ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG
INSTITUT FÜR INFORMATIK

Lehrstuhl für Rechnerarchitektur
Prof. Dr. Bernd Becker

Bounded Model Checking auf stochastischen Systemen

Diplomarbeit

Bettina Claudia Braitling

17. November 2008

Hinweis:
Diese Diplomarbeit wurde gemäß der Neuregelung der deutschen Rechtschreibung nach dem 1. August 2006 verfasst, wie sie in der 24. Auflage des Dudens zur deutschen Rechtschreibung festgehalten ist.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Außerdem erkläre ich, dass diese Diplomarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Bettina Claudia Braitling
Freiburg, den 17. November 2008

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Motivation

Je weiter die technologische Entwicklung voranschreitet, desto stärker ist der moderne Mensch von komplexen, elektronischen Systemen umgeben. Meistens ist er sich dessen gar nicht bewusst. Es beginnt mit der per Zeit und Außentemperatur gesteuerten Heizung, die das Wasser für die morgendliche Dusche anheizt, und endet noch lange nicht mit dem Multi-Media-PC, der im Wohnzimmer für Unterhaltung sorgt.

Diese Systeme werden teilweise durch äußere Einflüsse gesteuert, wie zum Beispiel eine Heizung durch einen Thermostat, der die Heizung je nach herrschender Raum- oder Außentemperatur an- beziehungsweise ausschaltet. Das Verhalten hängt in diesem Fall bis zu einem gewissen Grad vom Wetter ab. Ein weiteres Beispiel wäre eine vollautomatische Insulinpumpe, die, abhängig vom Blutzuckerspiegel eines Patienten, verschiedene Mengen an Insulin freigeben könnte. Hier wäre der individuelle Stoffwechsel des Patienten, der die künstliche Insulin-Versorgung benötigt, ausschlaggebend für das Verhalten des Systems.

Äußere Einflüsse, wie das Wetter oder der menschliche Stoffwechsel in den oberen Beispielen, sind meistens nicht vorhersehbar, sondern in einem gewissen Maße zufällig. Die Folge davon ist, dass das System mit einer gewissen Wahrscheinlichkeit von einem Zustand in einen anderen wechselt.

Ein weiteres „zufälliges“ Element eines Systems ist der Verschleiß einzelner Komponenten. Es muss abgeschätzt werden, wie lange und unter welcher Belastung einzelne Komponenten und das ganze System innerhalb gewisser Toleranzgrenzen arbeiten können.

Das korrekte Verhalten eines Systems ist wünschenswert, teilweise sogar lebenswichtig. Das Wasser im Heizungskessel sollte nicht kalt sein, auf der anderen Seite sollte es auch nicht so heiß werden, dass die Gefahr einer ernsthaften Verbrühung besteht. Die Insulinpumpe sollte einer Überzuckerung des Blutes entgegenwirken, auf der anderen Seite sollte sie aber auch keinen Unterzucker verursachen.

Durch äußere Einflüsse und Verschleißerscheinungen wird das Element des Zufalls zum Teil des Systems. Man bezeichnet solche Systeme als *stochastische Systeme*. Ein mögliches fehlerhaftes Verhalten eines Systems lässt sich als kritischer Zustand ansehen. Dieser Zustand sollte überhaupt nicht oder zumindest nur mit einer sehr geringen Wahrscheinlichkeit erreicht werden.

Es ist nun Aufgabe der Verifikation, sicherzustellen, dass diese Wahrscheinlichkeitsgrenzen bereits beim Entwurf des Systems eingehalten werden. Das Überprüfen eines System-Entwurfs auf gewünschte beziehungsweise unerwünschte Eigenschaften wird als *Model Checking* bezeichnet. Eine Methode dafür ist die Erzeugung von *Gegenbeispielen*. Diese bestehen im Fall von *stochastischen Systemen* aus einer Menge von Folgen von Zustandsübergängen beziehungsweise *Pfaden* innerhalb des jeweiligen Sys-

tems. Die *Wahrscheinlichkeitsmasse* dieser Menge liegt über der maximal zulässigen Wahrscheinlichkeitsgrenze.

Gegenbeispiele sind für die System-Entwicklung von einem unschätzbaren Wert. Sie zeigen nicht nur, dass ein System ein fehlerhaftes Verhalten aufweist, sondern ermöglichen auch, dieses fehlerhafte Verhalten gezielt zu reproduzieren. So können die genauen Ursachen des Fehlers untersucht werden. Mit Hilfe dieser „Ursachenforschung“ kann die Wahrscheinlichkeit des Auftretens des Fehlers bei einem neuen System-Entwurf verringert werden.

Die Erzeugung und Verwendung von *Gegenbeispielen* für *stochastische Systeme* ist bereits Gegenstand einiger anderer Arbeiten. Diese sind alle in den letzten Jahren (2005 bis 2008) entstanden und zum Großteil parallel zueinander erschienen.

Die Autoren von [2] und [3] suchen mit Hilfe von heuristischen Algorithmen wie beispielsweise Z^* nach dem Pfad mit der höchsten Wahrscheinlichkeit, der die Sicherheitsbedingung widerlegt. In [3] wird mit den gleichen Algorithmen auch nach Mengen von Pfaden gesucht. Die Autoren arbeiten auf einem expliziten Zustandsraum, dementsprechend nennen sie ihren Ansatz auch „Explicit State Model Checking“ oder kurz ESMC. In keiner dieser beiden Arbeiten wird versucht, die gefundenen *Gegenbeispiele* zu kompaktieren.

In [25] und [26] wurde ein anderer Ansatz gewählt: Hier soll das kleinste *Gegenbeispiel* mit der höchsten Wahrscheinlichkeit gefunden werden. Dabei kommt eine Kürzeste-Wege-Suche zum Einsatz. Auch hier wird die Möglichkeit einer *Gegenbeispiel-Kompaktierung* nicht angesprochen. Die gleichen Autoren sind ebenfalls an [14] beteiligt. Diese Arbeit befasst sich mit der Kompaktierung von *Gegenbeispielen* mit Hilfe von regulären Ausdrücken. Es wird ebenfalls erläutert, wie ein *stochastisches System* – dargestellt durch eine *Markov-Kette* – mit Hilfe von regulären Ausdrücken reduziert werden kann. In [14] werden außerdem einige Ergebnisse für den Ansatz der Kürzeste-Wege-Suche präsentiert. Sowohl [25] als auch [26] und [14] gehen von einem expliziten Zustandsraum aus.

[4] befasst sich ebenfalls mit der Kompaktierung von *Gegenbeispielen*. Zu diesem Zweck werden hier zunächst die starken Zusammenhangs-Komponenten – auch als SCCs bezeichnet für englisch „**S**trongly **C**onnected **C**omponents“ – eines Systems gesucht und durch Transitionen mit entsprechenden Wahrscheinlichkeiten ersetzt. Das *stochastische System* kann auf diese Weise in eine azyklische *Markov-Kette* umgewandelt werden. Ein *Gegenbeispiel* besteht bei dieser Arbeit aus mehreren Untermengen, *Zeugen* genannt, die wiederum aus Pfaden bestehen, die sich außerhalb der SCCs gleich verhalten. In [4] wird ebenfalls mit einem expliziten Zustandsraum gearbeitet.

Die Autoren von [28] befassen sich mit der Verwendung von *Gegenbeispielen* zur Verfeinerung der Prädikat-Abstraktion eines Systems. Diese Methode wird mit CEGAR für englisch „**C**ounter**e**xample-**g**uided **A**bstraction **R**efinement“ abgekürzt. Dabei wird ein System zuerst stark vereinfacht dargestellt. Mit Hilfe von *Gegenbeispielen* können die Unterschiede zwischen dieser Abstraktion und dem ursprünglichen System nachgewiesen werden, woraufhin die Abstraktion verfeinert wird. Dieser Vorgang wird so lange wiederholt, bis eine genügend genaue Approximation des Systems erreicht wurde. Um CEGAR auch für *stochastische Systeme* anweden zu können, verwenden

die Autoren von [28] den Ansatz von [25]. Es könnte aber auch ein anderer Ansatz, beispielsweise der Ansatz dieser Arbeit, verwendet werden.

Die hier vorliegende Arbeit basiert auf *Bounded Model Checking* [6]. *Bounded Model Checking* ist ursprünglich eine Methode zur Erzeugung von *Gegenbeispielen* für nichtstochastische Systeme. Die Grundlage für *Bounded Model Checking* ist das *Erfüllbarkeitsproblem*. Das System wird in einen *booleschen Ausdruck* umgewandelt, für den eine *erfüllende Belegung* gesucht wird. Diese Belegung entspricht einem Pfad einer festen Länge k innerhalb des Systems. Will man nach noch längeren Pfaden suchen, so muss der *boolesche Ausdruck* entsprechend angepasst werden.

Der Zustandsraum bei *Bounded Model Checking* wird nicht wie in den oben vorgestellten Arbeiten explizit dargestellt, sondern ist implizit in der Belegung bestimmter Variablen enthalten. Durch diese implizite Darstellung des Zustandsraums kann *Bounded Model Checking* auch bei sehr großen Systemen angewendet werden.

Ein weiterer Vorteil von *Bounded Model Checking* besteht darin, dass das gegebene System vor der Umwandlung in einen *booleschen Ausdruck* nicht lange bearbeitet werden muss. Eine aufwendige Reduktion im Vorfeld, wie sie beispielsweise in [14] oder [4] durchgeführt wird, entfällt.

Um mit Hilfe von *Bounded Model Checking* auch *Gegenbeispiele* für *stochastische Systeme* erzeugen zu können, sind besondere Vorgehensweisen erforderlich. Unter anderem muss das *Erfüllbarkeitsproblem* entsprechend aufgestellt und ausgewertet werden. Diese Vorgehensweisen sollen hier vorgestellt werden. Die dazu benötigten Grundlagen werden in Kapitel 2 erläutert. Zu diesen Grundlagen gehören das *Erfüllbarkeitsproblem*, *Bounded Model Checking* und *stochastische Systeme* in Form von *Markov-Ketten mit diskreter Zeit*.

Die kompakte Darstellung von *Gegenbeispielen* spielt ähnlich wie in den oben vorgestellten Arbeiten [14] und [28] auch hier eine wichtige Rolle. Durch eine kompakte Repräsentation kann zum einen Rechenleistung eingespart werden, zum anderen können dadurch *Gegenbeispiele* in einer verständlicheren Form dargestellt werden. Der in dieser Arbeit gewählte Ansatz zur Kompaktierung von *Gegenbeispielen* basiert auf dem Erkennen von Schleifen in bereits gefundenen Pfaden, die eine Sicherheitsbedingung widerlegen. Eine Schleife kann durch mehrfaches Durchlaufen zur Verlängerung eines bereits bekannten Pfades führen. Wird sie aber erkannt, so kann der Pfad in zwei Komponenten aufgeteilt werden: In einen *Basispfad* und eine Schleife. Der *Basispfad* gibt Auskunft über die Ursachen des aufgetretenen Fehlers, die Schleife spielt eine entscheidende Rolle beim Erreichen der benötigten *Wahrscheinlichkeitsmasse*. Der *Basispfad* und die Schleife bilden zusammen einen *erweiterten Pfad*.
Die Vorgehensweise für *Bounded Model Checking auf stochastischen Systemen* mitsamt den Optimierungen wird in Kapitel 3 dargestellt. Dabei wird auch in knapper Form auf die Implementierung, die im Zusammenhang mit dieser Arbeit entstanden ist, eingegangen.

Kapitel 4 befasst sich mit konkreten Experimenten, für die *Bounded Model Checking auf stochastischen Systemen* eingesetzt werden kann, und präsentiert einige empirisch ermittelte Ergebnisse zu diesen Experimenten.

In Kapitel 5 werden schließlich die Ergebnisse und Erkenntnisse dieser Arbeit ab-

schließend zusammengefasst und erläutert. Ebenso wird ein Ausblick mit möglichen Erweiterungen für zukünftige Entwicklungen gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für das *Bounded Model Checking auf stochastischen Systemen* vorgestellt.

Zunächst wird in Unterkapitel 2.1 das *Erfüllbarkeitsproblem* (*SAT*) behandelt. Das Erfüllbarkeitsproblem stellt die Grundlage von *Bounded Model Checking* dar und spielt damit eine wichtige Rolle für die hier vorliegende Arbeit. Um das Erfüllbarkeitsproblem zu verstehen, ist es nötig, einen Blick auf den *booleschen Kalkül* zu werfen. Im Anschluss daran wird ein Überblick über die wichtigsten Strategien zum Lösen des *Erfüllbarkeitsproblems* geliefert. Es folgt außerdem ein Exkurs über *Binary Decision Diagrams*, die ebenfalls für das *Bounded Model Checking auf stochastischen Systemen* wichtig sind. *Binary Decision Diagrams* – kurz BDDs genannt – werden benötigt, um später in Kapitel 3 ein stochastisches System in einen *booleschen Ausdruck* umzuwandeln, wie er für *Bounded Model Checking* gebraucht wird.

Mit *Bounded Model Checking* befasst sich Unterkapitel 2.2. Es wird gezeigt, wie eine Folge von Zustandsübergängen innerhalb eines Systems als boolesches *Erfüllbarkeitsproblem* formuliert werden kann.

Im letzten Teil dieses Kapitels, Unterkapitel 2.3, geht es um stochastische Systeme. Diese werden in dieser Arbeit durch *Markov-Ketten mit diskreter Zeit* dargestellt. Zu Beginn von Unterkapitel 2.3 werden darüber hinaus noch einige stochastische Grundlagen erläutert, die zum besseren Verständnis von stochastischen Systemen notwendig sind.

2.1 Das Erfüllbarkeitsproblem – SAT

Das Erfüllbarkeitsproblem oder kurz SAT – für englisch „**Sat**isfiability Problem“ – ist eines der wichtigsten und ältesten Themen der Informatik. Es besteht aus der Frage, ob ein gegebener logischer oder boolescher Ausdruck erfüllbar ist oder nicht.

Bereits 1971 bewies Cook, dass es sich bei SAT um ein NP-vollständiges Problem handelt [13]. Es ist damit das erste Problem, dem NP-Vollständigkeit nachgewiesen werden konnte. Deswegen ist es das Standard-Problem für NP-Vollständigkeit. Im Rahmen der Forschung an SAT sind bereits große Erfolge erzielt worden und es wird weiterhin intensiv an SAT gearbeitet.

Viele Probleme aus verschiedenen Anwendungsbereichen, wie zum Beispiel Künstliche Intelligenz, Verifikation, Testen, um nur ein paar zu nennen, lassen sich als Erfüllbarkeitsprobleme formulieren. SAT spielt auch beim Bounded Model Checking auf stochastischen Systemen eine entscheidende Rolle, denn es stellt die Grundlage für

Letzteres dar. Systemabläufe mit fester Länge lassen sich als boolescher Ausdruck formulieren und fallen damit in den Bereich des Erfüllbarkeitsproblems.

Im Folgenden soll SAT näher behandelt werden. Zunächst werden einige Begriffe des booleschen Kalküls erläutert, im Anschluss daran wird SAT formal definiert und die Grundlagen für die meisten der heute gängigen SAT-Solver werden kurz vorgestellt. Darauf folgt ein Exkurs über Binary Decision Diagrams, die ebenfalls zur Lösung von SAT herangezogen werden können.

2.1.1 Der boolesche Kalkül

Der boolesche Kalkül ist die Grundlage von SAT. Er ermöglicht, vielfältige Probleme als Formeln der Aussagenlogik darzustellen. Schaltkreise, aber auch andere Systeme, lassen sich als boolesche Funktion darstellen.

Die Definitionen dieses Kapitels sind in ihrer Form an [5] angelehnt.

Basis aller Überlegungen und Definitionen des booleschen Kalküls ist die Menge $\mathbb{B} = \{0, 1\}$. Die Werte 0 und 1 lassen sich dabei auch als **WAHR** oder **FALSCH** interpretieren.

Als Grundgerüst für alle Ausdrücke und Funktionen dient die boolesche Algebra:

Definition 2.1 (Boolesche Algebra)
Es sei M eine Menge, es seien $\cdot : M \times M \to M$ und $+ : M \times M \to M$ zwei binäre Verknüpfungen sowie $\neg : M \to M$ eine unäre Verknüpfung.
$(M, \cdot, +, \neg)$ *ist eine* boolesche Algebra*, wenn für alle $x, y, z \in M$ folgende Axiome gelten:*

$x + y = y + x$	*und*	$x \cdot y = y \cdot x$	*(*Kommutativität*)*
$x + (x + z) = (x + y) + z$	*und*	$x \cdot (x \cdot z) = (x \cdot y) \cdot z$	*(*Assoziativität*)*
$x + (x \cdot y) = x$	*und*	$x \cdot (x + y) = x$	*(*Absorption*)*
$x + (y \cdot z) = (x + y) \cdot (x + z)$	*und*	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	*(*Distributiviät*)*
$x + (y \cdot \neg y) = x$	*und*	$x \cdot (y + \neg y) = x$	*(*Auslöschung*)*

Aus dieser Definition folgt die Existenz der eindeutigen *Neutralen Elemente* 0 beziehungsweise 1 für $+$ beziehungsweise $\cdot$. Ebenso lassen sich daraus folgende Regeln ableiten:

Idempotenz:	$x + x = x$ und $x \cdot x = x$
de Morgan:	$\neg(x + y) = \neg x \cdot \neg y$ und $\neg(x \cdot y) = \neg x + \neg y$
Consensus:	$(x \cdot y) + (\neg x \cdot z) = (x \cdot y) + (\neg x \cdot z) + (y \cdot z)$ und
	$(x + y) \cdot (\neg x + z) = (x + y) \cdot (\neg x + z) \cdot (y + z)$

Wie aus der Definition 2.1 und den daraus folgenden Regeln ersichtlich ist, gilt in einer booleschen Algebra das Prinzip der *Dualität*. Gilt eine aus der booleschen Algebra abgeleitete Gleichung p, so gilt auch die Gleichung, die man erhält, wenn man alle $+$ und $\cdot$ und alle 0 und 1 in p vertauscht. Diese Gleichung p' bezeichnet man als *duale* Gleichung zu p.

Nachdem mit der booleschen Algebra nun das Grundgerüst festgelegt ist, ist es möglich, boolesche Funktionen zu definieren.

Definition 2.2 (Boolesche Funktion)
Es sei $\mathbb{B}^n$ *die Menge aller Wörter aus* 0 *und* 1 *der Länge* n.

- *Die Abbildung* $f : \mathbb{B}^n \to \mathbb{B}^m$ *heißt* boolesche Funktion.
- $\mathbb{B}_{n,m} := \{f \mid f : \mathbb{B}^n \to \mathbb{B}^m\}$ *ist die Menge aller* booleschen Funktionen *von* $\mathbb{B}^n$ *nach* $\mathbb{B}^m$.
- *Für* $m = 1$ *bezeichnet* $\mathrm{ON}(f)$ *die Menge aller Belegungen* α, *für die* f erfüllt *ist:*

$$\mathrm{ON}(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 1\}$$

  $\mathrm{OFF}(f)$ *bezeichnet die Menge aller Belegungen* α , *für die* f *nicht erfüllt ist:*

$$\mathrm{OFF}(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 0\}$$

- *Die Menge der* n *Variablen von* f *wird als* X_n *bezeichnet, die einzelnen Variablen von* f *haben die Bezeichnungen* $x_1, x_2, \ldots, x_n$.

Die Operatoren + (Disjunktion), $\cdot$ (Konjunktion) und $\neg$ (Negation) einer booleschen Algebra sind gewissermaßen bereits boolesche Funktionen. Da jeder Schaltkreis im Grunde genommen aus Bausteinen aufgebaut ist, die diesen Operationen entsprechen – OR, AND und NOT –, ist es offensichtlich, dass jeder Schaltkreis als boolesche Funktion dargestellt werden kann.

Boolesche Funktionen hängen eng mit booleschen Ausdrücken zusammen. Letztere sind eine Möglichkeit, boolesche Funktionen darzustellen. Boolesche Ausdrücke können verwendet werden, um alle Funktionen einer Menge $\mathbb{B}_{n,m}$ zu beschreiben. Ein boolescher Ausdruck wiederum erhält erst dadurch eine Bedeutung, dass ihm eine boolesche Funktion zugeordnet wird.

Definition 2.3 (Boolescher Ausdruck – Syntax)
Es sei $A := X_n \cup \{0, 1, \wedge, \vee, \neg, (,)\}$ *ein Alphabet.*
Die Menge $\mathrm{BE}(X_n)$ *der vollständig geklammerten* booleschen Ausdrücke *über der Variablenmenge* X_n *ist eine Teilmenge von* A^*. *Sie wird folgendermaßen induktiv definiert:*

- 0 *und* 1 *sind* boolesche Ausdrücke.
- *Die Variablen* $x_1, \ldots, x_n \in X_n$ *sind ebenfalls* boolesche Ausdrücke.
- *Sind* g *und* h boolesche Ausdrücke, *dann gilt dies auch für:*
  - $(g \wedge h)$ *(Konjunktion)*
  - $(g \vee h)$ *(Disjunktion)*
  - $\neg g$ *(Negation)*
- *Nichts sonst ist ein* boolescher Ausdruck.

Zur Vereinfachung der Schreibweise von booleschen Ausdrücken gelten fogende Konventionen:

- Die Negation $\neg$ bindet stärker als die Konjunktion $\wedge$.
- Die Konjunktion $\wedge$ bindet stärker als die Disjunktion $\vee$.

Auf die Weise können Klammern weggelassen werden, ohne dass der boolesche Ausdruck dadurch mehrdeutig wird.

Statt der Operator-Symbole $\vee$ und $\wedge$ werden auch synonym die Operator-Symbole $+$ und $\cdot$ verwendet. Um die Negation einer Variablen $\neg x$ anzuzeigen, schreibt man auch $\overline{x}$.

Es ist nun noch nötig, die Semantik von booleschen Ausdrücken zu definieren, indem man ihnen eine entsprechende boolesche Funktion zuordnet:

Definition 2.4 (Boolescher Ausdruck – Semantik)
Es sei $\phi : \mathrm{BE}(X_n) \to \mathbb{B}_n$ die Interpretationsfunktion boolescher Ausdrücke. Sie wird folgendermaßen induktiv definiert:

- $\phi(0) = 0$ *(Nullfunktion)*
- $\phi(1) = 1$ *(Einsfunktion)*
- $\phi(x_i)(\alpha_1, \ldots, \alpha_n) = \alpha_i \; \forall\, \alpha \in \mathbb{B}^n$ *(Projektion)*
- $\phi(x \vee y) = \phi(x) + \phi(x)$
- $\phi(x \wedge y) = \phi(x) \cdot \phi(y)$
- $\phi(\neg x) = \neg\phi(x)$

Es gibt noch weitere Möglichkeiten, boolesche Funktionen darzustellen. Man kann sie unter anderem auch durch Wahrheitstabellen oder Binary Decision Diagramms (siehe Kapitel 2.1.3) repräsentieren. Diese beiden Darstellungsarten haben gegenüber den booleschen Ausdrücken den Vorteil, dass sie kanonisch sind, das heißt, dass es zum Beispiel für jede boolesche Funktion (bis auf Isomorphie) genau eine Darstellung als Wertetabelle gibt.

Im Falle der booleschen Ausdrücke gibt es mehrere verschiedene Normalformen, um die Darstellung zu vereinheitlichen. Zu den wichtigsten gehört die Konjunktive Normalform – kurz KNF.

Für die Darstellung einer booleschen Funktion als KNF benötigen wir die Definitionen der Begriffe „Literal“ und „Klausel“.

Definition 2.5 (Literal)
Als Literal *einer Variable x bezeichnet man einen booleschen Ausdruck der Form x oder $\overline{x}$. x wird* positives Literal *genannt, $\overline{x}$* negatives Literal.

Definition 2.6 (Klausel)
Eine Klausel *ist eine Disjunktion von Literalen.*
Enthält eine Klausel *sowohl das positive als auch das negative Literal einer Variable, so heißt sie* Äquivalenzklausel. *Der Wert einer Äquivalenzklausel ist immer* 1.
Eine Klausel, *die sämtliche Variablen einer Funktion entweder als positives oder negatives Literal enthält, bezeichnet man als* Maxterm.

Definition 2.7 (Konjunktive Normalform – KNF)
Eine Konjunktive Normalform *ist eine Konjunktion von Klauseln.*

Die KNF ist zum Lösen und Darstellen von SAT-Problemen ausreichend. Liegt ein boolescher Ausdruck in anderer Form vor, beispielsweise als Schaltkreis oder als Binary Decision Diagramm, so kann er mit Hilfe der *Tseitin-Transformation* in eine KNF umgewandelt werden.

Durch die Tseitin-Transformation [45] lässt sich jeder boolesche Ausdruck p in eine KNF umwandeln. Die dadurch entstehende KNF p' ist nur linear größer als die ursprüngliche Formel. In p' werden Hilfsvariablen eingefügt, die für eine konsistente Belegung der Teilausdrücke von p sorgen. Es gilt:

$$p(x_1, \ldots, x_n) \equiv \exists\, h_1, \ldots, h_k : p'(x_1, \ldots, x_n, h_1, \ldots, h_k) \tag{2.1}$$

Der durch die Tseitin-Transformation entstehende boolesche Ausdruck p' ist aufgrund der zusätzlichen Variablen $h_1, \ldots, h_k$ nicht logisch äquivalent zum ursprünglichen Ausdruck p. Jedoch sind die beiden Ausdrücke erfüllbarkeitsäquivalent, das heißt, wenn p' erfüllbar ist, so ist es auch p.

Liegt der boolesche Ausdruck p beispielsweise als Schaltkreis vor, so stellen die Hilfsvariablen $h_q, \ldots, h_k$ die inneren Leitungen des Schaltkreises dar. Die Belegung der inneren Leitungen beziehungsweise der Hilfsvariablen muss konsistent sein zur Eingangsbelegung von p.

Zum Abschluss werden nun als Beispiel die Tseitin-Transformationen für die Basis-Funktionen angegeben, aus denen sich jede boolesche Funktion aufbauen lässt:

Beispiel 2.1 (Tseitin-Transformationen der Basis-Funktionen)

$$\begin{array}{lllll} p = & x_1 \cdot x_2 & \rightarrow & p' = & (h + \overline{x_1} + \overline{x_2}) \cdot \left(\overline{h} + x_1\right) \cdot \left(\overline{h} + x_2\right) \\ p = & x_1 + x_2 & \rightarrow & p' = & (h + \overline{x_1}) \cdot (h + \overline{x_2}) \cdot \left(\overline{h} + x_1 + x_2\right) \\ p = & \overline{x} & \rightarrow & p' = & (h + x) \cdot \left(\overline{h} + \overline{x}\right) \end{array}$$

2.1.2 SAT und SAT-Solver

Mit Hilfe des booleschen Kalküls kann nun das Erfüllbarkeitsproblem formal definiert werden:

Definition 2.8 (SAT)
Gegeben sei ein boolescher Ausdruck p.
Ist p erfüllbar, gibt es also eine Belegung α der Variablen von p, sodass $\phi(p)(\alpha) = 1$? Gibt es keine solche Belegung, so ist p unerfüllbar.

Algorithm 1 DPLL-Algorithmus

```
 1: function SATSolve(clause list S)
 2:     // unit propagation:
 3:     while something changed do
 4:         for each unit clause {l} ∈ S do
 5:             assign l = TRUE;
 6:             delete every clause from S containing l;
 7:             delete ¬l from every clause in S containing ¬l;
 8:         end for
 9:     end while
10:     if S is empty then
11:         return TRUE;
12:     else if S contains an empty clause {} then
13:         return FALSE;
14:     end if

15:     // branching:
16:     choose a literal l occuring in S;
17:     if SATSolve(S ∪ {l}) then
18:         return TRUE;
19:     else if SATSolve(S ∪ {¬l}) then
20:         return TRUE;
21:     else
22:         return FALSE;
23:     end if
24: end function
```

Wie oben bereits dargelegt, ist SAT ein NP-vollständiges Problem [13]. Es gibt jedoch Algorithmen, denen es gelingt, kleine und auch relativ große Instanzen des Problems effizient zu lösen. Dieses Unterkapitel soll im Folgenden kurz die wesentlichsten Mechanismen von aktuellen SAT-Solvern vorstellen. Einen detaillierteren Überblick darüber bietet [48].

Der in den meisten gängigen SAT-Solvern verwendete Algorithmus ist der Davis-Putnam-Logemann-Loveland-Algorithmus [15], abgekürzt nach den Entwicklernamen DPLL. Es handelt sich dabei um die Weiterentwicklung des Davis-Putnam-Algorithmus [16]. Algorithmus 1 zeigt eine mögliche Variante des DPLL-Algorithmus in Pseudocode.

Der DPLL-Algorithmus arbeitet auf einer Liste von Klauseln S, die einen booleschen Ausdruck p in KNF darstellt.

In den Zeilen 3 bis 9 wird zunächst nach einer Unit Clause gesucht. Eine *Unit Clause* ist eine noch unerfüllte Klausel, deren Literale alle bis auf eines mit 0 belegt worden sind. Auch eine Klausel mit einem einzigen unbelegten Literal ist eine Unit Clause. Das letzte unbelegte beziehungsweise freie Literal nennt man auch *Unit Literal*.

Die freie Variable der Unit Clause wird so belegt, dass die Klausel dadurch erfüllt wird. Zur Vereinfachung der Ausdrucksweise und ohne Beschränkung der Allgemein-

heit wird im Folgenden davon ausgegangen, dass es sich bei dem freien Literal l um ein positives Literal handelt.

Im Anschluss an die Belegung von l mit 1 werden alle Klauseln, die das Literal l der Variable enthalten, aus S gelöscht. Dies ist möglich, da alle diese Klauseln durch die Belegung von l ebenfalls erfüllt wurden. Somit tragen sie – zumindest vorerst – keine weiteren Informationen bei, die für die Erfüllbarkeit des ganzen Ausdrucks relevant sind.

Nachdem alle erfüllten Klauseln entfernt wurden, löscht der Algorithmus sämtliche Vorkommnisse von $\neg l$. Auch dies ist unbedenklich, $\neg l$ wurde bereits mit 0 belegt (da l mit 1 belegt wurde). Die betroffenen Klauseln können somit nicht durch $\neg l$ erfüllt werden. Um diese Klauseln zu erfüllen, muss also eine der verbliebenen freien Variablen entsprechend belegt werden.

