Model Checking mit SPIN

Sabine Daniela Bauer
Seminar Formal Methods for Fun and Profit

Institut für Informatik
SS 05

1 Einleitung

Programme sollen aus vielerlei Gründen fehlerfrei arbeiten. Entweder weil sie Einsatz in kritischen Umgebungen finden, wie z.B. im Flugzeug oder in einem Atomkraftwerk, wo eine Fehlfunktion unabsehbare Folgen haben kann, oder einfach deshalb, weil nachträgliche Fehlerkorrekturen sehr kostenintensiv sind. Zudem legen die Anwender immer mehr wert darauf, dass die Software, die sie erwerben, auch einwandfrei funktioniert. Es gibt verschiedene Methoden, um Programme auf ihre Korrektheit zu überprüfen. So kann man z.B. versuchen, durch einfaches „Testen“ alle Eingabekonstellationen durchzuspielen, so dass nach Möglichkeit alle Fehler gefunden werden. Das Problem ist nur, dass bei dieser Methode schon einmal ein vorhandener Fehler nicht gefunden wird, weil z.B. eine mögliche Eingabekonstellation übersehen wurde. Zudem ist das „Testen“ sehr zeitintensiv und damit auch sehr teuer. An dieser Stelle setzt die sogenannte formale Verifikation an, die mit Hilfe logischer Formeln die Korrektheit bzw. durch ein Gegenbeispiel die Unkorrektheit beweist [1]. Bei der formalen Verifikation findet entweder das deduktive Vorgehen oder das hier behandelte Model Checking Anwendung. Welche der beiden Varianten eingesetzt wird, hängt von dem zu verifizierenden Programm ab. Denn bei der deduktiven Verifikation wird das terminierende Programm durch Vor- und Nachbedingungen sowie Zusicherungen formalisiert. Model Checking dagegen ist konzipiert für nicht terminierende Computersysteme, die durch temporallogische Formeln wie z.B. „Jede Anfrage wird irgendwann beantwortet“ oder „Niemals sind zwei Prozesse gleichzeitig in ihrem kritischen Abschnitt“ beschrieben werden [2]. Im Grunde wird dann auch hier lediglich das System darauf getestet, ob z.B. die Formel „Jede Anfrage wird irgendwann beantwortet“ immer zutrifft. Nur läuft hier das „Testen“ durch den Model Checker automatisch ab und ist damit nicht mehr so teuer und zusätzlich noch wesentlich genauer. Durch Model Checking kann sowohl Soft- als auch Hardware verifiziert werden. Im weiteren Text wird dies nicht unterschieden, sondern einfach vom System geredet.

2 Grundlagen des Model Checking

2.1 Das Model Checking-Problem

Das zu verifizierende System wird beim Model Checking als Kripke-Struktur dargestellt. Die Anforderungen, die das System erfüllen muss, werden mit Hilfe der Linear Temporal Logic (LTL) formuliert. Sowohl die Kripke-Struktur als auch die LTL-Formeln werden einem vollautomatischen Verifikationstool, dem sogenannten Model Checker, übergeben. Als Ergebnis bekommt man entweder, dass das System alle Anforderungen erfüllt oder, falls dies nicht der Fall sein sollte, einen Hinweis darauf, wo der Fehler zu finden ist.

2.2 Kripke-Struktur

Bei einer Kripke-Struktur handelt es sich im Grunde um einen endlichen Automaten. Es fehlt allerdings das Ein- und Ausgabealphabet, stattdessen ist ei-

ne Beschriftungsfunktion vorhanden. Die Elemente der Kripke-Struktur $M = (S, S_0, R, L)$ sind folgendermaßen definiert [3]:

- S ist eine endliche Menge an Zuständen
- S_0 ist der Anfangszustand
- $R \subseteq S \times S$ ist eine Translationsrelation, wo für jeden Zustand $s \in S$ angegeben wird, ob und welche Nachfolgezustände existieren.
- $L : S \rightarrow 2^{\{p_1,...,p_n\}}$ ist eine Beschriftungsfunktion, die jedem Zustand die dort geltenden atomaren Aussagen (Eigenschaften) aus der Menge $\{p_1, ..., p_n\}$ zuweist.

