. B. in

$$\mathsf{id}\{2 \cdot i \mapsto 2 \cdot i + 1 \mid i \in \mathbb{N}\} \quad = \quad 2 \cdot \lceil (\mathsf{id} + 1)/2 \rceil - 1$$

Manchmal werden in der Mengenschreibweise im Zusammenhang mit Substitutionen auch Vektoren verwendet. So gelte für $\underline{a} \in A^n$, $\underline{b} \in B^n$ und $f \in [A \to B]$

$$[A \to B] \quad \ni \quad f\{\underline{a} \mapsto \underline{b}\} \quad := \quad f\{\underline{a}(i) \mapsto \underline{b}(i) \mid 1 \leq i \leq |a|\} \tag{A.1}$$

Dies macht nur dann Sinn, wenn man fordert, daß die einzelnen Komponenten von $\underline{a}$ paarweise verschieden sind ($\underline{a}(i) = \underline{a}(j) \;\Rightarrow\; i = j$), da sonst die resultierende Menge keine partielle Funktion mehr darstellt.

Definition A.9 (Totale Funktion) *Unter einer* totalen *Funktion f (kurz einfach nur Funktion oder Abbildung genannt) versteht man eine partielle Funktion $f \in [A \to B]$ mit $\mathsf{domain}(f) = A$.*

Wie schon mehrmals erwähnt, soll der Begriff Funktion alleine für eine totale Funktion stehen, da in der Regel die Verwendung einer partiellen Funktion anstatt einer totalen näherer Erläuterung bedarf.

Definition A.10 (Surjektive und injektive (partielle) Funktionen)
Eine partielle Funktion $f\colon A \to B$ heißt

$$\begin{array}{rcl} \textit{injektiv} & :\Leftrightarrow & \forall a, b \in \mathsf{domain}(f).\ f(a) = f(b) \Rightarrow a = b \\ \textit{surjektiv (auf } B) & :\Leftrightarrow & \mathsf{range}(f) = B \\ \textit{bijektiv (auf } B) & :\Leftrightarrow & f \textit{ injektiv und surjektiv (auf } B) \end{array}$$

A.4 Verbände und Fixpunkte

In diesem Abschnitt findet man, was für die Arbeit an Grundlagen für Verbände und Fixpunkte gebraucht wird. Die Hauptquelle für diese Definitionen und Sätze bildet /Schmitt, 1992b/, /Winskel, 1993/ und natürlich /Tarski, 1955/.

Definition A.11 (Schranken) *Bez. einer partiellen Ordnung $\leq \subseteq A^2$ (nun in infix-Schreibweise verwendet) heiße $a \in A$ für eine Menge $B \subseteq A$ eine*

$$\begin{array}{lll}
\textit{obere Schranke}\ (B \leq a) & :\Leftrightarrow & \forall b \in B.\ b \leq a \\
\textit{untere Schranke}\ (a \leq B) & :\Leftrightarrow & \forall b \in B.\ a \leq b \\
\textit{Supremum}\ (a \in \mathsf{sup}\, B) & :\Leftrightarrow & B \leq a \textit{ und } \forall c \in A.\ B \leq c \wedge c \leq a \ \Rightarrow\ c = a \\
\textit{Infimum}\ (a \in \mathsf{inf}\, B) & :\Leftrightarrow & a \leq B \textit{ und } \forall c \in A.\ c \leq B \wedge a \leq c \ \Rightarrow\ c = a \\
\textit{Maximum}\ (a \in \mathsf{max}\, B) & :\Leftrightarrow & a \in \mathsf{sup}\, B \textit{ und } a \in B \\
\textit{Minimum}\ (a \in \mathsf{min}\, B) & :\Leftrightarrow & a \in \mathsf{inf}\, B \textit{ und } a \in B
\end{array}$$

(Im letzten Fall spricht man auch üblicherweise von einem Minimum von *B.)*

Definition A.12 (Verbände) *Eine partielle Ordnung $\leq \subseteq A^2$ heißt*

$$\textit{unterer Halbverband} \quad :\Leftrightarrow \quad \forall \emptyset \subsetneq B \subseteq A.\ |B| < \infty \ \Rightarrow\ |\mathsf{inf}\, B| = 1 \tag{A.2}$$

$$\textit{oberer Halbverband} \quad :\Leftrightarrow \quad \forall \emptyset \subsetneq B \subseteq A.\ |B| < \infty \ \Rightarrow\ |\mathsf{sup}\, B| = 1 \tag{A.3}$$

Ein oberer und unterer Halbverband heißt Verband. *Wenn die rechte Seite der Gleichung $(A.2)$ bzw. $(A.3)$ für beliebige $\emptyset \subsetneq B \subseteq A$ gilt, so heißt $(A, \leq)$ ein* vollständiger *unterer (oberer) Halbverband und analog vollständiger Verband, wenn beide Gleichungen gelten.*

In einem unteren (oberen) Halbverband bezeichne man das zu einer endlichen Menge $B \subseteq A$ eindeutige Infimum c als das *Infimum (Supremum) geschrieben $\mathsf{inf}\, B = c$ ($\mathsf{sup}\, B = c$). Ebenso für vollständige Halbverbände und beliebiges (nichtendliches) B. Für $B = A$ definiere*

$$\bot := \mathsf{inf}\, A \qquad \textit{und} \qquad \top := \mathsf{sup}\, A$$

Eine Teilmenge B eines vollständigen Verbandes A heiße Kette *gdw.*

$$\mathsf{sup}\, B' \in B, \quad \textit{für alle } B' \subseteq B \textit{ mit } |B'| < \infty$$

In einem Verband A wird das Infimum bzw. das Supremum von $a, b \in A$ auch als

$$a \cap b := \mathsf{inf}\{a, b\}, \qquad a \cup b := \mathsf{sup}\{a, b\}$$

geschrieben und als „meet“ bzw. „join“ bezeichnet. Für $B \subseteq A$ schreibt man auch

$$\bigcap B := \mathsf{inf}\, B, \qquad \bigcup B := \mathsf{sup}\, B$$

wobei bei $|B| = \infty$ und nicht vollständigen A diese nicht zu existieren brauchen.

Definition A.13 (Monotone und Stetige Abbildungen auf Verbänden)
Eine Abbildung $f\colon A \to A$ auf einem vollständigen Verband $(A, \leq)$ heiße

$$\begin{array}{lll} \textit{monoton} & :\Leftrightarrow & \forall a,b \in A.\ a \leq b \Rightarrow f(a) \leq f(b) \\ \textit{stetig} & :\Leftrightarrow & \forall B \subseteq A,\ B\ \textit{Kette}.\ \sup f(B) = f(\sup B) \end{array}$$

Definition A.14 (Fixpunkte) *Für $\leq \subseteq A^2$, $f\colon A \to A$ und $a \in A$ gelte*

$$a \textit{ ist ein Fixpunkt für } f \quad :\Leftrightarrow \quad f(a) = a$$

Darüber hinaus heißt a kleinster (größter) Fixpunkt, wenn a Fixpunkt ist und zusätzlich für alle Fixpunkte $b \in A$ mit $b \leq a$ ($b \geq a$) schon $a = b$ gilt.

Satz A.15 (Fixpunkte monotoner Funktionen) *Sei $(A, \leq)$ ein vollständiger Verband $(A, \leq)$, $f\colon A \to A$ eine monotone Funktion, so existiert der kleinste Fixpunkt m und der größte Fixpunkt M. Sie lassen sich berechnen als*

$$M = \bigcup \{m \in A \mid m \leq f(m)\} \qquad m = \bigcap \underbrace{\{m \in A \mid f(m) \leq m\}}_{:=B}$$

Beweis: Zunächst ist $B \neq \emptyset$, da $f(\top) \leq \top$ und somit $\top \in B$, so daß $m := \bigcap B$ wohldefiniert ist. Z.Z. ist nun, daß m kleinster Fixpunkt ist. Nun sei $b \in B$ beliebig gewählt. Nach Definition von m folgt sofort $m \leq b$. Weiter liefert die Monotonie von f die Ungleichung $f(m) \leq f(b) \leq b$, letzteres da $b \in B$. Somit ist $f(m)$ auch eine untere Schranke von B, von denen m die größte ist, so daß $f(m) \leq m$. Damit erweist sich $m \in B$. Wiederum mit der Monotonie von f ergibt die letzte Ungleichung $f(f(m)) \leq f(m)$. Dies ergibt $f(m) \in B$ und schließlich $m \leq f(m)$. Die Antisymmetrie von $\leq$ zeigt nun, daß m tatsächlich ein Fixpunkt ist.

Weiterhin ist die Menge der Fixpunkte eine Untermenge von B, da aus $f(a) = a$ sofort $f(a) \leq a$ folgt. Damit ist der kleinste Fixpunkt, der in B liegt, nämlich m, auch ein globaler kleinster Fixpunkt. Analog verfährt man bei M. ❑

Interessant an diesem Beweis (dieser ist an den aus /Schmitt, 1992b/ angelehnt) ist, daß man meinen würde es reiche aus, für die Existenz eines kleinsten Fixpunktes nur untere Halbverbände zu betrachten. Dann aber ist die Wohldefiniertheit des Schnittes $\bigcap B$ nicht notwendigerweise erfüllt, da ein vollständiger unterer Halbverband nicht unbedingt ein größtes Element ($\top$) besitzen muß. Umgekehrt besagt der nächste Satz, daß jeder vollständige untere Halbverband mit größtem Element schon ein Verband ist. Dies verdeutlicht, daß die Voraussetzung des vollständigen Verbandes recht scharf gewählt ist. Ein weiteres Indiz dafür liefert das Beispiel der natürlichen Zahlen $\mathbb{N}$ mit der natürlichen Ordnung. Diese bilden genau so einen vollständigen unteren Halbverband, der *kein* vollständiger Verband ist, da $\mathbb{N}$ selbst keine obere Schranke besitzt. Die Nachfolgeoperation

$$f: \mathbb{N} \to \mathbb{N}, \qquad f: n \mapsto n+1$$

ist offensichtlich monoton und hat keinen Fixpunkt.

Satz A.16 (Kriterium für die Vollständigkeit von Verbänden)
Jeder vollständige untere (obere) Halbverband mit einem Maximum (Minimum) ist ein vollständiger Verband.

Beweis: Sei $(A, \leq)$ ein vollständiger unterer Halbverband mit Maximum $M = \mathsf{max}A$. Sei nun $B \subseteq A$ eine beliebige Menge. Z.Z. ist, daß $|\mathsf{sup}B| = 1$. Dazu betrachte $C := \{a \in A \mid B \leq a\}$, die Menge der oberen Schranken von B. Diese ist nicht leer, da $M \in C$, und nach Voraussetzung existiert $c := \bigcap C$. Für ein $b \in B$ gilt $b \leq d$ für alle $d \in C$, so daß $b \leq C$. Da nun c größte untere Schranke für C ist, muß also $b \leq c$ gelten, womit $c \geq B$. Sei nun $c' \in A$ mit $c' \geq B$, dann ist $c' \in C$ und somit $c \leq c'$. Damit ist $c \in \mathsf{sup}B$. Ebenso ist für $c' \in \mathsf{sup}B$ auch $c' \geq B$, so daß $c' \in C$ und somit $c \leq c'$. Nach Definition von „sup“ folgt schließlich $c = c'$. (Analog für obere Halbverbände) ❑

Die Wortwahl in diesem Satz ist sehr sorgfältig gewählt. So bedeutet die Konklusion, daß der vollständige untere (obere) Halbverband sogar ein vollständiger oberer (unterer) Halbverband ist! Insbesondere zeigt der Beweis

Korollar A.17 *Für einen vollständigen unteren bzw. oberen Halbverband $(A, \leq)$ gilt*

$$|\mathsf{sup}B| \leq 1 \quad \textit{bzw.} \quad |\mathsf{inf}B| \leq 1, \qquad \textit{für alle } B \subseteq A$$

d. h. das Supremum bzw. Infimum ist eindeutig, wenn es existiert.

Dies zeigt, daß ein vollständiger (z. B. endlicher) unterer Halbverband A nicht deshalb kein Verband sein kann, weil es zwei Elemente $a, b \in A$ gibt, die kein eindeutiges Supremum haben ($|\mathsf{sup}\{a, b\}| \geq 2$), sondern es muß zwei Elemente $a, b \in A$ geben, die überhaupt kein Supremum besitzen.

Am obigen Beispiel $(\mathbb{N}, \leq)$ soll eine Vorgehensweise geschildert werden, wie man von einem vollständigen unteren Halbverband zu einem vollständigen Verband gelangt ($\mathbb{N}$ ist zwar schon ein Verband, aber eben nicht ein vollständiger oberer Halbverband). Hierzu fügt man ein Maximum ∞ hinzu und definiert $\infty \geq n$ für alle $n \in \mathbb{N}^\infty := \mathbb{N} \cup \{\infty\}$. Damit bleibt $\mathbb{N}^\infty$ vollständiger unterer Halbverband. Zusätzlich ist $\mathbb{N}^\infty$ aber nun nach letztem Satz ein vollständiger oberer Halbverband. Diese Betrachtungen lassen sich verallgemeinern zu:

Satz A.18 (Vervollständigung eines vollständigen unteren Halbverbandes)
Sei $(A, \leq)$ ein vollständiger unterer Halbverband. Dann ist $B := A \dot\cup \{M\}$ ein vollständiger Verband, wenn man definiert

$$b \leq_B b' \quad :\Leftrightarrow \quad b' = M \textit{ oder } (b, b' \in A \textit{ und } b \leq_A b')$$

für alle $b, b' \in B$ und neuem $M \notin A$.

Es soll nicht verschwiegen werden, daß diese Konstruktion nicht unbedingt den kleinsten Verband liefert. Wenn z. B. der untere Halbverband schon ein einziges maximales Element besitzt, so kann man durch die Vereinbarung, daß alle Paare ohne Supremum diese Maximum als Supremum bekommen, einen kleineren Verband konstruieren.

Satz A.19 (Fixpunkte stetiger Funktionen)
Sei $(A, \leq)$ ein vollständiger Verband, $f : A \to A$ eine stetige Funktion, dann ist f monoton und für die kleinsten und größten Fixpunkte gilt

$$\begin{aligned} \bigcap\{a \in A \mid f(a) = a\} &= \bigcup_{i=0}^{\infty} f^i(\bot) \\ \bigcup\{a \in A \mid f(a) = a\} &= \bigcap_{i=0}^{\infty} f^i(\top) \end{aligned} \tag{A.4}$$

Beweis: Sei $a, b \in A$ mit $a \leq b$, so ist $B := \{a, b\}$ eine Kette, da $a \cup b = b$. Die Stetigkeit von f ergibt nun

$$f(a) \leq \mathsf{sup}\{f(a), f(b)\} = \mathsf{sup}\, f(B) = f(\mathsf{sup}\, B) = f(b)$$

Damit ist f monoton und der kleinste Fixpunkt m existiert nach Satz A.15. Da A vollständig ist, existiert die rechte Seite von $(A.4)$. Z.Z. ist noch, daß m gleich dieser rechten Seite $R := \bigcup f^i(\bot)$ ist. Die Menge $\{f^i(\bot)\}$ ist linear geordnet

$$f^{i+1}(\bot) \geq f^i(\bot)$$

Dies zeigt eine einfache Induktion nach i. Im Induktionsschritt von i nach $i+1$ gilt

$$f^{i+2}(\bot) = f(f^{i+1}(\bot)) \geq f(f^i(m)) = f^{i+1}(m)$$

mit Induktionshypothese und Monotonie von f. Insbesondere ist $\{f^i(\bot)\}$ eine Kette und mit der Stetigkeit von f gilt

$$f(R) = f(\bigcup f^i(\bot)) = \bigcup f(f^i(\bot)) = \bigcup f^{i+1}(\bot) = R$$

Somit ist R ein Fixpunkt von f. Ebenso durch einfache Induktion kann $f^i(\bot) \leq m$ gezeigt werden. Der Induktionsanfang ist trivial und den Induktionsschritt bestätigt

$$f^{i+1}(\bot) = f(f^i(\bot)) \leq f(m) = m,$$

wiederum nach Induktionshypothese und Anwendung der Monotonie von f. Damit ist m eine obere Schranke aller $f^i(\bot)$ und somit gilt für R als kleinste all dieser oberen Schranken $R \leq m$. Da R Fixpunkt ist und m der kleinste Fixpunkt, folgt daraus $R = m$. (Analog für den größten Fixpunkt) ❑

Der folgende Satz stellt ein hinreichendes Kriterium dar, um die Beziehung zwischen den Fixpunkten zweier stetiger Funktionen zu bestimmen.