Man bezeichnet diesen Vorgang des Löschens von negierten Literalen und erfüllten Klauseln als Deduktion, Implikation oder *Unit Propagation* (siehe Algorithmus 1, Zeile 2).

Ist S nach der Unit Propagation leer, so bricht der Algorithmus ab. Alle Klauseln wurden erfüllt. Damit ist p erfüllbar.

Enthält S dagegen eine leere Klausel, aus der alle Literale entfernt wurden, so ist p mit der aktuellen Variablenbelegung nicht erfüllbar. Eine leere Klausel repräsentiert eine Klausel, deren Literale alle mit 0 belegt wurden. Man bezeichnet diese Situation als *Konflikt*.

Bis hierhin setzt das Vorgehen des Algorithmus die Existenz einer Unit Clause voraus. Doch was geschieht, wenn es keine Unit Clause gibt? In diesem Fall sucht sich der DPLL-Algorithmus eine Variable aus, die er mit **WAHR** belegt und ruft sich selbst rekursiv mit dieser Belegung auf. Führt der Aufruf zu einem Konflikt, so wird die Variable neu belegt, diesmal mit **FALSCH**. Sollte auch dies zu einem Konflikt führen, so gibt der Algorithmus **FALSCH** zurück. Geschieht das auf der obersten Ebene, auf der noch keine anderen Variablen belegt wurden, so ist p generell unerfüllbar. Dieser Vorgang heißt *Branching* (siehe Algorithmus 1, Zeile 15). Die ausgewählte Variable wird *Entscheidungsvariable* oder englisch *Decision Variable* genannt[1]. Durch Branching schränkt sich der Algorithmus auf einen Teilbereich des möglichen Lösungsraums ein.

Seit der Entwicklung des DPLL-Algorithmus im Jahr 1962 (siehe [15]) wurde viel im Bereich der SAT-Solver erreicht. Wie bereits erwähnt, basieren jedoch die meisten aktuellen SAT-Solver nach wie vor auf diesem Algorithmus. Er wurde allerdings weiterentwickelt und durch zusätzliche Mechanismen ergänzt. Neben Unit Propagation und Branching spielen noch zwei andere Techniken eine wesentliche Rolle: Backtracking und Lernen. Diese vier Mechanismen und ihre Weiterentwicklungen werden im Folgenden kurz besprochen.

Die Unit Propagation ist wohl der wichtigste Mechanismus eines SAT-Solvers [48]. An dem Prinzip, wie es oben und in Algorithmus 1 dargestellt wird, hat sich kaum etwas geändert, jedoch wurde intensiv nach effektiven Ansätzen gesucht, es zu imple-

[1] Ein *Entscheidungsliteral* (*Decision Literal*) ist das Literal einer solchen Variable.

mentieren. Die Hauptschwierigkeit besteht darin, in einer großen Menge an Klauseln die Unit Clauses überhaupt zu finden. Der Grundansatz einiger erfolgreicher Implementierungen – beispielsweise die SAT-Solver SATO [46] und Chaff [35] – besteht darin, Zeiger auf verschiedene freie Literale einer Klausel zu richten. Wird eines der überwachten Literale mit 1 belegt, so muss nichts getan werden, die Klausel ist erfüllt. Wird eines der Literale jedoch mit 0 belegt, so wird der jeweilige Zeiger aktualisiert, indem er auf ein anderes freies Literal der Klausel gesetzt wird. Stellt sich dabei heraus, dass nur noch ein Literal frei ist und alle anderen mit 0 belegt sind, so liegt eine Unit Clause vor. Sind dagegen alle Literale mit 0 belegt, so wurde ein Konflikt gefunden. Dieser Ansatz wird beispielsweise von dem SAT-Solver SATO in Form von Head- und Tail-Listen verwendet [47] oder von dem SAT-Solver Chaff, dessen Entwickler hierfür den Begriff *Watch Literals* geprägt haben [35].

Neben effektiven Implementierungen für die Unit Propagation wurde auch nach effektiven Strategien zur Durchführung des Branching gesucht. Die Schwierigkeit an dieser Stelle besteht in der Auswahl der „richtigen“ Entscheidungsvariable. Je nachdem, in welcher Reihenfolge die Variablen ausgewählt werden, kann sich die Laufzeit des Algorithmus erheblich verlängern. Dies wird in Beispiel 2.2 demonstriert.

Beispiel 2.2 (DPLL-Algorithmus – Branching)
Gegeben sei ein boolescher Ausdruck $p := (\overline{x_1} + x_2) \cdot (x_1 + x_3) \cdot (x_1 + \overline{x_4})$
Am Anfang sind alle Variablen frei. Die Klauselliste S *sieht zu Beginn folgendermaßen aus:* $S = \{(\overline{x_1} + x_2), (x_1 + x_3), (x_1 + \overline{x_4})\}$
Im Folgenden sind zwei mögliche Fälle, wie sich der Algorithmus verhalten könnte, dargestellt:

Fall 1: *Der Algorithmus wählt zuerst die Variable* x_1 *aus und belegt sie mit dem Wert* 1. *Dadurch sind die zweite und die dritte Klausel bereits erfüllt, sie müssen nicht mehr betrachtet werden. Der Algorithmus ruft sich selbst auf mit* SATSolve($S \cup \{x_1\}$).
Bei der Unit Propagation (vergleiche Algorithmus 1, Zeile 2) wird zunächst die Klausel $\{x_1\}$ *als Unit Clause erkannt. Durch das Unit Literal* x_1 *werden die zweite und die dritte Klausel erfüllt, sie fallen weg.* $\overline{x_1}$ *wird aus der ersten Klausel gelöscht:*

$$S = \{(x_2)\}$$

Die letzte verbliebene Klausel ist ebenfalls eine Unit Clause. x_2 *wird mit* 1 *belegt und die letzte Klausel wird gelöscht. Dadurch wird* S *leer, was zum Ende des Durchlaufs führt.* p *ist erfüllbar.*

Fall 2: *Der Algorithmus wählt als erste Entscheidungsvariable* x_4 *aus und belegt* x_4 *mit* 0. *Der rekursive Aufruf erfolgt mit* SATSolve($S \cup \{\overline{x_4}\}$).
$\overline{x_4}$ *ist nun das Unit Literal und führt dazu, dass die dritte Klausel aus* S *entfernt wird:*

$$S = \{(\overline{x_1} + x_2), (x_1 + x_3)\}$$

Der Algorithmus wählt nun x_3 als zweite Entscheidungsvariable und ruft sich erneut selbst auf: SATSolve$S \cup \{x_3\}$.
Beim dritten Aufruf ist x_3 ein Unit Literal, die zweite Klausel wird gelöscht:

$$S = \{(\overline{x_1} + x_2)\}$$

Unabhängig davon, welches der beiden Literale der Algorithmus nun wählt, mit dem nächsten Aufruf wird der Durchlauf endgültig beendet. So kann der Algorithmus beispielsweise x_2 als Entscheidungsliteral wählen und es auf 1 setzen. Der letzte Aufruf erfolgt mit SATSolve*$(S \cup \{x_2\})$.*
x_2 ist nun ein Unit Literal, die letzte verbliebene Klausel ist erfüllt und wird aus S entfernt. Der Algorithmus beendet sich und gibt an, dass p erfüllbar ist.

Im ersten Fall wird der Algorithmus inklusive des initialen Aufrufs zweimal aufgerufen, wohingegen im zweiten Fall insgesamt vier Aufrufe gebraucht werden.

Wie aus Beispiel 2.2 ersichtlich ist, kommt der Auswahl der Entscheidungsvariable eine große Bedeutung bei. Zu diesem Zweck gibt es verschiedene Heuristiken. Eine einfache, gierige Heuristik ist „Dynamic Largest Combined Sum“, abgekürzt mit DLCS [32]. Eine Heuristik oder ein Algorithmus wird als *gierig* bezeichnet, wenn in jeder Iteration das aktuelle Optimum ausgewählt wird. Dabei wird außer Acht gelassen, ob diese lokalen Optima letzten Endes zu einem globalen Optimum führen oder nicht. Bei DLCS wird immer die Variable ausgewählt, deren Literale in den meisten unerfüllten Klauseln vorkommen.

Es existieren noch weit ausgefeiltere Heuristiken, wie zum Beispiel VSIDS [35]. Auf diese sowie auf weitere Beispiele dieser Art kann an dieser Stelle jedoch nicht Bezug genommen werden, da dies im Rahmen dieser Arbeit zu weit führen würde. Weiter gehende Informationen dazu finden sich jedoch in [48].

Ein anderes wichtiges Prinzip in SAT-Solvern ist *Backtracking*. Hier geht es darum, dass die Entscheidung rückgängig gemacht wird, wenn die Belegung einer Entscheidungsvariable zu einem Konflikt führte, und die Entscheidungsvariable mit dem anderen Wert belegt wird.

In diesem Zusammenhang ist der Begriff *Entscheidungsebene* oder englisch *Decision Level* von Bedeutung. Damit bezeichnet man die „Ebene“, auf der eine Entscheidung getroffen wird. Am Anfang, wenn alle Variablen unbelegt sind, befindet sich der Algorithmus auf Entscheidungsebene 0. Wird nun eine Variable ausgewählt, so befindet man sich auf Entscheidungsebene 1. Alle Variablen, die nun durch Unit Propagation ebenfalls belegt werden, befinden sich ebenfalls auf Entscheidungsebene 1. Die Entscheidungsebenen werden aufeinanderfolgend durchnummeriert, bis man bei der letzten Entscheidungsvariable angelangt ist.

Im Folgenden bedeute die Schreibweise $x = X@k$, dass die Variable x den Wert $X \in \{0, 1\}$ auf Entscheidungsebene k annimmt.

An Beispiel 2.2 lassen sich die Entscheidungsebenen verdeutlichen:

- Im ersten Fall des Beispiels 2.2 gehört x_1 zur Entscheidungsebene 1 und wird mit 1 belegt: $x_1 = 1@1$.
  Durch die Unit Propagation ergibt sich, dass x_2 mit 1 belegt wird. Damit befindet sich x_2 ebenfalls auf Entscheidungsebene 1: $x_2 = 1@1$.

- Im zweiten Fall von Beispiel 2.2 gibt es mehrere Entscheidungsebenen: x_4 ist auf Entscheidungsebene 1 ($x_4 = 0@1$), x_3 auf Entscheidungsebene 2 ($x_3 = 1@2$) und x_2 zusammen mit x_1 auf Entscheidungsebene 3. Für x_2 und x_1 sind mehrere Belegungen möglich, zum Beispiel $x_2 = 1@3$ und $x_1 = 0@3$.

Backtracking besteht nun darin, im Falle eines Konflikts zu einer früheren Entscheidungsebene zurückzukehren.

Im DPLL-Algorithmus, wie er auf Seite 10 dargestellt ist, ist bereits eine einfache Form des Backtrackings enthalten: Beim Branching (siehe Zeile 15) ruft der Algorithmus sich selbst mit einer weiteren Unit Clause auf, die von der aktuellen Entscheidungsvariable bestimmt wird. Führt dieser Aufruf zu einem Konflikt, so kehrt der Algorithmus zu dieser Entscheidungsebene zurück und belegt die Variable mit einem anderen Wert. Führt auch das zu einem Konflikt, so geht der Algorithmus noch eine Entscheidungsebene höher. Kommt der Algorithmus auf der obersten Ebene (Entscheidungsebene 0) an, so ist die Formel unerfüllbar. Dieses Durchlaufen der einzelnen aufeinanderfolgenden Entscheidungsebenen bezeichnet man auch als chronologisches Backtracking.

Chronologisches Backtracking hat jedoch einen Nachteil: Es ist denkbar, dass ein Konflikt nicht auf dem letzten, sondern auf einer früheren Entscheidungsebene verursacht wurde. In diesem Fall bietet es sich an, zu dieser früheren Entscheidungsebene zurückzukehren. Man bezeichnet diesen Vorgang als nicht-chronologisches Backtracking.

Die Unterschiede von chronologischem und nicht-chronologischem Backtracking sollen an Beispiel 2.3 verdeutlicht werden.

Beispiel 2.3 (Backtracking)
Gegeben sei der boolesche Ausdruck

$$p := (\overline{x_1} + x_2) \cdot (\overline{x_1} + x_3 + x_4) \cdot (\overline{x_2} + \overline{x_3}) \cdot (x_1 + x_5) \cdot (\overline{x_6} + \overline{x_5} + \overline{x_7}) \cdot (x_1 + x_6 + x_8)$$

Folgende Entscheidungen seien schon getroffen worden: $x_4 = 0@1$, $x_7 = 1@2$ *und* $x_8 = 0@3$. *Die Klausel-Liste* S *sieht nun folgendermaßen aus:*

$$S = \{(\overline{x_1} + x_2), (\overline{x_1} + x_3), (\overline{x_2} + \overline{x_3}), (x_1 + x_5), (\overline{x_6} + \overline{x_5}), (x_1 + x_6)\}$$

Nun wird $x_1 = 0@4$ *gewählt. Die ersten zwei Klauseln werden dadurch erfüllt, die dritte Klausel ist davon gar nicht betroffen, jedoch führen die letzten drei Klauseln zu einem Konflikt:*

- $x_1 = 0@4 : (x_1 + x_5) \Rightarrow x_5 = 1@4$
- $x_5 = 1@4 : (\overline{x_6} + \overline{x_5}) \Rightarrow x_6 = 0@4$

- $x_1 = 0@4$ *und* $x_6 = 0@4 : (x_1 + x_6) = 0 \Rightarrow$ *Konflikt!*

Beim chronologischen Backtracking wird nun die Entscheidung $x_1 = 0@4$ *rückgängig gemacht und stattdessen* $x_1 = 1@4$ *gesetzt. Allerdings führt das zu einem Konflikt mit den ersten drei Klauseln:*

- $x_1 = 1@4 : (\overline{x_1} + x_2) \Rightarrow x_2 = 1@4$
- $x_1 = 1@4 : (\overline{x_1} + x_3) \Rightarrow x_3 = 1@4$
- $x_2 = 1@4$ *und* $x_3 = 1@4 : (\overline{x_2} + \overline{x_3}) = 0 \Rightarrow$ *Konflikt!*

Da der Konflikt auf Entscheidungsebene 4 *auf diese Weise nicht gelöst werden konnte, würde man beim chronologischen Backtracking nun zur Entscheidungsebene* 3 *zurückgehen und* $x_8 = 1@3$ *setzen.*
Beim nicht-chronologischen Backtracking würde dagegen sofort zur Entscheidungsebene 3 *zurückgesprungen werden, wenn der Konflikt durch* $x_1 = 0@4$ *auftritt.*

Es ist offensichtlich, dass nicht-chronologisches Backtracking ausgefeilte Vorgehensweisen erfordert. Die gründliche Analyse der Ursachen eines Konflikts ist nötig, um auf die richtige Entscheidungsebene zurückzuspringen. Eine genauere Betrachtung der zu diesem Zweck entwickelten Vorgehensweisen würde jedoch den Rahmen dieser Arbeit sprengen, weshalb sie hier nicht näher behandelt werden sollen. Nicht-chronologisches Backtracking ist beispielsweise innerhalb der SAT-Solver BerkMin [24] und GRASP [33] implementiert worden.

Die heutigen SAT-Solver verfügen in der Regel über einen Lernmechanismus, der ebenfalls eine wichtige Rolle beim effizienten Lösen von SAT spielt. Ein SAT-Solver ist in der Lage zu lernen, welche Variablenbelegungen in der Vergangenheit zu Konflikten geführt haben, und kann sie dementsprechend in Zukunft meiden. Dies geschieht, indem der Solver im Falle eines Konflikts die Belegungen der früheren Entscheidungsvariablen, die zum Konflikt geführt haben, negiert und die dadurch entstehende Klausel in seine Klausel-Liste einfügt. Auf diese Weise wird verhindert, dass diese Variablenbelegungen noch einmal auftreten. Die gelernten Klauseln nennt man *Konfliktklauseln.*

Beispiel 2.4 (Konfliktklauseln)
Gegeben sei der boolesche Ausdruck aus Beispiel 2.3 mit

$$p := (\overline{x_1} + x_2) \cdot (\overline{x_1} + x_3 + x_4) \cdot (\overline{x_2} + \overline{x_3}) \cdot (x_1 + x_5) \cdot (\overline{x_6} + \overline{x_5} + \overline{x_7}) \cdot (x_1 + x_6 + x_8)$$

Erneut seien die Entscheidungen $x_4 = 0@1$, $x_7 = 1@2$ *und* $x_8 = 0@3$ *bereits getroffen worden. Für* $x_1 = 0@4$ *tritt nun der bereits bekannte Konflikt auf. Dieser wurde ausgelöst von der Teil-Belegung* $x_4 = 0$, $x_7 = 1$ *und* $x_8 = 0$. *Diese Belegung wird nun negiert:*

$$\overline{\overline{x_4} \cdot x_7 \cdot \overline{x_8}} = (x_4 + \overline{x_7} + x_8)$$

Die neue Klausel wird in die Klausel-Liste des Solvers eingefügt, der Solver geht – im Falle von nicht-chronologischem Backtracking – auf Entscheidungsebene 3 *zurück.*

Die in den Konfliktklauseln enthaltenen Informationen sind redundant, da sie sich aus den übrigen Klauseln ableiten. Jedoch ist es wesentlich schneller, auf eine Konfliktklausel zuzugreifen, als die darin enthaltene Information erneut abzuleiten. Allerdings nimmt die benötigte Rechenzeit auch durch die zusätzlichen Klauseln, die beachtet werden müssen, zu. Ebenso steigt der Bedarf an Speicherplatz durch das Anwachsen der Klausel-Liste. Um diese Nachteile zu verringern, verfügen viele Solver über Heuristiken, die einige der weniger nützlichen Konfliktklauseln wieder aus der Klausel-Liste löschen. Auch diese Heuristiken sollen hier nicht näher erläutert werden. Der SAT-Solver BerkMin verfügt beispielsweise über solche Techniken [24].

Dies stellt nur einen knappen Überblick über die wichtigsten Eigenschaften und Techniken von SAT-Solvern dar. Neben dem hier vorgestellten DPLL-Algorithmus gibt es auch noch andere Ansätze, allerdings sind die darauf basierenden SAT-Solver in der Regel nicht vollständig. Insbesondere können sie häufig die Unerfüllbarkeit eines booleschen Ausdrucks nicht nachweisen, was jedoch für das Bounded Model Checking auf stochastischen Systemen unbedingt erforderlich ist. Wer sich enger mit SAT-Solvern auseinandersetzen will, dem bietet [48] einen genaueren Überblick mit zahlreichen Leseanregungen.

2.1.3 Binary Decision Diagrams – BDD

In Kapitel 2.1.1 wurden boolesche Funktionen vorgestellt und kurz die möglichen Darstellungsformen angesprochen. Eine solche Darstellungsform sind *Binary Decision Diagrams* (*BDDs*). Ein *BDD* ist ein azyklischer, gerichteter Graph, dessen innere Knoten über genau zwei ausgehende Kanten verfügen. BDDs gehören zu den Entscheidungsdiagrammen, mit der Besonderheit, dass sie nur zwischen zwei Werten unterscheiden, zwischen 0 und 1 beziehungsweise **WAHR** und **FALSCH**. Sie werden verwendet, um boolesche Funktionen darzustellen [8]. Für das Bounded Model Checking auf stochastischen Systemen werden sie benötigt, um aus einem stochastischen System eine KNF zu generieren.

Eine Funktion f, die durch einen BDD repräsentiert wird, wird an einem inneren Knoten d in zwei Teile geteilt, abhängig von dem Wert der in d repräsentierten Variable x_i. Der eine Teil wird als f_{low}, der andere als f_{high} bezeichnet. Entsprechend heißen die beiden Kanten, die von d zu den jeweiligen Teilbäumen führen, auch *Low*- beziehungsweise *High*-Kanten.

Abbildung 2.1 zeigt ein Beispiel für einen einfachen BDD. Die durchgezogenen Linien sind *High*-Kanten, die durchbrochenen Linien sind *Low*-Kanten. Der vertikale Verlauf einer Kante gibt auch ihre Richtung im Graphen an, eine Kante führt immer von einem oberen zu einem unteren Knoten.

Ein BDD verfügt über zwei verschiedene Arten von Blättern: 0- und 1-Blätter. Ein Pfad von der Wurzel zu einem Blatt entspricht einer Variablenbelegung der Funktion f, der Wert des Blattes gibt über das Ergebnis der Funktion Auskunft.

Die Idee hinter der Funktions-Zerlegung besteht darin, dass in den daraus entstehenden Teilfunktionen die Variable des Knotens d nicht mehr vorkommt. Der Wert der Variable wird schon festgelegt, indem man sich für die *Low*- oder *High*-Kante

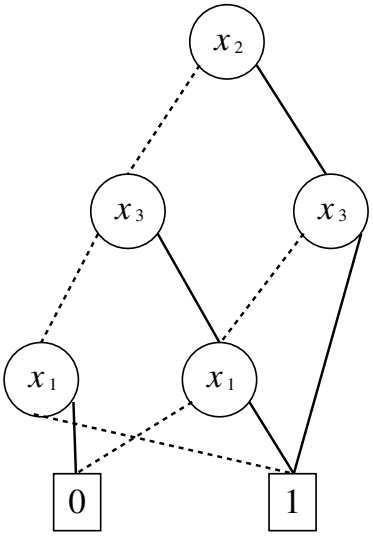

Abbildung 2.1: Ein einfacher BDD

entscheidet.

Es gibt mehrere Möglichkeiten für die Zerlegung. Bei den hier gezeigten BDDs wird ausschließlich die *Shannon-Zerlegung* [41] verwendet:

$$f(x) = \overline{x_i} \cdot f_{x_i=0} + x_i \cdot f_{x_i=1} \tag{2.2}$$

$f_{x_i=0}$ beziehungsweise $f_{x_i=1}$ bezeichnen die Funktionen, die entstehen, wenn in f die Variable x_i konstant auf 0 beziehungsweise 1 gesetzt wird. Man nennt diese Funktionen auch *negativer* oder *positiver Kofaktor* von f nach x_i beziehungsweise $\overline{x_i}$. Statt $f_{x_i=0}$ und $f_{x_i=1}$ schreibt man auch $f_{\overline{x_i}}$ und f_{x_i}

Der Vorteil von BDDs besteht in ihrer anschaulichen Darstellung von booleschen Funktionen. Sie sind darüber hinaus eine kanonische Funktionsdarstellung, wenn sie bestimmte Eigenschaften erfüllen:

Definition 2.9 (Eigenschaften von BDDs)

- *Ein BDD heißt* frei, *wenn auf jedem Pfad von einer Wurzel zu einem Blatt jede Variable höchstens einmal vorkommt.*
- *Ein BDD heißt* geordnet, *wenn auf allen Pfaden des BDDs eine bestimmte Variablenordnung eingehalten wird.*
- *Ein BDD ist* vollständig reduziert, *wenn sich keine der folgenden Reduktionsregeln mehr darauf anwenden lässt:*
  - *Zeigen sowohl die High- als auch die Low-Kante eines Knotens d auf denselben Knoten d', so wird d entfernt. Alle eingehenden Kanten von d gehen nun zu d' (siehe Abbildung 2.2(a)).*
  - *Sind zwei Knoten d' und d'' mit der gleichen Variable x_i markiert und führen ihre Low- und High-Kanten zu den gleichen Teilbäumen $f_{\overline{x_i}}$ und f_{x_i},*

so werden die beiden Knoten zu einem einzigen Knoten d zusammengefasst. Alle eingehenden Kanten von d′ und d″ zeigen nun auf d (siehe Abbildung 2.2(b)).

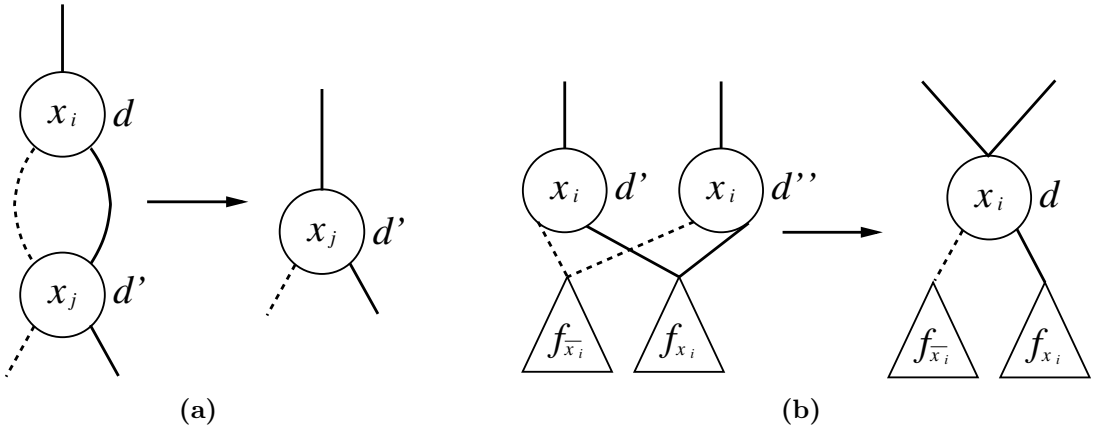

Abbildung 2.2: Reduktionsregeln für BDDs

Ein geordneter, freier und vollständig reduzierter BDD ist für eine feste Variablenordnung eindeutig und damit eine kanonische Darstellung von booleschen Funktionen. Zwei Funktionen, die den gleichen geordneten, freien und reduzierten BDD für die gleiche Variablenordnung haben, mit Ausnahme von Isomorphie, sind äquivalent (*Satz von Bryant* [8]). Insbesondere hat ein solcher BDD genau ein 0- und ein 1-Blatt.

Im Folgenden gehen wir stets davon aus, dass es sich bei einem BDD um einen geordneten, freien und reduzierten BDD handelt.

Liegt der BDD für eine boolesche Funktion f vor, so kann man die ON-Menge von f leicht daraus ablesen: Es reicht aus, von einem 1-Blatt aus rückwärts alle Pfade bis zum Startknoten zu durchlaufen. Ein SAT-Problem, das als BDD vorliegt, ist trivial lösbar. Da SAT jedoch ein NP-vollständiges Problem ist (siehe [13]), ist es offensichtlich, dass BDDs nicht nur Vorteile haben können: Der Bau eines BDDs nimmt in der Regel viel Zeit und Speicherplatz in Anspruch. BDDs können exponentiell mit der Anzahl an Funktionsvariablen wachsen. Dabei hat die Variablenordnung einen sehr starken Einfluss auf die Größe eines BDDs.

Beispiel 2.5 (Einfluss der Variablenordnung auf BDDs)
Die Funktion $f : \mathbb{B}^3 \times \mathbb{B}^3 \to \mathbb{B}$ vergleicht zwei dreistellige Binärzahlen $a = a_1a_2a_3$ und $b = b_1b_2b_3$. Wenn $a \geq b$, liefert f den Wert 1, ansonsten 0:

$$f(a,b) = \begin{cases} 1, & \text{wenn } a \geq b \\ 0, & \text{sonst} \end{cases}$$

Je nachdem, was für eine Variablenordnung gewählt wird, fällt der BDD für f unterschiedlich groß aus. Die Variablenordnung $a_1 \prec b_1 \prec a_2 \prec b_2 \prec a_3 \prec b_3$ führt zu einem

BDD mit acht inneren Knoten (siehe Abbildung 2.3).
Der BDD mit der Variablenordnung $a_1 \prec a_2 \prec a_3 \prec b_1 \prec b_2 \prec b_3$ *ist wesentlich größer und verfügt über* 18 *innere Knoten (siehe Abbildung 2.4).*

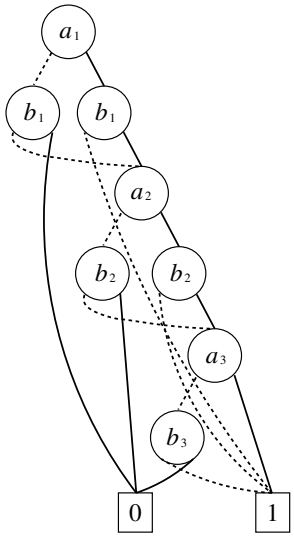

Abbildung 2.3: BDD für $a \geq b$ mit Variablenordnung $a_1 \prec b_1 \prec a_2 \prec b_2 \prec a_3 \prec b_3$

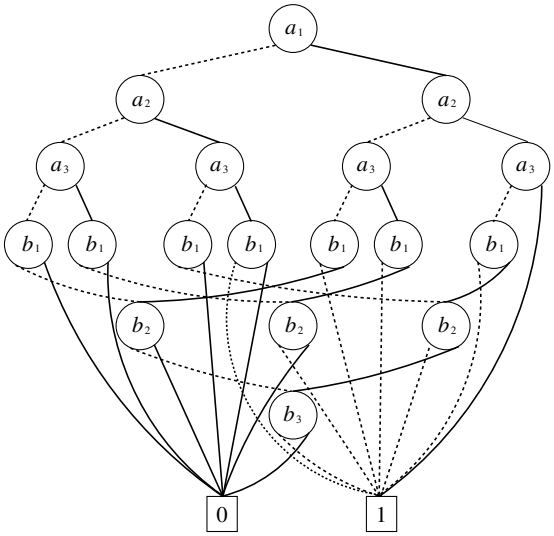

Abbildung 2.4: BDD für $a \geq b$ mit Variablenordnung $a_1 \prec a_2 \prec a_3 \prec b_1 \prec b_2 \prec b_3$

Es ist bereits ein NP-vollständiges Problem, eine existierende Variablenordnung zu verbessern [7]. Entsprechendes gilt für die Minimierung eines BDDs. Es gibt jedoch sowohl exakte (siehe [20]) als auch heuristische Verfahren, um diese Probleme zu lösen. Zu den heuristischen Verfahren gehören beispielsweise Sifting (siehe [40]) und Windows Optimization (siehe dazu [22] und [21]). Allerdings zeichnen sich alle diese Verfahren durch einen hohen Speicherplatzbedarf aus.