Beispiel
In diesem Beispiel [4] handelt es sich um eine Kripke-Struktur $M = (S, S_0, R, L)$

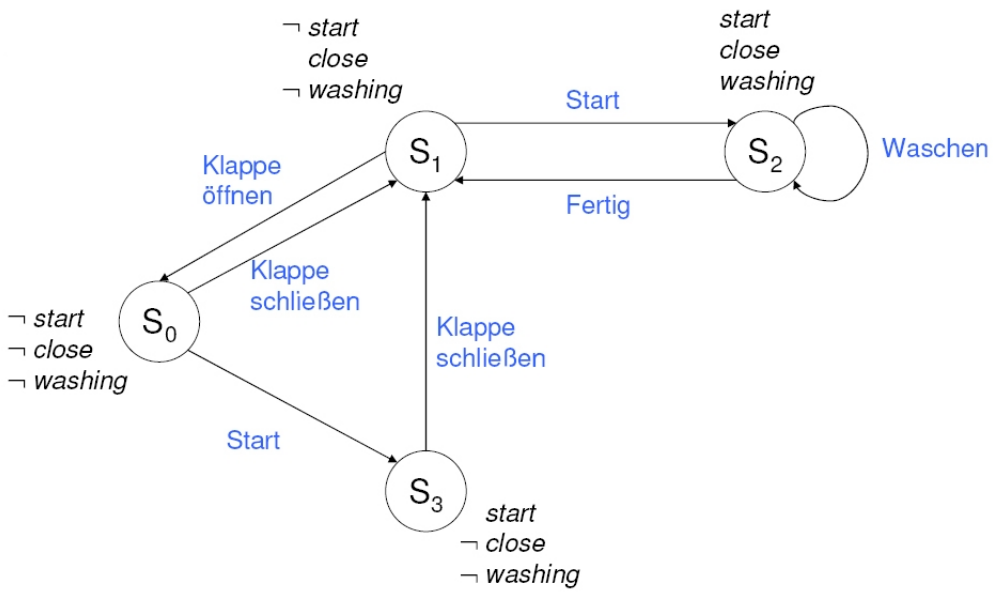

Abbildung 1. Kripke-Struktur [4]

mit

- $S = \{S_0, S_1, S_2, S_3\}$
- S_0 als Anfangszustand
- $R = \{(S_0, S_1), (S_0, S_3), (S_1, S_0), (S_1, S_2), (S_2, S_1), (S_2, S_2), (S_3, S_1)\}$
- $L(S_0) = \{\neg close, \neg start, \neg washing\}$
  $L(S_1) = \{close, \neg start, \neg washing\}$
  $L(S_2) = \{start, close, washing\}$
  $L(S_3) = \{\neg close, start, \neg washing\}$

Die Waschmaschine, die diese Kripke-Struktur beschreibt, soll folgendermaßen arbeiten: Anfangs ist die Waschmaschine im Ruhezustand S_0, d.h. sie wäscht nicht, der Startknopf wurde nicht gedrückt und die Klappe ist offen. Schließt man nun die Klappe, geht sie in den Zustand S_1 über, wo nicht gewaschen wird, der Startknopf nicht gedrückt wurde, die Klappe aber geschlossen ist. Drückt man jetzt den Startknopf, fängt die Maschine an zu waschen und ist damit im Zustand S_2. Dort bleibt sie solange, bis der Waschvorgang beendet ist. Dann tritt wieder der Zustand S_1 ein, wo man die Klappe öffnen kann. Tut man dies, befindet sich die Maschine wieder im Ruhezustand S_0. Falls der Benutzer aus Versehen den Startknopf drückt, obwohl die Klappe noch offen ist, soll die Maschine natürlich keinesfalls anfangen zu waschen. Daher gibt es den Zustand S_3, in welchem dementsprechend die Maschine nicht wäscht und die Klappe offen ist, der Startknopf aber betätigt wurde. Nun muss als erstes die Klappe geschlossen werden, damit die Waschmaschine in Zustand S_1 wechselt und sie nach erneutem Drücken des Startknopfes anfängt zu waschen [4].

2.3 Linear Temporal Logic

Nachdem anhand eines Beispieles gezeigt wurde, wie die Kripke-Struktur eines zu verifizierenden Systemes ausschaut, sollen nun Anforderungen an das System in der Spezifikationssprache „Linear Temporal Logic“ (LTL) vorgeführt werden. Der Name von LTL ist darauf zurückzuführen, dass mit Hilfe dieser Logik Aussagen über das zeitliche Auftreten von Ereignissen entlang eines ausgewählten Pfades in der Kripke-Struktur getroffen werden. Ein Pfad durch die Kripke-Struktur ist eine endliche Sequenz von Zuständen: $n = S_1 S_2 ...$ mit $S_i \in S$. Alle Pfade müssen der Anforderung gerecht werden, dass

- ein unerwünschter Zustand nie erreicht wird (Sicherheitsbedingung)
- ein erwünschter Zustand irgendwann erreicht wird (Lebendigkeitsbedingung).

