Model-Checking mit SPIN

Seminar Logik und Semantik von Programmen (FGI 3)

Wintersemester 2009/2010

Markus Kuppe, Sebastian Rockel
{8kuppe,8rockel}@informatik.uni-hamburg.de

Universität Hamburg

Zusammenfassung Dieser Bericht gibt eine Einführung in das Thema Model-Checking und führt anhand praktischer Beispiele mit dem Werkzeug SPIN durch den Verifikationsprozess.
Aktuelle Beispiele zeigen uns die Unzulänglichkeiten heutiger Systeme auf. Korrektheit kann nur mittels Model-Checking erreicht werden. Dabei werden temporale Logiken verwendet um Spezifikationsanforderungen am System zu überprüfen. Der SPIN Model-Checker arbeitet mit der Linear Temporal Logic und zahlreichen Optimierungen bei der Verifikation. Einige populäre Probleme von verteilten Prozessen werden anhand von SPIN beispielhaft geprüft. Dabei wird detailliert auf die in Promela spezifizierten Modelle eingegangen um die Vorteile von SPIN als Model-Checker zu zeigen.

1 Einführung

In unserer heutigen Zeit haben Computer nicht nur in speziellen Bereichen, sondern in nahezu unseren kompletten Alltag Einzug gehalten. Dabei werden die Soft- und Hardware-Systeme immer komplexer und dadurch schwieriger vor Fehlern zu schützen. Beispiele gibt es zahlreich, wie zum Beispiel den Stromausfall in den USA 2003. Eine falsche Software-Reaktion auf starke Netzschwankungen ließ einen Großteil des Stromnetzes ausfallen. Ein anderer Fall, indem das System Kontonummern mit weniger als Zehn Stellen mit Nullen von hinten (!) auffüllte, führte 2005 zu fehlenden Auszahlungen an Arbeitslosengeld- (ALG) 2-Empfänger.
Ein Ausweg könnte sein, die Fehler zu vermeiden, was offensichtlich bei komplexen Systemen nicht möglich ist. Ein pragmatischerer Weg wäre das Testen auf Fehler, was die Fehleranzahl in der Produktversion stark vermindert aber nicht

alle Fehler finden kann. Um alle Fehler zu finden bzw. zu garantieren, dass ein System fehlerfrei ist, muss man seine Korrektheit beweisen (verifizieren). Das erreicht man mittels Model-Checking, da in der Regel ein Verifikationsproblem einem Modellprüfungsproblem entspricht.

1.1 Model-Checking

Model-Checking umfasst Techniken, die

- verifizieren, ob die (System-) Spezifikation erfüllt wird,
- automatisch arbeiten,
- die Korrektheit bzgl. der Spezifikation feststellen oder
- ein Gegenbeispiel liefern.

Beim Model-Checking werden aber auch zusätzliche „Probleme“ eingeführt, die es zu beachten gilt. Ein Modell enthält in der Regel keine „unwichtigen“ Aspekte des Systems. Es kann so zu Fehlern bei der Abstrahierung (Modell-Erstellung) kommen. Auch der Aufwand bzw. die Kosten zur Erstellung eines Modells, sowie der Spezifikation und Verifikation dürfen nicht unterschätzt werden. In der heutigen Praxis ist Model-Checking meist nur bei sehr kritischen Systemen ökonomisch (z.B. i.d. Luft- und Raumfahrt).
Model-Checker arbeiten mit temporaler Logik um nicht nur aktuelle, sondern auch zukünftige (System-) Ereignisse zu überprüfen.

1.2 CTL (Computation Tree Logic)

CTL gehört zu den temporalen Logiken und wird von einigen Model-Checkern angewandt. Dabei werden unendliche Bäume von Sequenzen betrachtet. Ihre Interpretation ist Nicht-Determinismus, also die Existenz von mehreren möglichen Entwicklungen bei einem Ereignis. Der Model-Checker prüft typischerweise, ob dieser Baum eine gegebene CTL-Formel (der Korrektheitsanforderung) erfüllt.

1.3 LTL (Linear Temporal Logic)

LTL als temporale Logik wird von SPIN verwendet. Es betrachtet (LTL-) Formeln als zeitliche Modalitäten. Statt einzelner Belegungen werden unendliche Sequenzen betrachtet. In LTL wird für das Model-Checking-Problem geprüft, ob diese Sequenz ein Modell der (LTL-) Formel ist. Für den Model-Checker bleibt dann noch die Frage zu klären, ob alle möglichen Sequenzen die gegebene

Formel erfüllen.
Im Folgenden ein paar Beispiele von häufig verwendeten LTL-Formeln, wobei p und q beliebige logische Ausdrücke sein die zu *TRUE* oder *FALSE* ausgewertet werden können.