Satz A.20 *Für einen vollständigen Verband $(A, \leq)$, stetige Funktionen $f, g : A \to A$ gilt*

$$\bigcup_{i=0}^{\infty} f^i(\bot) \leq \bigcup_{i=0}^{\infty} g^i(\bot) \qquad \bigcap_{i=0}^{\infty} f^i(\top) \leq \bigcap_{i=0}^{\infty} g^i(\top)$$

falls $f(a) \leq g(a)$ für alle $a \in A$.

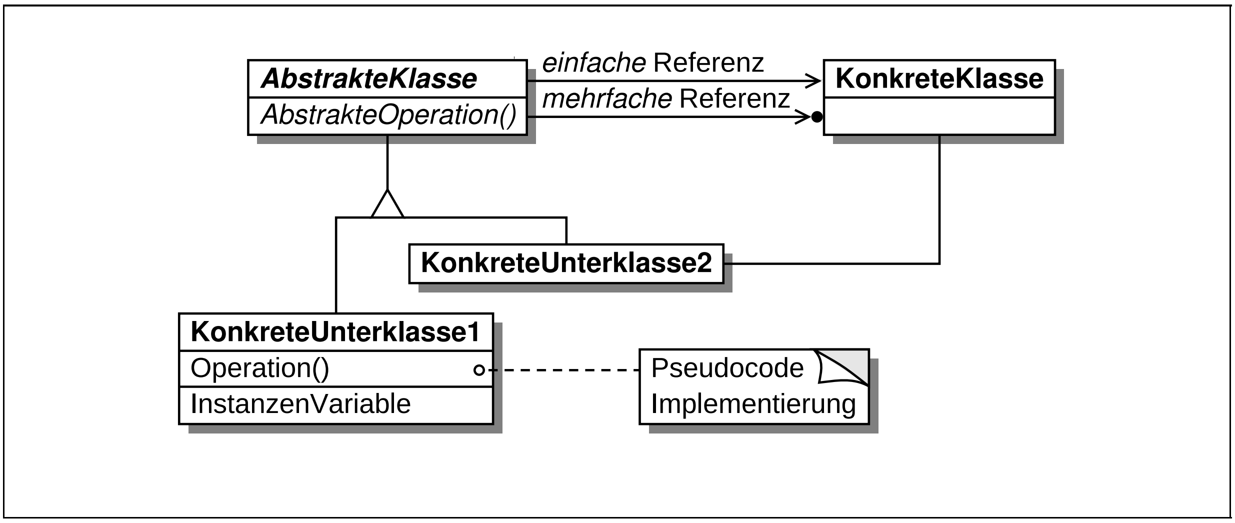


Abbildung A.1: Legende zu Diagrammen von Klassen.

Beweis: Seien $a, b \in A$ mit $a \leq b$. Dann ist

$$f(a) \leq f(b) \leq g(b)$$

nach Voraussetzung und da f monoton ist. Zeige nun durch Induktion über $i \in \mathbb{N}$, daß

$$f^i(a) \ \leq \ g^i(b)$$

Der Induktionsanfang ist trivial. Im Induktionsschritt gilt

$$f^{i+1}(a) = f^i(f(a)) \leq g^i(g(b)) = g^{i+1}(b)$$

und man erhält $f^i(\bot) \leq g^i(\bot) \leq \bigcup g^i(\bot)$. Somit ist auch der größte Fixpunkt von g eine obere Schranke aller $f^i(\bot)$ und nach Definition des Supremums folgt die Aussage für den kleinsten Fixpunkt. Der größte Fixpunkt wird genauso behandelt. ❑

A.5 Diagramme für Klassen

In dieser Arbeit wird eine einfache Version der Notation für die Beschreibung der Beziehung von Klassen aus /Gamma et al., 1995/ verwendet. Abbildung A.1 stellt eine Legende für die in dieser Notation verwendeten graphischen Elemente dar.

Die „Kästen“ stellen die einzelnen Klassen dar. Diese sind in drei Bereiche unterteilt. An der obersten Stelle steht der Klassenname in fetter Schrift. Darunter folgen die Methoden eines Objektes dieser Klasse in normaler Schrift. Als Methoden sind die Bezeichner mit Klammern und eventuell Argumenten versehen. Dies unterscheidet die Methoden von den Bezeichnern im dritten Bereich, die die Individuenvariablen einer Instanz (=Objekt) darstellen.

Abstrakte Klassen, die man als abstrakte Schnittstelle ansehen kann, werden kursiv dargestellt. Von abstrakten Klassen können selbst keine Instanzen gebildet werden. Dies geht nur von konkreten Unterklassen. Unterklassen werden mit ihrer (direkten) Oberklasse durch eine Linie und einem Dreieck verbunden. Die andere Beziehung zwischen Klassen besteht aus Referenzen (in dieser Arbeit wird keine Aggregation verwendet) in Individuenvariablen eines Objekt zu einem Objekt einer anderen Klasse.

Die Referenz wird durch einen Pfeil zwischen den Klassen gekennzeichnet. Wenn die Individuenvariable, die die Referenz beherbergt im Diagramm genannt wird, so nimmt entweder der Pfeil seinen Ursprung an der Position des Bezeichners der Variable im dritten Teil des Kastens und ist unbeschriftet, oder aber die Individuenvariable wird erst gar nicht in den Kasten eingetragen und die Beschriftung des Pfeils stellt den Bezeichner der Variable dar. Zum Schluß besteht noch die Möglichkeit die Implementierung einer Methode in Pseudocode anzugeben. Dieser Pseudocode ist in dieser Arbeit immer an C++ (/Stroustrup, 1991/) angelehnt.

A.6 O-Notation

Die O-Notation wird wie üblich in der Literatur definiert. Für diesen Abschnitt seien die betrachteten Funktionen alle total.

Definition A.21 **(O, Ω)** *Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ sei*

$$\begin{aligned}
\mathrm{O}(f) &:= \{g \in [\mathbb{N} \to \mathbb{N}] \mid \textit{es gibt } c, n \in \mathbb{N} \textit{ mit } g(m) \leq c \cdot f(m), \textit{ für alle } m \geq n\} \\
\Omega(f) &:= \{g \in [\mathbb{N} \to \mathbb{N}] \mid \textit{es gibt } c, n \in \mathbb{N} \textit{ mit } c \cdot g(m) \geq f(m), \textit{ für alle } m \geq n\}
\end{aligned}$$

Anhang B

Ergänzungen

B.1 Fakten zu $\mathbb{B}_\mu^V$

Haben zwei Belegungen für jede Variable äquivalente Bilder, so ist die Auswertung eines beliebigen Termes zu einem booleschen Ausdruck (dessen Semanik) unter diesen beiden Belegungen dieselbe modulo semantischer Äquivalenz. Dies ist Thema der nächsten Definition und des nächsten Satzes, der benötigt wird, um die Fixpunktiterationen mit BDDs berechnen zu können.

Definition B.1 (Äquivalente Belegungen)
Zwei Belegungen $\rho_P^1, \rho_P^2 \in [P \to \mathbb{B}^W]$ *heißen* äquivalent *genau dann, wenn*

$$\rho_P^1(X)\downarrow \Leftrightarrow \rho_P^2(X)\downarrow \quad \textit{und} \quad \rho_P^1(X) \equiv \rho_P^2(X), \textit{ für } \rho_P^2(X)\downarrow$$

für alle $X \in P$.

Satz B.2 *Für zwei äquivalente Belegungen* $\rho_P^1, \rho_P^2 \in [P \to \mathbb{B}^W]$ *gilt*

$$\langle\!\langle s \rangle\!\rangle \rho_P^1 \equiv \langle\!\langle s \rangle\!\rangle \rho_P^2$$

für alle $s \in T_\mu^V$.

Beweis: Der Beweis wird geführt durch Induktion über den Termaufbau von s. Für die einfachen Basisfälle ($s = u[i]$ oder $s = u \doteq v$) ergeben sich auf beiden Seiten dieselben booleschen Ausdrücke. Auch der Induktionsschritt für $s = \exists v.\, t$, $s = s_0 \wedge s_1$, $s = \neg t$ oder $s = X(\underline{u})$ für $\sigma(X) = \varepsilon$ ergibt sich unmittelbar aus der Induktionsbehauptung.

Mit der Beobachtung, daß bei $s = X(\underline{u})$ mit $\rho_P^1(X)\uparrow$ und $\sigma(X) \neq \varepsilon$ das zu bestimmende minimale n dasselbe ist, vollzieht sich auch hier der Induktionsschritt. Übrig bleibt der Fall $s = X(\underline{u})$, $\sigma(X) \neq \varepsilon$ und $\rho_P^1(X)\downarrow$. Nach Voraussetzung folgt aber sofort $\rho_P^1(X) \equiv \rho_P^2(X)$. ❑

Für die Optimierungen in Abschnitt 5.3 sind neben aussagen- und prädikatenlogischen Äquivalenzumformungen auch Umformungen auf Prädikatsdefinitionen notwendig. Dadurch werden in diesem Abschnitt mehrere Prädikatsvariablendefinitionen nebeneinander verwendet. Die Semantik unter einer Definition wird dann auf der linken Seite mit dem Namen der Definition gekennzeichnet. Zum Beispiel kommt neben der Prädikatsdefinition δ noch eine weitere δ'

vor. Mit ${}_{\delta}\langle\!\langle\cdot\rangle\!\rangle$ wird dann die Semantik unter der ersten Definition bezeichnet und entsprechend mit ${}_{\delta'}\langle\!\langle\cdot\rangle\!\rangle$ die unter der zweiten.

Bevor nun Äquivalenzumformungen betrachtet werden, muß hier nocheinmal genauer der Äquivalenz- und damit der Tautologiebegriff für μ-Kalkülterme (-formeln) fixiert werden.

Definition B.3 (Semantische Äquivalenz in $\mathbb{B}_\mu^V$) *Zwei Terme $s,t \in T_\mu^V$ heißen äquivalent, in Zeichen $s \equiv t$, gdw. für die „leere" Belegung $\rho_P = \{\}$ für die Semantik II $\langle\!\langle s\rangle\!\rangle\rho_P \equiv \langle\!\langle t\rangle\!\rangle\rho_P$ gilt.*

Die erste Umformung, die hier betrachtet wird, ist die Expansion von Fixpunkten. Dies wurde z. B. in /Kozen, 1983/ schon für den modalen μ-Kalkül angegeben. In $\mathbb{B}_\mu^V$ lautet sie (hier gleich auch für nicht rekursive Prädikate angegeben)

Satz B.4 (Expansion von Prädikaten) *Für $X \in P$ gilt $X(\pi(X)) \equiv \beta(X)$.*

Beweis: Sei zunächst $\sigma(X) = \mu$. Dann gibt es ein $n \in \mathbb{N}$ und ein $\zeta \in \mathbb{B}^W$ mit

$$\zeta = (\Phi_\rho^X)^n(0) \equiv (\Phi_\rho^X)^{n+1}(0)$$

für $\rho := \{\}$ und $X(\pi(X)) = \zeta\{\pi(X) \mapsto \pi(X)\}$ also $X(\pi(X)) = \zeta$. Satz B.2 und die Definition von Φ ergibt nun

$$\langle\!\langle\beta(X)\rangle\!\rangle\{X \mapsto \zeta\} = \Phi_\rho^X(\zeta) \equiv (\Phi_\rho^X)^{n+1}(0) \equiv \zeta = X(\pi(X))$$

mit folgender Gleichung $\langle\!\langle\beta(X)\rangle\!\rangle\rho\{X \mapsto \zeta\} \equiv \langle\!\langle\beta(X)\rangle\!\rangle\rho$ folgt der Rest. Diese wird bewiesen durch Induktion über den Termaufbau von $\beta(X)$. Der Fall $\sigma(X) \in \{\nu,\varepsilon\}$ folgt analog. ❑

Diese Umformung läßt sich besonders gut zusammen mit einer Substitution der Argumentvariable $\pi(X)$ von X verwenden. Diese Art α-Konversion wird hier nicht formal behandelt. Man muß dabei nur aufpassen, daß keine unerwünschten Nebeneffekte bei durch Quantoren gebundenen Variablen entstehen. Als Beispiel für diesen Satz betrachte man die Definition von R von Seite 19. Sie lautete

$$\mu R(s)\,.\,S(s)\;\vee\;\exists t\,.\,T(t,s)\wedge R(t)$$

Nun gilt nach letztem Satz

$$R(\pi(R)) \equiv R(s) \equiv S(s)\;\vee\;\exists t\,.\,T(t,s)\wedge R(t) \equiv S(s)\;\vee\;\exists t\,.\,T(t,s)\wedge R(t)$$

und man erhält die Tautologie $S(s) \to R(s)$, wobei der Begriff Tautologie wie in Definition 2.3 über den Äquivalenzbegriff definiert wird.

Definition B.5 *Zu $t \in T_\mu^V$, $X,R \in P$ mit $\tau(X) = \tau(R)$, $X \notin \mathsf{dep}(R)$ heiße X in t R-konjungiert gdw. für alle $Y \in \mathsf{scc}(X) \cup \{X\}$ und alle $s \leq t$ sowohl $\tau(Y) = \tau(X)$ gilt und aus $Y(\underline{u}) < s$ auch $Y(\underline{u}) \wedge R(\underline{u}) \leq s$ folgt.*

Das Beispiel von oben fortführend läßt sich folgende Äquivalenz zeigen

$$\exists s\,.\,S(s) \wedge X(s) \equiv \exists s\,.\,(S(s) \wedge R(s)) \wedge X(s) \equiv \exists s\,.\,S(s) \wedge (R(s) \wedge X(s)) \qquad \text{(B.1)}$$

Damit hat man einen Term bekommen, in dem X jetzt R-konjugiert ist. Intuitiv ist klar, daß die Semantik von X nur von den erreichbaren Zuständen abhängt. Man ist jetzt versucht, umgekehrt wie in obigem Satz, $R(s)$ in den Rumpf von $X(s)$ zu propagieren. Dies ist aber formal nur erlaubt, wenn in der SCC von X alle Prädikate „invariant" gegenüber R sind. Die nächste Definition liefert hierfür ein notwendiges syntaktisches Kriterium.

Definition B.6 *Zu $t \in T_\mu^V$, $X,R,T \in P$, $X \notin \mathsf{dep}(R) \cup \mathsf{dep}(T)$, $\underline{v} \in V^n$ mit $\tau(\underline{v}) = \tau(X) = \tau(R)$ und $\tau(X)^2 = \tau(T)$ heiße X in t T/R-konjugiert bez. $\underline{v}$ gdw. für alle $Y \in \mathsf{scc}(X) \cup \{X\}$ und alle $s \leq t$ sowohl $\tau(Y) = \tau(X)$ gilt und aus $Y(\underline{u}) < s$ auch entweder $Y(\underline{u}) \wedge R(\underline{u}) \leq s$ oder $Y(\underline{u}) \wedge T(\underline{v},\underline{u}) \leq s$ folgt.*

Mit E dem Prädikat, das die Fehlerzustände beschreibt (vgl. S. 18), kann man X als das Prädikat definieren, das diejenigen Zustände charakterisiert, von denen ein Fehlerzustand (der E erfüllt) erreichbar ist.

$$\mu X(s)\,.\,E(s) \;\vee\; \exists t\,.\,T(s,t) \wedge X(t)$$

Im Rumpf von X ist X selbst T/R-konjungiert bez. s. Nun betrachte das Vorkommen von X in den Rümpfen aller Prädikate einer Definition.

Definition B.7 *Ein $t \in T_\mu^V$ ist in RPNF (rekursiver positiver Normalform) gdw. es kein $s \in T_\mu^V$ und $X \in P$ gibt, mit $\mathsf{scc}(X) \neq \{\}$ und $X(\underline{u}) \leq \neg s \leq t$. Als boolesche Basisoperatoren sind nur „ $\neg$ ", „ $\vee$ " und „ $\wedge$ " erlaubt.*

Definition B.8 *Eine Prädikatsvariablendefinition δ ist in RPNF gdw. für alle $X \in P$ gilt, $\beta(X)$ ist in RPNF.*

Definition B.9 *Seien $X,R,T \in P$ mit $\tau(X) = \tau(R)$ und $\tau(X)^2 = \tau(T)$. Eine Prädikatsvariablendefinition δ heißt für ein X T/R-konjugiert gdw. X für alle $Z \in P \backslash \mathsf{scc}(X)$ in $\beta(Z)$ R-konjugiert ist und X für $Z \in \mathsf{scc}(X) \cup X$ in $\beta(Z)$ T/R-konjugiert ist bez. $\pi(Z)$.*

Für X ist die Definition des obigen Beispiel T/R-konjugiert. Wie der nächste Satz zeigt, darf man dann den Rumpf der Definition von X ersetzen durch

$$\mu X(s)\,.\,R(s) \wedge (E(s) \;\vee\; \exists t\,.\,T(s,t) \wedge X(t)),$$

wenn man nur noch Terme auswertet, in denen X R-konjugiert ist, was der Term aus Glg. $(B.1)$ erfüllt. Für die Korrektheit dieser Vorgehensweise muß noch sichergestellt werden, daß R unter T selbst „invariant" bleibt.