LTL baut dabei auf der Aussagenlogik auf und hat folgende fünf grundlegende Operatoren:

- $\mathbf{X}p$ (next time): Im nächsten Zustand muss die angegebene Eigenschaft erfüllt sein.
- $\mathbf{F}p$ (in the future): Irgendwann muss die angegebene Eigenschaft erfüllt sein.
- $\mathbf{G}p$ (globally): In jedem Zustand entlang des Pfades muss die angegebene Eigenschaft erfüllt sein.
- $p\mathbf{U}q$ (until): Falls es einen Zustand gibt, in dem q gilt, dann muss in jedem vorherigen Zustand p erfüllt sein.
- $q\mathbf{R}p$ (release): p gilt in allen Zuständen bis einschließlich dem Zustand, in dem auch q erfüllt ist.

Beispiel
Beispielanforderungen für unsere Waschmaschinen - Kripke-Struktur wären die folgenden:

- $Fwashing$

Dies bedeutet, dass irgendwann die Eigenschaft *washing* erfüllt sein muss. Dies ist ja auch sehr einleuchtend, denn eine Waschmaschine die nie waschen würde, wäre schlicht und ergreifend unbrauchbar. Diese Anforderung wäre in diesem Fall also sogar eine Lebendigkeitsbedingung.

- $G(washing \rightarrow Xclose)$

Immer wenn die Maschine wäscht, ist im nächsten Zustand die Klappe geschlossen. Entweder weil immer noch gewaschen wird oder die Maschine ist fertig und der Benutzer soll nun die Klappe öffnen.

- $G(\neg(start \vee close \vee washing)) \rightarrow X(start \vee close)$

In dieser Anforderung sind nun auch die Bausteine der Aussagenlogik zu sehen. So ist es möglich, mit Hilfe der De Morganschen Regel $\neg(start \vee close \vee washing)$ in $\neg start \wedge \neg close \wedge \neg washing$ umzuwandeln. Diese Eigenschaften kann man nun leicht dem Startzustand S_0 zuordnen. Verfolgt man jetzt die vom Startzustand ausgehenden Pfeile, sieht man, dass der nächste Zustand entweder die Eigenschaft *start* erfüllt oder die Eigenschaft *close* und dies ist genau, dass, was in diesem Term gefordert wird.

2.4 Arbeitsweise eines Model Checkers

Das als Kripke-Struktur modellierte System und die LTL formulierten Anforderungen werden nun dem Model Checker übergeben. Dieser benutzt zur Verifikation Büchi-Automaten. Büchi-Automaten arbeiten auf unendlichen Wörtern, was beim Model Checking unbedingt nötig ist, denn dort werden ja Systeme verifiziert, die nicht terminieren. Zunächst müssen sowohl die Kripke-Struktur als auch die LTL-Formel in Büchi-Automaten dargestellt werden. Bei der Kripke-Struktur ist dies leicht, da diese einfach als Büchi-Automat aufgefasst werden kann. Das einzige, was sich dadurch ändert, ist, wann der Automat eine Eingabe akzeptiert. Ein Büchi-Automat akzepiert dann, wenn der Endzustand vom Startzustand aus zu erreichen ist, und wenn der Enzustand dann in einem Zyklus unendlich oft durchlaufen wird. Bei der LTL-Formel modelliert man die negierte Form, also $\neg\varphi$, als Büchi-Automaten. Dies macht man, weil alle Pfade im Automat M φ erfüllen müssen. Gibt es aber einen Pfad, der $\neg\varphi$ erfüllt, verletzt er damit automatisch φ. Da in der Regel die meisten Anforderungen von dem System erfüllt werden, ist es vom Aufwand her gesehen besser, die Pfade zu suchen, welche φ verletzten oder $\neg\varphi$ erfüllen. Deshalb modelliert man den Büchi-Automaten für $\neg\varphi$ und bildet dann den Durchschnitt der beiden Automaten. Es gilt dabei nun der folgende Satz.

Satz 1 *φ gilt in M genau dann, wenn die akzeptierte Sprache leer ist.*

Akzeptiert also der resultierende Automat die leere Sprache erfüllt M φ, andernfalls enthält er diejenigen Pfade von M, die φ verletzen [1].