$\Box p$	immer p
$\Diamond p$	irgendwann einmal p
$p \rightarrow \Diamond q$	wenn p, dann irgendwann q
$p \rightarrow q \mathcal{U} r$	wenn p, dann q bis r auftritt
$\Box \Diamond p$	immer irgendwann p
$\Diamond \Box p$	irgendwann immer p

2 Model-Checking mit SPIN

2.1 Grundlagen: SPIN und Promela

Der Name SPIN kommt von Simple Promela INterpreter. Wie der Name schon sagt wird Promela interpretiert. Promela ist dabei die Modellierungssprache, in der das Modell des zu prüfenden Systems spezifiziert wird. Der Name Promela kommt dabei von Process Meta Language und deutet auf die Verwendung als abstrakte Beschreibungssprache hin. Beide Begriffe werden in Abschnitt 3.1 und 3.2 weiter behandelt.
SPIN hat seine Stärken bei bestimmten Anwendungen (wie wahrscheinlich die meisten Model-Checker). Dabei handelt es sich vorwiegend um Verifikationen nebenläufiger und oder verteilter Prozesse. Beispiele sind allgemein Client-Server und Web-Applikationen, um nur eine kleine Auswahl zu nennen. In der Industrie wurde (und wird) SPIN z.B. beim Pfadplanungsmodul des NASA Deep Space One (1998-2001) oder bei der Korrektheitsprüfung von Telekommunikationsprotokollen wie UMTS verwendet.

2.2 SPIN unter der Haube

Der Verifikationsprozess in SPIN läuft nach dem im Folgenden aufgelisteten Schema ab. Er ist auch grafisch dargestellt in Abbildung 1.

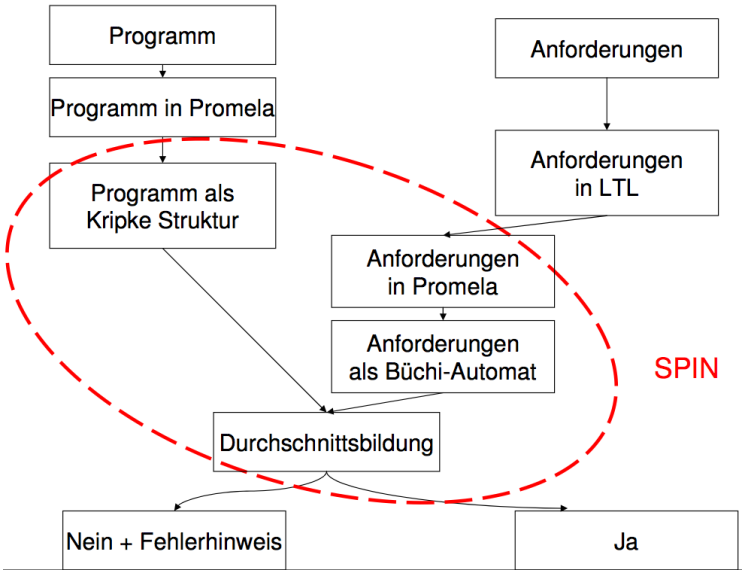


Abbildung 1. Model-Checking in SPIN

1. Jeder (in Promela modellierte) Prozess des Systems wird in einen endlichen Automaten umgewandelt.
2. Die Verschränkung aller Automaten ergibt den globalen System-Automat A_M. Dieser ist auch bekannt als *system state space*, *global reachability graph* oder Kripke-Struktur.
3. SPIN benutzt neben dem Modell die Korrektheitsanforderung als Eingabe. Diese in LTL formulierte Formel wird als (logisch negierte) Korrektheitsanforderung ($\neg\varphi$) in einen *Büchi-Automat* ($A_{\neg\varphi}$, SPIN: *Never Claim, Abschnitt 2.4*) umgewandelt und gespeichert (Abschnitt 2.3).
4. Der Durchschnitt von A_M und $A_{\neg\varphi}$ ergibt wieder einen Büchi-Automat.

Die akzeptierte Sprache des im viertem Schritt erhaltenen Büchi-Automaten ist leer, wenn die Korrektheitsanforderung an das System hält (A_M erfüllt φ) oder eben nicht und enthält dann genau die Gegenbeispiele (Sequenzen bzw. Pfade von A_M, die φ verletzen), die das System nicht erfüllt.
Zusammenfassend kann man sagen, dass in SPIN Korrektheitsanforderungen an unerwünschte (System-) Verhalten formuliert werden. Der Verifizierer (Model-Checker) prüft dann auf die Unmöglichkeit dieser oder gibt detaillierte Beispiele für solche an.

2.3 Büchi-Automat

Ein Büchi-Automat ist eine Erweiterung des endlichen Automaten um unendliche Wörter als Eingabe zu akzeptieren. Beim Model-Checking kann oft das Problem selber auf die Überprüfung auf leere Sprache von Büchi-Automaten überführt werden. Dessen Sprache ist nicht leer, wenn es einen Endzustand (s_1) gibt. Dieser muss vom Startzustand (s_0) und von sich selbst erreichbar sein (Akzeptanz-Zyklus). In Abbildung 2 wird ein Büchi-Automat für die einfache LTL-Formel $\Diamond\Box p$ dargestellt.