Definition B.10 *$R \in P$ heiße für $T \in P$ invariant gdw. $(R(\underline{u}) \wedge T(\underline{u}, \underline{v})) \rightarrow R(\underline{v})$ eine Tautologie ist ($\tau(R)^2 = \tau(T)$).*

Satz B.11 *Seien $X, R, T \in P$ mit $\tau(X) = \tau(R)$ und $\tau(X)^2 = \tau(T)$, δ eine Prädikatsdefinition in RPNF, die für X T/R-konjugiert ist, und δ' sei definiert durch*

$$\delta' := (\pi, \beta', \sigma), \qquad \textit{mit} \qquad \beta'(Y) := \begin{cases} \beta(Y) & \textit{für } Y \neq X \\ R(\pi(X)) \wedge \beta(X) & \textit{für } Y = X \end{cases}$$

Weiter sei R für T invariant (unter δ) und X in $t \in T_\mu^V$ R-konjugiert. Dann gilt

$$_\delta\langle\!\langle t \rangle\!\rangle \rho_P \equiv {}_{\delta'}\langle\!\langle t \rangle\!\rangle \rho_P$$

für alle Variablenbelegungen ρ_p bez. der Semantik II.

Beweis: Der Beweis wird durch Induktion über die Parameter aus Gleichung (2.19) geführt. Der interessante Fall ist $t = Y(\underline{u}) \wedge R(\underline{u})$, mit $Y \in \mathsf{scc}(X) \cup \{X\}$ und $\rho_P(Y)\uparrow$. O.B.d.A. sei $\pi(Y) = \underline{u}$. Dann gibt es ein $n \in \mathbb{N}$ (Maximum!) mit

$$_\delta\langle\!\langle t \rangle\!\rangle \rho_P \equiv \underbrace{_\delta\langle\!\langle R(\underline{u}) \rangle\!\rangle \rho_P}_{s:=} \wedge ({}_\delta\Phi^Y_{\rho_P})^n(0) \stackrel{?}{\equiv} \underbrace{_{\delta'}\langle\!\langle R(\underline{u}) \rangle\!\rangle \rho_P}_{s':=} \wedge ({}_{\delta'}\Phi^Y_{\rho_P})^n(0) \equiv {}_{\delta'}\langle\!\langle t \rangle\!\rangle \rho_P$$

Weiter gilt mit der Voraussetzung $X \notin \mathsf{dep}(R)$, $s \equiv s'$. Damit ist der Beweis beendet, wenn obige Terme (in $\mathbb{B}^W$) als äquivalent nachgewiesen werden können. Dazu zeige für alle $r \in T_\mu^V$

$$s \wedge {}_\delta\Phi^X_{\rho_P}(r) \equiv s \wedge {}_{\delta'}\Phi^X_{\rho_P}(r)$$

Für $\delta^* \in \{\delta, \delta'\}$ gilt nach Definition von δ'

$$s \wedge {}_{\delta^*}\Phi^X_{\rho_P}(r) \equiv {}_{\delta^*}\langle\!\langle R(\underline{u}) \wedge \beta(Y) \rangle\!\rangle \rho_P\{Y \mapsto r\}$$

Nach äußerer Induktionsvoraussetzung wäre man fertig, wenn X R-konjungiert ist in $R(\underline{u}) \wedge \beta(Y)$. Lemma B.12 liefert einen äquivalenten Term, der R-konjugiert ist (hier braucht man die Voraussetzung $X \notin \mathsf{dep}(T)$, damit R auch unter δ' für T invariant bleibt). ❑

Lemma B.12 *Sei $t \in T_\mu^V$ in RPNF, X in t T/R-konjugiert bez. $\underline{u}$ und R für T invariant. Dann gibt es ein $r \in T_\mu^V$ mit $\langle\!\langle r \rangle\!\rangle \rho_P \equiv \langle\!\langle R(\underline{u}) \wedge t \rangle\!\rangle \rho_P$ für alle Belegungen ρ_P für die Semantik II.*

Beweis: Hier ist nur der Fall $t = t_1 \otimes t_2$ mit $\otimes \in \{\wedge, \vee\}$ interessant.[1] Gilt $t_1, t_2 \neq X(\cdot)$, distributiere man $R(\underline{u})$ über t_1 und t_2 und die Induktionsvoraussetzung liefert den Rest. Sei nun O.b.d.A. $t_1 = X(\underline{w})$. Dann bleibt noch $t_2 = R(\underline{u})$, wobei $\underline{w} = \underline{u}$ und man fertig ist, oder es gilt $t_2 = T(\underline{u}, \underline{w})$. Die Invarianz von R ergibt dann aber sofort $t \equiv t \wedge R(\underline{w})$ und mit der Assoziativität von „$\wedge$“ ist der Beweis beendet. ❑

[1] Wiederum vernachlässige man α-Konversion bei Quantoren.

B.2 Übersetzungen nach $\mathbb{B}_\mu^V$

In diesem Abschnitt werden Übersetzungen für verschiedene Formalismen in den in dieser Arbeit vorgestellten μ-Kalkül angegeben. Alle diese Übersetzungen gab es schon für den in der Literatur beschrieben Parkschen oder modalen μ-Kalkül. Deshalb werden sie auch nicht formal definiert und bewiesen sondern nur an Hand von Beispielen erläutert.

B.2.1 Parkscher μ-Kalkül

Beim μ-Kalkül /Park, 1976/, wie er auch in /Burch et al., 1990, Enders et al., 1993/ oder im System von Janssen /Janssen, 1993/ benützt wird, werden Terme in zwei Klassen aufgeteilt. Die *einfachen* Terme entsprechen booleschen Ausdrücken in $\mathbb{B}_\mu^V$. Es gibt Quantoren und Anwendungen von Prädikatsvariablen z. B. ist

$$P(u) \vee \exists\, v\, [T(u,v) \wedge R(u)]$$

ein einfacher Term im Parkschen μ-Kalkül. Die dabei auftretenden Variablen stammen in der ursprünglichen Formulierung aus einem beliebigen (unsortierten) Bereich. Bei fest gewähltem endlichen Bereich kann man diesen binär kodieren, so daß die Variablen Werte aus $\mathbb{B}^n$ denotieren, für ein festes $n \in \mathbb{N}$. Mit dieser Identifikation kann man nun direkt einfache Terme nach T_μ^V übersetzen. Obiger Term stimmt damit syntaktisch mit seiner Übersetzung nach $\mathbb{B}_\mu^V$ überein bis auf die eckigen Klammern „[“ und „]“, die statt dem Punkt „.“ den Rumpf vom Quantor trennen.

Einfache Terme im Parkschen μ-Kalkül können durch λ-*Abstraktion* zu *relationalen Termen* gemacht werden. Die einzigen Operatoren, die aus einem relationalen Term wieder einen relationalen Term erzeugen, sind die Fixpunktoperatoren z. B.

$$\mu X\, [\lambda u\, [P(u) \vee \exists\, v\, [T(u,v) \wedge X(v)]]]$$

Daneben gelten nur noch einfache Prädikatsvariable als relationale Terme. Diese wiederum werden durch eine Anwendung auf einen zur Stelligkeit passenden Vektor von Variablen zu einem einfachen Term, wie das schon in obigen Beispiel geschehen ist. Mit dem letzten relationalen Term bekommt man z. B. folgenden einfachen Term

$$\forall\, w\, [S(w) \rightarrow (\mu X\, [\lambda u\, [P(u) \vee \exists\, v\, [T(u,v) \wedge X(u)]]])(w)] \tag{B.2}$$

In der ursprünglichen Version würde man nun für P, S und T bei der Auswertung der Semantik eine feste Belegung wählen. Hier zum Beispiel für S die Menge der Startzustände und für T die Menge der Übergänge.

Im Modellprüfer von Janssen wurde wie bei funktionalen Programmiersprachen ein „`let`“-Konstrukt eingeführt, daß dazu dient, Abkürzungen zu definieren, die textuell ersetzt werden. Damit könnte man zum Beispiel S als Abkürzung ansehen, die syntaktisch die Menge der Startzustände charakterisiert. Dies ist aber nur für nicht rekursive Prädikate erlaubt. In $\mathbb{B}_\mu^V$ wurde diese Einschränkung fallen gelassen, genauso wie man natürlicherweise in funktionalen Programmiersprachen ein „`letrec`“-Konstrukt einführt (siehe /Jones, 1987/). Damit kann man nun auch Fixpunkte direkt als Abkürzung definieren und braucht keine λ-Abstraktion mehr.

Das wichtigste Argument für diese Vorgehensweise ist, daß diese Syntax weitverbreiteten Programmiersprachen wie C ähnelt, und so einem unbedarften Anwender den Einstieg erleichtert. Darüberhinaus muß ein praktikables System sowieso Abkürzungen erlauben. Der Nachteil liegt eher auf theoretischer Seite, daß nämlich die Wohldefiniert der Semantik nicht mehr so einfach zu beweisen ist, was aber in Kapitel 2 durchgeführt wurde, und den Anwender *nach erfolgreichem Beweis* sowieso nicht interessiert. Auch vereinfacht dies die Behandlung der Frage der Variablenallokation.

Um nun Terme aus dem Parkschen μ-Kalkül zu übersetzen, führt man für jede λ-Abstraktion und jeden Fixpunktoperator eine neue Prädikatsvariable als Abkürzung ein. Eine Prädikatsdefinition zu obigen Beispiel lautet in $\mathbb{B}_\mu^V$ dann

$$\mu X(u) \,.\, P(u) \vee \exists\, v \,.\, T(u,v) \wedge X(v),$$

wobei die Definitionen der „konstanten" Prädikatsvariablen S, P und T nicht aufgeführt sind. Hier wurde, was immer geht, die Prädikatsvariable für die oberste λ-Abstraktion im Rumpf eines Fixpunktes weggelassen. Unter dieser Prädikatsdefinition wird nun der folgende Term evaluiert

$$\forall\, w \,.\, S(w) \rightarrow X(w)$$

Man beachte wie der Parameter der (immer auftretenden) λ-Abstraktion im Rumpf des Fixpunktoperators einfach zum formalen Parameter der Definition von X wird.

B.2.2 Modaler μ-Kalkül

Im modalen μ-Kalkül (s. /Cleaveland, 1993/, /Kozen, 1983/, /Winskel, 1989/, /Winskel, 1991/, /Cleaveland, 1990/, /Cleaveland und Steffen, 1993/, /Cleaveland, 1993/) kann man die Modelle (S und T im obigen Beispiel) nicht direkt syntaktisch beschreiben. In Systemen, die zur Beschreibung der Eigenschaften von Systemen den modalen μ-Kalkül verwenden, werden die Systeme selbst durch einen anderen Formalismus beschrieben. Ein Paradebeispiel hierfür ist die Concurrency Workbench /Cleaveland, 1992/, bei der Prozeßalgebraterme benützt werden. Diese lassen sich aber wiederum im μ-Kalkül darstellen (siehe unten).

Durch die Formeln des modalen μ-Kalkül werden demnach nur Eigenschaften des Systems beschrieben. Es gibt keine Individuenvariablen noch Quantoren, und die Prädikatsvariablen sind implizit einstellig und erwarten einen Zustand als Argument, was aber syntaktisch nicht hingeschrieben wird. Den Wahrheitswert einer Formel erhält man, wenn man das entsprechende Prädikat auf den Startzustand anwendet. Bei fest interpretiertem Prädikat P lautet obiges Beispiel aus Gleichung (B.2) im modalen μ-Kalkül

$$\mu X \,.\, P \vee \langle\rangle X$$

Der modale Operator „$\langle\rangle$" steht dafür, daß sein Rumpf (hier X) in einem Folgezustand gilt. Weiter gibt es noch den dualen modalen Operator „$[\,]$", den man als „in allen Folgezuständen gilt ..." interpretieren kann.

Für die Übersetzung der Fixpunktoperatoren führt man wie beim Parkschen μ-Kalkül neue Prädikatsvariablen als Abkürzung ein. Diese sind entsprechend der Semantik alle einstellig. Hierfür müssen formale Parameter definiert werden, die für den aktuellen Zustand stehen. Die modalen Operatoren werden nun durch Quantoren aufgelöst.

$$\exists\, v \,.\, T(u,v) \wedge \ldots \quad \text{für} \quad \langle\rangle \ldots \qquad \text{und} \qquad \forall\, v \,.\, T(u,v) \rightarrow \ldots \quad \text{für} \quad [\,] \ldots$$

Die Individuenvariable (v), über die quantifiziert wird, sollte verschieden sein von der Variable, die für den momentanen Zustand steht (u). Es muß also hier für jedes syntaktische Vorkommen eines modalen Operator eine eigene Individuenvariable eingeführt werden. Die Übersetzung des Rumpfes des modalen Operators erfolgt rekursiv, wobei bei der Bearbeitung des Rumpfes nun die durch den Quantor gebundene Variable für den momentanen Zustand steht. Als Beispiel betrachte die Formel

$$\mu Y . P \vee [\,](Q \wedge \langle\rangle Y)$$

In $\mathbb{B}_\mu^V$ entspricht dies der Prädikatsdefinition

$$\mu Y(u) . P(u) \vee \forall v . T(u,v) \to (Q(v) \wedge \exists w . T(v,w) \wedge Y(w))$$

und wie oben muß der Term

$$\forall w . S(w) \to Y(w)$$

ausgewertet werden.

In manchen Versionen des modalen μ-Kalküls werden auch durch Aktionen markierte Zustandsübergangsysteme betrachtet. Hiermit ist mit einem Übergang noch eine Aktion verbunden und die Syntax der Modaloperatoren wird um Aktionen erweitert. Zur Aktion a schreibt man „ $\langle a \rangle$ “ und meint damit, daß es einen Übergang gibt, der mit der Aktion a markiert ist, und dessen Folgezustand das dem Rumpf entsprechende Prädikat erfüllt. Die Übersetzung nach $\mathbb{B}_\mu^V$ lautet hier

$$\exists v . T_a(u,v) \wedge \ldots \quad \text{für} \quad \langle a \rangle \ldots \qquad \text{und} \qquad \forall v . T_b(u,v) \to \ldots \quad \text{für} \quad [b] \ldots$$

Dabei stellt T_α die Projektion der Übergangsrelation auf die Aktion α dar.

Mehrfach geschachtelte Fixpunkte

Für die Übersetzung von FairCTL (siehe unten) oder allgemein modalem μ-Kalkül mit einer Alternierungstiefe (/Emerson und Lei, 1986/) größer als 1, müssen mehrfach geschachtelte Fixpunkte behandelt werden. Als Beispiel betrachte die FairCTL-Formel EG P unter den zwei Fairneßbedingungen H_1 und H_2 (vgl. S. 25). Nach /Burch et al., 1990, Clarke et al., 1993/ ist dies äquivalent zu folgender Formel im modalen μ-Kalkül (dort im Parkschen μ-Kalkül beschrieben)

$$\nu X . P \wedge \langle\rangle (\mu Y_1 . H_1 \wedge X \;\vee\; P \wedge \langle\rangle Y_1) \wedge \langle\rangle (\mu Y_2 . H_2 \wedge X \;\vee\; P \wedge \langle\rangle Y_2)$$

Wie oben erläutert, führt man nun für jeden Fixpunktoperator eine Abkürzung ein und erhält folgende Prädikatsvariablendefinition in $\mathbb{B}_\mu^V$

$$\begin{aligned}
&\mu\, Y_1(s) . H_1(s) \wedge X(s) \;\vee\; P(s) \wedge \exists t . T(s,t) \wedge Y_1(t) \\
&\mu\, Y_2(s) . H_2(s) \wedge X(s) \;\vee\; P(s) \wedge \exists t . T(s,t) \wedge Y_2(t) \\
&\nu\, X(s) \;\; . P(s) \wedge (\exists t_1 . T(s,t_1) \wedge Y_1(t_1)) \wedge (\exists t_2 . T(s,t_2) \wedge Y_2(t_2))
\end{aligned}$$

Man erkennt hier, daß für jede Fairneßbedingung eine neue Prädikatsvariable eingeführt werden muß. Deren Rumpf hat aber eine konstante Größe. Insgesamt ist die Übersetzung von FairCTL also linear in der Größe der Terme.