Eine allgemeinere Form von BDDs sind *MTBDDs* beziehungsweise *Multi-Terminal BDDs*. Im Wesentlichen haben sie die gleichen Eigenschaften wie BDDs. Die Unterschiede liegen in der Anzahl an Blättern und im Wertebereich, aus dem die Werte der Blätter stammen. MTBDDs verfügen über eine beliebige Anzahl von Blättern[2]. Die Werte der Blätter sind nicht auf 0 und 1 beschränkt, stattdessen ist jede endliche Menge als Wertebereich denkbar [23]. Dies kann zum Beispiel eine Teilmenge der Zahlenbereiche $\mathbb{N}$, $\mathbb{Z}$ oder $\mathbb{R}$ sein. Insbesondere eignen sich MTBDDs zum Repräsentieren und Bearbeiten von Matrixen [23]. Abbildung 2.5 zeigt einen einfachen MTBDD.

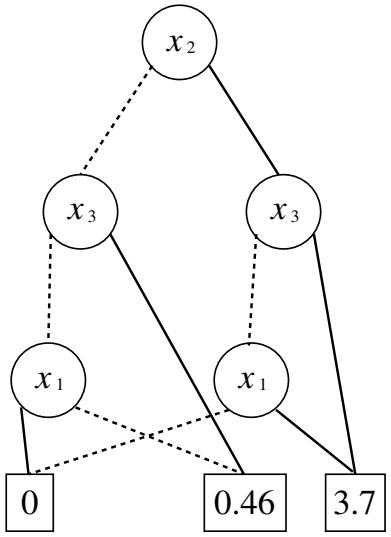

Abbildung 2.5: Ein einfacher MTBDD

MTBBDs können auch im Zusammenhang mit stochastischen Systemen eingesetzt werden. Hier werden sie verwendet, um die Zustandsübergänge eines solchen Systems darzustellen. In diesem Fall beschränkt sich der Wertebereich der Blätter auf das Intervall $[0, 1]$. Darauf wird in Kapitel 3.1.1 genauer eingegangen werden.

2.2 Bounded Model Checking – BMC

Model Checking ist ein wichtiger Bestandteil der Verifikation. Es geht darum, festzustellen, ob ein gegebenes System auch wirklich den gewünschten Anforderungen entspricht. Dies ist von erheblichem Interesse für Industrie und Forschung und sollte nach Möglichkeit mit einem automatisierten Verfahren realisiert werden.

Häufig wird ein System als deterministischer Automat in Form eines Zustandsgraphen dargestellt. Wie in [12] beschrieben, versuchte man in den 80er-Jahren des vorigen Jahrhunderts, die Systeme dementsprechend mit Algorithmen zur Graph-Traversierung zu untersuchen. Da die Zustände eines Systems jedoch exponentiell

[2]Es ist offensichtlich, dass ein nicht-trivialer MTBDD über mindestens zwei Blätter verfügt. Hätte ein MTBDD nur ein einzelnes Blatt, so würde er durch die Reduktionsregeln (siehe Abbildung 2.2) auf dieses Blatt reduziert werden.

in der Anzahl seiner Bestandteile anwachsen, ist dieser Ansatz mit erheblichen Einschränkungen bezüglich der Größe des jeweiligen Systems verbunden.

Eine weitere Methode ist Symbolic Model Checking. In diesem Fall wird der Zustandsgraph nicht explizit erstellt, stattdessen wird das System implizit durch BDDs dargestellt [10]. Doch auch dieses Verfahren beinhaltet Nachteile: Obwohl größere Systeme als bei den graphenbasierten Methoden betrachtet werden können, sind auch die BDD-basierten Ansätze in der Größe des zu betrachtenden Systems beschränkt. Wie in Kapitel 2.1.3 erwähnt, können BDDs ebenfalls exponentiell in der Anzahl ihrer Variablen anwachsen, wobei auch die Variablenordnung eine erhebliche Rolle spielt[3]. Da es ein NP-vollständiges Problem ist, eine Variablenordnung zu verbessern (siehe [7]), ist meistens das Eingreifen eines Anwenders vonnöten, der das System kennt und dementsprechend Richtlinen für die Variablenordnung vorgeben kann. Dies widerspricht jedoch dem Wunsch nach einem automatisierten Verifikationsverfahren. Es kommt hinzu, dass der BDD einiger wichtiger Standard-Komponenten, wie zum Beispiel der eines Multiplizierers, unabhängig von der Variablenordnung exponentiell groß ist [9].

Eine Lösung für diese Probleme bietet Bounded Model Checking – kurz BMC [6]. Hier kommen weder Zustandsgraphen noch BDDs zum Einsatz. Stattdessen basiert BMC auf SAT. Das System wird mit seinen Zustandsübergängen als boolesche Formel in KNF dargestellt. Diese Formel wird an einen SAT-Solver übergeben, womit dessen ausgefeilte Techniken für das Model Checking genutzt werden können.

Die Anzahl an betrachteten Zustandsübergängen ist dabei auf eine bestimmte Länge k begrenzt – englisch „bounded“. Lässt sich innerhalb dieser k Schritte zeigen, dass die zu untersuchende Eigenschaft des Systems *nicht* gilt beziehungsweise eine Bedingung *verletzt* wird, so hat man ein Gegenbeispiel gefunden.

BMC liefert also Gegenbeispiele der Länge k für die zu untersuchende Eigenschaft eines Systems. Auf diese Weise kann allerdings nicht gezeigt werden, dass ein System eine bestimmte Bedingung erfüllt. Wird kein Gegenbeispiel der Länge k gefunden, ist es durchaus möglich, dass für ein größeres k ein Gegenbeispiel gefunden werden könnte. Damit ist BMC nicht vollständig. Man versucht dieses Problem jedoch mit Hilfe der k-Induktion (vergleiche [42]) oder der Craig'schen Interpolation (siehe [34]) zu lösen. Beide Verfahren bauen auf BMC auf.

Für die Zwecke dieser Arbeit ist BMC jedoch vollkommen ausreichend. In der Regel handelt es sich bei den untersuchten Eigenschaften um Sicherheitsbedingungen, von denen bereits bekannt ist, dass sie verletzt werden könnten. BMC wird verwendet, um ein konkretes Gegenbeispiel für eine solche Sicherheitsbedingung zu erzeugen. Man kann dadurch Aufschluss darüber erhalten, was die Ursachen für die Verletzung der Sicherheitsbedingung sind. Durch genaue Untersuchung des Gegenbeispiels bieten sich unter Umständen Lösungsansätze.

Eine BMC-Formel setzt sich nun aus drei Teilen zusammen: Einem Prädikat für den Startzustand, einer KNF für die Transitionsrelation und einer negierten Bedingung

[3]Siehe dazu auch Beispiel 2.5 in Kapitel 2.1.3 auf Seite 18.

beziehungsweise Invariante (siehe [1]):

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k) \tag{2.3}$$

$I(s_0)$ in Formel 2.3 ist das Prädikat für den Startzustand, $T(s_i, s_{i+1})$ für $i = 0, \ldots, k-1$ ist die KNF für die Transitionsrelation und $P(s_k)$ ist die zu überprüfende Invariante im k-ten Zustand. Die betroffenen Variablen der Transitionsrelation und der Bedingung müssen einen Index-Shift vorweisen, der dem jeweiligen Zustandsübergang entspricht. Die Variablen im Startzustand können dagegen für das Prädikat $I(s_0)$ in der Regel direkt der System-Beschreibung entnommen werden.

Wurde für k kein Gegenbeispiel gefunden, so wird nun ein weiterer Zustandsübergang betrachtet: $k \rightarrow k+1$. Die Formel 2.3 wird um den Zustandsübergang $T(s_k, s_{k+1})$ ergänzt und die Bedingung $P(s_k)$ wird durch die neu berechnete Bedingung $P(s_{k+1})$ ersetzt.

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge T(s_k, s_{k+1}) \wedge \neg P(s_{k+1}) \tag{2.4}$$

Zum Schluss dieses Unterkapitels soll BMC noch einmal anhand eines konkreten Beispiels verdeutlicht werden. Es wurde an ein Beispiel aus [12] angelehnt.

Beispiel 2.6 (BMC für einen 2-Bit-Zähler)
Gegeben sei ein 2-Bit-Zähler, a und b seien die Variablen für das nieder- beziehungsweise höchstwertigste Bit. a' und b' bezeichnen die Werte, die a und b nach einem Berechnungsschritt annehmen. Die Transitionsrelation sei gegeben durch:

$$(a' \leftrightarrow \neg a) \wedge (b' \leftrightarrow a \oplus b)$$

*$(a \oplus b)$ entspricht dabei $(a \wedge \neg b) \vee (\neg a \wedge b)$ (**XOR**) und $(a \leftrightarrow b)$ entspricht $(a \vee \neg b) \wedge (\neg a \vee b)$ (**XNOR**).*

a und b werden beide mit 0 initialisiert. Es soll nun überprüft werden, ob (a, b) den Wert $(1, 1)$ nie annehmen kann. Daraus ergibt sich für die BMC-Formel:

$$\neg P(s_i) = \neg(\neg(a_i \wedge b_i)) = (a_i \wedge b_i)$$

$k = 0$ und $k = 1$ seien bereits berechnet worden[4]. Nun wird $k = 2$ berechnet:

$$\begin{array}{rcl} I(s_0) & : & (\quad \neg a_0 \wedge \neg b_0 \quad)\wedge \\ T(s_0, s_1) & : & (\quad (a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0)) \quad)\wedge \\ T(s_1, s_2) & : & (\quad (a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1)) \quad)\wedge \\ P(s_2) & : & (\quad a_2 \wedge b_2 \quad) \end{array}$$

Auch für $k = 2$ ist die Formel nicht erfüllbar.

Wird nun $k = 3$ betrachtet, ändert sich Folgendes an der Formel:

- *$T(s_2, s_3) = (a_3 \leftrightarrow \neg a_2) \wedge (b_3 \leftrightarrow (a_1 \oplus b_2))$ kommt hinzu.*
- *$P(s_2)$ wird durch $P(s_3) = a_3 \wedge b_3$ ersetzt.*

Die daraus resultierende Formel ist erfüllbar. (a, b) kann also nach $k = 3$ Schritten den Wert $(1, 1)$ annehmen.

[4]Für $k = 0$ und für $k = 1$ sind die resultierenden BMC-Formeln unerfüllbar, wie sich leicht nachrechnen lässt.

2.3 Stochastische Systeme

Ein stochastisches System lässt sich als endlicher Automat betrachten, dessen Zustandsübergänge mit Wahrscheinlichkeiten gewichtet sind.

Wir gehen hier von einem diskreten Zeitmodell aus, der Übergang von einem Zustand in den nächsten entspricht also der Zustandsänderung eines Systems in einem Zeitschritt. Einen Zeitschritt bezeichnet man auch als *Zeitrahmen* oder englisch *Timeframe*. Das System befindet sich zum Zeitpunkt t im Zustand s und wechselt zur Zeit $t+1$ in den Zustand s'.

Bevor jedoch in Form von Markov-Ketten eine formale Definition für stochastische Systeme vorgestellt wird, ist es nötig, sich mit einigen stochastischen Grundlagen vertraut zu machen. Im Anschluss daran werden stochastische Systeme genauer vorgestellt und schließlich die Sprache PCTL behandelt, mit der sich die Eigenschaften von stochastischen Systemen definieren lassen.

2.3.1 Stochastische Grundlagen

In diesem Unterkapitel werden einige stochastische Grundlagen vorgestellt, die für den Umgang mit stochastischen Systemen notwendig sind. Die folgenden Definitionen und Schlussfolgerungen sind an [17] angelehnt.

Zunächst ist es notwendig, einige Begriffe festzulegen:

- Ω ist eine nicht-leere Menge und wird als *Grundraum* bezeichnet.
- $\mathfrak{A} \subseteq \mathfrak{P}(\Omega)$ ist ein System von Teilmengen von Ω. $A \in \mathfrak{A}$ heißt *Ereignis*.
  $A^C := \Omega \setminus A$ ist das *Komplement-Ereignis* von A.

Für die Verwendung von stochastischen Systemen wird ein Wahrscheinlichkeitsraum mit einer Wahrscheinlichkeitsverteilung benötigt. Um diese zu erhalten, braucht es zuerst eine σ-Algebra:

Definition 2.10 (σ-Algebra)
$\mathfrak{A} \subseteq \mathfrak{P}(\Omega)$ *heißt* σ-Algebra, *wenn Folgendes gilt:*

- $\Omega \in \mathfrak{A}$
- $A \in \mathfrak{A} \Rightarrow \Omega \setminus A \in \mathfrak{A}$
- $A_1, A_2, \ldots \in \mathfrak{A} \Rightarrow \bigcup_{i \geq 1} A_i \in \mathfrak{A}$

Aus Definition 2.10 lassen sich einige Schlussfolgerungen ziehen:

- $\emptyset \in \mathfrak{A}$
- $A_1, A_2, \ldots \in \mathfrak{A} \Rightarrow \bigcap_{i \geq} A_i \in \mathfrak{A}$
- $A_1, \ldots, A_n \in \mathfrak{A} \Rightarrow A_1 \cup \ldots \cup A_n \in \mathfrak{A}$ und $A_1 \cap \ldots \cap A_n \in \mathfrak{A}$

Nun ist es möglich, einen allgemeinen Wahrscheinlichkeitsraum zu definieren:

Definition 2.11 (Wahrscheinlichkeitsraum)
Ein (allgemeiner) Wahrscheinlichkeitsraum *ist ein Tripel* $(\Omega, \mathfrak{A}, P)$*, bestehend aus einem Grundraum* Ω*, einer* σ*-Algebra* $\mathfrak{A}$ *und einer Wahrscheinlichkeitsverteilung* $P : \mathfrak{A} \to [0, 1]$.
P muss folgende Eigenschaften erfüllen:

- $P(\Omega) = 1$ *(*$A := \Omega$ *heißt* sichereres Ereignis*)*

- *Es seien* $A_1, A_2, \ldots \in \mathfrak{A}$ *paarweise disjunkte Mengen, dann gilt:*

$$P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$$

  Man bezeichnet diese Eigenschaft als σ-Additivität.

$P(A)$ *ist die* Wahrscheinlichkeit *des Ereignisses* A *und wird auch* Wahrscheinlichkeitsmasse *von* A *genannt.*

Aus Definition 2.11 lassen sich einige Folgerungen für die Wahrscheinlichkeitsverteilung P ziehen:

- $P(\emptyset) = 0$ ($A := \emptyset$ bezeichnet man als *unmögliches Ereignis.*)

- $P(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} P(A_i)$ für endlich viele paarweise disjunkte Mengen $A_i \in \mathfrak{A}$. Man nennt diese Eigenschaft *endliche Additivität.*

- $P(A \setminus \Omega) = 1 - P(A)$

- $A, B \in \mathfrak{A}, A \subset B \Rightarrow P(A) \leq P(B)$ (Monotonie).

- $A, B \in \mathfrak{A} \Rightarrow P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
  $A \cup B$ bedeutet, dass das Ereignis A oder das Ereignis B eintritt. $A \cap B$ bedeutet, dass sowohl das Ereignis A als auch das Ereignis B eintritt.

- $A_1, A_2, \ldots \in \mathfrak{A}$, die A_i sind nicht paarweise disjunkt $\Rightarrow P(\bigcup_{i=1}^{\infty} A_i) \leq \sum_{i=1}^{\infty} P(A_i)$
  Man bezeichnet diese Eigenschaft als Sub-σ-Additivität.

Handelt es sich bei Ω um eine höchstens abzählbare Menge und entspricht $\mathfrak{A}$ der gesamten Potenzmenge $\mathfrak{P}(\Omega)$, so liegt ein diskreter Wahrscheinlichkeitsraum vor.

2.3.2 Markov-Ketten mit diskreter Zeit – DTMCs

Eine Möglichkeit, ein stochastisches System formal zu definieren, sind Markov-Ketten mit diskreter Zeit (DTMCs für englisch „**D**iscrete-**T**ime **M**arkov **C**hains“). Die nun folgenden Definitionen sind an die entsprechenden Definitionen in [11] angelehnt.

Definition 2.12 (Markov-Kette mit diskreter Zeit – DTMC)
Eine Markov-Kette *ist ein Tupel* $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ *mit*

- S*: Eine endliche Menge an Zuständen (englisch „States“);*
- $s_0 \in S$*: Der Startzustand der Markov-Kette.*
- $T : S \times S \to [0, 1]$*: Eine Transitionsmatrix. Es muss gelten:*

$$\sum_{s' \in S} T(s, s') = 1 \; \forall \, s \in S$$

  T gibt die Wahrscheinlichkeiten der Zustandsübergänge an.

- AP*: Eine endliche Menge an atomaren Propositionen.*
- $\mathcal{L} : S \to 2^{AP}$*: Eine Beschriftungsfunktion, die jedem Zustand $s \in S$ eine Menge $\mathcal{L}(s)$ von atomaren Propositionen aus AP zuweist, die in s gelten.*

Ein Zustand s in $\mathcal{D}$ wird absorbierend *genannt, wenn gilt $T(s, s) = 1$.*
Insbesondere muss gelten, dass jeder Zustand s einen Nachfolgezustand hat.

Die Definition von Markov-Ketten lässt sich an einem Beispiel verdeutlichen:

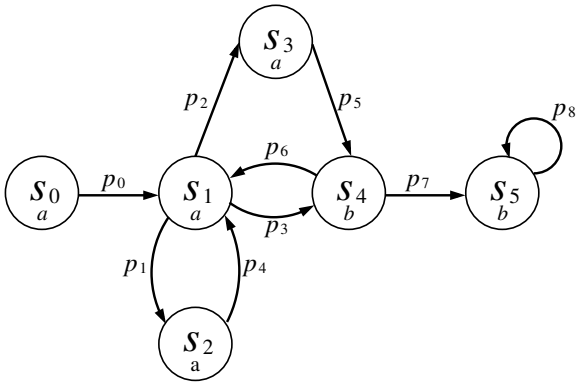

Abbildung 2.6: Beispiel für eine DTMC

Beispiel 2.7 (Beispiel für eine DTMC)
Abbildung 2.6 ist eine graphische Repräsentation einer DTMC $\mathcal{D}$.
$s_0, \ldots, s_5$ sind die Zustände, $p_0, \ldots, p_8$ die jeweiligen Werte aus der Matrix T (siehe unten). a und b sind atomare Propositionen, die in den jeweiligen Zuständen gelten.

s_0 ist der Startzustand von $\mathcal{D}$. Der Zustand s_5 hat als einzigen möglichen Nachfolger sich selbst, also handelt es sich dabei um einen absorbierenden Zustand. $s_1, \ldots, s_4$ bezeichnen die übrigen Zustände, $p_0, \ldots, p_8$ sind die Übergangswahrscheinlichkeiten. Die Übergangsmatrix ist gegeben mit:

$$T := \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0.5 & 0.4 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0.7 & 0 & 0 & 0 & 0.3 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Die einzelnen Zeilen der Matrix addieren sich zu 1 auf, was der Bedingung für die Transitionsmatrix in Definition 2.12 entspricht. Die Positionen der Matrix geben die einzelnen Zustandsübergänge an: Die erste Zeile zum Beispiel enthält die Wahrscheinlichkeiten aller Zustandsübergänge aus dem Startzustand s_0 in einen der anderen Zustände von $\mathcal{D}$, die zweite Zeile enthält entsprechend die Wahrscheinlicheiten der Übergänge aus s_1. Die erste Spalte gibt die Wahrscheinlichekeiten an, den Startzustand s_0 zu betreten, die zweite Spalte enthält die Wahrscheinlichkeiten für s_1 als Nachfolgezustand.
Der Eintrag in der zweiten Zeile in Spalte vier bezeichnet somit die Übergangswahrscheinlichkeit von s_1 nach s_3. In diesem Fall beträgt die Wahrscheinlichkeit 0.5.
Der Eintrag in der ersten Zeile und vierten Spalte lautet 0, es gibt also keinen direkten Übergang von Zustand s_0 nach s_3. Dies entspricht auch der Abbildung 2.6.
Insgesamt ergeben sich aus der Matrix T folgende Übergangswahrscheinlichkeiten:

$$p_0 := p_4 := p_5 := p_8 := 1, p_1 := 0.1, p_2 := 0.5, p_3 := 0.4, p_6 := 0.7, p_7 := 0.3$$

Aus der Forderung in Definition 2.12, dass jeder Zustand einer Markov-Kette $\mathcal{D}$ einen Nachfolgezustand besitzt, folgt, dass in $\mathcal{D}$ unendlich lange Pfade möglich sind.

Definition 2.13 (Pfad)
Es sei $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ eine Markov-Kette.
Dann ist ein Pfad *ω in $\mathcal{D}$ eine Folge $(s_1, s_2, \ldots)$ von Zuständen aus S, wobei gilt:*

$$\forall\, i \in \mathbb{N} : T(s_i, s_{i+1}) > 0$$

ω^i bezeichnet den i-ten Zustand von ω, dass heißt $\omega^i := s_i$.
Die Menge aller unendlich langen Pfade in $\mathcal{D}$ ist $Paths_{inf}$.

Für das Bounded Model Checking auf stochastischen Systemen spielen jedoch auch endliche Pfade eine große Rolle, weswegen es notwendig ist, auch diese zu definieren:

Definition 2.14 (Endlicher Pfad)
Ein endlicher Pfad *$\omega_n := (s_0, s_2, \ldots, s_n)$ der Länge n ist eine Folge von n+1 Zuständen aus S, sodass für alle $0 \leq i < n$ gilt: $T(s_i, s_{i+1}) \geq 0$.*
ω_n ist ein Präfix eines unendlichen Pfades ω.
$Paths_{fin}$ ist die Menge aller endlichen Pfade einer Markov-Kette $\mathcal{D}$.

Zum Beispiel ist in Abbildung 2.6 der endliche Pfad $\omega_n := (s_0, s_1, s_4, s_5)$ Präfix des unendlichen Pfades $\omega := (s_0, s_1, s_4, s_5, s_5, \ldots)$.

Für die Zwecke dieser Arbeit ist es notwendig, in der Lage zu sein, die Wahrscheinlichkeit eines endlichen Pfades ω_n berechnen zu können. Dazu ist es nötig, den Basis-Zylinder eines Pfades ω_n zu definieren.

Definition 2.15 (Basis-Zylinder)
Es sei $\omega_n \in Paths_{fin}$. Der Basis-Zylinder *von ω_n ist definiert als*

$$\Delta(\omega_n) := \{\omega \in Paths_{inf} \mid \omega_n \text{ ist Präfix von } \omega\}$$

Mit Hilfe der Basis-Zylinder kann nun der Wahrscheinlichkeitsraum einer Markov-Kette $\mathcal{D}$ erzeugt werden, über den die Wahrscheinlichkeitsverteilung für die Pfade definiert wird:

Definition 2.16 (Wahrscheinlichkeitsraum einer Markov-Kette)
Es sei $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ eine Markov-Kette und $s \in S$. Der Wahrscheinlichkeitsraum *von $\mathcal{D}$ ist dann folgendermaßen definiert:*

$$\Psi^s := (Paths_{inf}(s), \Delta^s, prob_s)$$

mit

- *$Paths_{inf}(s)$ ist die Menge aller unendlichen Pfade, die im Zustand s beginnen.*
- *Δ^s ist eine σ-Algebra. Sie wird von der leeren Menge und von den Basis-Zylindern über S, die in $Paths_{inf}(s)$ enthalten sind, generiert.*
- *$prob_s$ ist das induzierte Wahrscheinlichkeitsmaß.*
  $prob_s$ muss Folgendes erfüllen:

$$prob_s(\Delta(s, s_1, s_2, \ldots, s_n)) = \prod_{i=0}^{n-1} T(s_i, s_{i+1})$$

Das Wahrscheinlichkeitsmaß $prob(\omega_n)$ eines endlichen Pfades ω_n bezeichnen wir auch als *Wahrscheinlichkeitsmasse* von ω_n.

Beispiel 2.8 (Wahrscheinlichkeitsberechnung eines endlichen Pfades)
Es sei $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ die Markov-Kette aus Abbildung 2.6. Der Pfad $\omega := (s_0, s_1, s_4, s_5)$ ist ein endlicher Pfad in $\mathcal{D}$. Die Wahrscheinlichkeit von ω berechnet sich nun folgendermaßen:

$$prob(\omega) = T(s_0, s_1) \cdot T(s_1, s_4) \cdot T(s_4, s_5) = p_0 \cdot p_3 \cdot p_7 = 1 \cdot 0.4 \cdot 0.3 = 0.12$$

2.3.3 PCTL

Um bestimmte Eigenschaften für Pfade in einer Markov-Kette zu definieren, wird eine eigene logische Sprache benötigt, PCTL [27]. PCTL ist eine Übermenge der Sprache CTL, welche für nicht stochastische Systeme verwendet werden kann.

Definition 2.17 (PCTL – Syntax)
Die Syntax von PCTL setzt sich aus zwei verschiedenen Arten von Formeln zusammen:

- *PCTL-Zustandsformeln:* $\Phi := \top \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid P_{\bowtie p}[\phi]$
- *PCTL-Pfadformeln:* $\phi := \Phi_1 \,\mathcal{U}\, \Phi_2 \mid \mathcal{X}\Phi$

Wobei $p \in [0,1] \subset \mathbb{R}$, $a \in AP$ *und* $\bowtie \in \{>, \geq, \leq, <\}$.

$P_{\bowtie p}[\phi]$ ist der probabilistische Operator. Er bestimmt die Wahrscheinlichkeit p, mit der eine PCTL-Pfadformel ϕ zu erfüllen ist. Eine andere Schreibweise für $P_{\bowtie p}[\phi]$ ist $[\phi]_{\bowtie p}$.

Definition 2.18 (PCTL – Semantik)
Es sei $\models$ *die kleinste binäre Relation, welche die folgenden Bedingungen erfüllt.* s *ist dabei ein Zustand,* ω *ein unendlicher Pfad.*

$$
\begin{array}{lll}
s & \models \top & \\
s & \models a & \Leftrightarrow a \in \mathcal{L}(s) \\
s & \models \Phi_1 \wedge \Phi_2 & \Leftrightarrow s \models \Phi_1 \text{ und } s \models \Phi_2 \\
s & \models \neg\Phi & \Leftrightarrow s \not\models \Phi \\
s & \models P_{\bowtie p}[\phi] & \Leftrightarrow prob_s(\omega \in Paths_{inf}(s) : \omega \models \phi) \bowtie p \\
\omega & \models \Phi_1 \,\mathcal{U}\, \Phi_2 & \Leftrightarrow \exists\, i \in \mathbb{N} : \omega^i \models \Phi_2 \text{ und } \forall\, j : 0 \leq j < i : \omega^i \models \Phi_1 \\
\omega & \models \mathcal{X}\Phi & \Leftrightarrow \omega^1 \models \Phi
\end{array}
$$

Der logische Operator $\vee$, die logische Konstante $\bot$ sowie die PCTL-Operatoren $\mathcal{G}$ („es gilt immer“) und $\mathcal{F}$ („es gilt irgendwann“) lassen sich aus den obigen Definitionen ableiten:

$$
\begin{array}{lll}
s & \models \bot & \Leftrightarrow s \models \neg\top \\
s & \models \Phi_1 \vee \Phi_2 & \Leftrightarrow s \models \neg(\neg\Phi_1 \wedge \neg\Phi_2) \text{ (de Morgan)} \\
\omega & \models \mathcal{F}\Phi & \Leftrightarrow \omega \models \top \,\mathcal{U}\, \Phi \\
\omega & \models \mathcal{G}\Phi & \Leftrightarrow \omega \models \neg(\top \,\mathcal{U}\, \neg\Phi)
\end{array}
$$

Im Folgenden beschränken wir uns auf PCTL-Formeln der Form $P_{\leq p}[a \,\mathcal{U}\, b]$. Sie reichen aus, um Sicherheitseigenschaften auf Markov-Ketten zu formulieren.

Will man nun überprüfen, ob eine PCTL-Pfadformel $\phi := P_{\leq p}[a \,\mathcal{U}\, b]$ in einer Markov-Kette $\mathcal{D}$ gültig ist, so muss man die Wahrscheinlichkeitsmasse aller unendlichen Pfade ω mit $\omega \models a \,\mathcal{U}\, b$ bestimmen

Häufig bietet es sich an, die Gültigkeit einer Formel ϕ nicht zu beweisen, sondern zu widerlegen. In diesem Fall wird nach einem *Gegenbeispiel* gesucht. Ein *Gegenbeispiel* kann aus einem einzelnen endlichen Pfad bestehen, der $a \,\mathcal{U}\, b$ erfüllt und dessen Wahrscheinlichkeitsmasse die Wahrscheinlichkeitsgrenze p von ϕ übersteigt. Es kann

sich aber auch um eine Menge von Pfaden handeln, die alle $a \mathcal{U} b$ erfüllen und deren gemeinsame Wahrscheinlichkeitsmasse hoch genug ist, um p zu überschreiten. Einen Pfad, der $a \mathcal{U} b$ erfüllt, bezeichnet man als Zeugen.

Definition 2.19 (Zeuge und Gegenbeispiel)
Es sei $\phi := P_{\leq p}[a \mathcal{U} b]$ eine Sicherheitseigenschaft einer Markov-Kette $\mathcal{D}$.

- *Als* Zeuge *bezeichnet man einen (endlichen) Pfad $\omega \in Paths_{fin}(s_0)$, für den gilt:*

$$\omega \models a \mathcal{U} b$$

- *Als* Gegenbeispiel *bezeichnet man eine Menge E von Zeugen, für die gilt:*

$$prob(\{\Delta(\omega) \mid \omega \in E\}) > p$$

  Die $\Delta(\omega)$ sind dabei nicht notwendigerweise disjunkt.