3 Der Model Checker „SPIN“

3.1 Einführung

Hier wird nun der häufig eingesetzte Model Checker „SPIN“ präsentiert. SPIN steht für „Simple Promela Interpreter“. Dieser Name rührt von der C-ähnlichen Sprache Promela her, in welcher das zu verifizierende System beschrieben wird. SPIN wurde von Gerard Holzmann in den Bell Laboratories entwickelt und bekam 2001 den „Software System Award“ verliehen. Haupteinsatzgebiet von SPIN ist die Analyse von Kommunikationsprotokollen [5]. Der Ablauf des SPIN-Programmes ist in Abbildung 2 zu sehen.

Der Benutzer schreibt sein System in Promela und die zu erfüllenden Anfor-

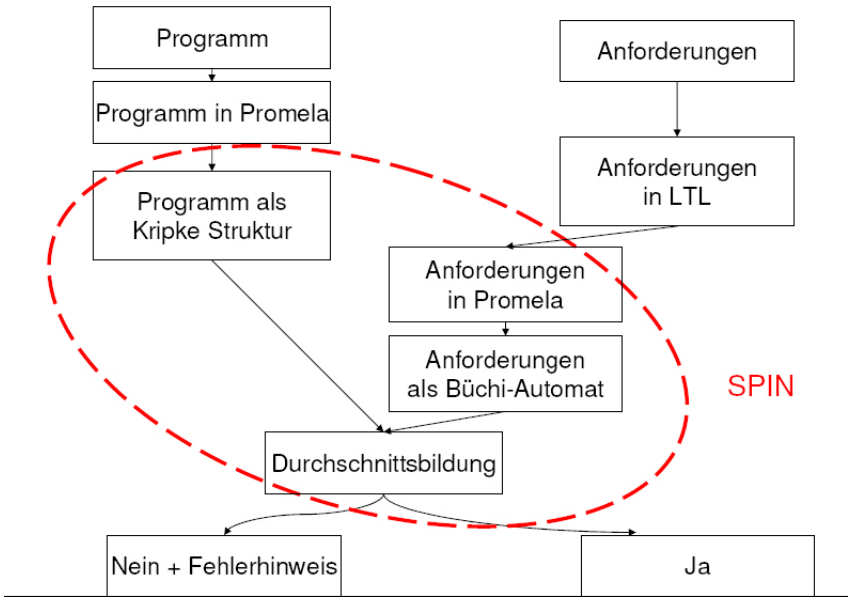

Abbildung 2. Spin

derungen in LTL, wobei in SPIN die LTL-Syntax von der gängigen Norm etwas abweicht. Sowohl der Promelacode als auch die LTL-Anforderungen werden dann an SPIN übergeben. SPIN modelliert aus dem Promelacode den Büchi-Automaten und übersetzt die LTL-Anforderungen in Promela, so dass sie dann

ebenfalls als Automat dargestellt werden können. Im letzten Schritt erfolgt die Durchschnittsbildung und der Benutzer bekommt das Ergebnis der Verifikation mitgeteilt [6].

3.2 Promela

Promela ist ein Akronym für „Process/Protocol Meta Language“. Die Beschreibung des Systems in Promela soll eine Vereinfachung für den Benutzer darstellen, da es relativ schwierig ist, das System direkt als Kripke-Struktur zu modellieren. So ist Promela dementsprechend auch keine Implementationssprache, sondern eine Systembeschreibungssprache. Um dies zu ermöglichen, liegt der Schwerpunkt der Sprache bei der Modellierung von Prozesssynchronisationen und nicht bei irgendwelchen Berechnungen. Weiterhin ist die Sprache darauf ausgelegt, simultane Softwaresysteme zu beschreiben und nicht dafür gedacht Hardware-Stromkreisen darzustellen. Um einen Eindruck über die Sprache Promela zu vermitteln und zu zeigen, wie die Modellierung eines Programmes in Promela funktioniert, wird an dieser Stelle ein kurzes Promela-System erklärt [6]. Das Beispielprogramm (Abbildung 3) hat zwei Prozesse, welche sich gegensei-

```
mtype = { P, C };
mtype turn = P;
active proctype producer()
{
        do
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        od
}
active proctype consumer()
{
        do
        :: (turn == C) ->
                printf("Consume\n");
                turn = P
        od
}
```