B.2.3 Weitere Übersetzungen

CTL* und LTL

Die Unterlogiken von CTL* lassen sich allesamt in den μ-Kalkül übersetzen. In allen in der Literatur genannten Methoden wird dabei eine Reduktion auf das Problem der Übersetzung von LTL verwendet. Für CTL und FairCTL gibt es direktere Methoden, die in /Burch et al., 1990, Clarke et al., 1993/ beschrieben sind. Verfahren zur Übersetzung von ganz CTL* in den μ-Kalkül findet man in der Diplomarbeit von Reffel /Reffel, 1996/. Hier seien nur die Arbeiten von Dam /Dam, 1994/ und /Clarke et al., 1994/ herausgegriffen. In /Reffel, 1996/ wurde unter anderem eine Verbesserung des Algorithmus von Dam angegeben und ein Übersetzer von CTL* in die μcke-Eingabesprache realisiert. Hierdurch ist es gelungen, weitaus stärkere Eigenschaften zu verifizieren (z. B. beim ABP, wo die in FairCTL formulierbare Fairneß viel zu stark ist).

Prozeßalgebra, Bisimulation und Sprachäquivalenz

In /Enders et al., 1993/ wurde angegeben, wie die Übergangsrelation eines durch einen Prozeßalgebraterm beschriebenen Systems als BDD repräsentiert werden kann. Die dabei auftretenden Operationen können aber alle auch in $\mathbb{B}_\mu^V$ durchgeführt werden. (vgl. Abschnitt C.5). Wie der Test auf Bisimulation dann im Parkschen μ-Kalkül durchgeführt werden kann, findet man in /Burch et al., 1990, Enders et al., 1993/.

Die Behandlung von Sprachäquivalenz ω-regulärer Automaten findet sich in den Arbeiten /Burch et al., 1990/, /Burch et al., 1992/ und zusätzlich /Park, 1981/.

B.3 Syntax der μcke-Eingabesprache

B.3.1 Typsystem

Der erste vom Autor entwickelte Prototyp der μcke, der in /Melcher, 1995/ verwendet wurde, hatte eine Lisp-ähnliche Eingabesprache und nur ein sehr einfaches Typsystem. So waren nur boolesche Variablen und Vektoren erlaubt, so wie das auch in $\mathbb{B}_{\mu}^{V}$ der Fall ist. Dies bereitete erhebliche Mühe bei der Implementierung des „front end“ für Argos, da die Übersetzung von Verbund- und Aufzählungstypen noch erledigt werden mußte. Auf der anderen Seite gibt es solche Typen in fast jeder formalen Beschreibungssprache, für die eine Übersetzung in die μcke erstrebenswert ist.

In der neuen Version der μcke, deren Syntax an C++ angelehnt ist, wurde deshalb ein Teil des Typsystems von C++ übernommen. Darunter fallen Aufzählungstypen (`enum`), Verbunde (`class`) und Vektoren (z. B. `bool a[10]`). Es gibt keine Abkürzungen für Typen (`typedef`), keine Vererbung und keine varianten Verbunde (`union`). In einer Nachfolgeversion der μcke soll dies alles integriert werden (auch polymorphe Prädikate).

Basistypen

Als einziger Basistyp ist zur Zeit der boolesche Typ `bool` implementiert. Dieser kann auch als Aufzählungstyp interpretiert werden, mit der Definition

```
enum bool { true, false };
```

Nach den Konventionen für Aufzählungstypen (s. u.) kann also eine Variable vom Typ `bool` genau für die Konstanten `0` oder `1` bzw. `true` oder `false` stehen.

Bereichstypen

Bereichstypen gibt es in C++ nicht. Deshalb wurde für ihre Definition das Schlüsselwort `enum` für Aufzählungstypen überladen. Durch

```
enum E { 1 .. 4 };
```

wird ein Typ mit dem Namen `E` definiert, der aus den Elementen `1`, `2`, `3` und `4` besteht. Die linke Grenze l muß eine positive Zahl (einschließlich der `0`) und kleiner als die rechte Grenze r sein. Der Typ besteht dann aus $r-l+1$ Elementen und bei Verwendung von booleschen Prädikaten (s. S. 144) werden zur Kodierung $\log_2(r-l+1)$ boolesche Variable benötigt.[2]

Aufzählungstypen

Ein Aufzählungstyp wird durch das Schlüsselwort `enum` definiert. Hier sei das Beispiel von Seite 7 wiederholt

```
enum Medici { leopold, cosimo, margarete, ferdinand };
```

Neben den symbolischen Konstanten `leopold` usw. kann auf einen Aufzählungstyp auch so zugegriffen werden, als wäre er als Bereichstyp definiert mit der `0` als der linken Grenze und der gleichen Kardinalität. Hierfür muß keine explizite Typkonversion durchgeführt werden.

[2]Bei Verwendung von BDDs ist noch wichtig, daß das MSB (most significant bit) der obersten BDD-Variable zugeordnet wird.

Vektoren

Vektoren oder (engl. array) sind *anonyme* Typen. Sie können nur an drei Stellen definiert werden:

1. als Typ eines Parameters einer Prädikatsdefinition
2. als Typ einer durch einen Quantor gebundenen Variable
3. als Typ einer Kompente eines Verbundtyps

Ihre abstrakte Syntax lautet

```
<IDENT> '[' <NUMBER> ']'
```

Dies bedeutet, daß nur feste Vektorenlängen verwendet werden können. Als eine zukünftige Erweiterung ist geplant, hier auch einen beliebigen Aufzählungs- oder Bereichstyp zuzulassen. Das macht natürlich nur dann Sinn, wenn auch beim Aufbau von Termen Variabeln als Vektorindex erlaubt sind, was momentan auch nicht möglich ist. Vektoren beginnen wie in C++ per definitionem bei `0`. Es sind also Indizes zwischen `0` einschließlich und der Vektorenlänge ausschließlich erlaubt.

Verbunde

Verbundtypen folgen in ihrer Syntax Klassendefinitionen in C++. Sie können dazu benutzt werden, um das kartesische Produkt von Typen zu bilden. Erst in einer zukünftigen Version der μcke sollen Vererbung und polymorphe Prädikate eingeführt werden. Die abstrakte Syntax für Verbundtypen lautet

```
'class' <IDENT> '{' <COMPOUNDS> '};'

<COMPOUNDS> ::= ( <COMPOUND> ';' )* <COMPOUND>
<COMPOUND>  ::= <IDENT> ( <CMPPAR> ',' )* <CMPPAR>
<CMPPAR>    ::= <IDENT> ( '[' <NUMBER> ']' )?
```

Gegenüber C++ fehlen Zugriffsregelungen (`public` etc.), Vererbung und die Möglichkeit Methoden zu definieren. Sei `T` ein beliebiger schon definierter Typ dann ist auch

```
class A { T a; bool b[4], c; T d; };
```

eine gültige Typdefinition für den Typ `A`. Wenn nun `x` als „`A x`“ definiert ist, dann sind entsprechend C++-Syntax

```
x.a   x.b  x.b[3]   x.c   x.d
```

alles gültige Terme.

B.3.2 Terme

Grundterme

Als Grundterme sind in der Eingabesprache der μcke, Konstanten, Variablen und Prädikatsanwendungen vorgesehen. Als Konstanten kommen nach den Erläuterungen zum Typsystem positive natürliche Zahlen und symbolische Konstanten, die durch einen Aufzählungstyp definiert wurden, in Frage.

Bei Variablen kann zusätzlich zum Namen noch eine „Zugriffsspezifikation" stehen, die eine einzelne Komponente einer Variable mit geschachteltem Typ (Verbunde oder Vektoren) auswählt. Dies kann rekursiv geschehen, wie oben im Unterabschnitt über Verbunde schon erläutert. Man beachte, daß diese Zugriffspezifikation beliebig lang sein kann. Zu den Klassendefinitionen

```
class A { bool a[4]; };
class B { A b, c; };
class C { B d[2], e; };
```

und der Deklaration „`C x`" ist „`x.d[1].c.a[2]` " ein gültiger Grundterm mit booleschem Typ.

Funktionsanwendungen

Funktionsanwendungen haben auch dieselbe Syntax wie in C, mit der Einschränkung, daß nur Grundterme (Konstanten und Variablen mit Zugriffspezifikation) als Argumente erlaubt sind. Für spätere Versionen der μcke ist hier eine Erweiterung auf beliebig geschachtelte Funktionsanwendungen sinvoll. Man beachte, daß erst die Möglichkeit Zugriffsspezifikationen bei Argumentvariablen zu verwenden, wie in

```
...
class State { SubState1 s1; SubState2 s2; Channel c; };
bool Trans1(Channel in, Channel out,
            SubState1 s, SubState1 t) ...
bool Trans1(Channel in, Channel out,
            SubState2 s, SubState2 t) ...
bool Trans(State now, State next)
  Trans1(now.c, next.c, now.s1, next.s1) & next.s2=now.s2 |
  Trans2(now.c, next.c, now.s2, next.s2) & next.s1=now.s1 ;
```

bei der Definition von „`Trans`", es erlaubt, Systeme schichtenweise zu beschreiben. Dies vereinfacht ungemein den Aufwand bei der Übersetzung von vielen formalen Beschreibungen.

Operatoren

Die einzige momentan implementierte Operation auf Grundtermen ist die Gleichheit. In zukünftigen Versionen der μcke ist geplant, daß zusätzlich arithmetische Operationen und Vergleiche erlaubt sind. Zur Zeit muß man diese Operationen als Prädikat definieren (vgl. Abschnitt C.2). So ist zum Beispiel „`x.d[1].b = x.e.c`" ein boolescher Term. Neben durch die Gleichheit (oder Ungleichheit „ `!=` ") gebildete Terme sind nur noch die zwei Konstanten `true` und `false` (oder `0` und `1`) und die Anwendung einer booleschen Funktion (Prädikat) einfache boolesche Terme.

Auf booleschen Termen sind dann die üblichen booleschen Operationen wie „`&`“ für die Konjunktion, „`|`“ für die Disjunktion, „`!`“ für die Negation, „`->`“ für die Implikation usw. definiert. Zusätzlich gibt es auch ein „kaskadierendes“ if-then-else. Hier ist die Syntax wie in C++, z. B.

```
if(x.e != x.d[0])
  x.e.b.a[0]
else
if(x.e = x.d[1])
  0
else
  x.d[0] != x[1]
```

Um Fallunterscheidungen zu unterstützen, die bei der Definition von Übergangsrelationen sehr häufig anzutreffen sind, wurde auch ein „`case`“-Konstrukt definiert. Dieses entspricht aber in seiner Semantik dem kaskadierenden if-then-else. Obiges Beispiel lautet damit

```
case
  x.e != x.d[0] : x.e.b.a[0];
  x.e = x.d[1]  : 0;
  1             : x.d[0] != x[1];
esac
```

Neben den normalen booleschen Operatoren sind noch die zwei BDD-Vereinfachungsoperatoren „$\downarrow$“ und „$\Downarrow$“ aus Abschnitt 3.8.3 in die Syntax mit aufgenommen worden. Bei Verwendung von BDD-Bibliotheken lassen sich so spezielle Optimierungen (s. Abschnitt 5.3) auf den μ-Kalkül übertragen. Der „$\downarrow$“-Operator (oder „constrain“, „generalized cofactor“ genannt) heißt „`cofactor`“ und der „$\Downarrow$“-Operator (in der Literatur auch mit „restrict“ bezeichnet) schreibt sich „`assume`“.

Annotationen

In der Eingabesprache gibt es keine Blöcke. Deshalb konnten die geschweiften Klammern „`{`“ und „`}`“ mißbraucht werden, um Terme zu annotieren. Eine solche Annotierung hat die generelle Syntax

```
<TERM> '{' <IDENT> ... '}'
```

Der Name innerhalb der geschweiften Klammern charakterisiert die Annotationsart. Die Punkte stehen für die jeweilige Annotationsart spezifischen Argumente. Neben Annotierungen zur Steuerung der Erzeugung von Gegenbeispielen (/Jäger, 1996/) seien hier nur Allokationsrandbedingungen genannt, die weiter unten nocheinmal näher betrachtet werden.

Quantoren

Quantoren haben eine Parameterliste wie die später noch behandelten Funktionsköpfe. Sie definieren die durch den Quantor gebundenen Variablen und sind typisiert. So lautet die Syntax einer Parameterliste

```
<PARAMLIST> ::=
     ( <IDENT> <IDENT> ( '[' <NUMBER> ']' ) ? ',' ) *
               <IDENT> <IDENT> ( '[' <NUMBER> ']' ) ?
```

Wie oben schon erwähnt, ist dies neben Komponenten von Verbunddefinitionen die einzige Stelle, an der Vektoren (der Stelligkei <NUMBER>) definiert werden können. Ein Quantor hat dann die Syntax

```
( 'forall' | 'exists' ) <PARAMLIST> '.' <TERM>
```

Ein Beispiel lautet

```
exists Action a, State s, State t . T1(a,s,t) | T2(a,s,t);
```

Funktionsdefinitionen

Wie in C verlangt die μcke, daß alle verwendeten Typen und Funktionen (Prädikate) vor ihrer Benutzung deklariert sind. Für mehrfach rekursive Funktionen gibt es demnach Vorwärtsdeklarationen. Diese haben dieselbe Syntax wie Funktionsdefinitionen ohne Rumpf. Die Funktionsköpfe genügen

```
<FUNHEAD> ::=
  ( 'mu' | 'nu' ) ? <IDENT> <IDENT> '(' <PARAMLIST> ')'
```

und damit ist die Syntax für Funktionsdefinitionen

```
<FUNHEAD> <TERM>
```

Gegenüber C und C++ fehlen hier die den Rumpf der Definition umschließenden geschweiften Klammern. Als Beispiel betrachte

```
bool Trans(bool a[2], bool b[2])
   (a[0] -> b[0]) &  (a[1] -> b[1]);
```

Das optionale „mu“ oder „nu“ wird verwendet, wenn es sich bei der Funktion um eine rekursiv definierte Funktion handelt, und gibt an, ob der kleinste oder größte Fixpunkt gemeint ist.

Kommandos

Neben den Annotationen gibt es noch weitere Kommandos, die nicht zum μ-Kalkül gehören, sondern zur Steuerung des Modellprüfers dienen. All diese Kommandos verwenden die in C übliche Syntax

```
'#' <IDENT> ...
```

Die Punkte stehen auch hier für weitere für das Kommando spezifischen Argumente. Man beachte, daß die Kommandos alle mit einem „;“ abgeschlossen werden müssen, wie dies übrigens auch für alle Definitionen und zu evaluierenden Terme gilt. Die wichtigsten Kommandos sind

'#print' (<STRING> | <IDENT>) im Falle einer in Anführungszeichen („"“) eingeschlossenen Zeichenkette wird einfach diese ausgegeben. Somit ist

```
#print "Hello World";
```

das berühmte „Hello World“-Programm für die μcke. Hier wird automatisch ein Zeilenumbruch angehängt. Ist ein Name als Argument angegeben, so wird die mit dem Namen assoziierte Symbolinformation ausgegeben. Weiter gibt es noch zwei spezielle Namen „symbols“, womit die Symbolinformation über alle Symbole angezeigt wird, und „statistics“, was die μcke veranlaßt, schichtenweise alle gesammelten statistischen Informationen auszugeben.

'#load' (<STRING> | <IDENT>) entspricht einem „#include“ in C zum Laden einer Datei mit weiteren Definitionen oder zu evaluierenden Termen. Geschachtelte „#load“-Befehle sind erlaubt. Dies ist besonders nützlich, um sich Bibliotheken mit zu evaluierenden Formeln anzulegen, die dann einfach nach Bedarf zu einem speziellen Modell hinzugeladen werden können.

'#size' <IDENT> gibt die Größe der Repräsentation einer definierten Funktion aus. Dazu muß sie natürlich erst bestimmt werden, wozu der Modellprüfungsalgorithmus verwendet wird. Im Falle der Verwendung einer BDD-Bibliothek wird hier die Größe, d. h. die Anzahl Knoten, des BDDs ausgegeben.

'#onsetsize' <IDENT> gibt die Anzahl erfüllender Belegungen sowohl absolut, im Zweierlogarithmus als auch in Prozent aus. (Abkürzung #ons)

'#witness' <TERM> generiert einen Zeugen für einen Term. Die Angabe eines beliebigen Termes ohne dieses Kommando führt nur zu dessen Evaluation. Der Modellprüfungsalgorithmus liefert nur die Aussage „true“ oder „false“. Der Algorithmus für die Zeugengenerierung stammt aus /Kick, 1996/ und wurde in /Jäger, 1996/ implementiert. (Abkürzung #wit)

'#cex' <TERM> erzeugt (dual zu dem Kommando „#witness“) ein Gegenbeispiel nach der Methode aus /Kick, 1996/ („#cex“ steht für „counter example“).

'#timer' ('go' | 'reset' | 'stop') ? Damit kann eine Stoppuhr ausgelesen, gestartet, zurückgesetzt oder angehalten werden. Um sie „abzulesen“ wird kein Argument angegeben.