Beispiel 2.9 (Zeuge und Gegenbeispiel)

- *Gegeben: $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$, die in Abbildung 2.6 gezeigte Markov-Kette. Die Übergangswahrscheinlichkeiten für $\mathcal{D}$ sind erneut durch die Übergangsmatrix T gegeben mit:*

$$p_0 := p_4 := p_5 := p_8 := 1, p_1 := 0.1, p_2 := 0.5, p_3 := 0.4, p_6 := 0.7, p_7 := 0.3$$

  $\phi := P_{\leq 0.8}[a \mathcal{U} b]$ sei eine Sicherheitseigenschaft für $\mathcal{D}$ mit Wahrscheinlichkeitsgrenze $p := 0.8$

- *Gesucht: Ein Gegenbeispiel für ϕ.*

- *Durchführung:*
  Zunächst sucht man nach Zeugen für $a \mathcal{U} b$. Der Pfad $\omega := (s_0, s_1, s_4)$ ist ein solcher Zeuge. Nun berechnet man die Wahrscheinlichkeit von ω:

$$prob(\omega) = p_0 \cdot p_3 = 1 \cdot 0.4 = 0.4$$

  Da $prob(\omega) \leq p = 0.8$ haben wir zwar einen Zeugen gefunden, jedoch noch kein gültiges Gegenbeispiel. Wir suchen also nach weiteren Zeugen:

$$\omega' := (s_0, s_1, s_3, s_4) \text{ mit } prob(\omega') = p_0 \cdot p_2 \cdot p_5 = 1 \cdot 0.5 \cdot 1 = 0.5$$

  Die Wahrscheinlichkeitsmasse von ω' reicht ebenfalls nicht aus, um diesen Zeugen allein zu einem Gegenbeispiel zu machen. Die gemeinsame Wahrscheinlichkeitsmasse von ω und ω' reicht jedoch aus, um p zu überschreiten:

$$prob(\omega) + prob(\omega') = 0.4 + 0.5 = 0.9 \geq p$$

  Damit ist $E := \{\omega, \omega'\}$ ein Gegenbeispiel für ϕ.

3 Bounded Model Checking auf stochastischen Systemen

Bounded Model Checking auf stochastischen Systemen – im Folgenden abgekürzt als SBMC – ist eine Erweiterung von BMC auf DTMCs. Genauso wie bei BMC geht es darum, zu überprüfen, ob auf einem gegebenen System eine Sicherheitsbedingung gilt. In diesem Fall handelt es sich bei dem System allerdings um eine DTMC.
Die Fragestellung lässt sich folgendermaßen formulieren:

> Gegeben sei eine Markov-Kette $\mathcal{D} = (S, s_0, T, AP, \mathcal{L})$. a und b seien Propositionen aus AP. Des Weiteren sei $\Phi = P_{\leq p}[a\,\mathcal{U}\,b]$ eine Sicherheitsbedingung auf $\mathcal{D}$. Gibt es ein Gegenbeispiel E auf $\mathcal{D}$, sodass Φ widerlegt wird?

Statt nach Zeugen beliebiger Länge zu suchen, sucht man nun nach Zeugen der Länge k. Haben die gefundenen Zeugen zusammengenommen keine ausreichende Wahrscheinlichkeitsmasse, um die Wahrscheinlichkeitsgrenze p zu überschreiten, so wird die Suche für $k + 1$ fortgesetzt. Dies wird so lange wiederholt, bis die Wahrscheinlichkeitsmasse der gefundenen Zeugen p übersteigt oder bis eine maximale Iterationstiefe erreicht wird. Ein Zustand, in dem b gilt, heißt *kritischer Zustand*.

3.1 Vorgehensweise

Bounded Model Checking auf stochastischen Systemen setzt sich aus verschiedenen Phasen zusammen, die in Abbildung 3.1 dargestellt sind.

Zuerst werden aus dem System, gegeben in Form einer DTMC $\mathcal{D}$ und der Sicherheitsbedingung Φ, mehrere Entscheidungsdiagramme generiert. In Abbildung 3.1 sind diese mit DD für englisch „**D**ecision **D**iagrams“ abgekürzt. Die Darstellung in Form von Entscheidungsdiagrammen ist der Startpunkt für das eigentliche SBMC-Verfahren.

Aus den Entscheidungsdiagrammen wird ein boolescher Ausdruck ψ in KNF generiert. Mit Hilfe von ψ wird nun ein Pfad beziehungsweise Zeuge der Länge k gesucht. Im Anschluss daran wird überprüft, ob die Gesamt-Wahrscheinlichkeit der resultierenden Pfadmenge größer ist als die Wahrscheinlichkeitsgrenze p von Φ. Ist dies der Fall, so stellt die Pfadmenge ein gültiges Gegenbeispiel dar und das Verfahren wird beendet. Eine andere Möglichkeit, wie das Verfahren beendet werden kann, ist durch das Erreichen einer maximalen Iterationstiefe k_{max} gegeben, die im Voraus festgelegt wird. In diesem Fall ist die Pfadmenge kein gültiges Gegenbeispiel, enthält aber alle Zeugen bis zur Länge k_{max}.

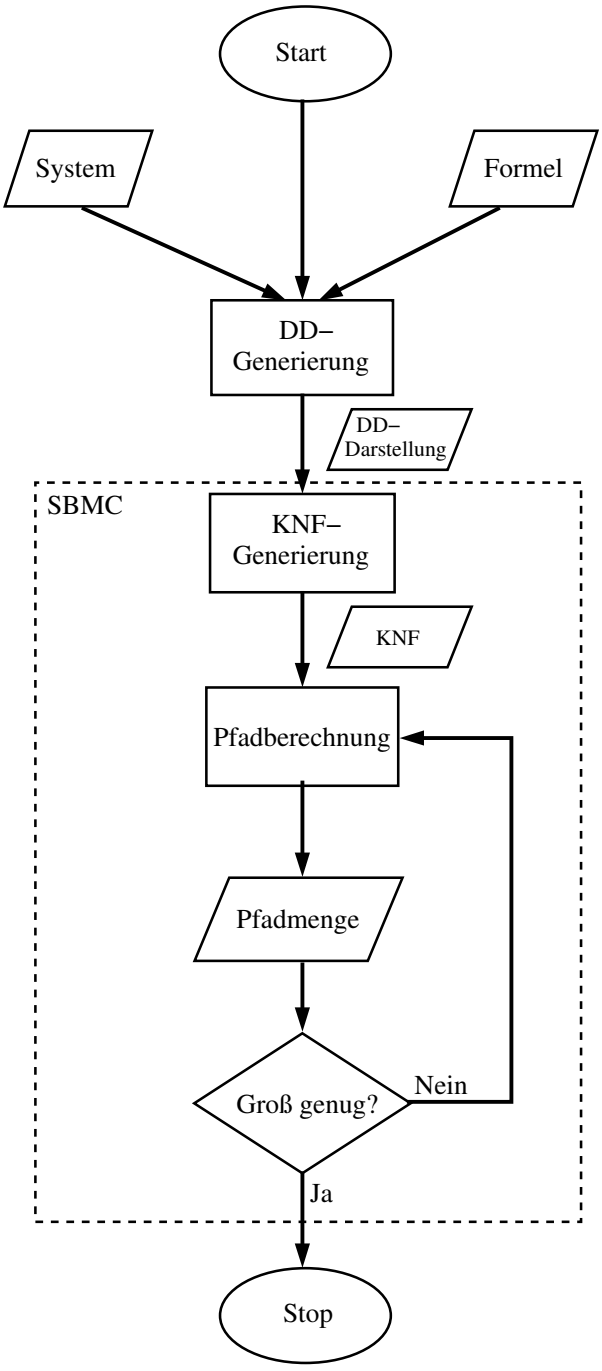

Abbildung 3.1: Flowchart zur Vorgehensweise von SBMC

Reicht die Gesamt-Wahrscheinlichkeitsmasse der Pfadmenge nicht aus und wurde die maximale Iterationstiefe k_{max} noch nicht erreicht, so wird weiter nach Zeugen gesucht. Zuerst werden weitere Pfade der Länge k berechnet, bis keine neuen mehr hinzu kommen. Dann wird die Iterationstiefe um 1 höhergesetzt und nach Zeugen der Länge $k + 1$ gesucht. Dies wird so lange durchgeführt, bis die oben erwähnten Abbruchkriterien erfüllt werden.

3.1.1 Erstellen der Entscheidungsdiagramme

Für SBMC wird ein boolescher Ausdruck ψ in konjunktiver Normalform benötigt, der die DTMC $\mathcal{D}$ darstellt. Ein solcher Ausdruck lässt sich direkt aus der formalen Beschreibung von $\mathcal{D}$ ableiten, seine Größe wäre in diesem Fall linear zu der Anzahl an Transitionen in $\mathcal{D}$. Nun sollte ψ aber möglichst kompakt sein, da unnötig viele Klauseln das SBMC-Verfahren verlangsamen würden. Um eine kompakte Darstellung von $\mathcal{D}$ in Form einer KNF zu erhalten, wird $\mathcal{D}$ zunächst in mehrere Entscheidungsdiagramme umgewandelt. Aus diesen Entscheidungsdiagrammen kann anschließend ein kompakter boolescher Ausdruck ψ in konjunktiver Normalform gewonnen werden.

In Kapitel 3.1.1 soll nun beschrieben werden, wie eine DTMC $\mathcal{D}$ in mehrere Entscheidungsdiagramme umgewandelt werden kann. Im darauf folgenden Kapitel 3.1.2 wird gezeigt, wie man aus diesen Entscheidungsdiagrammen eine KNF generiert.

Insgesamt werden vier Entscheidungsdiagramme erzeugt: Ein BDD für den Startzustand s_0, ein BDD für die kritischen Zustände und je ein MTBDD und ein BDD für die Transitionsrelation.

Um den BDD für die kritischen Zustände berechnen zu können, wird eine charakteristische Funktion $\mathcal{X}_c$ für die Zustände von $\mathcal{D}$ benötigt. $\mathcal{X}_c$ wird mit Hilfe der Beschriftungsfunktion $\mathcal{L}$ von $\mathcal{D}$ definiert:

$$\forall\, s \in S, \forall\, c \in AP:\ \mathcal{X}_c(s) := \begin{cases} 1, & \text{wenn } c \in \mathcal{L}(s) \\ 0, & \text{sonst} \end{cases} \tag{3.1}$$

Es handelt sich dabei um eine boolesche Funktion, die sich als BDD darstellen lässt.

Um den MTBDD für die Transitionsrelation zu bauen, wird die Transitionsmatrix T rekursiv in Quadranten aufgeteilt:
Es muss sichergestellt werden, dass es sich bei T um eine $n \times n$-Matrix handelt, mit $n = 2^i$, $i \in \mathbb{N}$. Trifft dies nicht zu, so wird T um weitere Spalten und Zeilen ergänzt, bis die nächstgrößere 2er-Potenz erreicht ist. Die Einträge dieser zusätzlichen Zeilen und Spalten lauten alle 0.

T wird nun in vier gleich große Quadranten unterteilt, die alle mit einem Variablenpaar (x_0, y_0) markiert sind. Dabei ist x_0 die Variable für die Zeilen und y_0 die Variable für die Spalten der Matrix.

Die vier Quadranten $T_0(x_0, y_0)$ werden ihrerseits wieder in Quadranten unterteilt. Die entsprechenden Variablen sind x_1 und y_1. Dieser Vorgang wird so lange fortgesetzt, bis jeder Quadrant nur aus einem einzigen Wert besteht. Dieser Wert wird zu einem

Blatt in dem zu bauenden MTBDD. Der Pfad zu diesem Blatt ist gegeben durch die Variablenbelegungen der x_i und y_i.

Zum besseren Verständnis wird der Bau eines MTBDDs aus einer Matrix anhand der in Abbildung 3.2 dargestellten DTMC demonstriert. Eine detaillierte und formale Beschreibung des Vorgangs findet sich in [23].

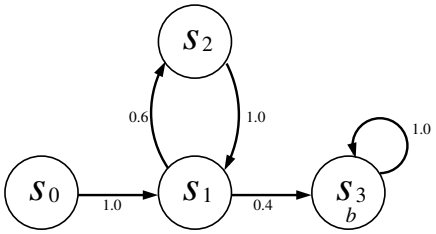

Abbildung 3.2: DTMC zu Beispiel 3.1

Beispiel 3.1 (Erstellung eines MTBDD aus einer Transitionsmatrix)
Es sei $\mathcal{D}$ die in Abbildung 3.2 dargestellte DTMC. Die Transitionsrelation ist durch folgende Matrix T gegeben:

$$T := \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0.6 & 0.4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

T wird nun in vier Quadranten unterteilt, die dazu gehörenden Variablen sind x_0 und y_0:

$$T_0(0,0) := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad T_0(0,1) := \begin{pmatrix} 0 & 0 \\ 0.6 & 0.4 \end{pmatrix}$$
$$T_0(1,0) := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad T_0(1,1) := \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

Jede dieser vier Matrizes wird wiederum in vier Quadranten unterteilt. Die Variablen sind nun x_1 und y_1. Die neuen Quadranten enthalten jeweils nur einen Wert. Die vier Quadranten von $T_0(0,1)$ sind:

$$T_1(0,0) := 0,\ T_1(0,1) := 0,\ T_1(1,0) := 0.6,\ T_1(1,1) := 0.4$$

Diese Quadranten sind Blätter des MTBDD. Der Pfad zu den Blättern ist durch die jeweilige Variablenbelegung gegeben. Das Blatt 0.6 zum Beispiel ist durch die Variablenbelegung $x_0 = 0, y_0 = 1, x_1 = 1$ und $y_1 = 0$ kodiert, wohingegen zu dem Blatt 0 mehrere Variablenbelegungen führen. Auf diese Weise wird eine Funktion definiert, die sich als MTBDD darstellen lässt. Abbildung 3.3(a) zeigt den vollständig reduzierten MTBDD zu $\mathcal{D}$.

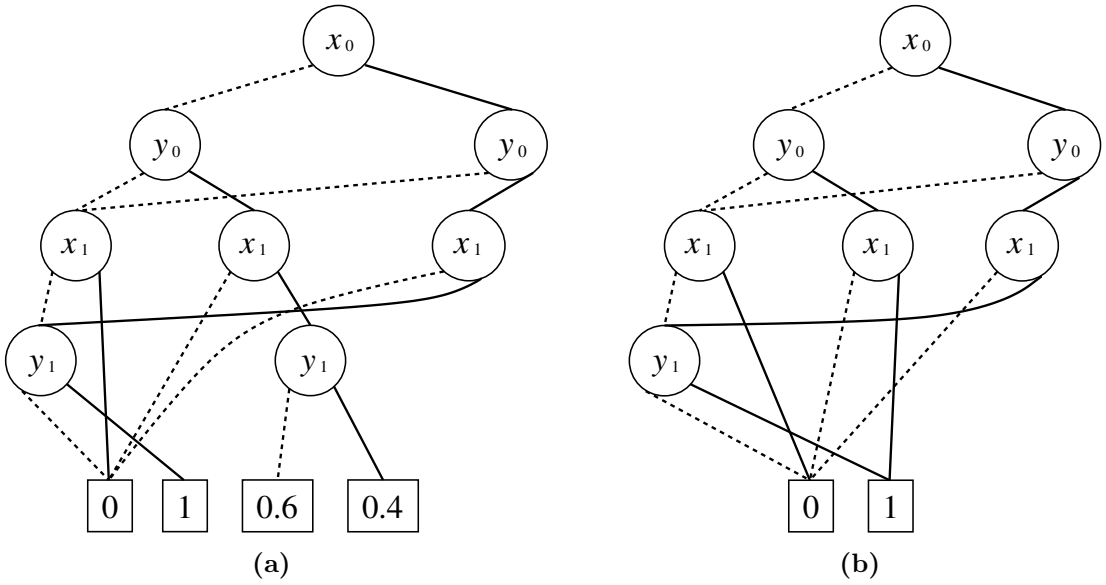

Abbildung 3.3: MTBDD und BDD zu Beispiel 3.1

Jedes Blatt gibt nun die Wahrscheinlichkeit eines Übergangs von einem Zustand s (englisch *State*) in einen Folgezustand (englisch *Nextstate*) s' an. Dabei wird der Zustand s durch die Belegung der Variablen x_i und der Folgezustand s' durch die Belegung der Variablen y_i kodiert. So wird zum Beispiel der Zustand s_1 in Abbildung 3.2 in Beispiel 3.1 durch die Variablenbelegung $x_0 = 0, x_1 = 1$ kodiert. Die Variablenbelegung $y_0 = 1, y_1 = 0$ verweist auf den möglichen Folgezustand s_2. Die Kombination von Zustands- und Folgezustandsvariablen ergibt den Zustandsübergang. Gibt es keine Kante zwischen zwei Zuständen, so führt die entsprechende Variablenbelegung zu einem 0-Blatt. Die Variablenbelegung $x_0 = y_0 = y_1 = 0, x_1 = 1$ führt beispielsweise in dem MTBDD in Abbildung 3.3(a) zum Blatt 0. Dies entspricht der DTMC in Abbildung 3.2. Dort ist zu sehen, dass es von dem kodierten Zustand s_1 ($x_0 = 0, x_1 = 1$) zu dem kodierten Folgezustand s_0 ($y_0 = y_1 = 0$) keine Kante gibt.

Zustände werden also nicht explizit, sondern implizit durch Belegungen von bestimmten Variablen dargestellt. Entsprechend ist der BDD für den Startzustand s_0 eine lineare Kette von Knoten, der die initiale Belegung der Zustandsvariablen darstellt.

Wie bereits erwähnt, werden für die durch die Matrix T gegebene Transitionsrelation sowohl ein MTBDD als auch ein BDD erzeugt. Letzterer lässt sich einfach aus dem MTBDD gewinnen: Alle Blätter, die einen Wert größer als 0 haben, werden zu einem 1-Blatt umgewandelt. Der entstehende BDD kann unter Umständen noch weiter reduziert werden. Alternativ dazu kann der BDD auch direkt durch Parallelkomposition aus der formellen Beschreibung der DTMC generiert werden (siehe [37]). Der BDD gibt die korrekten Zustandsübergänge einer DTMC $\mathcal{D}$ wider, allerdings ohne die damit verbundenen Wahrscheinlichkeiten. Abbildung 3.3(b) zeigt den BDD für die Transitionsrelation aus Beispiel 3.1.

Nachdem die DTMC $\mathcal{D}$ in Entscheidungsdiagramme umgewandelt wurde, müssen noch Zustandsübergänge von $\mathcal{D}$ entfernt werden, indem die entsprechenden Belegungen im Transitions-BDD statt zu einem 1- zu einem 0-Blatt geleitet werden.

Zuerst werden die ausgehenden Kanten aller Zustände entfernt, die weder a noch b erfüllen. Dies kann bedenkenlos getan werden, da diese Zustände ohnehin irrelevant für die Suche nach Zeugen für Φ sind: Kein Pfad, der solche Zustände enthält, erfüllt Φ. Wenn diese Kanten nicht entfernt werden würden, bestünde darüber hinaus die Gefahr, dass später bei der Suche nach Zeugen fälschlicherweise Pfade gefunden werden würden, die $[a \,\mathcal{U}\, b]$ nicht erfüllen.

Ebenfalls werden alle ausgehenden Kanten der kritischen Zustände entfernt. Auch dies hat keinerlei negative Auswirkungen auf die Korrektheit des Verfahrens, da die gesuchten Zeugen in einem kritischen Zustand enden. Außerdem ist es notwendig, diese Kanten zu entfernen. Würde es nicht geschehen, so wäre es möglich, dass „neue“ Zeugen durch das Aneinanderhängen bereits bekannter Zeugen entstehen, womit diese mehrfach in die Wahrscheinlichkeitsmasse einfließen würden. Darüber hinaus wird das System auf diese Art verkleinert, was die Suche nach Zeugen beschleunigt.

Die DTMC befindet sich nun in einer Form, aus der die KNF ψ für das SBMC-Verfahren erstellt werden kann.

3.1.2 KNF-Generierung

Aus den im vorangegangenen Kapitel erläuterten BDDs werden boolesche Ausdrücke in konjunktiver Normalform erzeugt. Dies geschieht mit Hilfe der rekursiven Anwendung der Tseitin-Transformation[1].

Die Tseitin-Transformation liefert für jeden Knoten d eines BDDs eine Menge von Klauseln. Die Variable x, mit welcher der Knoten d beschriftet ist, wird mit Hilfe von Hilfsvariablen l und h in Relation zu den ausgehenden Kanten von d gesetzt. Die eingehenden Kanten von d werden durch eine einzelne Hilfsvariable o dargestellt. Ohne Beschränkung der Allgemeinheit steht l für die *Low*- und h für die *High*-Kante von d. Abbildung 3.4 zeigt einen BDD-Knoten mit entsprechender Beschriftung.

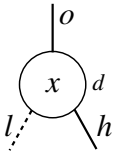

Abbildung 3.4: BDD-Knoten mit Kanten-Beschriftung

Die Tseitin-Transformation eines BDD-Knotens d entspricht der Zerlegung eines 2-Bit-Multiplexers mit x als Select-Eingang, o als Ausgang, und dem 0- beziehungsweise 1-Eingang l beziehungsweise h.

$$o \leftrightarrow \left(\overline{x}l + xh\right) \;\equiv\; \left(\overline{o} + x + l\right) \cdot \left(o + x + \overline{l}\right) \cdot \left(\overline{o} + \overline{x} + h\right) \cdot \left(o + \overline{x} + \overline{h}\right) \tag{3.2}$$

[1] Siehe Kapitel 2.1.1, Seite 9.

Für jeden BDD-Knoten d liefert Formel 3.2 maximal vier Klauseln mit jeweils drei Literalen. Die Anzahl pro Knoten kann in bestimmten Fällen geringer ausfallen:

- Führt eine der ausgehenden Kanten von d zu dem Blatt 0 oder 1, so kann die entsprechende Hilfsvariable durch die Konstante 0 oder 1 ersetzt werden. Dadurch reduziert sie die Anzahl von Klauseln für d um eins, wobei eine der drei verbliebenen Klauseln nur zwei Literale enthält.

- Führen die beiden ausgehenden Kanten von d zu Blättern, so stellt der BDD-Knoten d direkt das Literal x oder $\overline{x}$ dar. In diesem Fall wird keine KNF für d erstellt. Stattdessen wird das Literal von d statt der entsprechenden Hilfsvariable in den Klauseln der Vorgängerknoten von d verwendet.

Die Anzahl an Klauseln und Literalen ist damit linear in der Knotenanzahl des jeweiligen BDDs.

Bei der KNF-Erzeugung wird der BDD von den Blättern ausgehend rückwärts durchschritten. Die für die eingehende Kante erzeugte Hilfsvariable o_i eines Knotens d_i wird dabei zur Variable der entsprechenden ausgehenden Kante der Vorgängerknoten von d_i.
Wird der Wurzelknoten w erreicht, so muss Folgendes beachtet werden: w hat keine eingehenden Kanten. Trotzdem wird eine Variable o_i für w erzeugt. Diese Variable wird als Unit Clause[2] in die KNF eingefügt. Auf diese Weise wird sichergestellt, dass die KNF für den BDD genau dann erfüllbar ist, wenn sich durch die Variablenbelegung ein Pfad von w zu einem Blatt ergibt.

Jeder der drei BDDs – einer für den Startzustand, einer für die Transitionsrelation und einer für die kritischen Zustände – wird auf diese Weise in eine Menge von Klauseln umgewandelt. So erhält man ein Prädikat für den Startzustand, einen booleschen Ausdruck für die Transitionsrelation und eine Bedingung für die kritischen Zustände. Zusammen bilden sie einen logischen Ausdruck ψ in konjunktiver Normalform, ähnlich wie Formel 2.3 auf Seite 22. Wie beim normalen Model Checking wird die KNF für die Transitionsrelation entsprechend oft in ψ eingefügt, wobei die Indizes der Variablen verschoben werden. Auf diese Weise wird ψ zu einer Repräsentation eines Pfades der Länge k, der bei Startzustand s_0 beginnen und bei einem kritischen Zustand enden soll.

$$\psi := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \mathcal{X}_b(s_k) \tag{3.3}$$

$I(s_0)$ ist dabei das Prädikat des Startzustands s_0, $T(s_i, s_{i+1})$ ist die Transitionsrelation und $\mathcal{X}_b(s_k)$ steht für die kritischen Zustände.

Mit der KNF ψ kann nun der eigentliche BMC-Vorgang gestartet werden.

3.1.3 BMC

Die in Kapitel 3.1.2 erzeugte KNF ψ wird an einen SAT-Solver übergeben. Findet der Solver eine erfüllende Belegung, so steht diese für einen Pfad der Länge k in $\mathcal{D}$, der im

[2]Siehe Kapitel 2.1.2, Seite 10.

Startzustand s_0 beginnt und in einem kritischen Zustand endet. Es wurde ein Zeuge ω gefunden.

Mit Hilfe des aus der Transitionsmatrix erzeugten MTBDDs[3] wird die Wahrscheinlichkeit des Zeugen errechnet: Entsprechend den Belegungen der Zustands- und Folgezustandsvariablen x_i und y_i wird der MTBDD durchschritten. Das schließlich erreichte Blatt liefert die Wahrscheinlichkeit für einen Zustandsübergang. Um die Wahrscheinlichkeit für ganz ω zu berechnen, muss der MTBDD k-mal durchlaufen werden, immer der Belegung der Variablen mit dem entsprechenden Index-Shift folgend. k solche Durchläufe reichen aus, da die k-te Transition vom vorletzten zum letzten Zustand von ω führt. Die Wahrscheinlichkeiten, die bei den jeweiligen Durchläufen errechnet werden, werden aufmultipliziert. Dies ist anlaog zur Wahrscheinlichkeitsberechnung eines endlichen Pfades wie in Beispiel 2.8 auf Seite 27.

Falls ω über genügend Wahrscheinlichkeitsmasse verfügt, um die Wahrscheinlichkeitsgrenze der Sicherheitsbedingung Φ zu überschreiten, so wurde mit ω ein Gegenbeispiel für Φ gefunden. Sollte dies nicht der Fall sein, so wird der SAT-Solver erneut aufgerufen.

Es muss nun verhindert werden, dass der Solver den Pfad ω erneut findet. Dies geschieht durch das Negieren von ω in Form einer weiteren Klausel: Wie bereits erwähnt, entspricht ω einer erfüllenden Belegung der Formel 3.3. Die Belegung der Zustandsvariablen der einzelnen Zeitrahmen kodiert die Zustände des Pfades. Es seien nun $x_{i,0}, \ldots, x_{i,n}$ die Literale der Zustandsvariablen für den Zeitrahmen i. Indem diese Literale für jeden Zeitschritt negiert werden, erhält man folgende Klausel:

$$\left(\bigvee_{i=1}^{k} \bigvee_{j=0}^{n} \overline{x_{i,j}} \right) \tag{3.4}$$

Die Klausel-Datenbank des SAT-Solvers wird um diese *Pfadklausel* ergänzt. Da sie einer Negation von ω entspricht, wird der Solver so gezwungen, neue Pfade zu finden.

Bei Formel 3.4 ist noch zu beachten, dass nicht mit dem Zeitrahmen 0, sondern mit dem Zeitrahmen 1 begonnen wird. Dies ist möglich, da ausschließlich nach Pfaden gesucht wird, die im Startzustand s_0 beginnen, was durch Formel 3.3 sichergestellt ist. Da dieser Zustand also der Beginn aller Pfade ist, die der SAT-Solver möglicherweise findet, muss er nicht ausgeschlossen werden. Das bietet den Vorteil, dass die Pfadklausel kleiner ausfällt.

Durch die Pfadklauseln ist der Solver in der Lage, Pfade zu lernen. Dies ist vergleichbar mit dem Lernen von Konflikten bei gewöhnlichen SAT-Problemen[4].

Mit Hilfe des SAT-Solvers wird so lange nach Zeugen der Länge k gesucht, bis keine mehr gefunden werden. Reicht die Wahrscheinlichkeitsmasse der bis zu diesem Zeitpunkt gefundenen Zeugen nicht aus, um die Wahrscheinlichkeitsgrenze p zu überschreiten, so wird nach Zeugen der Länge $k+1$ gesucht. Der SAT-Solver wird komplett neu gestartet und erhält einen neuen booleschen Ausdruck ψ, der sich nun über einen

[3] Zur MTBDD-Erzeugung aus der Transitionsrelation siehe Seite 33 in Kapitel 3.1.1.

[4] Vergleiche dazu Kapitel 2.1.2, Seite 15.

weiteren Zeitrahmen erstreckt. Durch den Neustart verliert der SAT-Solver alle bisher übergebenen und gelernten Klauseln. Dies ist notwendig, damit nur noch Pfade der Länge $k + 1$ berechnet werden.

Die Suche nach immer längeren Zeugen setzt sich fort, bis entweder die Wahrscheinlichkeitsmasse für ein Gegenbeispiel ausreicht oder bis eine vorgegebene Iterationstiefe erreicht wird. Startwert für k ist dabei nicht der Zeitrahmen 0, sondern die minimale Distanz $dist_{min}$ zwischen Startzustand und einem kritischen Zustand, also die Länge des kürzesten Pfades zwischen Startzustand s_0 und einem kritischen Zustand. Es ist offensichtlich, dass Pfadlängen, die kürzer sind als $dist_{min}$, nicht betrachtet werden müssen. Kein Pfad, der kürzer ist, kann ein Zeuge sein, da kein kritischer Zustand über einen solchen Pfad erreichbar ist. Man erhält $dist_{min}$ durch Rückwärts-Traversierung, ausgehend von den kritischen Zuständen, oder durch vom Startzustand s_0 ausgehende Vorwärts-Traversierung.

Algorithmus 2 fasst die bisherige Vorgehensweise noch einmal zusammen. Beim Aufruf der Funktion SBMCSolve bezeichnet $\mathcal{D}$ die zu betrachtende DTMC, die bereits in Entscheidungsdiagramme umgewandelt wurde. Φ ist die zu überprüfende Sicherheitseigenschaft, p die Wahrscheinlichkeitsgrenze und k_{max} die maximale Iterationstiefe. Der Algorithmus rechnet so lange, bis entweder p überschritten oder k_{max} erreicht wurde.

Zunächst wird im Preprocessing $\mathcal{D}$ auf die oben beschriebene Weise reduziert. Im Anschluss daran wird $dist_{min}$ ermittelt.

Auf das Preprocessing folgt die Initialisierung (siehe Algorithmus 2, Zeile 5): Die KNF ψ wird für die Distanz $k := dist_{min}$ erstellt. *prob* ist die Wahrscheinlichkeitsmasse der gefundenen Zeugen und beträgt anfangs 0. Die Menge der gefundenen Zeugen, *Paths*, ist zu Beginn leer.

In Zeile 9 beginnt das eigentliche SBMC-Verfahren: Die KNF ψ wird an einen SAT-Solver übergeben. Findet dieser eine Lösung, so wird daraus der neue Zeuge ω berechnet. Dieser wird zu *Paths* hinzugefügt. Die Wahrscheinlichkeitsmasse von ω wird berechnet und zu der bis dahin erreichten Gesamt-Wahrscheinlichkeitsmasse *prob* addiert. Danach wird ω ausgeschlossen, sodass der SAT-Solver beim nächsten Schleifendurchlauf einen anderen Pfad finden muss.