Abbildung 3. Promela Programm [6]

tig beeinflussen. Die jeweiligen Aktionen werden mit Hilfe einer gemeinsamen, globalen Variable koordiniert. In der ersten Zeile des Programmes werden die symbolischen Namen P und C deklariert. P steht für Producer (Hersteller) und C für Consumer (Konsument). Bei dieser Deklaration werden P und C eindeutige, positive Integerwerte zugewiesen, durch die sie dann intern repräsentiert werden. P und C sind vom Typ *mtype*, welcher ein Nachrichtentyp ist und in

der Regel beim Nachrichtenaustausch zwischen Prozesse eingesetzt wird. Was hier auch der Fall ist. Als nächstes folgt die globale Deklaration der Variablen *turn*, die ebenfalls vom Typ *mtype* ist. Diese Variable kann nun alle Werte annehmen, die in der Zeile zuvor in den geschweiften Klammern deklariert wurden. In diesem Fall also P oder C. Lässt man *turn* uninitialisiert, wird ihr der Wert null zugewiesen. Dies ist aber außerhalb des Bereiches der möglichen *mtype* Werte. Daher muss *turn* direkt der Wert P oder C zugewiesen werden.
Im folgenden werden zwei Prozesse initialisiert. Dabei gibt *proctype* jeweils an, dass es sich um Prozesse handelt und *producer* und *consumer* sind jeweils die Prozessnamen. Damit wäre der Prozess deklariert, würde aber nicht automatisch auch ausgeführt werden. Dazu muss er noch explizit instantiiert werden, was mit dem Prefix *active* passiert.
Die Kontrollstruktur in den beiden Prozessen ist nahezu identisch. Beide enthalten eine Schleife, die mit *do* beginnt und mit *od* endet.
Mittels :: werden Optionen angegeben, wovon mindestens eine, aber auch mehrere vorhanden sein können. Falls es mehrere gibt, wird nichtdeterministisch eine Anweisung ausgewählt, deren Guard erfüllt ist. Der Guard ist im Grunde nichts anderes als eine Abbruchbedingung. In diesem System ist jeweils nur eine Option vorhanden. Trifft z.B. $turn == P$ zu, d.h. hat *turn* gerade den Wert P, wird der String "Produce" gedruckt und durch das \n wird ein Zeilenvorschub ausgelöst. Nun wird noch *turn* die Variable C zugewiesen und damit die Schleife verlassen. Da ja nun *turn* den Wert C hat, kann der zweite Prozess ausgeführt werden. Welcher völlig analog verläuft [6].

3.3 Linear Temporal Logic in SPIN

Die Syntax der LTL-Formeln ist in SPIN, wie gesagt, etwas anders als normalerweise üblich. So wird der Operator F in SPIN durch $<>$ repräsentiert und der Operator G durch $[]$. X und U werden in SPIN genauso wie sonst auch verwandt. Der Operator R fehlt vollständig. Häufig benutzte LTL-Formeln sind die folgenden [6]:

- $[]p$

In jedem Zustand entlang des Pfades gilt p.

- $<> p$

Irgendwann muss p einmal gelten.

- $p \rightarrow <> q$

Wenn p auftritt, muss irgendwann q einmal gelten.

- $p \rightarrow qUr$

Wenn p auftritt, dann muss q in jedem Zustand erfüllt sein, bis zum ersten Mal r gilt.

– [] <> p

Zu jedem Zeitpunkt muss es einen späteren Zeitpunkt geben, wo p erfüllt ist (p gilt immer wieder).

– <> []p

Irgendwann in der Zukunft muss in jedem folgenden Zustand p erfüllt sein.

3.4 Übersetzung einer LTL-Formel in einen Büchi-Automaten

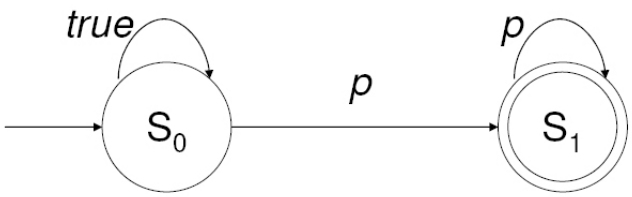

Abbildung 4. Büchi-Automat für die Formel <> []p [6]