'#reset' <IDENT> Die μcke verwendet für jeden zu evaluierenden Term in der Eingabe einen neuen Evaluator (vgl. Abschnitt 5.2.2). Um zu vermeiden, daß eine Funktion in einem zweiten Evaluator nocheinmal ausgewertet werden muß, obwohl in einem ersten Evaluator ihr Wert (z. B. der BDD dafür) schon bestimmt wurde, gibt es einen Ergebnisspeicher (engl. cache) für Funktionen, in denen Repräsentationen für Funktionen abgelegt werden.

Da die μcke nicht wissen kann, ob später eine schonmal berechnete Funktion nocheinmal benötigt wird, speichert sie grundsätzlich alle errechneten Repräsentationen für Funktionen in diesem Speicher. Um Platz zu sparen, kann der Benutzer über diesen Befehl eine nicht mehr benötigte Repräsentation aus dem Ergebnisspeicher löschen. Der spezielle Name „all“ führt zur Löschung aller gespeicherten Resultate.

'#visualize' <IDENT> startet eine Visualisierung einer Funktion. Diese Visualisierung benötigt die BDDsimple-Bibliothek (durch Laden des Managers simplebman.so) und verwendet den vom Autor implementierten „Graphlayouter“ lg. (Abkürzung #vis)

'#verbose' ('on' | 'off') ? erhöht oder senkt den „verbosity level“. Dieser gibt an, wieviel die μcke dem Benutzer darüber berichten soll, was sie gerade tut. In der höchsten Stufe (viermal mit on aufgerufen) wird zum Beispiel nach jedem Schritt im Modellprüfungsalgorithmus die Größe der Repräsentation und alle Statistikinformation ausgegeben. Ohne Argument verhält sich der Befehl als wäre er mit on als Argument aufgerufen worden.

`'#frontier' ( 'on' | 'off' )` Hiermit wird die interne Optimierung der „frontier set simplification" ein- bzw. ausgeschaltet.

`'#quit'` beendet die μcke-Sitzung. Daselbe geschieht natürlich wenn die letzte aller geöffneten Datei vollständig gelesen wurde. Bei einer interaktiven Sitzung kann der Benutzer auch durch Eingabe der Tastenkombination Ctrl und D die Eingabe beenden.

Mehrere der durch diese Kommandos gesteuerten Optionen können auch von der Kommandozeile aus angesprochen werden. So wird z. B. durch den Aufruf der μcke mittels

```
mucke -f -v model.mu spec.mu -
```

zunächst die „frontier set simplfication" eingeschaltet und der „verbosity level" um eins erhöht. Dann werden die Dateien `model.mu` und `spec.mu` bearbeit, um danach in den interaktiven Modus zu wechseln (`-`). Eine Übersicht über die Kommandozeilenparameter erhält man durch den Aufruf von `mucke -h`.

Allokationsrandbedingungen

Die μcke versucht mit dem Algorithmus aus Kapitel 4 automatisch gute Allokation zu generieren. Leider gelingt das nicht immer, ohne daß der Benutzer zusätzliche Informationen beisteuert. Wenn bei der Verifikation die μcke plötzlich sehr viel Speicher beansprucht und die Berechnungen nicht mehr zu Ende führen kann, dann liegt dies oft daran, daß eine Allokation schlecht gewählt wurde. Um hier einzugreifen, kann der Benutzer durch Erhöhen des „verbosity levels" sich erstens genau anschauen, welche Allokationen die μcke generiert hat, und zweitens kann er die Modellprüfung verfolgen bis an die Stelle, an der sehr große BDDs erzeugt werden. Wenn dies nach einer Substitution geschieht, dann ist mit größter Wahrscheinlichkeit die Allokation, in die hineinsubstituiert wird, schlecht gewählt.[3] Ebenso kann er für kleinere BDDs (kleiner 3000 Knoten) den BDD visualisieren, um dem Grund für die Größenexplosion herauszufinden.

Hier kann der Benutzer steuernd eingreifen, indem er eine Randbedingung angibt, die die μcke beim erneuten Start bei der Konstruktion einer Allokation beachtet. Es sollte untersucht werden, wie hier dynamische Variablenumordnung den Benutzer entlasten könnte. Der naive Ansatz, die dynamische Variablenumordnung von /Rudell, 1993/ für die Schaltnetzverifikation auf die Schaltwerkverifikation (Modellprüfung) zu übertragen, führt nicht zum Erfolg, wie der Vergleich in Abschnitt 5.4.2 und der erste Prototyp der μcke zeigte. Diese Technik hat nur dann Aussicht auf Erfolg, wenn man sie mit Allokationsrandbedingungen kombiniert.

Allokationsrandbedingungen in der Eingabesprache der μcke bestehen aus einer Liste von elementaren Randbedingungen

```
<ALLOCCS_ATOM> ::=
    <IDENT>  ( '~+' | '~-' | '-<' | '+<' ) <IDENT>
<ALLOCCS>        ::= ( <ALLOCCS_ATOM> ',' ) *  <ALLOCCS_ATOM>
```

Die Operatoren in den elementaren Randbedingungen (`<ALLOCCS_ATOM>`) stehen für die Bestandteile der Allokationsrandbedingungen aus Kapitel 4:

„Verschränken", „Blocken" und „Ordnung"

[3] Je nach Allokator wird für jede Funktion oder sogar für jeden Quantor eine eigene Allokation verwendet.

In der Schreibweise aus Abschnitt 4.4 waren dies „$\sim^+$“, „$\sim^-$“, „$\leq^-$“ und „$\leq^+$“. Diese Listen von Randbedingungen können nun an drei Stellen in der Eingabesprache der μcke auftreten. Zunächst wie oben schon erwähnt in einer Annotation wie im Beispiel auf Seite 126.

Zweitens werden Allokationen immer für die Parameter von Funktionen benötigt. Hier treten auch die meisten Probleme mit automatisch erzeugten Allokationen auf. Deshalbg wurde hierfür die Abkürzung eingeführt, damit man direkt nach einem Funktionskopf (`<FUNHEAD>`) eine Liste von Allokationsrandbedingungen (`<ALLOCCS>`) hinschreiben darf. Damit ist auch nur ein minimaler Änderungsaufwand beim Test verschiedener Randbedingungen nötig.[4]

Hat man zum Beispiel folgende Definition einer Übergangsrelation mit Aktionen

```
bool Trans(State s, Action a, State t)
   ...
```

dann würde der Standard-Allokator diese drei Variablen geblockt allokieren (wenn nicht die ASTI-Heuristik von S. 138 eingeschaltet ist). Der Benutzer würde bei der Auswertung dieser Funktion merken, daß sehr große BDDs entstehen. Deshalb würde er diese Definition näher anschauen und es eventuell mit folgenden Allokationsrandbedingungen versuchen

```
bool Trans(State s, Action a, State t)  a -< s, s ~+ t
   ...
```

Hätte er den Parameter `a` zuerst hingeschrieben, so würde die ASTI-Heuristik von S. 138 automatisch die zweite Randbedingung generieren und der Allokator würde auch versuchen `a` vor `s` und `t` zu allokieren. Dies hätte also denselben Effekt wie die Verwendung dieser Randbedingung. Aber beim Änderen der Parameterreihenfolge muß er zusätzlich überall, wo `Trans` aufgerufen wurde, konsistenz die Argumente vertauschen.

Drittens können Randbedingungen auch für die Namen der einzelnen Komponenten in Verbundtypen angegeben werden. Dies war besonders nützlich bei einer Vorstudie zur Berechnung der maximalen Länge der minimalen Zugfolge zur Lösung des „Rubik’s Cube“.

Die Beschreibung in der Eingabesprache der μcke findet man in Abschnitt C.3. Die Version des dort angegebenen Rubik’s Cube hat keine Farben, ist dafür aber durchnumeriert. Die Zugfolgen für den Würfel sind dieselben, wie beim richtigen Würfel. Es ist nur erlaubt, daß pro Zug eine Ebene des Würfels bestehend aus acht kleinen Würfeln um 90 Grad um ihre Achse gedreht werden darf. Die Anzahl erreichbare Zustände errechnet die μcke zu 40320 und benötigt dazu 9 Fixpunktiterationen (hier sind noch keine Gesamtdrehungen des Würfels eingerechnet). Damit läßt sich jede erreichbare Konfiguration ohne die räumliche Lage des Würfels zu verändern in höchstens 8 Zügen in die Ausgangsposition überführen und es gibt eine Konfiguration, für die es nicht mit weniger Zügen geht.

Verwendet man die Allokationsrandbedingungen, wie sie im Quelltext angegeben sind, so besteht der BDD für die Übergangsrelation aus 9305 Knoten. Wenn die Komponenten des globalen Zustandes alle geblockt allokiert werden, dann werden daraus 59180 BDD-Knoten. Die Berechnungsdauer erhöht sich von 31 Sekunden auf 76 Sekunden und der Speicherverbrauch von 4.3 MB auf 7.5 MB. Der Grund dafür, daß die Auswirkung auf die Zeit nicht noch drastischer ausfällt, liegt daran, daß die Anzahl Knoten des größten BDDs für eine Approximation von 11678 auf nur 14620 anstieg.

[4] Und die Syntax blieb LR(1)!

Anhang C

Quelltexte

C.1 $\mathsf{cite}_{\exists}$ in der BDD-Bibliothek `BDDsimple`

```
BDDsimple * BDDsimple::cITEexists(
  BDDsimple * a, BDDsimple * b, BDDsimple * c,
  BDDsimple * x, BDDsimpleIntToBDDAssoc * assoc)
{
  if(a == False) return exists(c,x);
  if(a == True) return exists(b,x);

  if(c == False)         // this is an 'and'
    {
      if(b==False) return False;
      if(b==True) return _composeExists(a,x,assoc);
    }
  else
  if(c == True)
    {
      if(b == True) return True;
      if(b == False)
        {
          BDDsimple * tmp = not(a);
          BDDsimple * res = _composeExists(tmp,x,assoc);
          _free(tmp);
          return res;
        }
    }

  BDDsimple * dest = (*assoc) [ a -> var ];

  if(dest)
    {
      if(dest == False) return exists(c,x);
      if(dest == True) return exists(b,x);
    }

  while(x!=True) {              // normalize x
```

```
    if(assoc -> min_var(a -> var) <= x -> var) break;
    if(!b -> isConstant() && b -> var <= x -> var) break;
    if(!c -> isConstant() && c -> var <= x -> var) break;

    x = x -> right;
  };

  if(x==True)
    {
      if(dest) _free(dest);
      return composeITE(a,b,c,assoc);
    }

  BDDsimple * res =
    cached(BDDsimpleCITEEXISTS(a,b,c,x,assoc->index()));

  if(res)
    {
      if(dest) _free(dest);
      return res;
    }

  if(!dest)
    {
      dest = find(a->var, False, True);
    }

  extra_cite_calls++;

  if(dest -> isVariable())
    {
      if((b -> isConstant() || dest -> var <= b -> var) &&
         (c -> isConstant() || dest -> var <= c -> var))
        {
          extra_cite_exists_part1++;

          BDDsimple * c0 = c->cf_left(dest -> var),
                    * c1 = c->cf_right(dest -> var),
                    * b0 = b->cf_left(dest -> var),
                    * b1 = b->cf_right(dest -> var);

          BDDsimple * z = _subtract_vars(x, dest -> var);

          BDDsimple * t = cITEexists(a -> right, b1,c1,z, assoc);
          BDDsimple * e = cITEexists(a -> left, b0,c0,z, assoc);

          _free(z);

          res = fastIteExistsWithVar(dest -> var, dest, t, e, x);

          _free(t); _free(e);
```

```
        }
      else      // 2nd case for dest -> isVariable()
        {
          extra_cite_exists_part2++;

          int m = minVar(b,c);
          BDDsimple * c0 = c->cf_left(m), * c1 = c->cf_right(m),
                    * b0 = b->cf_left(m), * b1 = b->cf_right(m);

          BDDsimple * z = _subtract_vars(x, m);
          BDDsimple * t = cITEexists(a, b1, c1, z, assoc);
          BDDsimple * e = cITEexists(a, b0, c0, z, assoc);

          _free(z);

          res = fastIteExistsWithVar(m, 0, t, e, x);

          _free(t); _free(e);
        }
    }
  else        // ! dest -> isVariable()
    {
      int m = minVar(a,b,c);
      if(a -> var == m)
        {
          extra_cite_exists_part3++;

          BDDsimple * d_vars = _vars(dest);
          BDDsimple * y = _and_vars(d_vars, x);
          _free(d_vars);
          BDDsimple * z = _subtract_vars(x,y);

          BDDsimple * t = cITEexists(a -> right,b,c,z, assoc);
          BDDsimple * e = cITEexists(a -> left,b,c,z, assoc);

          _free(z);

          res = iteExists(dest, t, e, y);

          _free(t); _free(e); _free(y);
        }
      else    // 2nd case for ! dest -> isVariable()
        {
          extra_cite_exists_part4++;

          BDDsimple * c0 = c->cf_left(m), * c1 = c->cf_right(m),
                    * b0 = b->cf_left(m), * b1 = b->cf_right(m);

          BDDsimple * f = find(m, False, True);
          BDDsimple * y = _and_vars(f, x);
          BDDsimple * z = _subtract_vars(x,y);
```

```
          BDDsimple * t = cITEexists(a, b1, c1, z, assoc);
          BDDsimple * e = cITEexists(a, b0, c0, z, assoc);

          _free(z);

          res = iteExists(f, t, e, y);

          _free(t); _free(e); _free(y); _free(f);
        }
    }

  _free(dest);

  store(BDDsimpleCITEEXISTS(a,b,c,x,assoc->index(),res));
  return res;
}
```

C.2 Arithmetische Operationen in der μcke

```
enum Data { 0..15 };

bool inc(Data s, Data t)
  case
    s =  0 : t =  1;
    s =  1 : t =  2;
    s =  2 : t =  3;  // Mit bits das zu kodieren geht
    s =  3 : t =  4;  // natuerlich auch. Aber man kann dann
    s =  4 : t =  5;  // (momentan) in den zu ueberpruefenden
    s =  5 : t =  6;  // Formeln keine Zahlen schreiben:
    s =  6 : t =  7;  // z. B. class Data { bool bits[4]; };
    s =  7 : t =  8;  // exists Data a. a = 11;
    s =  8 : t =  9;  // geht leider noch **nicht**
    s =  9 : t = 10;
    s = 10 : t = 11;
    s = 11 : t = 12;
    s = 12 : t = 13;
    s = 13 : t = 14;
    s = 14 : t = 15;
    s = 15 : t =  0;
  esac;

bool dec(Data s, Data t) inc(t,s);

mu bool add(Data s, Data t, Data r)      // s + t = r
  if(s = 0) r = t
  else
    exists Data tmp1, Data tmp2.
        dec(s, tmp1) & add(tmp1,t,tmp2) & inc(tmp2,r);

mu bool mult(Data s, Data t, Data r)     // s * t = r
  if(s = 0) r = 0
  else
  if(s = 1) r = t
  else
    exists Data tmp1, Data tmp2.
        dec(s, tmp1) & mult(tmp1,t,tmp2) & add(t,tmp2,r);

// Was ist das Inverse von 5 modulo 16 ?

#wit exists Data x. mult(5,x,1);
```