Findet der SAT-Solver keine Lösung für ψ, so wurden entweder alle existierenden Zeugen für das aktuelle k bereits gefunden oder es existieren keine Zeugen für diese Iterationstiefe. Der SAT-Solver wird neu gestartet, wodurch seine Klausel-Datenbank gelöscht wird. k wird um 1 erhöht, ψ wird neu berechnet und das Verfahren für eine neue Iterationstiefe gestartet (siehe Zeile 16 in Algorithmus 2).

Das SBMC-Verfahren wird beendet, wenn die erreichte Wahrscheinlichkeitsmasse *prob* die Wahrscheinlichkeitsgrenze p überschreitet oder wenn k_{max} erreicht wird. Nun wird noch überprüft, ob ein Gegenbeispiel gefunden wurde oder nicht und eine entsprechende Meldung ausgegeben. Im Anschluss daran gibt der Algorithmus die erreichte Wahrscheinlichkeitsmasse *prob*, zusammen mit der erreichten Iterationstiefe k und der Menge der bis dahin gefundenen Zeugen *Paths* zurück.

Die in diesem Kapitel geschilderte Vorgehensweise für SBMC enthält bereits einige der kleineren Optimierungen. Ein wesentlicher Punkt wurde aber bisher ausgespart:

Algorithm 2 SBMC-Algorithmus

```
1:  function SBMCSolve(𝒟, Φ, p, k_max)
2:      // preprocessing:
3:      reduce 𝒟;
4:      compute dist_min;

5:      // initialization:
6:      compute initial CNF ψ for k := dist_min;
7:      prob := 0.0;
8:      Paths := {};

9:      // SBMC procedure:
10:     while (prob ≤ p) && (k ≤ k_max) do
11:         if SATSolve(ψ) then
12:             get current solution for ψ and generate path ω;
13:             Paths := Paths ∪ ω;
14:             prob := prob + prob(ω);
15:             ExcludePath(ω);
16:         else
17:             restart SAT-Solver;
18:             update ψ to k := k + 1;
19:         end if
20:     end while

21:     if prob > p then
22:         print: "Counterexample found!";
23:     else
24:         print: "No counterexample found!";
25:     end if
26:     return prob, k, Paths
27: end function
```

Das Erkennen von Schleifen beziehungsweise *Loops*, das es ermöglicht, Gegenbeispiele kompakt zu repräsentieren und Pfade im Voraus zu berechnen und auszuschließen. Die nächsten beiden Unterkapitel sind diesen Optimierungen gewidmet.

3.2 Looperkennung und Gegenbeispiel-Kompaktierung

Eine DTMC $\mathcal{D}$ ist in der Regel nicht azyklisch. Das hat zur Folge, dass Pfade ab einer gewissen Länge häufig über Schleifen beziehungsweise englisch *Loops* verfügen. Schleifen spielen eine wichtige Rolle in der Erzeugung und kompakten Darstellung von Gegenbeispielen.

Bisher ist das SBMC-Verfahren nicht in der Lage, Schleifen zu erkennen. Verfügt ein Pfad ω nun über eine Schleife l, so besteht die Gefahr, dass der SAT-Solver für

eine höhere Iterationstiefe neue Pfade erzeugt, indem er l mehrfach durchläuft oder *abwickelt*. Dies hat zwei wesentliche Nachteile: Zum einen wird unnötig Rechenzeit für die Berechnung eines im Prinzip bekannten Pfades verbraucht. Zum anderen steigt damit die Anzahl an Zeugen innerhalb eines Gegenbeispiels E sprunghaft an, ohne dass die Wahrscheinlichkeitsmasse von E wesentlich zunimmt.

Beispiel 3.2 (SBMC ohne Looperkennung)
Es sei $\mathcal{D} := (S, P, AP, \mathcal{L})$ die DTMC aus Abbildung 3.2 auf Seite 34. Gegeben sei die Sicherheitsbedingung $\Phi := P_{\leq 0.9}[\top \mathcal{U}\, b]$ mit Wahrscheinlichkeitsgrenze $p := 0.9$.
Die minimale Distanz, um s_3 zu erreichen, ist $dist_{min} = 2$ deswegen startet das Verfahren mit $k := 2$. Als maximale Iterationstiefe wird $k_{max} := 9$ festgelegt.
Im Folgenden sind die Zeugen, die gefunden werden, und ihre Wahrscheinlichkeiten angegeben:

$$
\begin{aligned}
k := 2: &\quad \omega := (s_0, s_1, s_3),\ prob(\omega) = 0.4 \\
k := 3: &\quad - \\
k := 4: &\quad \omega' := (s_0, s_1, s_2, s_1, s_3),\ prob(\omega') = 0.24 \\
k := 5: &\quad - \\
k := 6: &\quad \omega'' := (s_0, s_1, s_2, s_1, s_2, s_1, s_3),\ prob(\omega'') = 0.144 \\
k := 7: &\quad - \\
k := 8: &\quad \omega''' := (s_0, s_1, s_2, s_1, s_2, s_1, s_2, s_1, s_3),\ prob(\omega''') = 0.0864 \\
k := 9: &\quad -
\end{aligned}
$$

Nur für gerade k werden Zeugen gefunden, für ungerade k ist s_3 nicht erreichbar. Es ist deutlich erkennbar, dass die Wahrscheinlichkeitsmasse der einzelnen Zeugen mit ihrer Länge abnimmt.
Nachdem k_{max} erreicht wurde, bricht das Verfahren ab. Die erreichte Wahrscheinlichkeitsmasse aller bis dahin gefundenen Zeugen beträgt 0.8704 und liegt damit unter p. Es wurde also kein Gegenbeispiel gefunden.

In Beispiel 3.2 wird der Pfad $\omega := (s_0, s_1, s_3)$ immer wieder durch die Schleife (s_1, s_2, s_1) verlängert. Am Ende wurden vier Zeugen gefunden, deren gemeinsame Wahrscheinlichkeitsmasse jedoch nicht für ein Gegenbeispiel ausreicht. Solche Effekte können vermieden werden, indem Schleifen in bereits gefundenen Pfaden erkannt und entsprechend behandelt werden. So ist es möglich, Pfade, die durch das Abwickeln bereits bekannter Schleifen entstehen würden, im Voraus auszuschließen. Außerdem erhält man in Form von erweiterten Pfaden Zeugen mit hoher Wahrscheinlichkeit (siehe dazu Definition 3.2). Beides kann die Berechnung von Gegenbeispielen enorm beschleunigen, was sich auch in den experimentellen Ergebnissen in Kapitel 4 widerspiegelt.

Definition 3.1 (Loop)
Es sei $\omega := (s_0, \ldots, s_n)$ ein Pfad in einer DTMC $\mathcal{D} := (S, P, AP, \mathcal{L})$.
Eine Folge von Zuständen $l := (s_i, s_{i+1}, \ldots, s_{i+m})$ bezeichnen wir als Schleife *oder* Loop *in ω der Länge m, wenn gilt:*

- *l ist eine Teilfolge von ω.*
- *$s_i = s_{i+m}$, wobei s_i der Start- und s_{i+m} der Endzustand von l ist.*

Einen Pfad, dessen Loops entfernt wurden oder der über keine Loops verfügt, bezeichnen wir als *Basispfad* ω_b. Die Berechnung der Wahrscheinlichkeitsmasse eines Loops ist analog zu der Berechnung der Wahrscheinlichkeitsmasse eines Pfades (vergleiche dazu Definition 2.16 auf Seite 27).

Wird zur Looperkennung Definition 3.1 ohne irgendwelche Einschränkungen verwendet, so kann folgendes Problem auftreten: Besteht ein Loop aus Zustandsübergängen, die wieder zurück zu einem früheren Zustand führen, werden zwei verschiedene Loops statt einem erkannt.

Beispiel 3.3 (Doppeltes Erkennen von Loops)
Es sei $\mathcal{D} := (S, P, AP, \mathcal{L})$ die DTMC aus Abbildung 2.6 auf Seite 25. Zwischen den Zuständen s_1 und s_4 gibt es zwei Kanten: $s_1 \to s_4$ und $s_4 \to s_1$. Zusammen bilden sie einen Loop.
Es ist nun folgender Pfad gegeben:

$$\omega := (s_0, s_1, s_4, s_1, s_4, s_5)$$

ω ist offensichtlich durch Verlängerung des Basispfads $\omega_b := (s_0, s_1, s_4, s_5)$ mit Hilfe des Loops zwischen s_1 und s_4 entstanden. Geht die Looperkennung strikt nach Definition 3.1 vor, werden fälschlicherweise zwei verschiedene Loops erkannt: $l := (s_1, s_4, s_1)$ und $l' := (s_4, s_1, s_4)$.

Eine solche Situation entspricht jedoch nicht der DTMC $\mathcal{D}$, denn es existiert an dieser Stelle nur ein Loop. Um das doppelte Erkennen von Loops zu verhindern, führen wir zur Looperkennung eine Zusatzbedingung ein:

> Der Startzustand s_i eines Loops l ist immer der früheste Zustand in einem Pfad ω, der ein zweites Mal in ω vorkommt.

Durch die zusätzliche Bedingung zur Looperkennung wird bei einem entsprechenden Pfad nur noch ein Loop erkannt. So wird in Beispiel 3.3 für den Pfad $\omega := (s_0, s_1, s_4, s_1, s_4, s_5)$ nur noch der Loop $l := (s_1, s_4, s_1)$ gefunden[5].

Mit Hilfe von Basispfaden und den dazu gehörenden Loops lassen sich Zeugen berechnen, die eine besonders hohe Wahrscheinlichkeitsmasse aufweisen. Den Verbund von Basispfad und zugehörigen Loops bezeichnen wir als erweiterten Pfad.

Definition 3.2 (Erweiterter Pfad)
Es sei ω_b ein Basispfad in einer DTMC $\mathcal{D}$ und $L := \{l_1, \ldots, l_n\}$ die Menge der Loops, deren Start- beziehungsweise Endzustand in ω enthalten ist. Zusammen bilden sie einen erweiterten Pfad *$\omega_e := (\omega_b, L)$. ω_e repräsentiert alle Pfade, die durch beliebig*

[5]Durch entsprechende Umformulierung der Zusatz-Bedingung zur Looperkennung könnte ebenfalls dafür gesorgt werden, dass in Beispiel 3.3 nur $l' := (s_4, s_1, s_4)$ als Loop anerkannt wird. Dies wäre ebenfalls korrekt. Es muss nur die Eindeutigkeit des Loops in solchen Fällen garantiert sein.

viele Abwicklungen der Loops $l_1, \ldots l_n$ aus dem Basispfad ω_b entstehen können. Haben dabei zwei oder mehr Loops den gleichen Startzustand in ω_b, so ist die Reihenfolge der Abwicklungen beliebig.

Ein erweiterter Pfad ω_e beinhaltet also alle Pfade, die durch Abwickeln der Loops, die mit einem Basispfad ω_b zusammenhängen, entstehen könnten. Dementsprechend ist die Wahrscheinlichkeit eines erweiterten Pfades höher als die der einzelnen, durch Abwickeln entstehenden Pfade.

Satz 3.1 (Wahrscheinlichkeit eines erweiterten Pfades)
Es sei ω_b ein Basispfad in einer DTMC $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ und $L = \{l_1, \ldots, l_n\}$ die Menge der zu ω_b gehörenden Loops.
$L_{s_0}, \ldots, L_{s_n}$, wobei $s_0, \ldots, s_n$ die Zustände in ω_b sind, sind disjunkte Teilmengen von L. L_{s_i} enthält alle Loops, die im Zustand s_i beginnen. Es gilt: $L = \dot{\bigcup}_{s_i \in \omega} L_{s_i}$
Die Wahrscheinlichkeit des erweiterten Pfades $\omega_e = (\omega_n, L)$ errechnet sich nun wie folgt:

$$prob(\omega_e) = prob(\omega_b) \cdot \prod_{s_i \in \omega_b} \frac{1}{1 - \sum_{l \in L_{s_i}} prob(l)}$$

Beweis 3.1
Es sei nun $f := \frac{1}{1-\sum_{l \in L_{s_i}} prob(l)}$ und $p_l := prob(l)$ für alle $l \in L_{s_i}$, $s_i \in \omega$.
Es gilt nun, die Korrektheit von f zu beweisen. Dies geschieht über eine Fallunterscheidung über die Größe der einzelnen Mengen L_{s_i}. Es werden drei Fälle unterschieden:

Fall 1: $|L_{s_i}| = 0$

L_{s_i} ist demzufolge leer, der Zustand s_i hat keine Loops. Daraus ergibt sich für f:

$$f = \frac{1}{1 - \sum_{l \in L_{s_i}} p_l} = \frac{1}{1 - 0} = 1$$

Es ist offensichtlich, dass der Loop-Faktor f in diesem Fall korrekt ist.

Fall 2: $|L_{s_i}| = 1$

Der Zustand s_i verfügt in diesem Fall nur über einen einzigen Loop l. Daraus ergibt sich:

$$\begin{aligned} f &= 1 + p_l + p_l^2 + \ldots \\ &= \sum_{n=0}^{\infty} p_l^n \text{ (Geometrische Reihe)} \\ &= \frac{1}{1 - p_l} \end{aligned}$$

Dies entspricht der Summe der Wahrscheinlichkeitsmassen von beliebig vielen Abwicklungen des Loops l. Damit ist die Gültigkeit von f für den zweiten Fall gezeigt.

Fall 3: $|L_{s_i}| > 1$

In diesem Fall verfügt der Zustand s_i über mehrere Loops. In die Wahrscheinlichkeitsmasse des erweiterten Pfades ω_e müssen alle möglichen Kombinationen der Loops aus L_{s_i} einfließen.

Für alle Loops $l_1, \ldots, l_m \in L_{s_i}$ gilt:

$$\begin{aligned} f &= 1 + (p_{l_1} + p_{l_2} + \cdots + p_{l_m}) + \\ &\quad (p_{l_1}^2 + p_{l_1}p_{l_2} + p_{l_2}p_{l_1} + p_{l_2}^2 + p_{l_1}p_{l_3} + \cdots + p_{l_m}^2) + \cdots \\ &= \sum_{n=0}^{\infty} \sum_{\substack{j_1, j_2, \ldots, j_m \in \mathbb{N} \\ j_1 + j_2 + \cdots j_m = n}} \binom{n}{j_1 j_2 \cdots j_m} p_{l_1}^{j_1} p_{l_2}^{j_2} \cdots p_{l_m}^{j_m} \\ &= \sum_{n=0}^{\infty} (p_{l_1} + p_{l_2} + \cdots + p_{l_m})^n \\ &= \frac{1}{1 - (p_{l_1} + p_{l_2} + \cdots + p_{l_m})} \text{ (Geometrische Reihe)} \\ &= \frac{1}{1 - \sum_{l \in L_{s_i}} p_l} \end{aligned}$$

Damit ist die Korrektheit von f auch im dritten Fall gezeigt.

Die Kombinationen dieser drei Fälle decken alle denkbar möglichen Situationen für einen erweiterten Pfad ω_e ab. Da sich die Loop-Faktoren der jeweiligen Zustände s_i einfach aufmultiplizieren lassen, ist mit der Korrektheit der drei Fälle auch die Korrektheit des ganzen Satzes 3.1 gezeigt.

Mit Hilfe der Looperkennung und der erweiterten Pfade lässt sich die Erzeugung von Gegenbeispielen beschleunigen, wie sich am Beispiel 3.4 erkennen lässt.

Beispiel 3.4 (SBMC mit Looperkennung und erweiterten Pfaden)
Es sei $\mathcal{D} := (S, P, AP, \mathcal{L})$ die DTMC aus Abbildung 3.2 auf Seite 34. Weiterhin sollen die gleichen Bedingungen gelten wie in Beispiel 3.2: $\Phi := P_{\leq 0.9}[\top \,\mathcal{U}\, b]$ und $k_{max} := 9$. Startwert ist wieder $k := 2$:

$k := 2:$ $\omega := (s_0, s_1, s_3),\ prob(\omega) = 0.4$
$k := 3:$ –
$k := 4:$ $\omega' := (s_0, s_1, s_2, s_1, s_3),\ prob(\omega') = 0.24 \Rightarrow$ *Loop* $l := (s_1, s_2, s_1)$
$\Rightarrow$ *Erweiterter Pfad* $\omega_e := (\omega, \{l\}),\ prob(\omega_e) = 0.4 \cdot \frac{1}{1-0.6} = 1$

Bei Iterationstiefe $k := 4$ wird der erweiterte Pfad ω_e gefunden. Er setzt sich zusammen aus dem Basispfad $\omega_b := \omega = (s_0, s_1, s_3)$ und dem Loop $l := (s_1, s_2, s_1)$.
Die Wahrscheinlichkeitsmasse von ω_e beträgt 1. Damit wurde die Wahrscheinlichkeitsgrenze $p := 0.9$ überschritten und das Verfahren wird beendet. $E := \{\omega_e\}$ ist ein gültiges Gegenbeispiel.

Bei Beispiel 3.2 wurde der SBMC-Algorithmus durch Erreichen der maximalen Iterationstiefe $k_{max} := 9$ beendet und kein Gegenbeispiel errechnet, obwohl vier Zeugen gefunden worden waren. Mit Hilfe der Looperkennung reicht es in Beispiel 3.4 aus, bis zur Iterationstiefe $k := 4$ vorzudringen. Es wird ein Gegenbeispiel gefunden, das aus einem einzelnen erweiterten Pfad besteht. Das Gegenbeispiel in diesem Fall enthält folglich weniger Zeugen als das unvollständige Ergebnis in Beispiel 3.2.

Looperkennung und erweiterte Pfade beschleunigen also nicht nur das Verfahren, sondern ermöglichen auch kompaktere Gegenbeispiele. Letztere haben den Vorteil, dass sie für den Anwender leichter nachvollziehbar sind: Der Basispfad ω_b eines erweiterten Pfades führt bereits ohne die Loops zu einem kritischen Zustand. In der durch ω_b dargestellten Folge von Zustandsübergängen kommt es also bereits zur Verletzung der Sicherheitseigenschaft Φ. Ein System-Entwickler kann sich dementsprechend auf die Analyse von ω_b konzentrieren, ohne erst die Gemeinsamkeiten einer großen Anzahl von Zeugen, die alle durch Abwickeln der gleichen Loops entstanden sind, erarbeiten zu müssen.

Es wäre jedoch falsch, die Loops zugunsten der Basispfade komplett außer Acht zu lassen. Zum einen werden sie gebraucht, um durch die erweiterten Pfade die benötigte Wahrscheinlichkeitsmasse zu erreichen, zum anderen können auch sie Informationen zum fehlerhaften Verhalten eines Systems enthalten. Außerdem können sie auch benutzt werden, um das SBMC-Verfahren noch mehr zu beschleunigen: Durch Vorausberechnen von Pfaden, die daraufhin vom SAT-Solver ausgeschlossen werden.

3.3 Im Voraus ausgeschlossene Pfade

Die in Beispiel 3.2 dargestellte Situation ist durch Looperkennung und erweiterte Pfade noch nicht behoben. Loops können jetzt zwar erkannt werden, der Solver würde aber nach wie vor Pfade finden, die durch mehrfaches Abwickeln von bereits gefundenen Loops entstehen. Um dies zu verhindern, werden die entsprechenden Pfade im Voraus berechnet und ausgeschlossen. Dadurch wird die Anzahl an Aufrufen des SAT-Solvers vermindert, was zu einer Verringerung der benötigten Rechenzeit führt.

Das um die Looperkennung erweiterte Verfahren geht bisher folgendermaßen vor: Sobald ein neuer Pfad ω gefunden wurde, wird dieser auf Loops hin untersucht. Wird ein Loop l gefunden, so wird als Nächstes überprüft, ob auf dem zugehörigen Basispfad ω_b bereits ein Loop gefunden wurde. Ist dies der Fall, so existiert bereits ein erweiterter Pfad ω_e, der nun um den Loop l ergänzt wird. Existiert noch kein erweiterter Pfad, so wird ω_e jetzt berechnet. Der gefundene Pfad ω wird, wie in Kapitel 3.1.3 auf Seite 38 vorgestellt, vom SAT-Solver ausgeschlossen.

Dies setzt sich so lange fort, bis der SAT-Solver für die aktuelle Iterationstiefe keine neuen Pfade mehr findet. Wie im ursprünglichen Verfahren wird nun die nächst höhere Iterationstiefe betrachtet. Die KNF ψ wird neu berechnet. Bevor allerdings der SAT-Solver auf ψ aufgerufen wird, erhält er noch eine Reihe von weiteren Klauseln. Diese stehen für alle Pfade der Länge der neuen Iterationstiefe k, die durch Abwickeln der bis dahin gefundenen Loops entstehen könnten.

Um diese Klauseln zu erhalten, muss zunächst berechnet werden, wie oft die Loops eines erweiterten Pfades ω_e abgewickelt werden müssen, um aus ihnen und dem Basispfad ω_b einen Pfad der Länge k zu erhalten:
Es seien $l_1, \ldots, l_m$ die Loops von ω_e. Um nun die Anzahl der benötigten Abwicklungen zu erreichen, muss folgende Gleichung gelöst werden:

$$|l_1| \cdot r_1 + |l_2| \cdot r_2 + \cdots + |l_m| \cdot r_m = k - |\omega_b|,\ r_1, \ldots, r_m \in \mathbb{N} \tag{3.5}$$

Eine Lösung besteht aus passenden Werten für die Faktoren r_1 bis r_m. Sie geben an, wie oft der jeweilige Loop abgewickelt werden muss. Dabei kann es durchaus mehrere Lösungen geben.

Gleichung 3.5 kann mit einem einfachen rekursiven Algorithmus gelöst werden. Eine Möglichkeit dafür zeigt Algorithmus 3. $R := (r_1, \ldots, r_m)$ ist dabei ein Vektor, der die

Algorithm 3 Algorithmus zum Lösen der Loop-Gleichung

```
1:  // initial values:
2:  R := (r_1, ..., r_m) := (0, ..., 0);
3:  S := {};
4:  dist := k - |ω_b|;
5:  current loop index i = 1;
6:  current distance d := 0;

7:  function EQSolve(i, d)
8:      if (i ≡ m) then
9:          if (d ≡ dist) then
10:             S := S ∪ R;
11:         end if
12:         return;
13:     end if
14:     d_max := ⌊(dist - d) / |l_i|⌋
15:     for j = 0, ..., d_max do
16:         R[i] = j;
17:         EQSolve(i + 1, d + j · |l_i|);
18:     end for
19: end function
```

Faktoren r_1 bis r_m enthält. Zu Beginn sind die Werte in R alle auf 0 gesetzt. S ist die Menge der gefundenen Lösungen, anfangs ist S leer. *dist* ist die durch die Loops zu erreichende Länge. Der Wert d gibt die bis dahin erreichte Länge, zu Beginn ebenfalls 0, an. i verweist auf den Index des aktuell zu betrachtenden Loops und wird mit 1 initialisiert.

Die Funktion EQSolve wird mit den Werten für i und d aufgerufen. Zuerst wird überprüft, ob bereits alle Loops betrachtet wurden („**if** $(i \equiv m)$“ in Zeile 8). Ist dies der Fall, so wird nun geprüft, ob eine Lösung für die Gleichung vorliegt („**if** $(d \equiv dist)$“). Trifft dies ebenfalls zu, so wird die aktuelle Lösung zu S hinzugefügt. Unabhängig

davon, ob eine Lösung vorliegt oder nicht, beendet sich die Funktion, wenn alle Loops betrachtet wurden.

Nach der Überprüfung wird d_{max} berechnet, dies ist die maximale Abwicklungstiefe für den aktuell betrachteten Loop l_i. In einer Schleife werden alle möglichen Abwicklungstiefen j für l_i durchlaufen (siehe Zeile 15 in Algorithmus 3). Innerhalb der Schleife wird r_i auf die aktuelle Abwicklungstiefe von l_i gesetzt. Die Funktion EQSolve wird nun rekursiv mit dem nächsten Loop l_{i+1} und der durch das j-fache Abwickeln von l_i erreichten aktuellen Länge d aufgerufen.

Sollte Algorithmus 3 keine gültigen Lösungen für Gleichung 3.5 liefern, so gibt es keinen Pfad der Länge k, der aus dem Basispfad ω_b durch Abwickeln der dazu gehörenden Loops entstehen könnte. Gibt es jedoch eine oder mehrere Lösungen, so müssen für diese nun die entsprechenden Klauseln gebildet werden.

Zu diesem Zweck wird für jede von Algorithmus 3 gefundene Lösung der Basispfad ω_b durchlaufen. Die Zustandsvariablen werden negiert und zu einer Klausel c hinzugefügt, ähnlich wie beim Ausschließen von gefundenen Pfaden im ursprünglichen Verfahren (siehe Kapitel 3.1.3, Seite 38). Wird ein Zustand erreicht, welcher der Startpunkt von Loops ist, so werden nun zwei Fälle unterschieden: Entweder hat der Zustand nur einen oder mindestens zwei Loops.

Hat der Zustand nur einen Loop l, so reicht es aus, den Loop so oft abzuwickeln, wie es die aktuell betrachtete Lösung von Algorithmus 3 vorgibt. Die Zustandsvariablen von l werden mit einem entsprechenden Index-Shift versehen und negiert in die Klausel c eingefügt. Das folgende Beispiel soll diesen Vorgang illustrieren:

Beispiel 3.5 (Ausschließen eines einzelnen Loops)
Für die DTMC $\mathcal{D}$ aus Abbildung 2.6 auf Seite 25 und die Sicherheitsbedingung $\Phi := P_{\leq p}[\top \,\mathcal{U}\, b]$ wurde folgender erweiterter Pfad gefunden:

$$\omega_e := ((s_0, s_1, s_4), \{(s_1, s_2, s_1)\})$$

Nun wird die Iterationstiefe $k := 6$ betrachtet. Aus ω_e lässt sich dadurch ein Pfad der Länge 6 generieren, dass der Loop $l := (s_1, s_2, s_1)$ zweimal abgewickelt wird. Um diesen Pfad auszuschließen, wird ω_b durchlaufen und negiert in die Klausel c eingefügt, bis der Zustand s_1 erreicht wird. Hier beginnt der Loop l. Der Loop wird nun zweimal durchlaufen, die Zustände von l werden negiert und ebenfalls in c eingefügt. Danach wird der Rest von ω_b durchlaufen und zur Klausel c hinzugefügt.
Daraus ergibt sich folgende Klausel:

$$c := (\overline{s_{0,0}} \vee \overline{s_{1,1}} \vee \overline{s_{2,2}} \vee \overline{s_{1,3}} \vee \overline{s_{2,4}} \vee \overline{s_{1,5}} \vee \overline{s_{4,6}})$$

Zugunsten eines besseren Überblicks wurden hier statt mehrerer Zustandsvariablen für jeden Zustand s_i die Zustandsbezeichnungen verwendet. Der zweite Index symbolisiert den Index-Shift für den jeweiligen Zeitrahmen.

Verfügt ein Zustand über mehrere Loops, so erfordert es etwas mehr Aufwand, diese auszuschließen. Die Loops müssen in allen möglichen Reihenfolgen, welche die aktuell betrachtete Lösung von Algorithmus 3 zulässt, ausgeschlossen werden.

Eine – zumindest auf den ersten Blick – einfache Art, dies zu bewerkstelligen, wäre, alle Pfade für alle möglichen Reihenfolgen auszurechnen und sie auszuschließen. Angenommen der erweiterte Pfad $\omega_e := (\omega_b, L)$ verfügt über Zustände mit mehreren Loops. $W_{g,k}$ sei die Menge der Pfade, die aus ω_e für eine Lösung g der Gleichung 3.5 für die Länge k entstehen. Jeder Pfad $\omega \in W_{g,k}$ besteht aus Literalen für den Basispfad $\omega_b := (s_0, \ldots, s_n)$ und den Literalen für die Loops aus L. Es sei C_b nun die Menge der Literale des Basispfads und es bezeichne $C^{\omega}_{L_{s_i}}$ die Menge der Literale für die Loops eines Zustands s_i aus ω. Dann lassen sich die Klauseln zum Ausschließen der Pfade in $W_{g,k}$ folgendermaßen errechnen:

$$\left\{C_b\right\} \times \left\{C^{\omega}_{L_{s_0}} \mid \omega \in W_{g,k}\right\} \times \cdots \times \left\{C^{\omega}_{L_{s_{n-1}}} \mid \omega \in W_{g,k}\right\}. \tag{3.6}$$

Der Endzustand s_n ist ein kritischer Zustand und hat keine Loops. Dies wird durch das Entfernen aller ausgehenden Kanten der kritischen Zustände nach der Erzeugung der Entscheidungsdiagramme sichergestellt[6]. Dementprechend gibt es keine Menge $C_{L_{sn}}$.

Diese einfache Methode zum Ausschließen von mehreren Loops führt allerdings zu einer hohen Anzahl an Klauseln und Literalen. Um diese Anzahl zu berechnen, bezeichne nun $l^i_1, \ldots, l^i_{m_i}$ die einzelnen Loops in L_{s_i}. Die Funktion $g_k : L \to \mathbb{N}$ gibt eine Lösung von Gleichung 3.5 für ω_e und Länge k an. $g_k(l^i_j)$ liefert die Anzahl, wie oft der Loop l^i_j des Zustands s_i abgewickelt werden soll. $g(L_{s_i})$ ist die Summe der Lösungen aller Loops aus L_{s_i}. Die Anzahl an Literalen, die pro Zustand benötigt werden, wird mit *lps* bezeichnet.
Die Anzahl an Klauseln und Literalen zum Ausschließen des Pfades ω_e für eine Lösung g_k ergibt sich dann als:

$$\begin{aligned} \#Klauseln &= \prod_{i=0}^{n-1} \binom{g_k(L_{s_i})}{g_k(l^i_1) \cdots g_k(l^i_{m_i})} \\ \#Literale &= \#Klauseln \cdot (k+1) \cdot lps\,. \end{aligned}$$

Die Formel für die Berechnung der Anzahl an Klauseln besteht aus dem Produkt der Multinomialkoeffizienten für die Lösung g_k. Mit Hilfe des Multinomialkoeffizienten wird die Anzahl an möglichen Reihenfolgen, in denen die Loops eines Zustands s_i auftreten können, berechnet.
Die Anzahl an Literalen setzt sich zusammen aus der Anzahl an Klauseln, der Anzahl an Literalen pro Zustand (*lps*) und der Anzahl an Zuständen des Pfades. Letztere beträgt für Iterationstiefe k immer $k+1$.