Die LTL-Formeln, welche wie gesagt, die Anforderungen des Systems beschreiben, müssen nun wie das System selbst auch als Büchi-Automaten dargestellt werden. Da dieser Schritt nicht ganz einfach ist, wird dieser Schritt ebenfalls von dem Model Checker für uns erledigt. An dieser Stelle soll daher auch nicht gezeigt werden, wie der Model Checker aus der Formel den Automaten generiert, sondern lediglich Beispiele gezeigt werden, wie fertige Automaten für eine bestimmte Formel aussehen. Für die bereits im letzten Kapitel vorgestellte Formel <> []p ergibt sich z.B. der Büchi-Automat in Abbildung 4. Diese Formel besagte, dass irgendwann in der Zukunft nur noch p auftritt. Dies kann man hier nun gut am Automaten nachvollziehen. Zunächst bleibt der Automat für jede beliebige Eingabe in S_0, wozu auch p zählt, welches durch den mit *true* gekennzeichneten Pfeil erreicht wird. Erst irgendwann einmal tritt p auf und der Automat wechselt in den Zustand S_1. Ab da darf die Eingabe nur noch p enthalten. Ansonsten wäre die zweite Akzeptanzbedingung des Büchi-Automaten nicht erfüllt, welche fordert, dass der Endzustand in einem Zyklus immer wieder durchlaufen wird [6].

3.5 Suchverfahren

Liegt sowohl das System als auch die Anforderungen an das System als Büchi-Automaten vor, wird der Durchschnitt der beiden Automaten gebildet. Dieser

Durchschnitt enthält dann wiederum einen Büchi-Automaten, der darauf überprüft werden muss, ob er die leere Sprache akzeptiert oder nicht. Dies erfolgt bei SPIN mit Hilfe einer geschachtelten Tiefensuche. Dabei wird zunächst danach gesucht, ob vom Startzustand aus ein Endzustand erreicht werden kann und falls dies der Fall ist, ob vom Endzustand aus ein Pfad vorhanden ist, auf welchem man wieder den gleichen Endzustand erreicht. Werden Pfade vom Startzustand zum Endzustand und vom Endzustand zum Endzustand gefunden, verletzen diese Pfade die Anforderungen an das System. Werden keine Pfade gefunden, erfüllt das System alle Anforderungen [6].

4 Zusammenfassung

Nachdem Model Checking ursprüglich nur zur Hardware-Verifikation benutzt wurde, wird es inzwischen auch erfolgreich in der Software-Verifikation eingesetzt. Der hier vorgestellte Model Checker SPIN ist dabei Marktführer. Ein wesentlicher Vorteil des Model Checkings gegenüber anderen Verikationswerkzeugen, wie z.B. „das Testen“, liegt in der Automatisierung des Verifikationsprozesses. Diese ermöglicht es einem, Systeme größerer Komplexität behandeln zu können, welche mit anderen Methoden nicht oder nur mit sehr hohem zeitlichen Aufwand verifziert werden könnten. Außerdem werden im Fehlerfall unmittelbar die verletzten Eigenschaften der Spezifikation ermittelt bzw. die Zustände des Systems und der dorthin führende Fehlerpfad aufgedeckt, welche die Ursache des ermittelten Fehlers sind. „Dadurch wird die Fehleranalyse optimiert und es entsteht insgesamt ein effizienterer Entwicklungsprozess.“ [3] Nachteile des Model Checking sind, dass das System bei dieser Methode eigentlich nur eine endliche Menge an Zuständen haben darf. Systeme mit einer unendlichen Zustandsmenge können zwar auch verifiziert werden, dies wird dann aber schon deutlich schwieriger. Ein weiteres Problem liegt beim Model Checking in der Zustandsexplosion, die z.B. auftreten kann bei Nebenläufigen Prozessen, wo die Anzahl der möglichen Zustände exponentiell anwächst. Dieses Problem konnte aber inzwischen einigermaßen durch die Technik der „Binary Decision Diagrams“ (BDDs) und der „Partial Order Reduction“ bewältigt werden [3].

Literatur

1. Thomas, Prof. Dr. W. und Löding, C.: Folien zur Vorlesung „Model Checking“ im WS 03/04. (2003)
2. Unbekannt: Informationsseite der Arbeitsgruppe „Logik in der Informatik“ der Universität Bonn. (2002)
3. Fraunhofer IESE: Kompetenzzentrum mit dem Ziel Informationen zu dem Thema Software-Engineering einfach und schnell zur Verfügung zu stellen. (Ein Projekt des Bundesministeriums für Bildung und Forschung)
4. Tretow, A.: Model Checking - Grundlagen. Seminar Logik in der Informatik (2004) 3–10
5. Ciesinski, F.: Spin/Promela, Modelchecker und Spezifikationssprache. (2004)
6. Holzmann, G.: The SPIN Model Checker, Boston (2003)