C.3 Vorstudie zum Rubik’s Cube

```
/*
             Y              (x,y,z)

             ^
             |
             |
             |
             +-----+-----+ <--- (1,1,0) = 6 f.e.
            /:   2/:   6/|
           / :   / :   / |
          +-----+-----+  |
         /:  +3/...+7/|..+
        / : ':/     / | /|
       +-----+-----+  |/ |
       |  :  |     |  +  |
       | ':  |.....|./|..+-----> x
       |' : '|  0 '|/ |4/
       +-----+-----+  |/
       |  +..|..+..|..+
       | '   |1'   |5/
       |'    |'    |/
       +-----+-----+
      /
     /
    /
   V

 Z

Here we have the following basic moves:

    xy0 = (6 2 0 4) xy1 = (5 7 3 1)
    yx0 = (6 4 0 2) yx1 = (5 1 3 7)
    xz0 = (4 5 1 0) xz1 = (6 7 3 2)
    zx0 = (4 0 1 5) zx1 = (6 2 3 7)
    yz0 = (2 3 1 0) yz0 = (6 7 5 4)
    zy0 = (2 0 1 3) yz1 = (6 4 5 7) */

enum P { 0 .. 7 };
class State { P p0, p1, p2, p3, p4, p5, p6, p7; }
  p0 ~+ p1, p1 ~+ p2, p2 ~+ p3, p3 ~+ p4,
  p4 ~+ p5, p5 ~+ p6, p6 ~+ p7;                 // important

bool _xy0(State s, State t)
  s.p6 = t.p2 & s.p2 = t.p0 & s.p0 = t.p4 & s.p4 = t.p6;
bool _xy1(State s, State t)
  s.p5 = t.p7 & s.p7 = t.p3 & s.p3 = t.p1 & s.p1 = t.p5;
bool _xz0(State s, State t)
  s.p4 = t.p5 & s.p5 = t.p1 & s.p1 = t.p0 & s.p0 = t.p4;
```

```
bool _xz1(State s, State t)
  s.p6 = t.p7 & s.p7 = t.p3 & s.p3 = t.p2 & s.p2 = t.p6;
bool _yz0(State s, State t)
  s.p2 = t.p3 & s.p3 = t.p1 & s.p1 = t.p0 & s.p0 = t.p2;
bool _yz1(State s, State t)
  s.p6 = t.p7 & s.p7 = t.p5 & s.p5 = t.p4 & s.p4 = t.p6;
bool cxy0(State s, State t)
  s.p5 = t.p5 & s.p7 = t.p7 & s.p3 = t.p3 & s.p1 = t.p1;
bool cxy1(State s, State t)
  s.p6 = t.p6 & s.p2 = t.p2 & s.p0 = t.p0 & s.p4 = t.p4;
bool cxz0(State s, State t)
  s.p6 = t.p6 & s.p7 = t.p7 & s.p3 = t.p3 & s.p2 = t.p2;
bool cxz1(State s, State t)
  s.p4 = t.p4 & s.p5 = t.p5 & s.p1 = t.p1 & s.p0 = t.p0;
bool cyz0(State s, State t)
  s.p6 = t.p6 & s.p7 = t.p7 & s.p5 = t.p5 & s.p4 = t.p4;
bool cyz1(State s, State t)
  s.p2 = t.p2 & s.p3 = t.p3 & s.p1 = t.p1 & s.p0 = t.p0;

bool xy0(State s, State t) cxy0(s,t) & _xy0(s,t);
bool xy1(State s, State t) cxy1(s,t) & _xy1(s,t);
bool xz0(State s, State t) cxz0(s,t) & _xz0(s,t);
bool xz1(State s, State t) cxz1(s,t) & _xz1(s,t);
bool yz0(State s, State t) cyz0(s,t) & _yz0(s,t);
bool yz1(State s, State t) cyz1(s,t) & _yz1(s,t);

bool T(State s, State t)
  xy0(s,t) | xy1(s,t) | xz0(s,t) | xz1(s,t) | yz0(s,t) |
  yz1(s,t) | xy0(t,s) | xy1(t,s) | xz0(t,s) | xz1(t,s) |
  yz0(t,s) | yz1(t,s) ;

bool S(State s)
  s.p0 = 0 & s.p1 = 1 & s.p2 = 2 & s.p3 = 3 &
  s.p4 = 4 & s.p5 = 5 & s.p6 = 6 & s.p7 = 7;

mu bool R(State s) S(s) | (exists State t. T(t,s) & R(t));

#ons R;
```

C.4 Alternating Bit Protokoll

Dies ist das Alternating Bit Protokoll (vgl. /Bartlett et al., 1969, Clarke et al., 1986/) mit expliziter Modellierung der Kanäle und der Daten.

C.4.1 Datentypen

```
class Data { bool bits[8]; };           // eight bit data

enum ControlStateOfSender { get, send, wait_for_ack };

class StateOfSender {
  ControlStateOfSender state;
  bool abp;
  Data data;
};

enum ControlStateOfReceiver { receive, deliver, send_ack };

class StateOfReceiver {
  ControlStateOfReceiver state;
  bool abp;
  Data data;
};

enum S2RTag { data0, data1, error, mt };
class S2RData { S2RTag tag; Data data; };
class S2RChannel { S2RData in; S2RData out; };

enum R2STag { ack0, ack1, error, mt };
class R2SData { R2STag tag; };
class R2SChannel { R2SData in; R2SData out; };

enum Running { sender, receiver, s2r, r2s };

class State {
  Running running;      // interleaving model!
  StateOfSender sender;
  S2RChannel s2r;
  StateOfReceiver receiver;
  R2SChannel r2s;
};
```

C.4.2 Übergangsrelation des Senders

```
bool StartOfSender(State s)
  s.sender.abp = 0 & s.sender.state = get;

bool CoStabSender(State s, State t)
  t.s2r.out = s.s2r.out & t.r2s.in = s.r2s.in &
```

```
  t.receiver = s.receiver;

bool StabSender(State s, State t)
  t.s2r.in = s.s2r.in & t.r2s.out = s.r2s.out &
  t.sender = s.sender;

bool TransSender(State s, State t)
  s.running = sender & CoStabSender(s,t) &
  (
    case

      s.sender.state = get :

         t.sender.state = send &

           t.sender.abp = s.sender.abp &
           t.sender.data = s.sender.data &       // arbitrary
           t.s2r.in = s.s2r.in &
           t.r2s.out = s.r2s.out;

      s.sender.state = send :

        if(s.s2r.in.tag = mt)
          (
            t.sender.state = wait_for_ack &

              t.sender.abp = s.sender.abp &
              t.sender.data = s.sender.data &
              (
                if(s.sender.abp) t.s2r.in.tag = data1
                else t.s2r.in.tag = data0
              )
              &
              t.s2r.in.data = s.sender.data &
              t.r2s.out = s.r2s.out
          )
        else StabSender(s,t);

      s.sender.state = wait_for_ack :

        if(s.r2s.out.tag = ack0 & !s.sender.abp |
           s.r2s.out.tag = ack1 &  s.sender.abp
          )
          (
            t.sender.state = get &

              // nondeterministically choose data to send:
              1 &
              t.sender.abp != s.sender.abp &
              t.s2r.in = s.s2r.in &
              t.r2s.out.tag = mt
```

```
        )
       else
       if(
           s.r2s.out.tag = ack0 &  s.sender.abp |
           s.r2s.out.tag = ack1 & !s.sender.abp |
           s.r2s.out.tag = error
         )
         (
           t.sender.state = send &

             t.sender.abp = s.sender.abp &
             t.sender.data = s.sender.data &
             t.s2r.in = s.s2r.in &
             t.r2s.out.tag = mt
         )
      else
      if(s.r2s.out.tag = mt) StabSender(s,t);

    esac
  )
  ;
```

C.4.3 Gesamtsystem

```
bool Start(State s)
  StartOfSender(s) & StartOfReceiver(s) &
  StartOfR2S(s) & StartOfS2R(s);

bool _Trans(State s, State t)
  TransSender(s,t) | TransReceiver(s,t) |
  TransS2R(s,t) | TransR2S(s,t);

mu bool Reachable(State s)
  Start(s) | (exists State t. _Trans(t,s) & Reachable(t));

// this is faster than just anding as in the SMV system:

bool Trans(State s, State t) _Trans(s,t) assume Reachable(s);
```

C.4.4 Übersetzung von AG AF `sender.state` = `get`

Bewiesen wird eine prototypische Lebendigkeitsaussage, die besagt, daß der Sender immer wieder ein Datum von seinem Benutzer annimmt. Da der Benutzer nicht modelliert wurde, sondern der Sender einfach im Kontrollzustand `get` nichdeterministisch irgendein Datum generiert, ist dies äquivalent dazu, daß der Sender immer wieder diesen Kontrollzustand erreicht. Dies gilt natürlich nur unter den Fairneßbedingungen, daß jeder der vier Prozesse unendlich oft am Zuge ist und das die beiden Kanäle nicht unendlich oft hintereinander einen Fehler erzeugen. In FairCTL formuliert lautet dies

$$\mathrm{AG\,AF}\,\texttt{sender.state} = \texttt{get}$$

Bei der Übersetzung nach Abschnitt B.2 in den μ-Kalkül ergibt dies für jede Fairneßbedingung eine eigene Fixpunktvariable. Als Optimierung neben der Vorwärtsanalyse, die durch Verunden jedes Rumpfes mit `Reachable(s)` realisiert wird, wird zusätzlich noch die Übergangsrelation unter der Annahme vereinfacht, daß der Ausgangszustand erreichbar sein muß. Beim SMV-System wird hier nur verundet!

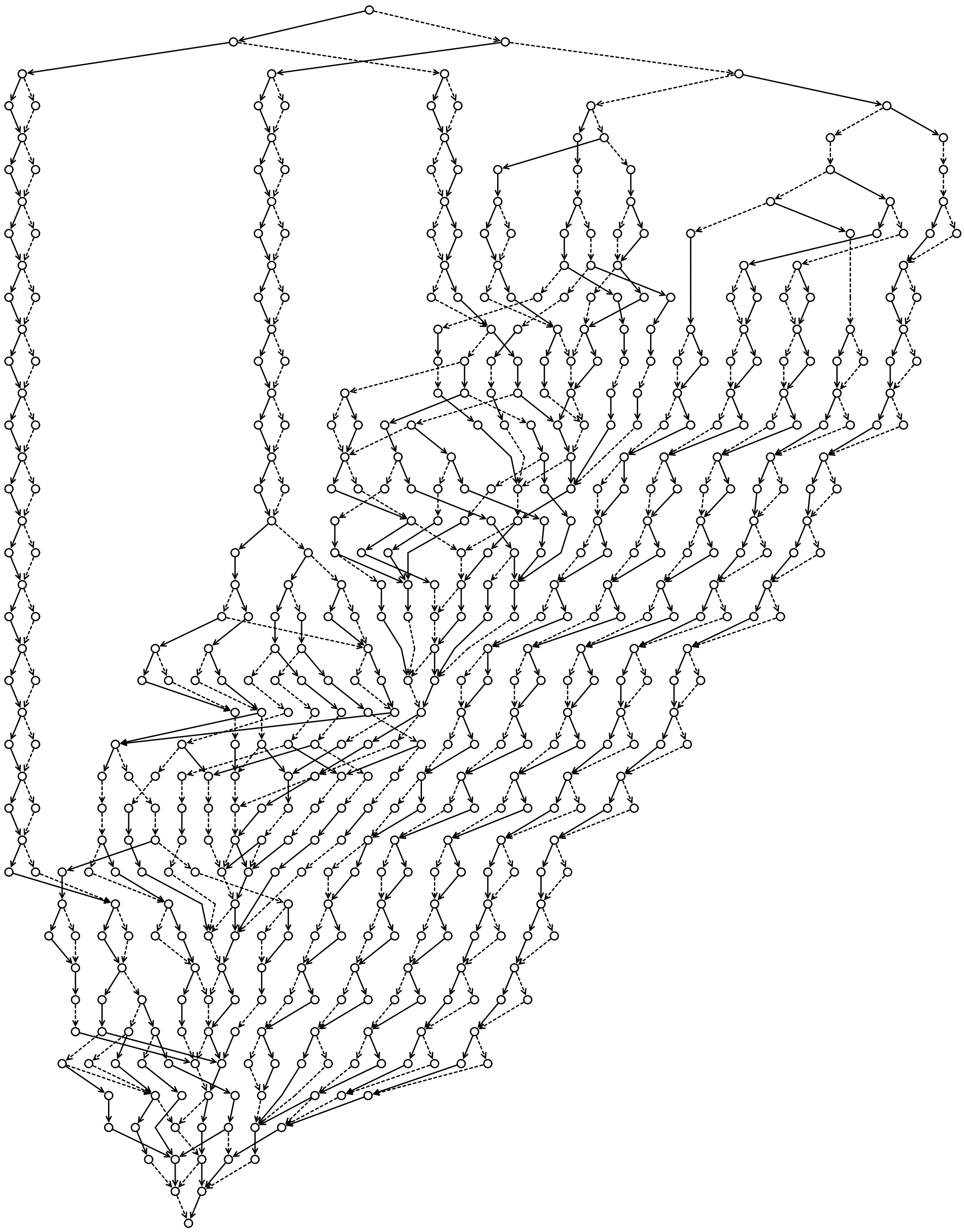
```
nu bool EGfair_sender_not_get(State s);

mu bool EGfair_sender_not_get_Sender(State s)
  (
    (s.running = sender & EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
                     EGfair_sender_not_get_Sender(t))
  )
  & Reachable(s)
  ;

mu bool EGfair_sender_not_get_Receiver(State s)
  (
    (s.running = receiver & EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
                     EGfair_sender_not_get_Receiver(t))
  )
  & Reachable(s)
  ;

mu bool EGfair_sender_not_get_S2R(State s)
  (
    (s.running = s2r & EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
                     EGfair_sender_not_get_S2R(t))
  )
  & Reachable(s)
```

```
  ;

mu bool EGfair_sender_not_get_R2S(State s)
  (
    (s.running = r2s & EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
                     EGfair_sender_not_get_R2S(t))
  )
  & Reachable(s)
  ;

mu bool EGfair_sender_not_get_S2Rfair(State s)
  (
    ((s.s2r.out.tag = data0 | s.s2r.out.tag = data1) &
      EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
       EGfair_sender_not_get_S2Rfair(t))
  )
  & Reachable(s)
  ;

mu bool EGfair_sender_not_get_R2Sfair(State s)
  (
    ((s.r2s.out.tag = ack0 | s.r2s.out.tag = ack1) &
      EGfair_sender_not_get(s))
    |
    s.sender.state != get &
    (exists State t. Trans(s,t) &
      EGfair_sender_not_get_R2Sfair(t))
  )
  & Reachable(s)
  ;

nu bool EGfair_sender_not_get(State s)
  s.sender.state != get &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_Sender(t)) &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_S2R(t)) &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_S2Rfair(t)) &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_Receiver(t)) &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_R2S(t)) &
  (exists State t. Trans(s,t) &
                   EGfair_sender_not_get_R2Sfair(t))
```

```
  & Reachable(s)
  ;

mu bool EF_EGfair_sender_not_get(State s)
  (
    EGfair_sender_not_get(s) |
    (exists State t. Trans(s,t) &
                     EF_EGfair_sender_not_get(t))
  )
  & Reachable(s)
  ;

#print "
'AG AF sender.state.get' with fairness that each process
is running infinitetly often and each channel is fair:
";

forall State s. Start(s) -> ! EF_EGfair_sender_not_get(s);
```

C.4.5 BDD der Übergangsrelation des ABP

Dies ist der BDD der Übergangsrelation „`Trans`“ des ABP, erzeugt mit der μcke durch den Befehl „`#vis Trans;`“ unter Verwendung des vom Autor für die Visualisierung von BDDs entwickelten „Graphenlayouter“ `lg`.



C.5 Der Scheduler von Milner

Dieses Beispiel stammt aus /Milner, 1989/, wird aber in der Version von /Enders et al., 1993/ verwendet. Diese Version verwendet eine viel schwächere Spezifikation als in der ursprünglichen Formulierung.[1] Zusätzlich wird hier die Übergangsrelation des Gesamtsystems unter der Annahme, daß nur erreichbare Zustände betrachtet werden, vereinfacht. Die Erreichbarkeitsanalyse wird nur für den Scheduler und nicht für die Spezifikation durchgeführt (in der Spezifikation sind alle Zustände) erreichbar), wobei von den Aktionen abstrahiert wird.