Die durch die einfache Methode erstellten Klauseln enthalten redundante Informationen: Wie bereits erwähnt, enthält jede dieser Klauseln zumindest den Basispfad, wenn nicht sogar einzelne ausgeschlossene Loops. Sie haben also einen gemeinsamen Teil und Bereiche, in denen sie sich unterscheiden. In diesem Fall kann der gemeinsame Teil ausfaktorisiert werden:

$$(C \vee D_1) \wedge (C \vee D_2) \Leftrightarrow (C \vee h) \wedge \left(\overline{h} \vee D_1\right) \wedge \left(\overline{h} \vee D_2\right) \tag{3.7}$$

[6]Siehe dazu Kapitel 3.1.1, Seite 35.

C repräsentiert den gemeinsamen Teil der Klauseln, D_1 und D_2 entsprechen den Bereichen, in denen sich die Klauseln unterscheiden. Um C auszufaktorisieren, wird eine neue Klausel gebildet, die aus C und einer Hilfsvariablen h besteht. h wird in negierter Form anstelle von C in die ursprünglichen Klauseln eingesetzt. Der dadurch entstehende Ausdruck ist erfüllbarkeitsäquivalent zum vorherigen.

Konkret angewendet auf einen erweiterten Pfad ω_e bedeutet das Folgendes: Der Basispfad ω_b wird durchschritten und ausgeschlossen – inklusive möglicher einzelner Loops –, bis ein Zustand s_i erreicht wird, der zwei oder mehr Loops besitzt. Es wird eine Hilfsvariable h_{s_i} in die bisherige Pfadklausel c eingefügt. Anschließend wird für jede mögliche Reihenfolge r_j der Loops von s_i eine Klausel c_{r_j,s_i} gebildet. Diese Loop-Klauseln c_{r_j,s_i} enthalten, entsprechend der Gleichung 3.7, nicht nur die negierten Zustände der Loops, sondern auch die negierte Hilfsvariable h_{s_i}. Nachdem die Loop-Klauseln erzeugt wurden, wird das Durchschreiten von ω_b fortgesetzt. Die übrigen Zustände des Pfades werden mit entsprechendem Index-Shift in die Pfadklausel c eingefügt. Sollte es mehrere Zustände in ω_b geben, die über mehr als einen Loop verfügen, so wird dieser Vorgang wiederholt, sobald der nächste dieser Zustände erreicht wird.

Beispiel 3.6 demonstriert das Ausschließen mehrerer Loops an einem Zustand.

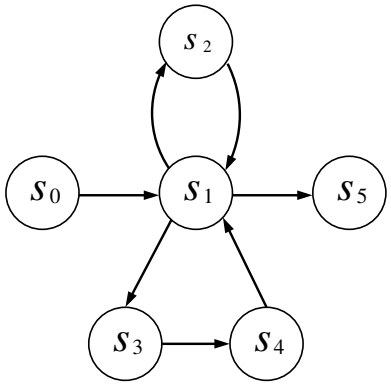

Abbildung 3.5: Graphische Darstellung des erweiterten Pfades aus Beispiel 3.6

Beispiel 3.6 (Ausschließen von mehreren Loops)
Es sei $\omega_e := ((s_0, s_1, s_5), \{(s_1, s_2, s_1), (s_1, s_3, s_4, s_1)\})$ *ein erweiterter Pfad in einer DTMC* $\mathcal{D}$. ω_e *verfügt über zwei Loops im Zustand* s_1*:* $l_1 := (s_1, s_2, s_1)$ *und* $l_2 := (s_1, s_3, s_4, s_1)$. *Abbildung 3.5 ist eine graphische Repräsentation von* ω_e.
Es soll nun die Iterationstiefe $k := 7$ *betrachtet werden. Wenn sowohl* l_1 *als auch* l_2 *jeweils einmal abgewickelt werden, erhält man aus dem erweiterten Pfad* ω_e *einen Pfad der Länge* 7.
Zunächst wird $\omega_b := (s_0, s_1, s_5)$ *durchschritten und negiert in eine Klausel* c *eingefügt. Wenn der Zustand* s_1 *erreicht wird, wird eine Hilfsvariable in* c *eingefügt:*

$$c := (\overline{s_{0,0}} \vee \overline{s_{1,1}} \vee h)$$

Wie in Beispiel 3.5 wurden hier statt der Zustandsvariablen die Zustandsbezeichnungen verwendet. Der zweite Index steht wieder für den jeweiligen Zeitrahmen.

Nun werden die Loop-Klauseln gebildet. Es gibt zwei Möglichkeiten, die Loops zu durchlaufen: Zuerst l_1 und dann l_2 oder zuerst l_2 und dann l_1. Entsprechend gibt es zwei Loop-Klauseln:

$$\begin{aligned} c_l &:= \left(s_{2,2} \vee s_{1,3} \vee s_{3,4} \vee s_{4,5} \vee s_{1,6} \vee \overline{h}\right) \\ c_l' &:= \left(s_{3,2} \vee s_{4,3} \vee s_{1,4} \vee s_{2,5} \vee s_{1,6} \vee \overline{h}\right) \end{aligned}$$

Jede der beiden Loop-Klauseln enthält die Hilfsvariable h in negierter Form.
Zum Schluss wird noch das Ende des Basispfads ω_b – in diesem Fall der Zustand s_5 – in die Klausel c eingefügt:

$$c := (\overline{s_{0,0}} \vee \overline{s_{1,1}} \vee h \vee \overline{s_{5,7}})$$

Mit Hilfe dieser drei Klauseln lassen sich nun alle Möglichkeiten, die aus dem erweiterten Pfad ω_e für die Iterationstiefe $k := 7$ gebildet werden könnten, ausschließen.

Die Loop-Klauseln werden mit Hilfe eines rekursiven Algorithmus, der dem Algorithmus 3 zur Berechnung der Abwicklungstiefen ähnelt, berechnet. Allgemein lässt sich die Menge der auf diese Weise erstellten Klauseln durch folgende Formel beschreiben:

$$\left\{C_b \dot\cup \left\{h_{s_0}, \ldots, h_{s_{n-1}}\right\}\right\} \dot\cup \bigcup_{i=0}^{n-1} \left\{C^{\omega}_{L_{s_i}} \dot\cup \left\{\overline{h_{s_i}}\right\} \mid \omega \in W_{g,k}\right\} \tag{3.8}$$

Die Variablen h_{s_i} sind die Hilfsvariablen zum Abwickeln der Loops eines Zustands s_i.

Verglichen mit den expliziten Pfadklauseln können durch die Ausfaktorisierung erheblich Literale eingespart werden. Betrachten wir beispielsweise noch einmal den erweiterten Pfad $\omega_e := ((s_0, s_1, s_5), \{(s_1, s_2, s_1), (s_1, s_3, s_4, s_1)\})$ aus Abbildung 3.5, diesmal für die Iterationstiefe $k := 23$. Eine Möglichkeit, die Länge 23 zu erreichen, ist, die Schleife $l_1 := (s_1, s_2, s_1)$ dreimal und die Schleife $l_2 := (s_1, s_3, s_4, s_1)$ fünfmal zu durchlaufen. Dafür gibt es insgesamt $\binom{8}{3\,5} = 56$ Möglichkeiten. Da man zum Kodieren der sechs verschiedenen Zustände von ω_e mindestens drei Bits beziehungsweise Literale benötigt, hätte man im Falle der expliziten Pfad-Berechnung 56 Klauseln zu je $24 \cdot 3 = 72$ Literalen. Das wären insgesamt 4 032 Literale.
Wird stattdessen der Basispfad ausfaktorisiert, sind erheblich weniger Literale nötig: Wir erhalten eine Klausel mit $3 \cdot 3 + 1 = 10$ Literalen für den Basispfad. Die 56 Loop-Klauseln enthalten nun $21 \cdot 3 + 1 = 64$ Literale. Das ergibt insgesamt 3 594 Literale in 57 Klauseln. Es können in diesem Fall also rund 10% der Literale eingespart werden.

In vielen Fällen wird durch die Ausfaktorisierung nicht nur die Anzahl an Literalen, sondern auch die Anzahl an Klauseln verringert. Dies ist in der Regel immer dann der Fall, wenn es auf einem Pfad mehrere Zustände gibt, die über mehr als einen Loop verfügen. Allgemein lässt sich die Anzahl an Klauseln und Literalen zum Ausschließen

von Loops durch Ausfaktorisierung mit folgenden Formeln abschätzen:

$$\begin{aligned} \#Klauseln &\leq 1 + \sum_{i=0}^{n-1} \binom{g_k(L_{s_i})}{g_k(l_1^i) \cdots g_k(l_{m_i}^i)} \\ \#Literale &\leq (|\omega_b|+1) \cdot lps + |\omega_b| + \sum_{i=0}^{n-1} \left(\binom{g_k(L_{s_i})}{g_k(l_1^i) \cdots g_k(l_{m_i}^i)} \cdot \left(1 + lps \cdot \sum_{l \in L_{s_i}} |l| \cdot g(l) \right) \right) \end{aligned}$$

Die Anzahl an Klauseln setzt sich zusammen aus einer Klausel für den Basispfad ω_b und der Summe über alle Möglichkeiten zum Ausschließen der Loops eines Zustands. Diese Möglichkeiten lassen sich wieder durch einen Multinomialkoeffizienten berechnen. Im Gegensatz zu dem einfachen Verfahren müssen die verschiedenen Möglichkeiten jedoch nicht aufmultipiziert, sondern aufaddiert werden, da es für jeden Zustand s_i eine separate Menge von Loop-Klauseln gibt.
Die Anzahl an Literalen besteht aus den Literalen für den Basispfad ω_b, der Anzahl an eingeführten Hilfsvariablen und der Summe der Literale der einzelnen Loop-Klauseln. Diese Summe setzt sich zusammen aus dem Produkt der Anzahl an Möglichkeiten zum Abwickeln der Loops – erneut berechnet durch einen Multinomialkoeffizienten – und der Anzahl an Literalen pro Loop-Klausel. Letztere ergibt sich aus der in die Loop-Klausel eingeführten Hilfsvariable und der Summe über die Zustände der Loops eines Zustands s_i.

Durch das Ausschließen von im Voraus berechneten Pfaden ergeben sich mehrere Vorteile: Zum einen verringert sich die Rechenzeit, da der SAT-Solver nicht mehr so oft aufgerufen werden muss. Zum anderen vereinfacht sich die Looperkennung, da sich auf jedem neu gefundenen Pfad maximal nur ein neuer Loop befinden kann. Wären es mehr, so hätte jeder dieser Loops bereits allein auf einer niedrigeren Iterationstiefe auftreten müssen. Das würde aber bedeuten, dass die Loops auch früher erkannt worden wären, was zur Folge hätte, dass der Pfad mitsamt den Loops ausgeschlossen worden wäre. Es kann also kein Pfad mit mehreren neuen Loops gefunden werden.

Bevor nun das erweiterte SBMC-Verfahren noch einmal in einem Algorithmus zusammengefasst wird, müssen noch zwei Fälle von Loops erwähnt werden, die bisher ausgespart wurden: Ineinander verschachtelte und sich überschneidende Loops.

Die in dieser Arbeit vorgestellten Methoden zum Erkennen und Ausschließen von Loops erlauben keine explizite Behandlung von ineinander verschachtelten oder sich überschneidenden Loops. Stattdessen werden diese implizit repräsentiert.

Befassen wir uns zunächst mit verschachtelten Loops. Als solche werden Loops bezeichnet, die innerhalb einer DTMC vom Startzustand s_0 nur über einen anderen Loop l erreichbar sind. Solche verschachtelten Loops können aufgrund der hier zur Looperkennung eingesetzten Definition 3.1[7] nicht gefunden werden. Der früheste Zustand auf dem gefundenen Pfad ω, der ein zweites Mal vorkommt, ist immer ein Teil von l. Allerdings würde ein solcher „verborgener“ Loop in Form von größeren Loops in den erweiterten Pfad ω_e einfließen.

[7] Siehe dazu Seite 41.

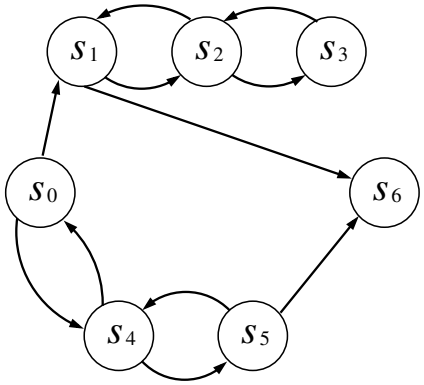

Abbildung 3.6: DTMC zu Beispiel 3.7 und Beispiel 3.8

Beispiel 3.7 (Verschachtelte Loops)
Es sei $\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ die in Abbildung 3.6 dargestellte DTMC. s_6 ist ein kritischer Zustand.
Betrachten wir die oberen Zustände s_1, s_2 und s_3: Hier liegen zwei ineinander verschachtelte Loops vor: $l_1 := (s_1, s_2, s_1)$ und $l_2 := (s_2, s_3, s_2)$. l_2 kann allerdings nicht vom erweiterten SBMC-Verfahren als eigenständiger Loop gefunden werden, da er nur über den Loop l_1 erreichbar ist.
l_2 wird frühestens für Iterationstiefe $k := 6$ in Form eines größeren Loops entdeckt: $l_3 := (s_1, s_2, s_3, s_2, s_1)$. l_2 ist gewissermaßen in l_1 „eingebettet“ worden und ist damit in l_3 implizit enthalten. l_3 wird zu dem erweiterten Pfad $\omega_e := ((s_0, s_1, s_6), \{l_1\})$ hinzugefügt.
Im Anschluss daran würden für die kommenden Iterationstiefen k alle möglichen Kombinationen von l_1 und l_3 ausgeschlossen werden. Kombinationen von l_1 und l_2, die sich nicht daraus bilden lassen, werden wieder als neuer Loop gefunden, zum Beispiel der Loop $l_4 := (s_1, s_2, s_3, s_2, s_3, s_2, s_1)$.

Eine andere Situation liegt bei sich überschneidenden Loops vor. Dies sind Loops, die vom SBMC-Verfahren erkannt werden, sich allerdings einen oder mehrere Zustände teilen. Diese Überschneidungen haben zur Folge, dass die Grundrichtung eines Pfades ω gewissermaßen wechseln kann. Statt auf einen kritischen Zustand „zuzusteuern“, verläuft zumindest eine Teilfolge von ω in Richtung Startzustand. Das SBMC-Verfahren ist nicht in der Lage, solche Pfade im Voraus zu berechnen, da nicht von einem Loop direkt in einen anderen gewechselt werden kann. Allerdings wird auch innerhalb eines solchen Pfades ω ein größerer Loop erkannt, der diese Möglichkeit implizit enthält.

Beispiel 3.8 (Sich überschneidende Loops)
$\mathcal{D} := (S, s_0, T, AP, \mathcal{L})$ ist die DTMC aus Abbildung 3.6, wie im vorangehenden Beispiel 3.7 ist s_6 ein kritischer Zustand.
Versuchen wir über die unteren Zustände von s_0 zu s_6 zu gelangen, so finden wir ebenfalls zwei Loops: $l_1 := (s_0, s_4, s_0)$ und $l_2 := (s_4, s_5, s_4)$. Beide Loops beinhalten den Zustand s_4. Durch diese Überschneidung kann zwischen den beiden Loops hin und

her gewechselt werden, wie zum Beispiel im folgenden Pfad:

$$\omega := (s_0, s_4, s_5, s_4, s_0, s_4, s_5, s_6)$$

Dieser Pfad kann nicht im Voraus berechnet werden, auch wenn die Loops l_1 und l_2 bekannt sind. Allerdings wird in ω ein neuer Loop erkannt: $l_3 := (s_0, s_4, s_5, s_4, s_0)$
l_3 ermöglicht es, einige weitere Kombinationen von l_1 und l_2 im Voraus zu berechnen und auszuschließen. Andere Kombinationen werden durch weitere Loops erkannt.

Zwar können mit dem erweiterten SBMC-Verfahren nie alle Kombinationsmöglichkeiten von ineinander verschachtelten oder sich überschneidenden Loops erfasst werden, aber mit zunehmender Iterationstiefe werden immer mehr Möglichkeiten ausgeschlossen.

Algorithmus 4 zeigt das erweiterte SBMC-Verfahren in Pseudocode. Der Aufruf der Funktion ESBMCSolve erfolgt mit den gleichen Parametern wie der Aufruf der Funktion SBMCSolve in Algorithmus 2: Es wird eine DTMC $\mathcal{D}$ in Form von Entscheidungsdiagrammen, eine Sicherheitsbedingung Φ, eine Wahrscheinlichkeitsgrenze p und eine maximale Iterationstiefe k_{max} übergeben.

Auch das Preprocessing ist identisch mit dem aus Algorithmus 2: Die DTMC $\mathcal{D}$ wird reduziert und die minimale Distanz $dist_{min}$ berechnet. Die Initialisierung ist ebenfalls nahezu gleich: k wird auf $dist_{min}$ gesetzt und die KNF ψ wird berechnet. Die Variable *prob* für die erreichte Gesamt-Wahrscheinlichkeitsmasse wird mit 0 initialisiert, die Menge *Paths* wird für den Anfang auf die leere Menge gesetzt. Es kommt die Menge *Loops* hinzu, die später die gefundenen Loops enthalten soll. *Loops* ist anfangs ebenfalls leer.

Das erweiterte SBMC-Verfahren beginnt in Zeile 10. Der SAT-Solver wird mit der KNF ψ aufgerufen. Falls eine Lösung gefunden wird, so wird diese in einen Pfad ω umgewandelt. Danach wird ω analysiert: Enthält ω einen Loop l, so müssen die Menge *Paths* und die Wahrscheinlichkeitsmasse *prob* aktualisiert werden. Für *Paths* bedeutet das, dass nach einem erweiterten Pfad $\omega_e \in$ *Paths* gesucht wird, dessen Basispfad mit dem von ω identisch ist. Existiert ein solcher erweiterter Pfad ω_e, so wird er um den Loop l ergänzt. Existiert ω_e noch nicht, so wird er nun generiert. Der Basispfad ω_b als einzelnes Element wird dabei aus *Paths* entfernt. Da ω_b bei einer früheren Iteration gefunden worden sein muss, muss er in *Paths* enthalten sein, sei es innerhalb eines erweiterten Pfades oder als einzelner Pfad. *Paths* enthält im erweiterten SBMC-Verfahren ausschließlich Zeugen in Form von Basispfaden und erweiterten Pfaden.
Nachdem *Paths* aktualisiert wurde, wird *prob* anhand von *Paths* neu berechnet. Der Loop l wird zur Menge *Loops* hinzugefügt.

Enthält der gefundene Pfad ω keinen Loop (siehe Zeile 18 von Algorithmus 4), so handelt es sich bei ω um einen neuen Basispfad. Er wird zur Menge *Paths* hinzugefügt. Die Wahrscheinlichkeitsmasse von ω wird berechnet und zu *prob* hinzuaddiert.

Sowohl in dem Fall, dass ω einen Loop enthält, als auch in dem, dass ω keinen Loop enthält, wird ω auf die aus dem ursprünglichen Verfahren bekannte Weise[8], vom SAT-Solver ausgeschlossen (siehe Zeile 22). Ein neuer Schleifen-Durchlauf beginnt.

[8]Das Ausschließen von Pfaden wurde auf Seite 38 in Kapitel 3.1.3 beschrieben.

Algorithm 4 Erweiterter SBMC-Algorithmus

```
1: function ESBMCSolve(𝒟, Φ, p, k_max)
2:     // preprocessing:
3:     reduce 𝒟;
4:     compute dist_min;

5:     // initialization:
6:     compute initial CNF ψ for k := dist_min;
7:     prob := 0.0;
8:     Paths := {};
9:     Loops := {};

10:    // extended SBMC procedure:
11:    while (prob ≤ p) && (k ≤ k_max) do
12:        if SATSolve(ψ) then
13:            get current solution for ψ and generate path ω;
14:            analyse ω;
15:            if ω has a loop l then
16:                update Paths;
17:                recompute prob;
18:                Loops := Loops ∪ l;
19:            else
20:                Paths := Paths ∪ ω;
21:                prob := prob + prob(ω);
22:            end if
23:            ExcludePath(ω);
24:        else
25:            restart SAT-Solver;
26:            update ψ to k := k + 1;
27:            for all ω_e ∈ Paths do
28:                ExcludeLoops(ω_e, k);
29:            end for
30:        end if
31:    end while

32:    if prob > p then
33:        print: "Counterexample found!";
34:    else
35:        print: "No counterexample found!";
36:    end if
37:    return prob, k, Paths, Loops
38: end function
```

Findet der SAT-Solver keine Lösung, so wird zuerst die Klausel-Datenbank des SAT-Solvers geleert, indem letzterer neu gestartet wird. Danach wird die Iterationstiefe um eins hochgesetzt: $k := k + 1$ und die KNF ψ aktualisiert. In einer Schleife werden nun alle erweiterten Pfade der Menge *Paths* durchlaufen. Die Funktion ExcludeLoops berechnet für jeden erweiterten Pfad ω_e alle möglichen Pfade, die durch Abwickeln der Loops von ω_e entstehen könnten. Die dabei entstehenden Loop- und Pfadklauseln werden direkt von der Funktion an den SAT-Solver übergeben. Danach beginnt ein neuer Durchlauf des SBMC-Verfahrens.

Wie beim ursprünglichen Verfahren wird das Verfahren so lange fortgetzt, bis entweder die Wahrscheinlichkeitsgrenze p überschritten oder die maximale Iterationstiefe k_{max} erreicht worden ist. Es wird überprüft, ob ein gültiges Gegenbeispiel vorliegt, und eine entsprechende Meldung ausgegeben. Die Menge *Paths*, welche die gefundenen Zeugen enthält, sowie die Menge *Loops*, die erreichte Wahrscheinlichkeitsmasse *prob* und die erreichte Iterationstiefe k werden zurückgegeben. Damit hat Algorithmus 4 die Suche nach einem Gegenbeispiel beendet.

3.4 Implementierung

In diesem Kapitel soll nun kurz auf die Implementierung eingegangen werden, die das SBMC-Verfahren mit allen bisher vorgestellten Optimierungen und Erweiterungen umsetzt. Da die verschiedenen Algorithmen zur Berechnung von Klauseln, Pfaden, Loops und Wahrscheinlichkeiten in den vorangehenden Unterkapiteln ausführlich beschrieben wurden, wird hier nicht noch einmal darauf eingegangen werden.

Als Programmiersprache wurde C++ gewählt. Der objektorientierte Ansatz von C++ erlaubt es, auf andere, bereits existierende Programme und Pakete zurückzugreifen. So wird für die Erzeugung und Verwaltung der BDDs und MTBDDs das CUDD-Paket (siehe [43]) eingesetzt. Die DTMC wird von dem Model Checker PRISM (siehe dazu [29]) in die benötigten Entscheidungsdiagramme umgewandelt. Die Entscheidungsdiagramme werden als XML-Datei an das eigentliche SBMC-Programm übergeben. PRISM wurde auch dazu verwendet, die Benchmarks, die für die Experimente in Kapitel 4 verwendet wurden, zu erzeugen. Als SAT-Solver kommt MiniSat (siehe [18]) zum Einsatz. Auch andere SAT-Solver ließen sich verwenden, einzige Bedingung ist, dass sich jederzeit weitere Klauseln einfügen lassen, damit Pfade und Loops ausgeschlossen werden können.

Der Aufbau des eigentlichen Programms wurde modular gehalten. Es gibt mehrere Klassen, denen unterschiedliche Aufgaben zukommen.

Die Klasse `BMCSolver` dient als Schnittstelle zwischen dem SBMC-Verfahren und dem SAT-Solver MiniSat. Über `BMCSolver` werden Klauseln an den SAT-Solver übergeben und Letzterer aufgerufen.

Aufgabe der Klasse `iterationManager` ist es, die Zustands- und Folgezustands- sowie die benötigten Hilfsvariablen zu generieren und zu verwalten. Hier werden die Index-Shifts für die Zeitrahmen errechnet sowie eine Abbildung erzeugt, die von den Variablen der verschiedenen Zeirahmen wieder auf die Ursprungs-Variablen verweist.

In der Klasse `ClauseCreator` erfolgt die Umwandlung der BDDs in boolesche Ausdrücke. Hier gibt es Funktionen, die einen BDD durchlaufen und entsprechend der in Kapitel 3.1.2 auf Seite 36 vorgestellten Methode Klauseln erzeugen. Die eigentliche KNF ψ wird innerhalb von `iterationManager` erzeugt und verwaltet.

Eine wichtige Rolle spielt die Klasse `PathAnalyzer`: Hier wurden die Looperkennung, die Wahrscheinlichkeitsberechnung sowie die Funktionen zum Ausschließen von Pfaden und Loops implementiert. Außerdem verwaltet `PathAnalyzer` auch die gefundenen Basispfade und die gefundenen Loops. Dazu werden zwei Vektoren angelegt: `PathTable` enthält die Basispfade und `LoopTable` enthält die Loops. Die Loops in `LoopTable` sind eindeutig. Wird der gleiche Loop auf mehreren Pfaden gefunden, so befindet er sich trotzdem nur einmal in der Loop-Tabelle. Um auf die verschiedenen Loops zu verweisen, werden die jeweiligen Indizes in der Loop-Tabelle genutzt.

Als Basis für die Darstellung von Loops und Pfaden dient die Datenstruktur `State`. Im Wesentlichen besteht diese Struktur aus zwei Integer-Vektoren: Der eine enthält die Zustandskodierungen, der andere enthält die Indizes der Loops, die in dem jeweiligen Zustand beginnen. Sollte es keine Loops geben, so ist dieser Loop-Vektor leer.

Die Datenstrukturen `LoopInfo` und `PathInfo` repräsentieren Loops und Basispfade. Eine Instanz von `LoopInfo` besteht, ebenso wie eine Instanz von `PathInfo`, aus einem Vektor von `States`, der eine Folge von Zuständen darstellt. Sowohl `LoopInfo` als auch `PathInfo` verfügen außerdem über einen Wert *prob*, der die Wahrscheinlichkeitsmasse des dargestellten Loops oder Pfades enthält. `PathInfo` enthält noch zwei weitere Werte: `loops` gibt an, wie viele Loops mit dem jeweiligen Basispfad verknüpft sind, `loop-prob` gibt die Wahrscheinlichkeit des aus dem Basispfad und den Loops entstehenden erweiterten Pfades an. Die Loops eines Pfades können gefunden werden, indem man den Pfad-Vektor durchläuft und die Loops aus den darin enthaltenen `States` ausliest. Ein direkter Zugriff auf die Loops eines erweiterten Pfades ist nicht möglich. Letzteres stellt jedoch kein Hindernis dar. Um Pfade im Voraus berechnen und ausschließen zu können, ist es ohnehin erforderlich, den jeweiligen Basispfad zu durchlaufen. Die Wahrscheinlichkeit eines erweiterten Pfades kann bei der Looperkennung berechnet beziehungsweise aktualisiert werden. Auch dabei ist ein Durchlaufen des Pfades erforderlich.
Es gibt also keine explizite Datenstruktur zur Darstellung von erweiterten Pfaden, nur Instanzen der Datenstruktur `PathInfo`, deren Werte `loop` und `loop-prob` gesetzt sind und deren Zustände auf Loops verweisen.

In einer `main`-Routine werden die verwendeten Pakete, Klassen und Datenstrukturen miteinander verknüpft. Die `main`-Routine selbst ist im Wesentlichen eine direkte Umsetzung von Algorithmus 4 auf Seite 54.

4 Experimente und Ergebnisse

In diesem Kapitel werden nun einige Modelle vorgestellt, auf die das SBMC-Verfahren angewendet werden kann. Zu jedem der Modelle werden darüber hinaus Ergebnisse präsentiert, die mit der in Kapitel 3.4 kurz vorgestellten Implementierung erzielt wurden.

Bei allen hier gezeigten Experimenten handelt es sich um Fallstudien, die auch auf der PRISM-Homepage zu finden sind [38]. Die Berechnungen wurden auf einem Dual Core AMD Opteron^{TM} Prozessor mit 2.6 GHz Taktfrequenz und 4 GB Hauptspeicher durchgeführt. Für jedes Experiment gab es ein Zeitlimit von zwei Stunden.

4.1 Würfel-Modelle

Einer der klassischen stochastischen Prozesse ist das Werfen einer Münze oder eines Würfels. Ist die Münze fair, so ist die Wahrscheinlichkeit für das Ergebnis *gleichverteilt*, das heißt, jedes Ergebnis eines Münzwurfs hat die gleiche Wahrscheinlichkeit. Die nun folgenden Modelle befassen sich mit dem Werfen einer Münze zur Simulation von einem Würfel beziehungszweise von zwei Würfeln. Die Modelle wurden von Knuth und Yao entwickelt [31].

Bei dem ersten hier vorgestellten Modell handelt es sich um die Simulation eines sechsseitigen Würfels mit Hilfe einer fairen Münze. Die Münze wird wiederholt geworfen und die Ergebnisse – Kopf oder Zahl – kodieren die verschiedenen Werte des Würfels. Abbildung 4.1 zeigt die DTMC zu diesem Würfel-Modell. Die Endzustände

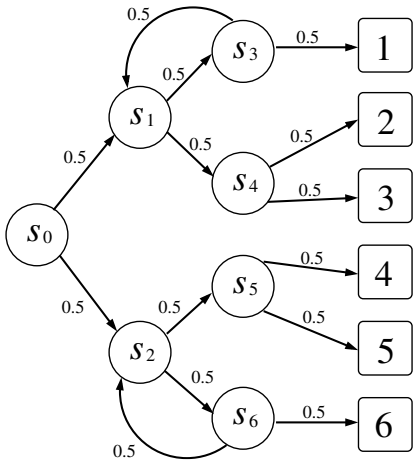

Abbildung 4.1: DTMC für Würfel-Modell (nach [38])

stehen für den kodierten Würfelwert. Je nach Ergebnis eines Münzwurfs werden die Kanten ausgewählt. Die obere ausgehende Kante eines Zustands steht für „Kopf“, die untere für „Zahl“.