Für die Übersetzung von CCS (s. /Milner, 1989/) nach μ-Kalkül wurde die Methode aus /Enders et al., 1993/ von BDDs auf den μ-Kalkül übertragen.

```
#print "
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% this is the famous scheduler of Milner %
% with forward analysis optimization     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
";
#print;
#print "=>  2  schedulers  <=";
#print;

enum StarterState { start, end };
enum ControllerState {
  start, next, decision, firstInternal, firstExternal
};
enum PreAction { tau, a0, c0, a1, c1 };

class Action { bool isInput; PreAction action; };
class SchedState {
  StarterState starter;
  ControllerState controllers[2];
};

bool isTau(Action a) !a.isInput & a.action = tau;

bool StartOfSched(SchedState s)
  s.starter = start &
  s.controllers[0] = start &
  s.controllers[1] = start &
  1;

bool TransOfStarter(Action a, SchedState s, SchedState t)
  s.starter = start &
    a.isInput & a.action = c0 &
    t.starter = end;

bool TransOfController0(Action a, SchedState s, SchedState t)
  s.controllers[0] = start &
    !a.isInput & a.action = c0 &
```

[1] In der μcke konnte auch die ursprüngliche Spezifikation beschrieben werden (s. /Jäger, 1996/).

```
    t.controllers[0] = next              |

  s.controllers[0] = next &
    !a.isInput & a.action = a0 &
    t.controllers[0] = decision          |

  s.controllers[0] = decision &
    a.isInput & a.action = c1 &
    t.controllers[0] = firstExternal     |

  s.controllers[0] = decision &
    isTau(a) &
    t.controllers[0] = firstInternal     |

  s.controllers[0] = firstExternal &
    isTau(a) &
    t.controllers[0] = start             |

  s.controllers[0] = firstInternal &
    a.isInput & a.action = c1 &
    t.controllers[0] = start;

bool TransOfController1(Action a, SchedState s, SchedState t)
  s.controllers[1] = start &
    !a.isInput & a.action = c1 &
    t.controllers[1] = next              |

  s.controllers[1] = next &
    !a.isInput & a.action = a1 &
    t.controllers[1] = decision          |

  s.controllers[1] = decision &
    a.isInput & a.action = c0 &
    t.controllers[1] = firstExternal     |

  s.controllers[1] = decision &
    isTau(a) &
    t.controllers[1] = firstInternal     |

  s.controllers[1] = firstExternal &
    isTau(a) &
    t.controllers[1] = start             |

  s.controllers[1] = firstInternal &
    a.isInput & a.action = c0 &
    t.controllers[1] = start;

bool CoStabStarter(SchedState s, SchedState t)
  s.controllers[0] = t.controllers[0] &
  s.controllers[1] = t.controllers[1] &
  1;
```

```
bool CoStab0to1(SchedState s, SchedState t)
  1;

bool CoStab0from0to1(SchedState s, SchedState t)
  s.controllers[1] = t.controllers[1]
  ;

bool CoStab1from0to1(SchedState s, SchedState t)
  s.controllers[0] = t.controllers[0]
  ;

bool sched0to1(Action a, SchedState s, SchedState t)
  TransOfController0(a,s,t) & CoStab0from0to1(s,t) |
  TransOfController1(a,s,t) & CoStab1from0to1(s,t) |
  isTau(a) & CoStab0to1(s,t) &
  (exists Action b, Action c.
    b.isInput != c.isInput & b.action = c.action &
    TransOfController0(b,s,t) &
    TransOfController1(c,s,t));

bool SchedNonProjected(Action a, SchedState s, SchedState t)
  sched0to1(a,s,t) & s.starter = t.starter |
  TransOfStarter(a,s,t) & CoStabStarter(s,t) |
  isTau(a) &
  (exists Action b, Action c.
    b.isInput != c.isInput & b.action = c.action &
    TransOfStarter(b,s,t) &
    sched0to1(c,s,t));

bool _Sched(Action a, SchedState s, SchedState t)
  SchedNonProjected(a,s,t) &
  a.action != c0 &
  a.action != c1 ;

bool _SchedWithoutAction(SchedState s, SchedState t)
  exists Action a. _Sched(a,s,t);

mu bool ReachableSched(SchedState s)
  StartOfSched(s) |
  (exists SchedState t. _SchedWithoutAction(t, s) &
                        ReachableSched(t));

bool Sched(Action a, SchedState s, SchedState t)
  _Sched(a,s,t) assume ReachableSched(s);

enum SpecState { 0 .. 1 };

bool StartOfSpec(SpecState s) s = 0;

bool TransSpec(Action a, SpecState s, SpecState t)
```

```
  !a.isInput &
  (
    s = 0 & a.action = a0 & t = 1 |
    s = 1 & a.action = a1 & t = 0
  );

bool TSpec(SpecState s, SpecState t)
  exists Action a. isTau(a) & TransSpec(a,s,t);

mu bool TSpecStar(SpecState s, SpecState t)
  s = t |
  (exists SpecState intermediate.
    TSpec(s, intermediate) & TSpecStar(intermediate, t));

bool DeltaSpec(Action a, SpecState x, SpecState y)
  isTau(a) & TSpecStar(x, y) |
  (exists SpecState z1. TSpecStar(x, z1) &
    (exists SpecState z2. TransSpec(a, z1, z2) &
                          TSpecStar(z2, y)));

bool TSched(SchedState s, SchedState t)
  exists Action a. isTau(a) & Sched(a,s,t);

mu bool TSchedStar(SchedState s, SchedState t)
  s = t |
  (exists SchedState intermediate.
    TSched(s, intermediate) & TSchedStar(intermediate, t));


bool DeltaSched(Action a, SchedState x, SchedState y)
  isTau(a) & TSchedStar(x, y) |
  (exists SchedState z1. TSchedStar(x, z1) &
    (exists SchedState z2. Sched(a, z1, z2) &
                           TSchedStar(z2, y)));

nu bool bisimulates(SchedState p1, SpecState p2) p1 ~+ p2
 (forall Action a, SchedState q1.
   DeltaSched(a, p1, q1) ->
     (exists SpecState q2. DeltaSpec(a, p2, q2) &
                           bisimulates(q1, q2))) &
 (forall Action b, SpecState r2.
   DeltaSpec(b, p2, r2) ->
     (exists SchedState r1. DeltaSched(b, p1, r1) &
                            bisimulates(r1, r2)));

forall SchedState s, SpecState t.
  StartOfSched(s) & StartOfSpec(t) -> bisimulates(s,t);
```

C.6 DME

Diese Version des asynchronen DME-Schaltkreis aus /Martin, 1985/ (und /McMillan, 1993a/) wird hier mit inkrementeller Generierung der Übergangsrelation formuliert. Dies bedeutet, daß die einzelnen Zellen auch als Fixpunkte definiert werden. Wie im SMV-System fest implementiert wird aber über alle Konjunkte der Übergangsrelation das `assume` hinwegezogen. Das ergibt die „`assume Reachable(s)`“-Terme im Rumpf der Zellen.

Als weitere Optimierung wurde auf oberster Ebene statt `assume` der „$\downarrow$“-Operator verwendet, welcher in der Eingabesprache der μcke `cofactor` lautet. Damit reduziert sich das Relationenprodukt zu einem einfachen Quantor, was bei mehr als 4 Zellen eine Ersparnis von etwa 10% ergab.

```
class GateState { bool out; };

bool TransAndGate(bool in1, bool in2,
                  GateState state, GateState next)
  (next.out <-> (in1 & in2)) |
  (next.out <-> state.out);

bool TransOrGate(bool in1, bool in2,
                 GateState state, GateState next)
  (next.out <-> (in1 | in2)) |
  (next.out <-> state.out);

bool TransCElement(bool in1, bool in2,
                   GateState state, GateState next)
  case
    in1 = in2 : next.out = in1 | next.out = state.out;
    1 : next.out = state.out;
  esac
  ;

// here we have a slightly different semantic than
// that of the smv description

bool TransMutexHalf(bool inp, bool next_otherOut,
                    GateState state, GateState next)
  (next.out = inp | next.out = state.out)
  &
  !(next.out & next_otherOut);

class UserState { bool req; };

bool TransUser(bool ack, UserState state, UserState next)
  next.req != ack | next.req = state.req;

// this order of variables is also used in the
// asynchronous version of the SMV-description

class StateCell {
  GateState q, f, d, b, i, h, n;
```

```
  UserState u;
  GateState  a, c, g, e, k, l, p, m, r, j;
};

class State { StateCell e_2, e_1; };

bool StartCell(StateCell state, bool token)
  state.u.req = 0 &
  state.a.out = 0 &
  state.b.out = 0 &
  state.c.out = 0 &
  state.d.out = 0 &
  state.g.out = 0 &
  state.e.out = 0 &
  state.f.out = 0 &
  state.h.out = 0 &
  state.k.out = 0 &
  state.i.out = 0 &
  state.l.out = 0  &
  state.j.out = 0  &
  state.p.out = 0 &
  state.n.out != token &
  state.m.out = token &
  state.r.out = 0 &
  state.q.out = 0;

mu bool Reachable(State s);

mu bool TransCell1(State s, State t)
  ((TransUser(s.e_1.r.out, s.e_1.u, t.e_1.u) &
  TransMutexHalf(s.e_1.u.req, t.e_1.b.out, s.e_1.a, t.e_1.a) &
      (exists bool l_req. l_req = s.e_2.p.out &
  TransMutexHalf(l_req, t.e_1.a.out, s.e_1.b, t.e_1.b)
      ) &
      (exists bool not_l_ack. not_l_ack != s.e_1.q.out &
  TransAndGate(s.e_1.a.out, not_l_ack, s.e_1.c, t.e_1.c)
      )) assume Reachable(s) &
      ((exists bool not_u_ack. not_u_ack != s.e_1.r.out &
  TransAndGate(s.e_1.b.out, not_u_ack, s.e_1.d, t.e_1.d)
      ) &
  TransOrGate(s.e_1.c.out, s.e_1.d.out, s.e_1.g, t.e_1.g) &
  TransCElement(s.e_1.c.out, s.e_1.i.out, s.e_1.e, t.e_1.e) &
  TransCElement(s.e_1.d.out, s.e_1.i.out, s.e_1.f, t.e_1.f) &
  TransCElement(s.e_1.g.out, s.e_1.j.out, s.e_1.h, t.e_1.h)
         )assume Reachable(s)) assume Reachable(s) &
      (((exists bool not_h_out. not_h_out != s.e_1.h.out &
  TransAndGate(s.e_1.g.out, not_h_out, s.e_1.k, t.e_1.k)
      ) &
      (exists bool not_j_out. not_j_out != s.e_1.j.out &
  TransAndGate(s.e_1.h.out, not_j_out, s.e_1.i, t.e_1.i)
      ) &
```

```
  TransAndGate(s.e_1.k.out, s.e_1.m.out, s.e_1.l, t.e_1.l) &
      (exists bool ack. ack = s.e_2.q.out &
  TransOrGate(s.e_1.l.out, ack, s.e_1.j, t.e_1.j)
      )) assume Reachable(s) &
  (TransAndGate(s.e_1.k.out, s.e_1.n.out, s.e_1.p, t.e_1.p) &
      (exists bool not_e_out, bool not_m_out.
 (not_e_out != s.e_1.e.out & not_m_out != s.e_1.m.out) &
  TransAndGate(not_e_out, not_m_out, s.e_1.n, t.e_1.n)
      ) &
      (exists bool not_f_out, bool not_n_out.
 (not_f_out != s.e_1.f.out & not_n_out != s.e_1.n.out) &
  TransAndGate(not_f_out, not_n_out, s.e_1.m, t.e_1.m)
      ) &
  TransAndGate(s.e_1.e.out, s.e_1.m.out, s.e_1.r, t.e_1.r) &
  TransAndGate(s.e_1.f.out, s.e_1.n.out, s.e_1.q, t.e_1.q))
      assume Reachable(s)) assume Reachable(s)
  ;

mu bool TransCell2(State s, State t)
  ((TransUser(s.e_2.r.out, s.e_2.u, t.e_2.u) &
  TransMutexHalf(s.e_2.u.req, t.e_2.b.out, s.e_2.a, t.e_2.a) &
      (exists bool l_req. l_req = s.e_1.p.out &
  TransMutexHalf(l_req, t.e_2.a.out, s.e_2.b, t.e_2.b)
      ) &
      (exists bool not_l_ack. not_l_ack != s.e_2.q.out &
  TransAndGate(s.e_2.a.out, not_l_ack, s.e_2.c, t.e_2.c)
      )) assume Reachable(s) &
      ((exists bool not_u_ack. not_u_ack != s.e_2.r.out &
  TransAndGate(s.e_2.b.out, not_u_ack, s.e_2.d, t.e_2.d)
      ) &
  TransOrGate(s.e_2.c.out, s.e_2.d.out, s.e_2.g, t.e_2.g) &
  TransCElement(s.e_2.c.out, s.e_2.i.out, s.e_2.e, t.e_2.e) &
  TransCElement(s.e_2.d.out, s.e_2.i.out, s.e_2.f, t.e_2.f) &
  TransCElement(s.e_2.g.out, s.e_2.j.out, s.e_2.h, t.e_2.h)
         )assume Reachable(s)) assume Reachable(s) &
      (((exists bool not_h_out. not_h_out != s.e_2.h.out &
  TransAndGate(s.e_2.g.out, not_h_out, s.e_2.k, t.e_2.k)
      ) &
      (exists bool not_j_out. not_j_out != s.e_2.j.out &
  TransAndGate(s.e_2.h.out, not_j_out, s.e_2.i, t.e_2.i)
      ) &
  TransAndGate(s.e_2.k.out, s.e_2.m.out, s.e_2.l, t.e_2.l) &
      (exists bool ack. ack = s.e_1.q.out &
  TransOrGate(s.e_2.l.out, ack, s.e_2.j, t.e_2.j)
      )) assume Reachable(s) &
  (TransAndGate(s.e_2.k.out, s.e_2.n.out, s.e_2.p, t.e_2.p) &
      (exists bool not_e_out, bool not_m_out.
 (not_e_out != s.e_2.e.out & not_m_out != s.e_2.m.out) &
  TransAndGate(not_e_out, not_m_out, s.e_2.n, t.e_2.n)
      ) &
      (exists bool not_f_out, bool not_n_out.
```

```
 (not_f_out != s.e_2.f.out & not_n_out != s.e_2.n.out) &
  TransAndGate(not_f_out, not_n_out, s.e_2.m, t.e_2.m)
      ) &
  TransAndGate(s.e_2.e.out, s.e_2.m.out, s.e_2.r, t.e_2.r) &
  TransAndGate(s.e_2.f.out, s.e_2.n.out, s.e_2.q, t.e_2.q))
      assume Reachable(s)) assume Reachable(s)
  ;

bool Start(State s) StartCell(s.e_1, 0) & StartCell(s.e_2, 1);

mu bool Reachable(State s);

mu bool _Trans(State s, State t) s ~+ t
  TransCell1(s,t) cofactor Reachable(s) &
  TransCell2(s,t) cofactor Reachable(s) ;

bool Save(State s) ! ( s.e_1.r.out & s.e_2.r.out ) ;

// because we used cofactor above we do not have
// to use a relational product below!

mu bool Reachable(State s)
  Start(s) | (exists State prev. _Trans(prev,s));

#ons Reachable;

forall State s. Reachable(s) -> Save(s);
```

C.7 4-Bit Zähler

```
#print "+---------------+";
#print "| N-Bit Counter |";
#print "+---------------+";
#print;

#print "  -> 4 bits <-";
#print;

class Data { bool bits[ 4 ]; };

bool zero(Data a)
  a.bits[0] = 0 & a.bits[1] = 0 &
  a.bits[2] = 0 & a.bits[3] = 0 ;

bool inc3(Data a, Data b)
  if(a.bits[3] = 0)
    (
      b.bits[3] = 1
    )
  else
    (
      b.bits[3] = 0
    )
  ;

bool inc2(Data a, Data b)
  if(a.bits[2] = 0)
    (
      b.bits[2] = 1 &
      b.bits[3] = a.bits[3]
    )
  else
    (
      b.bits[2] = 0 &
      inc3(a,b)
    )
  ;

bool inc1(Data a, Data b)
  if(a.bits[1] = 0)
    (
      b.bits[1] = 1 &
      b.bits[2] = a.bits[2] &
      b.bits[3] = a.bits[3]
    )
  else
    (
      b.bits[1] = 0 &
      inc2(a,b)
```

```
    )
  ;

bool inc0(Data a, Data b) a ~+ b
  if(a.bits[0] = 0)
    (
      b.bits[0] = 1 &
      b.bits[1] = a.bits[1] &
      b.bits[2] = a.bits[2] &
      b.bits[3] = a.bits[3]
    )
  else
    (
      b.bits[0] = 0 &
      inc1(a,b)
    )
  ;

mu bool Reachable(Data a)
  zero(a) |
  (exists Data b. inc0(b,a) & Reachable(b));

#print "calculate number of reachable states";
#print "(forward analysis)";
#print;

#ons Reachable;

#print statistics;
```

C.8 Arbiter

```
#print "  -> 4 cells <-";

class Arbiter { bool token, request, writer; };
class State { Arbiter arbiter[4]; };

bool only_one4(bool a0, bool a1, bool a2, bool a3)
  a0 & !a1 & !a2 & !a3 |
  a1 & !a0 & !a2 & !a3 |
  a2 & !a0 & !a1 & !a3 |
  a3 & !a0 & !a1 & !a2 ;

bool Start(State s)
  !s.arbiter[0].writer & !s.arbiter[1].writer &
  !s.arbiter[2].writer & !s.arbiter[3].writer &

  only_one4(s.arbiter[0].token, s.arbiter[1].token,
            s.arbiter[2].token, s.arbiter[3].token);

#size Start;

bool NoRequest(State s)
  !s.arbiter[0].request & !s.arbiter[1].request &
  !s.arbiter[2].request & !s.arbiter[3].request ;

bool TransArbiter(Arbiter s, Arbiter t)
  t.writer <-> (s.writer | s.token) & s.request;

bool Trans(State s, State t) s ~+ t
  t.arbiter[0].token = s.arbiter[3].token &
  t.arbiter[1].token = s.arbiter[0].token &
  t.arbiter[2].token = s.arbiter[1].token &
  t.arbiter[3].token = s.arbiter[2].token &

  TransArbiter(s.arbiter[0], t.arbiter[0]) &
  TransArbiter(s.arbiter[1], t.arbiter[1]) &
  TransArbiter(s.arbiter[2], t.arbiter[2]) &
  TransArbiter(s.arbiter[3], t.arbiter[3])
  ;

#size Trans;

mu bool Reachable(State s)
  Start(s) | (exists State t. Trans(t,s) & Reachable(t));

bool RealReachable(State s) Reachable(s) & NoRequest(s);

#ons RealReachable;
#print statistics;
```

Literaturverzeichnis

/Aho et al., 1986/ A. V. Aho, R. Sethi und J. D. Ullman. *COMPILERS Principles, Techniques and Tools*. Bell Telephone Laboratories.

/Alur und Henzinger, 1996/ R. Alur und T. Henzinger, Herausgeber. *Computer Aided Verification, 8th International Conference, CAV'96*, Band 1102. Springer-Verlag.

/Arnold, 1994/ A. Arnold. *Finite Transition Systems*. Prentice Hall.

/Aziz et al., 1996/ A. Aziz, K. Sanwal, V. Singhal und R. Brayton. Verifying Continous Markov Chains. In /Alur und Henzinger, 1996/, S. 269–276.

/Bartlett et al., 1969/ K. Bartlett, R. Scantlebury und P. Wilkinson. A note on reliable full-duplex transmissions over half-duplex lines. *Communications of the ACM*, 5(2):260–261.

/Beer et al., 1994/ I. Beer, S. Ben-David, D. Geist, R. Gewirtzman und M. Yoeli. Methodology and System for Practical Formal Verification of Reactive Hardware. In /Dill, 1994/, S. 182–193.

/Berman, 1991/ C. L. Berman. Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams. *IEEE Transactions on Computer-Aided Design*, 10(8):1059–1066.

/Bern et al., 1995/ J. Bern, C. Meinel und A. Slobodová. Global Rebuilding of BDDs, Avoiding Memory Requirement Maxima. In /Wolper, 1995/, S. 4–15.

/Bhat und Cleaveland, 1996/ G. Bhat und R. Cleaveland. Efficient Local Model Checking for Fragments of the Modal μ-Calculus. In T. Margaria und B. Steffen, Herausgeber, *Tools and Algorithms for the Construction and Analysis of Systems*, Band 1055 von *LNCS*, S. 107–126, Berlin. Springer.

/Biere, 1996/ A. Biere. Efficient μ-Calculus Model Checking with BDDs – Abstract. In U. Herzog und H. Hermanns, Herausgeber, *Formale Beschreibungstechniken für Verteilte Systeme*, Nummer 8 Band 29 in Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung (Informatik), S. 75–76.

/Biere und Kick, 1995/ A. Biere und A. Kick. A case study on different modelling approaches based on model checking - verifying numerous versions of the alternating bit protocol with SMV. Technischer Bericht 5/95, Faculty of Computer Science, University of Karlsruhe, D-76128 Karlsruhe, Germany.

/Boigelot und Godefroid, 1996/ B. Boigelot und P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. In /Alur und Henzinger, 1996/, S. 1–12.

/Brace et al., 1990/ K. S. Brace, R. L. Rudell und R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, S. 40–45. IEEE.

/Bryant, 1986/ R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

/Bryant, 1991/ R. E. Bryant. On the complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication. *IEEE Transactions on Computing*, 40(2):205–213.

/Burch et al., 1994/ J. R. Burch, 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.

/Burch et al., 1992/ J. R. Burch, E. M. Clarke und K. L. McMillan. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98:142–170.

/Burch et al., 1990/ J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill und K. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, S. 428–439. IEEE.

/CCITT, 1992/ CCITT. Recommendations Z.100, Specification and Description Language (SDL).

/Clarke et al., 1994/ E. Clarke, O. Grumberg und K. Hamaguchi. Another Look at LTL Model Checking. In /Dill, 1994/, S. 415–427.

/Clarke et al., 1993/ E. Clarke, O. Grumberg und D. Long. Verification Tools for Finite-State Concurrent Systems. In *A Decade of Concurrency*, Band 803 von *LNCS*, S. 124–175. REX School Symposium, Springer.

/Clarke et al., 1986/ E. M. Clarke, 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*, 8(2):244 – 263.

/Cleaveland, 1990/ R. Cleaveland. Tableau-Based Model Checking in the Propositional Mu-Calculus. *Acta Informatica*, 27:725–747.

/Cleaveland, 1992/ R. Cleaveland. Analyzing Concurrent Systems Using the Concurrency Workbench. In P. E. Lauer, Herausgeber, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, International Lecture Series 1991–1992, McMaster University, Hamilton, Ontario, S. 129–144. Springer-Verlag.

/Cleaveland, 1993/ R. Cleaveland. The Concurrency Workbench: A Semantics-Based Verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72.

/Cleaveland und Steffen, 1993/ R. Cleaveland und B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *Formal Methods in System Design*, 2:121–147.

/Coplien, 1992/ J. O. Coplien. *Advanced C++ programming styles and idioms*. Bell Telephone Laboratories.

/Coudert et al., 1989/ O. Coudert, C. Berthet und J.-C. Madre. Verification of Sequential Machines Using Functional Vectors. In L. J. M. Claesen, Herausgeber, *Formal VLSI Correctness Verification. VLSI Design Methods-II*, S. 197–204. IFIP, North-Holland.

/Coudert und Madre, 1990/ O. Coudert und J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In /ICCAD90, 1990/, S. 126–129.

/Dam, 1994/ M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96.

/Dershowitz und Jouannaud, 1990/ N. Dershowitz und J.-P. Jouannaud. Rewrite Systems. In /van Leeuwen, 1990/, S. 243–320.

/Deussen, 1992/ P. Deussen. Vorlesung Reduktionssysteme, SS92. Mitschrieb, Universität Karlsruhe Fakultät für Informatik, Postfach 6980, D-76128 Karlsruhe, Germany.

/Dicky, 1986/ A. Dicky. An algebraic and algorithmic method for analysing transition systems. *Theorectical Computer Science*, (46):285–303.

/Dill, 1994/ D. L. Dill, Herausgeber. *Computer Aided Verification, 6th International Conference, CAV'94*, Band 818 von *LNCS*. Springer-Verlag.

/Dsouza und Bloom, 1995/ A. Dsouza und B. Bloom. Generating BDD models for process algebra terms. In /Wolper, 1995/, S. 17–30.

/Ehrig, 1979/ H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proc. 1st Graph Grammar Workshop*, Band 73 von *Lecture Notes in Computer Science*, S. 1–69. Springer-Verlag.

/Eiríksson und McMillan, 1995/ Á. T. Eiríksson und K. L. McMillan. Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study. In /Wolper, 1995/, S. 367–380.

/Emerson, 1990/ E. A. Emerson. Temporal and Modal Logic. In /van Leeuwen, 1990/, S. 996 – 1072.

/Emerson und Lei, 1986/ E. A. Emerson und C.-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *IEEE Symposium on Logic in Computer Science*, S. 267–278.

/Enders et al., 1993/ R. Enders, T. Filkorn und D. Taubner. Generating BDDs for symbolic model checking. *Distributed Computing*, 6:155–164.

/ESA-CNES, 1996/ ESA-CNES, Joint Press Release Ariane 501, `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`.

/Fairley, 1985/ R. Fairley. *Software Engineering Concepts*. McGraw-Hill.

/Flanagan, 1996/ D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates.

/Fujii et al., 1993/ H. Fujii, G. Ootomo und C. Hori. Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams. In /ICCAD93, 1993/, S. 38–41.

/Fujita et al., 1988/ M. Fujita, H. Fujisawa und N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In /ICCAD88, 1988/, S. 2–5.

/Gamma et al., 1995/ E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

/Garey und Johnson, 1979/ M. R. Garey und D. S. Johnson. *Computers and Intractability*. W. H. Freeman, San Francisco.

/Gollner, 1996/ M. Gollner. *Eine abstrakte Maschine für das logisch-funktionale Programmieren*. Dissertation, Fakultät für Informatik, Universität Karlsruhe.

/Habel und Plump, 1996/ A. Habel und D. Plump. Graph Unification and Matching. In J. Cuny, H. Ehrig, G. Engels und G. Rozenberg, Herausgeber, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, Band 1073 von *Lecture Notes in Computer Science*, S. 75–88. Springer-Verlag.

/Halbwachs et al., 1991/ N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. The synchronous dataflow programming language LUSTRE. 9(79):1305–1320.

/Harel, 1988/ D. Harel. On visual formalisms. *Communications of the ACM*, 31(5).

/Henzinger et al., 1993/ T. A. Henzinger, Z. Manna und A. Pnueli. Temporal Verification Proof Methodologies for timed transition systems. *Information and Control*.

/Hoffmann und Plump, 1988/ B. Hoffmann und D. Plump. Jungle evaluation for efficient term rewriting. In J. G. P. L. W. Wechler, Herausgeber, *Proceedings of the International Workshop on Algebraic and Logic Programming*, Band 343 von *LNCS*, S. 191–203, Berlin. Springer-Verlag.

/Hogrefe, 1989/ D. Hogrefe. *Estelle, LOTUS und SDL*. Springer Compass. Springer-Verlag, Berlin.

/Holzmann, 1991/ G. J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall.

/ICCAD88, 1988/ ICCAD88. *IEEE Intl. Conference on Computer-Aided Design*.

/ICCAD90, 1990/ ICCAD90. *IEEE Intl. Conference on Computer-Aided Design*.

/ICCAD93, 1993/ ICCAD93. *IEEE Intl. Conference on Computer-Aided Design*.

/Jacobson, 1985/ N. Jacobson. *Basic Algebra I*. W. H. Freeman, New York, 2. Ausgabe.

/Jäger, 1996/ U. Jäger. Gegenbeispielgenerierung für den μ-Kalkül-Modellprüfer μcke. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

/Janssen, 1993/ G. Janssen. ROBDD Software. Technischer Bericht, Department of Electrical Engineering, Eindhoven University of Technology.

/Janssen, 1996a/ G. Janssen, bdd: routine package for logic function manipulation, `http://www.es.ele.tue.nl/geert/research/research_bdd.html`.

/Janssen, 1996b/ G. Janssen, mu: Boolean mu-Calculus Tool, `http://www.es.ele.tue.nl/geert/research/research_mu.html`.

/Jones und Lins, 1996/ R. Jones und R. Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.

/Jones, 1987/ S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall.

/Kick, 1996/ A. Kick. *Generierung von Gegenbeispielen und Zeugen bei der Modellprüfung*. Dissertation, Fakultät für Informatik, Universität Karlsruhe.

/Kit, 1995/ E. Kit. *Software testing in the real world: improving the process*. ACM-press books. Addison-Wesley.

/Knuth, 1973/ D. E. Knuth. *The Art of Computer Programming (I), Fundamental Algorithms*. Addison-Wesley, 2. Ausgabe.

/Kogge, 1991/ P. M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill.

/Kozen, 1983/ D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354.

/Lloyd, 1987/ J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag.

/Long, 1994/ D. E. Long, The BDD Library, `http://www.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html`.

/Malik et al., 1988/ S. Malik, A. R. Wang, R. K. Brayton und A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams. In /ICCAD88, 1988/, S. 6–9.

/Manber, 1989/ U. Manber. *Introduction to Algorithms, A Creative Approach*. Addison-Wesley.

/Martin, 1985/ A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, Herausgeber, *1985 Chapel Hill Conference on VLSI*, S. 245–260. Computer Science Press.

/McCalla, 1992/ T. R. McCalla. *Digital Logic and Computer Design*. Macmillan Publishing Company.

/McMillan, 1993a/ K. L. McMillan. *Symbolic Model Checking*. Kluwer.

/McMillan, 1993b/ K. L. McMillan, The SMV System, `http://www.cs.cmu.edu/afs/cs/project/modck/pub/www/smv.html`.

/McMillan, 1996/ K. L. McMillan. A Conjunctively Decomposed Boolean Representation for Symbolic Model Checking. In /Alur und Henzinger, 1996/, S. 13–25.

/Meinel und Slobodova, 1994/ C. Meinel und A. Slobodova. On the Complexity of Constructing Optimal OBDD's. Technischer Bericht 94-05, FB IV - Informatik, Universität Trier.

/Melcher, 1995/ H. Melcher. Komponentenorientierter Entwurf eines Modellprüfers am Beispiel von Argos. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

/Milner, 1989/ D. Milner. *Communication and Concurrency*. Prentice Hall.

/Paoli, 1996/ H. Paoli. Visualisierung von BDD-Algorithmen in Java. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

/Park, 1976/ D. Park. Finiteness is Mu-Ineffable. *Theoretical Computer Science*, 1(3):173–181.

/Park, 1981/ D. Park. Concurrency and automata on infinite sequences. In P. Deussen, Herausgeber, *Theoretical Computer Science, 5th GI-Conference*, Band 104 von *LNCS*, S. 167–183. Springer-Verlag.

/Philipps und Scholz, 1997/ J. Philipps und P. Scholz. Formal Verification of Statecharts With Instantaneous Chain Reactions. Eingereicht für TACAS'97.

/Plump, 1992/ D. Plump. Collapsed Tree Rewriting: Completeness, Confluence, and Modularity. In M. Rusinowitch und J.-L. Rémy, Herausgeber, *Conditional Term Rewriting Systems, Third International Workshop*, Band 342 von *LNCS*, S. 97–112, Pont-à-Mousson, France. Springer-Verlag.

/Rauzy, 1995/ A. Rauzy. Toupie = μ-calculus + constraints. In /Wolper, 1995/, S. 114–126.

/Reffel, 1996/ F. Reffel. Modellprüfung von Unterlogiken von CTL*. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

/Rinderspacher, 1996/ M. Rinderspacher. *Mathematische Modellierung und Verifikation von SDL-Systemen*. Dissertation, Fakultät für Informatik, Universität Karlsruhe.

/Rudell, 1993/ R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In /ICCAD93, 1993/, S. 42–47.

/Schmitt, 1992a/ P. Schmitt. Skriptum zur Vorlesung Nichtklassische Logiken. Skriptum, Universität Karlsruhe Fakultät für Informatik, Postfach 6980, D-76128 Karlsruhe, Germany.

/Schmitt, 1992b/ P. Schmitt. *Theorie der logischen Programmierung*. Springer-Verlag.

/Sieling, 1995/ D. Sieling. *Algorithmen und untere Schranken für verallgemeinerte OBDDs*. Dissertation, Universität Dortmund.

/Stroustrup, 1991/ B. Stroustrup. *The C++ programming language*. Bell Telephone Laboratories, 2. Ausgabe.

/Tarski, 1955/ A. Tarski. A Lattice Theoretic Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309.

/Theobald und Meinel, 1996/ T. Theobald und C. Meinel. State Encodings and OBDD-Sizes. Technischer Bericht 96-04, FB IV - Informatik, Universität Trier.

/Touati et al., 1990/ H. J. Touati, H. Savoj, B. Lin, R. K. Brayton und A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In /ICCAD90, 1990/, S. 130–133.

/van Leeuwen, 1990/ J. van Leeuwen, Herausgeber. Band B, Formal Methods and Semantics. Elsevier, Amsterdam.

/Warren, 1983/ D. H. D. Warren. An Abstract Prolog Instruction Set. Tech. Note 309, SRI International.

/Winskel, 1991/ Winskel. A Note on Model Checking the Modal ν-Calculus. *Theoretical Computer Science*, 83:157 – 167.

/Winskel, 1989/ G. Winskel. A note on model checking the modal ν-calculus. In G. Ausiello, M. Dezani-Ciancaglini und S. R. D. Rocca, Herausgeber, *Automata, Language and Programming, 16th International Colloquium, ICALP'89*, Band 372 von *Lecture Notes in Computer Science*, S. 761–772. Springer-Verlag.

/Winskel, 1993/ G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press.

/Wirth, 1986/ N. Wirth. *Algorithmen und Datenstrukturen mit Modula-2*. B.G. Teubner Stuttgart, 4. Ausgabe.

/Wolper, 1995/ P. Wolper, Herausgeber. *Computer Aided Verification, 7th International Conference, CAV'95*, Band 939 von *LNCS*. Springer-Verlag.

Index

Lebenslauf

Geburt:	22. April 1967 in Villingen-Schwenningen
Heirat:	24. Februar 1995
Grundschule:	1973–1977 Berneckschule in Schramberg
Gymnasium:	1977–1986 Gymnasium Schramberg
Wehrdienst:	1986–1987 Stetten a. k. M.
Studium:	1987–1993 Fakultät für Informatik, Universität Karlsruhe. 1989 Vordiplom, 1993 Diplom
Graduiertenkolleg:	1993–1996 Graduiertenkolleg „Berherrschbarkeit komplexer Systeme“, Fakultät für Informatik, ILKD, Lehrstuhl Prof. P. Deussen, Universität Karlsruhe