Wie bei einem tatsächlichen sechseitigen Würfel so ist auch hier die Wahrscheinlichkeit für jede mögliche Augenzahl $\frac{1}{6}$. Als zu verifizierende Eigenschaft für den SBMC-Algorithmus bietet sich also $\Phi := P_{\leq \frac{1}{6}}[\top \mathcal{U}\, x]$ mit $x \in \{1, \ldots, 6\}$ an.

Für jeden möglichen Wert von x erreichte bei einem Testlauf das einfache SBMC-Verfahren für $k := 23$ die Wahrscheinlichkeitsgrenze $p := \frac{1}{6}$, wobei die Berechnung bis auf die siebte Nachkommastelle genau erfolgte. Es wurden insgesamt elf Pfade gefunden, der SAT-Solver wurde 31-mal aufgerufen. Die benötigte Rechenzeit betrug 0.5 Sekunden.

Schon bei einem so kleinen System zeigen sich die Vorteile der Looperkennung: Das erweiterte Verfahren erreichte die Wahrscheinlichkeitsgrenze bereits für die Iterationstiefe $k := 5$. Es wurde ein erweiterter Pfad mit einem Loop gefunden. Der SAT-Solver musste nur viermal aufgerufen werden und die benötigte Rechenzeit lag unterhalb der Messgrenze.

Eine Erweiterung des Würfel-Modells berechnet die Summe aus den Augenzahlen zweier Würfel, ähnlich wie das soeben vorgestellte Modell. Auch auf dieses wurde SBMC angewandt, die Ergebnisse sind in Tabelle 4.1 zu sehen.

Tabelle 4.1: Ergebnisse für die Summe aus zwei Würfeln

		Ohne Looperkennung				Mit Looperkennung				
$\sum$	**p**	$\#\omega$	**k**	**#SAT**	**Zeit (s)**	$\#\omega$	**#l**	**k**	**#SAT**	**Zeit (s)**
2	1/36	10	24	28	0.38	3	1	14	14	0.09
3	1/18	11	24	30	0.36	3	1	13	14	0.08
4	1/12	11	24	31	0.39	3	2	14	16	0.09
5	1/9	12	24	32	0.39	3	2	12	14	0.06
6	5/36	10	21	28	0.27	3	2	14	17	0.10
7	1/6	11	23	31	0.32	3	2	13	16	0.08
8	5/36	10	21	28	0.24	3	1	14	17	0.08
9	1/9	12	24	32	0.38	3	1	12	14	0.06
10	1/12	11	24	31	0.37	3	1	14	16	0.09
11	1/18	11	24	30	0.39	3	1	13	14	0.06
12	1/36	10	24	28	0.31	3	1	14	14	0.08

Tabelle 4.1 ist folgendermaßen aufgebaut: Die ersten beiden Spalten geben die Summe aus den Augenzahlen und die Wahrscheinlichkeitsgrenze p für das jeweilige Ergebnis an. Die nächsten vier Spalten sind die Ergebnisse für SBMC ohne Looperkennung: Die Anzahl an gefundenen Pfaden, die erreichte Iterationstiefe, die Anzahl an Aufrufen des SAT-Solvers und die benötigte Rechenzeit in Sekunden. Die letzten fünf Spalten enthalten die Ergebnisse des erweiterten SBMC-Verfahrens. Hier wird die Anzahl an

gefundenen Pfaden, an gefundenen verschiedenen Loops, die erreichte Iterationstiefe, die Anzahl an SAT-Aufrufen und die benötigte Rechenzeit in Sekunden angegeben.

Die Wahrscheinlichkeitsgrenzen ergeben sich aus dem Modell und entsprechen den Wahrscheinlichkeiten für die verschiedenen Summen. Wie aus der zweiten Spalte der Tabelle 4.1 zu erkennen ist, liegt hier keine Gleichverteilung mehr vor. Für einige der möglichen Ergebnisse gibt es mehr Augenzahl-Kombinationen als für andere. Die Summe 12 kann beispielsweise nur durch die Kombination von zwei Sechsen erreicht werden, für die Summe 7 gibt es dagegen drei Kombinationen (1 und 6, 2 und 5, 3 und 4). Die Wahrscheinlichkeiten werden auch bei diesem Modell vom SBMC-Verfahren bis auf die siebte Nachkommastelle genau berechnet.

Das Modell für die Summe aus zwei Würfeln umfasst 67 Zustände und ist damit ebenfalls relativ klein. Wie beim einfachen Würfel-Modell zeigt sich auch hier, dass die Optimierungen des SBMC-Verfahrens schon für eine geringe Anzahl an Loops greifen.

Das einfache SBMC-Verfahren benötigt, wie in Tabelle 4.1 zu sehen, für die Berechnung der verschiedenen Instanzen höchstens 0.39 Sekunden. Es werden jeweils zehn bis zwölf Pfade gefunden, die erreichte Iterationstiefe liegt zwischen 21 und 24, der SAT-Solver wird 28- bis 32-mal aufgerufen. Mit dem optimierten SBMC-Verfahren fallen die jeweiligen Ergebnisse niedriger aus. So wurden maximal 0.1 Sekunden benötigt, um die Wahrscheinlichkeitsgrenze p zu erreichen. Für jede Instanz wurden drei Pfade und ein bis zwei Loops gefunden. Die Iterationstiefe k liegt zwischen 12 und 13 und der SAT-Solver musste nur 14- bis 17-mal aufgerufen werden.

Anhand des einfachen Würfel-Modells und dem Modell für die Summe aus zwei Würfeln lässt sich erkennen, dass die Optimierungen des SBMC-Verfahrens bereits für wenige Loops auf kleinen Systemen greifen.

4.2 Das synchrone Leader-Election-Protokoll

Bei den vorangegangen Würfel-Modellen handelte es sich um kleine Systeme, für die sowohl mit als auch ohne Optimierungen die SBMC-Berechnung nicht viel Zeit benötigt. Zwar bestätigen die Ergebnisse dieser Berechnung die Vermutung, dass die Optimierungen das SBMC-Verfahren beschleunigen, jedoch sind sie ansonsten nicht besonders aussagekräftig. Deswegen beschäftigen wir uns nun mit einem etwas größeren Modell: Dem synchronen Leader-Election-Protokoll (siehe [30]).

Das synchrone Leader-Election-Protokoll befasst sich mit folgender Situation: In einem Ring von N Prozessoren soll ein führender Prozesser, ein *Leader*, ausgewählt werden. Dies geschieht mit Hilfe von Nachrichten, die sich die Prozessoren über den Ring zuschicken. Dabei wählt jeder Prozessor als ID zufällig eine Zahl aus dem ganzzahligen Intervall $[1, K]$. K ist eine im Voraus gegebene Konstante. Jeder Prozessor verschickt nun seine ID über den Ring. Der Prozessor, der die größte eindeutige ID hat, wird zum Leader. Sollte kein Leader gefunden werden, so wird dieser Vorgang so lange wiederholt, bis ein Leader gefunden wird.

Es wird hier von einem Zeit-synchronen Verhalten ausgegangen. Es gibt eine globale Uhr und in jedem Zeitschritt liest ein Prozessor die Nachricht, die im vorangegangenen

Zeitschritt gesendet wurde, falls es eine solche Nachricht gibt. Der Prozessor macht höchstens einen Zustandsübergang und verschickt unter Umständen anschließend seine eigene Nachricht.

Tabelle 4.2 enthält die Ergebnisse des SBMC-Verfahrens für verschiedene Werte von N und K. Es wurde überprüft, ob tatsächlich ein Leader gewählt wird. Die erste

Tabelle 4.2: Ergebnisse für das synchrone Leader-Election-Protokoll

			Ohne Looperkennung				Mit Looperkennung				
N	**K**	**p**	**#ω**	**k**	**#SAT**	**Zeit (s)**	**#ω**	**#l**	**k**	**#SAT**	**Zeit (s)**
3	2	0.99	66	16	78	0.19	6	2	8	22	0.03
3	3	0.99	143	12	151	0.48	24	3	8	95	0.18
3	4	0.99	276	8	280	0.57	60	4	8	267	0.47
3	5	0.99	589	8	593	3.57	120	5	8	575	3.24
3	6	0.99	1040	8	1044	4.16	210	6	8	1023	3.16
3	8	0.99	1979	8	1983	8.02	504	8	8	1961	7.71
3	10	0.99	990	4	990	4.32	990	0	4	990	4.31
3	12	0.99	1711	4	1711	9.98	1711	0	4	1711	8.29
4	2	0.99	– Timeout –				8	8	10	77	0.17
4	3	0.99	”				60	21	10	1279	13.00
4	4	0.90	3903	10	3908	114.63	216	40	10	3419	84.89
4	5	0.90	2123	12	2128	153.46	560	57	10	2060	121.41
4	6	0.90	1167	5	1167	34.89	1167	0	5	1167	26.84
4	8	0.90	3687	5	3687	108.06	3687	0	5	3687	88.34
5	2	0.90	– Timeout –				10	22	12	223	1.41
5	3	0.90	9585	12	9591	563.73	180	63	12	7297	324.00
5	4	0.90	23019	12	23035	3903.55	900	124	12	21237	3261.03
5	5	0.90	2813	6	2813	1336.80	2813	0	6	2813	1208.32
5	6	0.90	6999	6	6999	5361.11	6999	0	6	6999	5036.71
6	2	0.90	– Timeout –				12	52	14	626	8.01
7	2	0.90	– Timeout –				14	113	16	1597	38.89
8	2	0.90	– Timeout –				16	240	18	3837	166.59
9	2	0.90	– Timeout –				18	494	20	8883	669.85
10	2	0.90	– Timeout –				20	1004	22	20061	2468.66

und die zweite Spalte enthalten die Werte für N und K. Die dritte Spalte gibt die gewählte Wahrscheinlichkeitsschranke p an. Da das Leader-Election-Protokoll früher oder später immer zur erfolgreichen Wahl eines Leaders führt, wurde für p der Wert 0.99 beziehungsweise für die größeren Instanzen 0.9 gewählt. Die nächsten vier Spalten enthalten die Ergebnisse des SBMC-Verfahrens ohne Looperkennung: Die Anzahl an gefundenen Pfaden, die erreichte Iterationstiefe k, die Anzahl an Aufrufen des SAT-Solvers und die verbrauchte Rechenzeit in Sekunden.

Die achte bis zwölfte Spalte enthalten schließlich die Ergebnisse für das erweiterte

SBMC-Verfahren. Zunächst wird wieder die Anzahl an gefundenen erweiterten Pfaden angegeben, im Anschluss daran die Anzahl an gefundenen verschiedenen Loops. Die neunte Spalte enthält die erreichte Iterationstiefe k, die elfte Spalte gibt die Anzahl der Aufrufe des SAT-Solvers an. Die letzte Spalte enthält die benötigte Rechenzeit in Sekunden.

Wie bei den Würfel-Modellen, so lässt sich auch am Leader-Election-Protokoll erkennen, dass die Looperkennung mit Gegenbeispiel-Kompaktierung erhebliche Vorteile bringt. Die Anzahl an Aufrufen des SAT-Solvers sowie die Rechenzeit werden erheblich vermindert. Für einige Instanzen reicht das Zeitlimit von zwei Stunden für das einfache SBMC-Verfahren nicht aus, das erweiterte SBMC-Verfahren dagegen findet für die gleichen Instanzen Lösungen in akzeptabler Zeit.

Eine solche Instanz ist beispielsweise das Leader-Election-Protokoll für die Werte $N := 5$ und $K := 2$. Das erweiterte Verfahren benötigte 1.41 Sekunden, um die Wahrscheinlichkeitsgrenze $p := 0.9$ zu überschreiten. Dazu musste der SAT-Solver 223-mal aufgerufen und bis zur Iterationstiefe $k := 12$ gerechnet werden. Es wurden 10 Zeugen und 22 verschiedene Loops gefunden. Das einfache SBMC-Verfahren dagegen musste nach zwei Stunden abgebrochen werden, ohne dass ein Ergebnis feststand. Zu diesem Zeitpunkt war das einfache Verfahren auf Iterationstiefe 30 angelangt und 249 341 Pfade waren gefunden worden. Die Gesamt-Wahrscheinlichkeitsmasse dieser Pfade betrug ungefähr 0.78. An den unterschiedlichen erreichten Iterationstiefen für diese Instanz des Leader-Election-Protokolls lässt sich auch gut erkennen, dass das erweiterte SBMC-Verfahren nicht notwendigerweise bis zu der gleichen Iterationstiefe rechnen muss wie das einfache Verfahren.

Die hier gezeigten Instanzen des Leader-Election-Protokolls enthalten zwischen 22 (für $N := 3$, $K := 2$) und 12 529 Zustände (für $N := K := 5$). Mit Ausnahme von zwei Instanzen wurden weniger als 150 MB Speicherplatz benötigt, neunzehn der hier gezeigten Instanzen benötigten sogar weniger als 80 MB. Für die Instanzen $N := K := 5$ beziehungsweise $N := 5$, $K := 6$ wurden 165.13 MB beziehungsweise 275.15 MB gebraucht. Der Speicherplatzbedarf von SBMC mit und ohne Looperkennung lag dabei in ähnlichen Dimensionen.

4.3 Das Contract-Signing-Protokoll

Bei dem Contract-Signing-Protokoll (siehe [19]) geht es um folgende Situation: Zwei Parteien A und B haben einen Vertrag ausgehandelt, der nun mit einer digitalen Signatur unterschrieben werden soll. Da sich A und B nicht vollkommen vertrauen, soll mit Hilfe des Contract-Signing-Protokolls sichergestellt werden, dass sich beide Parteien fair verhalten . Wenn beispielsweise A den Vertrag bereits unterschrieben hat, so muss A auch die Unterschrift von B erhalten, unabhängig davon, wie sich B verhält.

A und B erzeugen jeweils $2n$ Bit-Folgen der Länge L. Diese Bit-Folgen werden *Geheimnisse* genannt. Sie sind zu Beginn nur der Partei bekannt, die die Geheimnisse generiert hat. Das jeweils i-te und das $(i+n)$-te Geheimnis bilden ein Paar. A verfügt

also über die Geheimnis-Paare $(a_1, a_{n+1}), \ldots, (a_n, a_{2n})$ und B über die Geheimnis-Paare $(b_1, b_{n+1}), \ldots, (b_n, b_{2n})$. Die Geheimnis-Paare bilden die digitale Signatur, die ausgetauscht werden soll.

Mit Hilfe eines einzelnen Geheimnisses und eines Verschlüsselungssystems F wird der Vertrag c kodiert. $F(c, a_i)$ ist die Kodierung von c durch F mit dem Geheimnis a_i. Das Verschlüsselungssystem F sowie die Kodierung von c mit jedem der Geheimnisse steht beiden Parteien zur Verfügung. Sie dienen dazu, die ausgetauschten Geheimnisse zu verifizieren.

Der Austausch der Geheimnis-Paare läuft in zwei Phasen ab. Zunächst sendet eine der beiden Parteien, zum Beispiel A, eines ihrer Geheimnis-Paare (a_i, a_{i+n}). Das verwendete Übertragungsprotokoll ist so gestaltet, dass B mit Wahrscheinlichkeit 0.5 das Geheimnis a_i oder das Geheimnis a_{i+n} empfängt. A erfährt dabei nicht, welches der beiden Geheimnisse B empfangen hat. Anschließend sendet B auf dem gleichen Weg ein Geheimnis-Paar an A. Dies wird so lange fortgesetzt, bis jede Partei über die Hälfte der Geheimnisse der anderen Partei verfügt. In der zweiten Phasen werden einzelne Bits der Geheimnisse verschickt. Die Partei, die in Phase 1 begonnen hat, in diesem Fall A, schickt die j-ten Bits der Geheimnisse $a_1, \ldots a_{2n}$ an B. Im Anschluss daran sendet B die j-ten Bits seiner Geheimnisse $b_1, \ldots b_{2n}$ an A. Der Vorgang wird so lange wiederholt, bis jede Partei mindestens ein Geheimnis-Paar der jeweils anderen Partei kennt. Dies entspricht dem Contract Signing-Protokoll, wie es in [19] vorgestellt wird. Im Folgenden wird das Contract-Signing-Protokoll nach den Autoren von [19] mit EGL abgekürzt.

Das Protokoll geht davon aus, dass beide Parteien sich daran halten. Ist dies nicht der Fall, so ist die beginnende Partei A im Nachteil. Sobald B eines der Geheimnis-Paare von A kennt, könnte B einfach aus dem Protokoll ausscheiden, indem B keine weiteren Bits mehr verschickt. A kann in diesem Fall die ihm fehlenden Bits mit Hilfe des Verschlüsselungssystems F und den öffentlichen Kodierungen von c errechnen, verliert allerdings Zeit dadurch. Die Situation, dass B ein Geheimnis-Paar von A und A kein Geheimnis-Paar von B kennt, tritt auf jeden Fall ein, da A zuerst die L-ten Bits seiner Geheimnisse verschickt.

Im Folgenden bedeutet $known_A$ beziehungsweise $known_B$, dass A beziehungsweise B ein Geheimnis-Paar der anderen Partei kennt. Wir überprüfen deswegen die Eigenschaft $\Phi := P_{\leq 0.99}[\top \mathcal{U} (\neg known_A \wedge known_B)]$. Tabelle 4.3 enthält die Ergebnisse für verschiedene Instanzen von EGL. Da keine Loops gefunden wurden, sind nur die Ergebnisse für SBMC ohne Looperkennung angegeben.

Die erste Spalte von Tabelle 4.3 enthält die Anzahl N an Geheimnis-Paaren, die zweite Spalte enthält die Länge L der Geheimnisse in Bits. Die dritte Spalte gibt die Anzahl an Zuständen der jeweiligen Instanz an. Wie dort abzulesen ist, haben die hier aufgeführten Instanzen zwischen 28 830 (für $N := 5$ und $L := 2$) und 623 486 (für $N := 7$ und $L := 2$) Zustände.

Die nächsten vier Spalten geben die Ergebnisse des SBMC-Verfahrens für die einzelnen Instanzen an: Spalte 4 enthält die Anzahl an gefundenen Zeugen, Spalte 5 die erreichte Iterationstiefe k. In Spalte 6 steht die Anzahl an Aufrufen des SAT-Solvers. In Spalte 7 und 8 sind der benötigte Speicherplatz in MB und die Laufzeit in Sekunden

Tabelle 4.3: Ergebnisse für das Contract-Signing-Protokoll

N	L	#s	#ω	k	#SAT	MB	Zeit (s)
5	2	28830	1014	36	1019	57.05	108.47
5	3	49310	1014	56	1019	112.00	427.74
5	4	69790	1014	76	1019	261.57	1559.78
5	5	90270	1014	96	1019	376.17	2883.82
6	2	135550	4056	43	4062	89.54	625.78
6	3	233854	4056	67	4062	178.05	1808.50
6	4	332158	4056	91	4062	410.62	5785.07
7	2	623486	16221	49	16227	159.77	3016.32

angegeben.

Am meisten Speicherplatz wurde für die Instanz $N := 6, L := 4$ mit 410.62 MB verbraucht. Dies liegt noch gut innerhalb der verfügbaren Ressourcen von 4 GB Hauptspeicher. Für $N := L := 5$ wird mit 376.17 MB ähnlich viel Speicherplatz benötigt, die übrigen in Tabelle 4.3 gezeigten Instanzen haben einen deutlich geringeren Speicherplatzbedarf.

An den Ergebnissen zeigt sich, dass das SBMC-Verfahren auch auf relativ großen Systemen ohne Loops angewendet werden kann. Sowohl die Rechenzeit als auch der verbrauchte Speicherplatz befinden sich in einem vertretbaren Rahmen.

In [36] werden mehrere Varianten des Contract-Signing-Protokolls vorgestellt, die versuchen, die unfaire Situation des ursprünglichen Contract-Signing-Protokolls zu beheben. Diese Varianten unterscheiden sich alle in Phase 2.

In der ersten Variante – im Folgenden EGL2 genannt – versendet A nur die Bits für die erste Hälfte seiner Geheimnisse. Anschließend macht B das Gleiche für seine Geheimnisse. Der Vorgang wird für die restlichen Geheimnisse wiederholt.

In Phase 2 der zweiten Variante – EGL3 – wird abwechselnd von A und B ein Bit für die ersten n Geheimnisse gesendet, bis diese Geheimnisse komplett übertragen wurden. Im Anschluss daran geschieht das Gleiche für die zweite Hälfte der Geheimnisse.

Die letzte Variante – EGL4 – geht nach folgendem Schema vor: Zuerst verschickt A ein Bit für sein erstes Geheimnis a_1 und wartet darauf, dass B je ein Bit für seine Geheimnisse b_1 bis b_N schickt. Sobald dies geschehen ist, versendet A die Bits für die Geheimnisse a_2 bis a_N. Dies wird so lange wiederholt, bis die ersten N Geheimnisse abgearbeitet wurden. Danach wird die zweite Hälfte der Geheimnisse nach dem gleichen Prinzip verschickt.

Auch diese drei Varianten von EGL können unfaire Situationen nicht komplett ausschließen, wobei dies in EGL4 noch am ehesten gelingt (vergleiche [36]). Im Folgenden sind in Tabelle 4.3 die Ergebnisse von SBMC für die drei Varianten von EGL repräsentiert, mit Sicherheitseigenschaft $\Phi := P_{\leq p}[\top \mathcal{U} (\neg\, known_A \wedge known_B)]$.

Spalte 1 von Tabelle 4.3 gibt Auskunft, welche der drei Varianten betrachtet wurde.

Tabelle 4.4: Ergebnisse für Varianten des Contract-Signing-Protokolls

	N	**L**	**#s**	**P**	**p**	**#ω**	**k**	**#SAT**	**MB**	**Zeit (s)**
EGL2:	5	2	33790	0.9697266	0.9	922	34	925	65.95	99.69
	5	3	54270	”	”	922	54	925	139.39	477.18
	5	4	74750	”	”	922	74	925	308.75	1370.05
	5	5	95230	”	”	922	94	925	439.38	2769.84
	6	2	159742	0.9846191	0.9	3687	40	3690	99.80	545.46
	6	3	258046	”	”	3687	64	3690	194.67	1536.6
	6	4	356350	”	”	3687	88	3690	504.86	5826.88
	7	2	737278	0.9922485	0.9	14746	46	14749	193.75	3286.45
EGL3:	5	2	33790	0.6669922	0.6	615	23	617	65.27	60.72
	5	3	54270	”	”	615	33	617	118.41	220.8
	5	4	74750	”	”	615	43	617	229.71	1227.71
	5	5	95230	”	”	615	53	617	385.99	2789.18
	5	6	115710	”	”	615	63	617	579.81	2518.44
	6	2	159742	0.6667480	0.6	2458	27	2460	88.68	378.08
	6	3	258046	”	”	2458	39	2460	170.09	1167.19
	6	4	356350	”	”	2458	51	2460	405.50	4793.49
	7	2	737278	0.666687	0.6	9831	31	9833	158.80	2135.30
	7	3	1196030	”	”	9831	45	9833	314.31	5768.83
EGL4:	5	2	33790	0.5156250	0.5	512	21	512	44.86	29.63
	5	3	54270	”	”	512	31	512	79.21	77.74
	5	4	74750	”	”	512	41	512	171.43	220.83
	5	5	95230	”	”	512	51	512	272.92	488.84
	5	6	115710	”	”	512	61	512	358.36	611.68
	6	2	159742	0.5078125	0.5	2048	25	2048	74.18	194.62
	6	3	258046	”	”	2048	37	2048	140.39	448.86
	6	4	356350	”	”	2048	49	2048	283.22	1167.12
	6	5	454654	”	”	2048	61	2048	441.73	2334.04
	6	6	552958	”	”	2048	73	2048	646.44	3404.29
	7	2	737278	0.5039063	0.5	8192	29	8192	140.94	1072.86
	7	3	1196030	”	”	8192	43	8192	248.52	2622.88

Die nachfolgenden Spalten 2 bis 4 enthalten die Anzahl N an Geheimnis-Paaren, die Bit-Länge L und die Anzahl an Zuständen der betrachteten Instanzen. Das größte Modell enthält 1 196 030 Zustände (für $N := 7$ und $L := 3$), das kleinste 33 790 Zustände ($N := 5$ und $L := 2$).

Die fünfte Spalte enthält die tatsächlichen Wahrscheinlichkeiten P, mit denen die unfaire Situation eintritt. Sie wurden von PRISM errechnet und auf die siebte Nachkommastelle gerundet. Für die Wahrscheinlichkeitsgrenzen p des SBMC-Verfahrens wurden diese Wahrscheinlichkeiten P auf die erste Stelle nach dem Komma abgerundet (siehe Spalte 5).

Da auch in den in Tabelle 4.3 betrachteten Fällen keine Loops gefunden wurden, sind hier ebenfalls nur die Ergebnisse für das SBMC-Verfahren ohne Looperkennung angegeben.
Die sechste Spalte enthält die Anzahl an gefundenen Zeugen, die siebte die erreichte Iterationstiefe k. In Spalte 8 steht die Anzahl an Aufrufen des SAT-Solvers, in den Spalten 9 und 10 der verbrauchte Speicherplatz in MB und die benötigte Rechenzeit in Sekunden.

Die Instanz von EGL4 für $N := L := 6$ verbrauchte mit 646.44 MB den meisten Speicherplatz. Auch dies befindet sich noch im Rahmen der zur Verfügung stehenden Ressourcen. Die übrigen Instanzen hatten einen – in den meisten Fällen sogar deutlich – geringeren Speicherplatzbedarf.

Wie schon bei dem einfachen Contract-Signing-Protokoll zeigt sich auch hier, dass auch große Systeme ohne Loops durch das SBMC-Verfahren innerhalb akzeptabler Ressourcen-Grenzen behandelt werden können.

4.4 Das Crowds-Protokoll

Das Crowds-Prokoll befasst sich mit anonymem Webbrowsing [39]. Innerhalb einer Gruppe – englisch „Crowd“ – werden Nachrichten verschickt, indem sie zufällig über die Router der einzelnen Gruppenmitglieder geleitet werden. Dabei kann der Pfad, über den die Nachricht verschickt wird, beliebig viele Gruppenmitglieder enthalten. Auf diese Art ist nicht feststellbar, von welchem Anwender die Anfrage wirklich stammt. Jedes Gruppenmitglied könnte der ursprüngliche Versender der Nachricht sein.

Der Pfad wird auf folgende Weise erzeugt: Der Versender der Nachricht wählt zufällig ein Gruppenmitglied aus, über dessen Router die Nachricht geleitet werden soll. Der ausgewählte Router leitet die Nachricht mit Wahrscheinlichkeit $1 - p_f$ an ihr Ziel weiter. Mit Wahrscheinlichkeit p_f wird die Nachricht an den Router eines anderen Gruppenmitglieds weitergeleitet, das ebenfalls zufällig ausgewählt wurde. Die *Forwarding Probability* p_f ist dabei im Voraus gegeben. Dieser Prozess wird so lange fortgesetzt, bis die Nachricht ihr Ziel erreicht. Bei der zuälligen Auswahl des Gruppenmitglieds, über das als Nächstes die Nachricht weitergeleitet werden soll, kann jedes Gruppenmitglied auch sich selbst auswählen. Die Kommunikation zwischen zwei Gruppenmitgliedern wird mit einem paarweisen symmetrischen Schlüssel kodiert.

Wurde einmal ein Nachrichten-Pfad von einem Sender zu einem Ziel gebaut, so steht dieser Pfad fest, bis es notwendig wird, neue Nachrichten-Pfade zu berechnen. Eine Neuberechnung wird zum Beispiel notwendig, wenn sich die Größe der Gruppe ändert. Die Anonymität der einzelnen Gruppenmitglieder bleibt jedoch gewährleistet.

Die Situation ändert sich, wenn es korrupte Gruppenmitglieder beziehungsweise *Gegner* in der Gruppe gibt, die versuchen, herauszufinden, wer der ursprüngliche Versender einer Nachricht ist. Es wird davon ausgegangen, dass ein Gegner nur sein eigenes lokales Netzwerk überwachen kann. Ein Gegner kann also nur dann eine Nachricht beobachten, wenn er Teil des jeweiligen Nachrichten-Pfades ist. In diesem Fall sieht er, von welchem Mitglied die Nachricht an ihn weitergeleitet wurde. Da aber nicht bekannt ist, ob es sich bei dem vorhergehenden Mitglied um dem Versender handelt, so ist die Anonymität von Letzterem nach wie vor gewahrt. Kritisch wird es jedoch, wenn der Nachrichten-Pfad neu berechnet wird und erneut korrupte Mitglieder zu dem Pfad gehören. Die zusammenarbeitenden Gegner können nun unter Umständen den Ursprung der Nachricht auf die gleiche Quelle zurückführen, womit der Versender indentifiziert wäre. Die Gegner können dazu eine Reihe von Daten ausnutzen, wie zum Beispiel Cookies, Muster im Browsing-Verhalten und zeitliche Beobachtungen.

Im Folgenden bezeichnet N die Größe der Gruppe, G ist die Anzahl an nicht-korrupten Mitgliedern und R ist die Anzahl an Neuberechnungen der Nachrichten-Pfade. $observe_s$ ist die Häufigkeit, mit der ein Gruppenmitglied s als Versender einer Nachricht eingeschätzt wurde. Gilt $observe_s > 1$, so ist s in der Tat als Versender identifiziert worden.

Wir überprüfen nun die Sicherheitseigenschaft $\Phi := P_{\leq p}[\top \mathcal{U} (observe_s > 1)]$ für verschieden viele nicht-korrupte Mitglieder und Pfad-Berechnungen und unterschiedliche Wahrscheinlichkeitsgrenzen p. Die Gruppengröße beträgt immer $N := 20$, die Forwarding Probability ist $p_f := 0.8$. Tabelle 4.4 zeigt die Ergebnisse.

Spalte 1 von Tabelle 4.4 gibt die Anzahl an nicht-korrupten Mitgliedern an. Es wurden Instanzen für zwei, vier, fünf, zehn und 15 nicht-korrupte Mitglieder berechnet. 20 nicht-korrupte Mitglieder in einer Gruppe der Größe 20 zu betrachten, macht keinen Sinn, da die Wahrscheinlichkeit, dass der Versender von einem korrupten Mitglied entdeckt werden könnte, in diesem Fall offensichtlich 0 betragen muss. Deswegen wurde dieser Fall nicht berechnet, auch wenn er in dem Modell, wie es auf [38] bereitgestellt wird, einstellbar ist.

Die zweite Spalte enthält die Anzahl an Berechnungen der Nachrichten-Pfade. Da es unmöglich ist, den Versender zu entdecken, wenn der Nachrichten-Pfad nur einmal berechnet wird, wurde mit mindestens zwei Berechnungen begonnen.

In der dritten Spalte steht die Anzahl an Zuständen der verschiedenen Instanzen. Letztere haben zwischen 77 (für $G := 2$, $R := 2$) und 2 464 168 (für $G := 15$, $R := 6$) Zustände.

Spalte 4 gibt die tatsächlichen Wahrscheinlichkeiten P für die Entdeckung des Versenders an, gerundet bis auf die siebte Stelle nach dem Komma. Sie wurden erneut von PRISM errechnet. Die Wahrscheinlichkeit, dass der Versender von einem korrupten Mitglied entdeckt wird, steigt mit der Anzahl an Neuberechnungen des Nachrichten-Pfades und sinkt mit der Anzahl an Gegnern. Die in Spalte 5 enthaltenen Wahrschein-

Tabelle 4.5: Ergebnisse für das Crowds-Protokoll

					Ohne Looperkennung				Mit Looperkennung				
G	**R**	**#s**	**P**	**p**	**#ω**	**k**	**#SAT**	**Zeit (s)**	**#ω**	**#l**	**k**	**#SAT**	**Zeit (s)**
2	2	77	0.0450664	0.045	– Timeout –				1	16	35	41	4.65
2	3	183	0.1160649	0.100	”				9	25	29	108	5.45
2	4	356	0.1999541	0.199	”				57	98	46	1546	649.09
2	5	612	0.2880612	0.250	”				313	113	44	4889	484.69
2	6	967	0.3748147	0.300	”				1504	166	48	17893	3133.87
4	2	216	0.0229960	0.022	– Timeout –				1	567	29	586	11.87
4	3	726	0.0620134	0.050	”				93	936	30	3401	98.33
4	4	1891	0.1116642	0.070	”				2445	2998	34	23954	1252.82
4	5	4187	0.1678260	0.073	”				3532	3012	34	28818	2063.45
4	6	8275	0.2273823	0.075	”				5599	3168	35	35219	3038.83
5	2	311	0.0194646	0.019	– Timeout –				1	10801	32	10823	361.33
5	3	1198	0.0529623	0.030	5366	27	5382	157.63	129	214	24	639	23.72
5	4	3515	0.0961986	0.050	– Timeout –				3480	3470	31	20769	1027.64
5	5	8653	0.1458039	0.050	”				3442	3467	31	20753	1525.02
5	6	18817	0.1991599	0.050	”				3291	3470	31	20577	1724.69
10	2	1041	0.0132843	0.012	24026	26	24041	401.76	1	1457	23	1470	28.01
10	3	6563	0.0367907	0.020	4139	24	4152	172.84	420	1684	24	2352	94.94
10	4	30070	0.0679861	0.025	52795	27	52811	3216.86	3068	15185	27	23809	1982.41
10	5	111294	0.1047860	0.026	67067	28	67084	5504.32	4129	15393	27	31752	3573.33
10	6	352535	0.1454839	0.027	– Timeout –				8678	16007	27	43524	5180.80
15	2	2196	0.0114858	0.010	366	20	375	12.75	1	182	20	192	8.12
15	3	19228	0.0319954	0.020	28352	26	28367	1936.74	4840	6294	24	14121	914.82
15	4	119800	0.0594626	0.020	17677	25	17691	1906.22	4802	6289	24	14107	1724.59
15	5	592060	0.0921605	0.020	17677	25	17691	2854.80	4748	6279	24	14091	2617.21
15	6	2464168	0.1286525	0.020	17677	25	17691	3474.33	4653	6270	24	14054	3351.25

lichkeitsgrenzen p wurden für das SBMC-Verfahren so gewählt, dass sie möglichst nah an den jeweiligen tatsächlichen Wahrscheinlichkeiten P lagen und gleichzeitig gut innerhalb von der Rechenzeit-Begrenzung von zwei Stunden erreicht werden konnten.

Die Spalten 6 bis 9 enthalten die Ergebnisse für das SBMC-Verfahren ohne Looperkennung: Die Anzahl an Pfaden, die erreichte Iterationstiefe k, die Anzahl an Aufrufen des SAT-Solvers und die Rechenzeit in Sekunden. Die übrigen Spalten (10 bis 14) enthalten die Ergebnisse für das Verfahren mit Looperkennung. In Spalte 10 steht die Anzahl an Zeugen, in Spalte 11 die Anzahl an gefundenen Loops und Spalte 12 enthält die erreichte Iterationstiefe k. Die Spalten 13 und 14 geben die Anzahl an Aufrufen des SAT-Solvers und die benötigte Rechenzeit in Sekunden an.

Aus den Ergebnissen des Crowds-Protokolls in Tabelle 4.4 ist erneut ersichtlich, welche deutliche Beschleunigung die Looperkennung erbringt. Bei 15 der hier gezeigten 25 Instanzen konnte ohne Looperkennung die Wahrscheinlichkeitsgrenze innerhalb von zwei Stunden nicht erreicht werden, bei den übrigen zehn Instanzen liegt die Rechenzeit höher als bei der entsprechenden Berechnung mit Looperkennung. Als Beispiel sei hier die Instanz für $G := 10$, $R := 2$ und $p := 0.012$ herausgegriffen: Ohne Looperkennung läuft das Verfahren bis zur Iterationstiefe 26, es werden dabei 24 026 Zeugen gefunden. Der SAT-Solver wird innerhalb der 401.76 Sekunden Rechenzeit 24 041-mal aufgerufen. Mit Looperkennung werden dagegen nur 28.01 Sekunden und 1470 Aufrufe gebraucht. Der Algorithmus dringt bis zur Iterationstiefe 23 vor und findet einen Zeugen mit 1457 Loops, bis die Wahrscheinlichkeitsgrenze überschritten ist.

Den höchsten Speicherplatzbedarf hatte mit 641.25 MB die Instanz für $G := 2$ und $R := 6$, was ebenfalls deutlich innerhalb der verfügbaren Ressourcen von 4 GB Hauptspeicher liegt. Für $G := 4$ und $R := 6$ wurde mit 422.13 MB deutlich weniger Speicherplatz gebraucht, dies ist die Instanz mit dem zweithöchsten Bedarf. Mit zwei Ausnahmen – $G := 2, R := 4$ mit 244.49 MB und $G := 15, R := 6$ mit 212.04 MB – benötigten alle übrigen Instanzen weniger als 200 MB.

Am Crowds-Protokoll zeigt sich, ähnlich wie beim Contract-Signing Protokoll in Kapitel 4.3, dass mit dem SBMC-Vefahren auch größere Systeme verifiziert werden können.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, wie Bounded Model Checking (BMC) auf stochastischen Systemen, dargestellt durch Markov-Ketten mit diskreter Zeit (DTMCs), angewendet werden kann. Es gilt, eine Sicherheitseigenschaft $\Phi := P_{\leq p}[a \,\mathcal{U}\, b]$ auf einer DTMC $\mathcal{D}$ zu widerlegen. Dies geschieht durch die Erzeugung eines Gegenbeispiels E, welches aus einer Menge von Zeugen für $[a \,\mathcal{U}\, b]$ besteht. Die Wahrscheinlichkeitsmasse von E muss dabei über der Wahrscheinlichkeitsgrenze p liegen.

In Kapitel 2 wurden zuerst einige Grundlagen vorgestellt. Diese umfassen das Erfüllbarkeitsproblem zusammen mit den Grundzügen von SAT-Solvern und BDDs, Bounded Model Checking und stochastische Systeme in Form von DTMCs.

Im Anschluss daran wurde in Kapitel 3 das Verfahren zum Bounded Model Checking auf stochastischen Systemen (SBMC) vorgestellt. Zuerst wurde in Kapitel 3.1 das grundlegende Verfahren erläutert. Dabei wird die DTMC $\mathcal{D}$ unter Berücksichtigung von Φ in Entscheidungsdiagramme umgewandelt, aus denen ein boolescher Ausdruck ψ in konjunktiver Normalform (KNF) generiert wird. Dieser Ausdruck ψ wird an einen SAT-Solver übergeben. Findet dieser eine erfüllende Belegung, so entspricht diese Belegung einem Pfad ω in $\mathcal{D}$, der ein Zeuge für $[a \,\mathcal{U}\, b]$ ist. ω wird der Menge E hinzugefügt. Dieser Vorgang wird wiederholt, bis die Wahrscheinlichkeitsmasse von E die Wahrscheinlichkeitsgrenze p überschreitet oder eine maximale Pfadlänge k_{max} erreicht wurde.

In den Kapiteln 3.2 und 3.3 wurde vorgestellt, wie das SBMC-Verfahren durch Gegenbeispiel-Kompaktierung und Ausschließen von im Voraus berechneten Pfaden optimiert werden kann. Durch Looperkennung werden die Pfade in erweiterte Pfade umgewandelt. Ein solcher erweiterter Pfad $\omega_e := (\omega_b, L)$ besteht aus einem Basispfad ω_b und einer Menge L von Loops. Da erweiterte Pfade alle Pfade, die durch Abwickeln der Loops in L entstehen können, repräsentieren, verringert sich die Größe des Gegenbeispiels erheblich. Die Wahrscheinlichkeitsmasse eines erweiterten Pfades liegt über der Wahrscheinlichkeitsmasse eines einfachen Pfades.

Durch die erweiterten Pfade lassen sich auch im Voraus die Pfade berechnen, die durch mehrfaches Abwickeln der Loops in L entstehen könnten. Diese Pfade müssen nicht mehr von dem SAT-Solver betrachtet werden. Durch die Vorausberechnung sind sie bereits bekannt und können dementsprechend von der Suche nach weiteren Zeugen ausgeschlossen werden.

In Kapitel 3.4 wurde schließlich ein kurzer Einblick auf eine mögliche Implementierung des SBMC-Verfahrens mit allen vorgestellten Optimierungen gegeben.

Zuletzt wurden in Kapitel 4 einige Experimente vorgestellt, bei denen SBMC angewendet werden kann, und Ergebnisse für verschiedene Instanzen dieser Experimente präsentiert. In den Ergebnissen zeigt sich, dass das SBMC-Verfahren auch für

größere Systeme mit über 10^6 Zuständen verwendet werden kann. Laufzeit und Speicherplatzbedarf liegen dabei innerhalb akzeptabler Ressourcen-Grenzen. Durch die Gegenbeispiel-Kompaktierung und das Ausschließen von im Voraus berechneten Pfaden lässt sich das Verfahren außerdem erheblich beschleunigen.

In dieser Arbeit wurden bereits einige Optimierungen für SMBC vorgestellt und auch umgesetzt. Es sind jedoch weitere Optimierungen und Erweiterungen für das Verfahren denkbar, die an dieser Stelle für zukünftige Forschungen in diesem Bereich aufgeführt werden sollen:

In [44] werden drei mögliche Optimierungen für BMC vorgeschlagen, die sich auch bei SBMC anwenden ließen: Die erste Optimierung betrifft die Reihenfolge, in der Variablen vom SAT-Solver betrachtet werden. Nach der jetzigen Lösung kann es vorkommen, dass Variablen, die im gleichen Zeitrahmen vorkommen, zu weit auseinander liegenden Zeitpunkten belegt werden. Treten durch diese Belegung Konflikte innerhalb der dem Zeitrahmen entsprechenden Klausel auf, so müssen alle Belegungen, die in der Zwischenzeit vorgenommen wurden, rückgängig gemacht werden. Um dies zu vermeiden, schlägt der Autor von [44] vor, Variablen, die auf diese Weise voneinander „abhängig“ sind, möglichst direkt nacheinander zu betrachten. Aufgrund der Struktur der KNF, die für BMC beziehungsweise für SMBC verwendet wird[1], sind die Abhängigkeiten zwischen Variablen immer auf zwei Klauseln beschränkt. Dadurch ließe sich die Variablenreihenfolge bereits ansatzweise im Voraus festlegen.

Eine andere in [44] vorgeschlagene Optimierung betrifft die vom SAT-Solver gelernten Konfliktklauseln[2]. Bei dem derzeitigen Verfahren gehen durch den Neustart des SAT-Solvers alle diese Klauseln verloren, wenn von der Iterationstiefe k zur Iterationstiefe $k + 1$ übergegangen wird. Dabei könnten einige dieser Konfliktklauseln auch für die neue Iterationstiefe gültig sein. Betrachten wir dazu die SBMC-Formeln ψ_k und ψ_{k+1} für die Iterationstiefen k und $k + 1$:

$$\begin{aligned} \psi_k &:= I(s_0) \wedge \textstyle\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \quad \neg\mathcal{X}_b(s_k) \\ \psi_{k+1} &:= I(s_0) \wedge \textstyle\bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \wedge \quad \neg\mathcal{X}_b(s_{k+1}) \end{aligned}$$

Die beiden Formeln verfügen über einen gemeinsamen Teil:

$$\phi := \psi_k \cap \psi_{k+1} = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Alle Konflikte, die von den Klauseln in ϕ ausgelöst werden, können sowohl für ψ_k als auch für ψ_{k+1} auftreten. Es bietet sich also an, die entsprechenden Konfliktklauseln in der neuen Iterationstiefe wieder zu verwenden. Auf diese Art könnte die Anzahl an SAT-Aufrufen vermindert werden, was sich positiv auf die benötigte Rechenzeit auswirken würde.

Die dritte Optimierung aus [44] befasst sich ebenfalls mit gelernten Konflikten. Die Konfliktklauseln können nicht nur für die nächste Iterationstiefe behalten werden, sie

[1] Siehe hierzu Seite 22 in Kapitel 2.2 beziehungsweise Seite 37 in Kapitel 3.1.

[2] Das Lernen von Konflikten wird in Kapitel 2.1.2 auf Seite 15 beschrieben.

können auch dazu verwendet werden, einen symmetrischen Konflikt vorauszuberechnen. Der Autor von [44] bezeichnet diese Methode als „Contraint Replication“. Es wird dabei ausgenutzt, dass die Klauseln für die Transitionsrelation immer gleich aufgebaut sind. Sie unterscheiden sich nur in den verschobenen Indizes der Variablen. Tritt beispielsweise ein Konflikt zwischen den Klauseln für $T(s_{i-1}, s_i)$ und $T(s_i, s_{i+1})$ auf, so ist dieser Konflikt auch zwischen $T(s_i, s_{i+1})$ und $T(s_{i+1}, s_{i+2})$ möglich. Die entsprechende Konfliktklausel muss nur mit dem gleichen Index-Shift versehen werden wie die Klauseln für die Transitionsrelation, um diesen Konflikt auch für die nächste Iterationtiefe zu vermeiden. Das Gleiche gilt für Konflikte zwischen dem letzten Zustandsübergang $T(s_{k-2}, s_{k-1})$ und der Invariante $\mathcal{X}_b(s_k)$ für die kritischen Zustände. Wird die neue Iterationstiefe $k + 1$ betrachtet, so können diese Konflikte auch zwischen $T(s_{k-1}, s_k)$ und $\mathcal{X}_b(s_{k+1})$ auftreten. Nach einem entsprechenden Index-Shift können die Konfliktklauseln in diesem Fall ebenfalls übernommen werden.

Es sind noch weitere Optimierungen möglich: In Kapitel 3.1.1 wurde beschrieben, wie nach der Erstellung der Entscheidungsdiagramme aus der DTMC $\mathcal{D}$ Kanten entfernt werden. Wie an dieser Stelle bereits erwähnt worden ist, ist es notwendig, diese Kanten zu entfernen. Es können jedoch weitere Kanten entfernt werden: Kanten, die kein Bestandteil von Pfaden sind, die vom Startzustand s_0 zu einem kritischen Zustand führen, werden nicht benötigt. Ebenso können Kanten gelöscht werden, die zu Zuständen führen, die vom Startzustand s_0 aus nicht erreichbar sind. Durch das Entfernen dieser Kanten kann $\mathcal{D}$ weiter reduziert werden, es ist jedoch nicht notwendig. Unter Umständen kann es außerdem vorkommen, dass der Transitions-BDD dadurch größer ausfällt als ohne diese Reduktion.

Eine zusätzliche Optimierungsmöglichkeit besteht in der Verwendung von partiellen Variablenbelegungen. Häufig ist eine SAT-Formel schon erfüllt, bevor ein SAT-Solver alle Variablen der Formel belegt hat. Würde es gelingen, diese partiellen Belegungen vom SAT-Solver zu erhalten, so könnten damit unter Umständen mehrere verschiedene Pfade gleichzeitig dargestellt werden. Die Gegenbeispiele ließen sich dadurch weiter kompaktieren. Die Menge und Größe der Klauseln, die zum Ausschließen von Pfaden verwendet werden, könnte man dadurch ebenfalls reduzieren.

In manchen Fällen können aufgrund der Struktur einer DTMC $\mathcal{D}$ die Klauseln zum Ausschließsen von gefundenen Pfaden zusätzlich verkleinert werden: Verfügt $\mathcal{D}$ nur über einen einzigen kritischen Zustand, so kann in den Pfadklauseln der Zeitrahmen k wie schon der Zeitrahmen 0 weggelassen werden[3]. Durch die Invariante für den kritischen Zustand wird sichergestellt, dass alle gefundenen Pfade dort enden, der Zeitrahmen k ist deswegen beim Ausschluss von Pfaden nicht zwingend erforderlich.

Eine weitere Optimierungen bezieht sich auf die konkrete Implementierung des SBMC-Verfahrens. Bei dem vorgestellten Algorithmus zum Lösen von Gleichung 3.5 zum Ausschließen von mehreren Loops handelt es sich um eine relativ einfache Rekursion[4]. Es sollte nach einem effizienteren Algorithmus gesucht werden, dadurch ließe sich die Laufzeit des Verfahrens verringern.

[3]Vergleiche dazu Kapitel 3.1.3, Seite 38.
[4]Siehe dazu Algorithmus 3 auf Seite 46.

Außerdem sollte auch eine andere Darstellung von Gegenbeispielen im SBMC-Verfahren in Betracht gezogen werden. Im Augenblick sind Gegenbeispiele nichts anderes als Mengen von Pfaden, die sich aus Folgen von Zuständen zusammensetzen. Gegenbeispiele könnten jedoch auch durch Bäume repräsentiert werden. Gemeinsame Präfixe von Pfaden würden nur einmal dargestellt werden. Dies hätte den Vorteil, dass Loops, die an einem solchen Präfix gefunden werden würden, automatisch für alle Pfade mit diesem Präfix verwendet werden könnten.

Für die letzte hier aufgeführte Optimierung soll noch einmal die Looperkennung näher betrachtet werden. Wie auf Seite 51 in Kapitel 3.3 angesprochen, ist die Looperkennung in der hier vorgestellten Form nicht in der Lage, verschachtelte oder sich überschneidende Loops zu erkennen[5]. Derartige Loops werden stattdessen in größeren Loops implizit behandelt. Dies hat jedoch den Nachteil, dass diese Loops nicht vollständig im Voraus ausgeschlossen werden können. Durch eine mächtigere Looperkennung, die in der Lage wäre, solche verschachtelten und sich überschneidenden Loops zu erkennen und zu behandeln, könnten nicht nur noch mehr Pfade im Voraus ausgeschlossen werden. Es wäre dadurch ebenfalls möglich, die Gegenbeispiele noch stärker zu kompaktieren.

Grundsätzlich ist das SBMC-Verfahren in der in dieser Arbeit vorgestellen Form praktikabel, es kann jedoch durch die hier vorgeschlagenen Optimierungen weiter verbessert werden.

[5]Siehe Seite 51.

Literaturverzeichnis

[1] Ábrahám, Erika, Bernd Becker, Felix Klaedtke und Martin Steffen: *Optimizing Bounded Model Checking for Linear Hybrid Systems.* In: Cousot, Radhia (Herausgeber): *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Band 3385 der Reihe *Lecture Notes in Computer Science*, Seiten 396–412, Paris, France, Januar 2005. Springer.

[2] Aljazzar, Husain, Holger Hermanns und Stefan Leue: *Counterexamples for Timed Probabilistic Reachability.* In: *Formal Modeling and Analysis of Timed Systems (FORMATS)*, Band 3829, Seiten 177–195. Springer, 2005.

[3] Aljazzar, Husain und Stefan Leue: *Extended Directed Search for Probabilistic Timed Reachability.* In: *Formal Modeling and Analysis of Timed Systems (FORMATS)*, Band 4202, Seiten 33–51. Springer, 2006.

[4] Andrés, Miguel E., Pedro D'Argenio und Peter van Rossum: *Significant Diagnostic Counterexamples in Probabilistic Model Checking.* In: *Haifa Verification Conference (HVC)*, Lecture Notes in Computer Science. Springer, Oktober 2008.

[5] Becker, Bernd: *Technische Informatik I.* Vorlesungsfolien zur gleichnamigen Vorlesung an der Albert-Ludwigs-Universität Freiburg, Wintersemester 2000/2001.

[6] Biere, Armin, Alessandro Cimatti, Edmund M. Clarke und Yunshan Zhu: *Symbolic Model Checking without BDDs.* In: *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Seiten 193–207, London, UK, 1999. Springer.

[7] Bollig, Beate und Ingo Wegener: *Improving the variable ordering of OBDDs is NP-complete.* IEEE Transactions on Computers, 45(9):993–1002, 1996.

[8] Bryant, Randal E.: *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, 35(8):677–691, 1986.

[9] Bryant, R.E.: *On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication.* IEEE Transactions on Computers, 40(2):205–213, 1991.

[10] Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill und L.J. Hwang: *Symbolic model checking:* 10^{20} *states and beyond.* Information and Computation, 98(2):142–170, Juni 1992. Originally presented at the 1990 Symposium on Logic in Computer Science (LICS90).

[11] Ciesinski, Frank und Marcus Grösser: *On probabilistic computation tree logic.* In: Baier, C., B.R. Haverkort, H. Hermanns, J.-P. Katoen und M. Siegle

(Herausgeber): *Validation of Stochastic Systems – A guide to Current Research*, Band 2925 der Reihe *Lecture Notes in Computer Science*, Seiten 147–188. Springer, August 2004.

[12] Clarke, Edmund M., Armin Biere, Richard Raimi und Yunshan Zhu: *Bounded Model Checking Using Satisfiability Solving.* Formal Methods in System Design, 19(1):7–34, 2001.

[13] Cook, Stephen A.: *The complexity of theorem-proving procedures.* In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, Seiten 151–158, New York, NY, USA, 1971. ACM Press.

[14] Damman, Berteun, Tingting Han und Joos-Pieter Katoen: *Regular Expressions for PCTL Counterexamples.* In: Rubino, Gerardo (Herausgeber): *5th International Conference on Quantitative Evaluation of Systems*, Seiten 179–188, Saint-Malo, France, September 2008. IEEE Computer Society Press.

[15] Davis, Martin, George Logemann und Donald Loveland: *A machine program for theorem-proving.* Communications of the ACM, 5(7):394–397, 1962.

[16] Davis, Martin und Hilary Putnam: *A Computing Procedure for Quantification Theory.* Journal of the ACM, 7(3):201–215, 1960.

[17] Eberlein, Ernst: *Einführung in die Stochastik.* Skript zur gleichnamigen Vorlesung an der Albert-Ludwigs-Universität Freiburg, Wintersemester 2001/2002.

[18] Eén, N. und N. Sörensson: *An Extensible SAT-solver.* In: *6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Band 2919 der Reihe *Lecture Notes in Computer Science*, Seiten 502–518, Santa Margherita Ligure, Italy, Mai 2004. Springer.

[19] Even, S., O. Goldreich und A. Lempel: *A randomized rotocol for signing contracts.* Communications of the ACM, 28(6):637–647, 1985.

[20] Friedman, Steven J. und Kenneth J. Supowit: *Finding the Optimal Variable Ordering for Binary Decision Diagrams.* IEEE Transactions on Computers, 39(5):710–713, 1990.

[21] Fujita, M., H. Fujisawa und Y. Matsunaga: *Variable Ordering Algorithms for Ordered Binary Decision Diagrams and their Evaluation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12(1):6–12, 1993.

[22] Fujita, M., Y. Matsunaga und T. Kakuda: *On Variable Ordering of Binary Decision Diagrams for the Application of multi-level Logic Synthesis.* In: *Proceedings of the European Conference on Design Automation (EADC)*, Seiten 50–54. IEEE Computer Society Press, 1991.

[23] Fujita, Masahiro, Patrick C. McGeer und Jerry Chih-Yuan Yang: *Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure For Matrix Representation.* Formal Methods in System Design, 10(2/3):149–169, 1997.

[24] Goldberg, E. und Y. Novikov: *BerkMin: A fast and robust SAT-Solver.* Discrete Applied Mathematics, 155(12):1549–1561, 2007.

[25] Han, Tingting und Joost-Pieter Katoen: *Counterexamples in Probabilistic Model Checking.* In: Grumberg, Orna und Michael Huth (Herausgeber): *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Band 4424 der Reihe *Lecture Notes in Computer Science*, Seiten 72–86, Braga, Portugal, März 2007. Springer.

[26] Han, Tingting und Joost-Pieter Katoen: *Providing Evidence of Likely Being on Time: Counterexample Generation for CTMC Model Checking.* In: Namjoshi, Kedar S., Tomohiro Yoneda, Teruo Higashino und Yoshio Okamura (Herausgeber): *5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Band 4762 der Reihe *Lecture Notes in Computer Science*, Seiten 331–346, Tokyo, Japan, Oktober 2007. Springer.

[27] Hansson, Hans und Bengt Jonsson: *A Logic for Reasoning about Time and Reliability.* Formal Aspects of Computing, 6(5):512–535, 1994.

[28] Hermanns, Holger, Björn Wachter und Lijun Zhang: *Probabilistic CEGAR.* In: Gupta, Aarti und Sharad Malik (Herausgeber): *International Conference on Computer Aided Verification (CAV)*, Band 5123 der Reihe *Lecture Notes in Computer Science*, Princeton, NJ, USA, 2008. Springer. Also available as technical report.

[29] Hinton, A., M. Kwiatkowska, G. Norman und D. Parker: *PRISM: A Tool for Automatic Verification of Probabilistic Systems.* In: Hermanns, H. und J. Palsberg (Herausgeber): *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Band 3920 der Reihe *Lecture Notes in Computer Science*, Seiten 441–444, Vienna, Austria, März 2006. Springer.

[30] Itai, A. und M. Rodeh: *Symmetry Breaking in Distributed Networks.* Information and Computation, 88(1):60–87, 1990.

[31] Knuth, D. und A. Yao: *Algorithms and Complexity: New Directions and Recent Results*, Kapitel The complexity of nonuniform random number generation. Academic Press, 1976.

[32] Marques-Silva, J.P.: *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms.* In: Barahona, Pedro und José Júlio Alferes (Herausgeber): *9th Portuguese Conference on Artificial Intelligence (EPIA)*, Band 1695 der Reihe *Lecture Notes in Computer Science*, Seiten 62–74, Évora, Portugal, September 1999. Springer.

[33] Marques-Silva, J.P. und K.A. Sakallah: *GRASP – A New Search Algorithm for Satisfiability.* In: *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, Seiten 220–227. IEEE Computer Society, 1997.

[34] McMillan, Kenneth L.: *Interpolation and SAT-Based Model Checking.* In: Jr., Warren A. Hunt und Fabio Somenzi (Herausgeber): *15th International Conference on Computer-Aided Verification (CAV)*, Band 2725 der Reihe *Lecture Notes in Computer Science*, Seiten 1–13, Boulder, CO, USA, Juli 2003. Springer.

[35] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang und S. Malik: *Chaff: engineering an efficient SAT solver.* In: *Proceedings of the Design Automation Conference (DAC)*, Seiten 530–535. IEEE Computer Society, 2001.

[36] Norman, G. und V. Shmatikov: *Analysis of probabilistic contract signing.* Journal of Computer Security, 14(6):561–589, 2006.

[37] Parker, Dave: *Implementation of Symbolic Model Checking for Probabilistic Systems.* Doktorarbeit, University of Birmingham, 2002.

[38] *PRISM Homepage.* http://www.prismmodelchecker.org.

[39] Reiter, M.K. und A.D. Rubin: *Crowds: Anonymity for Web transactions.* ACM Transactions on Information and System Security (TISSEC), 1(1):66–92, 1998.

[40] Rudell, Richard: *Dynamic Variable Ordering for Ordered Binary Decision Diagrams.* In: Lightner, Michael R. und Jochen A. G. Jess (Herausgeber): *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Seiten 42–47, Santa Clara, CA, USA, November 1993. IEEE Computer Society.

[41] Shannon, C.E.: *A symbolic analysis of relay and switching circuits.* Massachusetts Institute of Technology, 1940.

[42] Sheeran, Mary, Satnam Singh und Gunnar Stålmarck: *Checking Safety Properties Using Induction and a SAT-Solver.* In: Jr., Warren A. Hunt und Steven D. Johnson (Herausgeber): *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Band 1954 der Reihe *Lecture Notes in Computer Science*, Seiten 108–125, Austin, Texas, USA, November 2000. Springer.

[43] Somenzi, F.: *CUDD: CU Decision Diagram Package Release 2.4.1.* University of Colorado at Boulder, 2005.

[44] Strichman, Ofer: *Accelerating Bounded Model Checking of Safety Properties.* Formal Methods in System Design, 24(1):5–24, 2004.

[45] Tseitin, G. S.: *On the complexity of derivation in propositional calculus.* Studies in Constructive Mathematics and Mathematical Logic, Part 2, Seiten 115–125, 1970.

[46] Zhang, H.: *SATO: An Efficient Propositional Prover.* In: *13th International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, Seiten 272–275. Springer, 1997.

[47] Zhang, H. und M.E. Stickel: *An efficient algorithm for unit propagation.* International Symposium on Artificial Intelligence and Mathematics, Seiten 166–169, 1996.

[48] Zhang, Linato und Sharad Malik: *The Quest for Efficient Boolean Satisfiability Solvers.* In: *18th International Conference on Automated Deduction (CADE)*, Band 2392 der Reihe *Lecture Notes in Computer Science*, Seiten 295–313, Copenhagen, Denmark, Juli 2002. Springer.