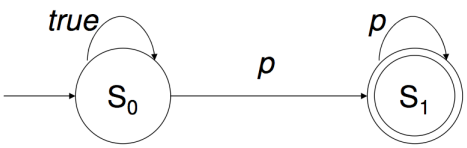


Abbildung 2. Büchi-Automat für $\Diamond\Box p$

2.4 Never Claim

Ein Never Claim ist ein Promela-Konstrukt, dass eine (negierte) LTL-Korrektheitsanforderung spezifiziert. Praktisch gesehen ist es ein in Promela repräsentierter Büchi-Automat.
Die Akzeptanz des Modells wird auf dem Ausführungsgraph des Automaten geprüft. Zusammenfassend ist ein Never Claim der Ausdruck einer negativen Korrektheitsanforderung (also das was nicht auftreten darf).
Im Promela-Beispiel (siehe Algorithmus 1) ist der Never Claim für $\neg(\Diamond\Box p)$ für die Korrektheitsanforderung $\Diamond\Box p$ (beides LTL-Formeln) aufgelistet. Abbildung 3 zeigt dafür den korrespondierenden Büchi-Automaten.

Algorithmus 1 Promela: $\Diamond\Box p$ als Never Claim: $\neg(\Diamond\Box p)$

```
never { /* !(<>[]p) */
T0_init :    /* init */
  if
  :: (!p) -> goto accept_S1
  :: (1) -> goto T0_init
  fi;
accept_S1 :    /* 1 */
  if
  :: (!p) -> goto accept_S1
  :: (1) -> goto T0_init
  fi;
}
```

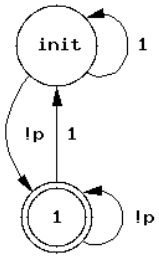


Abbildung 3. Büchi-Automat für $\neg(\Diamond\Box p)$

2.5 Zustandsraumsuche

SPIN führt eine geschachtelte Tiefensuche über dem Graphen des System-Automaten durch.

1. Dabei werden in einem ersten Durchgang alle Zustände gesucht, die vom Startzustand aus erreichbar sind.
2. Im zweiten Durchgang (deshalb geschachtelte Suche) wird nun für jeden der in 1. gefundenen Zustände geprüft, ob dieser von sich selbst aus erreichbar ist (siehe Abschnitt 2.3).

Die Suche terminiert, wenn ein Akzeptanz-Zyklus gefunden wurde (also ein Gegenbeispiel zur Korrektheitsanforderung bzw. Pfade, die φ verletzen). Das wird

auch „on-the-fly“-Prüfung des Zustandsraumes genannt, da nicht erst eine erschöpfenden Suche über den vollständigen Zustandsraum nötig ist. Dies kann jedoch trotzdem der Fall sein, wenn kein Gegenbeispiel gefunden wird (also A_M erfüllt φ). Es wird allgemein mindestens ein Zyklus gefunden, wenn auch mindestens einer existiert.
Abbildung 4 zeigt den beispielhaften Verlauf einer solchen Tiefensuche. Dabei wird ausgehend von s_0 erst s_3 gefunden, jedoch durch einen fehlenden Akzeptanz-Zyklus abgewiesen. Weiter über s_1, wird s_2 gefunden und als gültigen Endzustand akzeptiert.

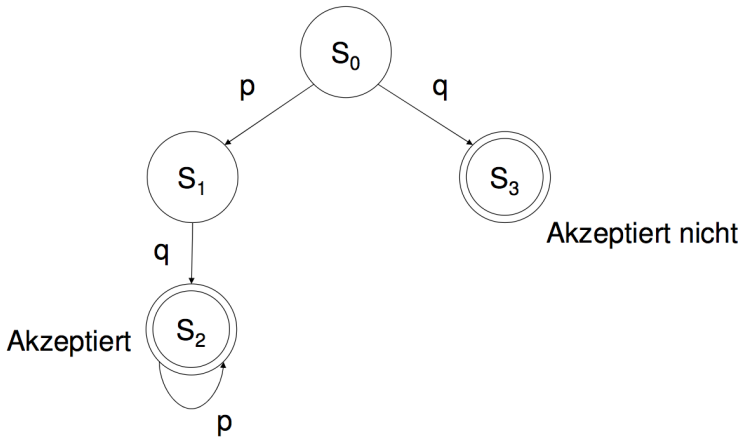


Abbildung 4. Geschachtelte Tiefensuche

2.6 Komplexitätsverwaltung

Ein Problem für jeden Model-Checker ist die so genannte Zustandsexplosion. Die Anzahl der Zustände, die geprüft werden müssen, hängt unter anderem von der Anzahl der Variablen, deren Wertebereich und der Anzahl nebenläufiger Prozesse im System ab.
Im Abschnitt 1.1 wurde bereits erwähnt, dass Modelle möglichst keine unwichtigen Aspekte des Systems enthalten sollte. Dies ist in erster Linie auf die Performance des Model-Checkers über dem Zustandsraum zurück zu führen.
Auswege aus dem „Fluch der Zustände“ sind Modelle:

- mit wenigeren Prozessen,
- die keine unnötigen Variablen enthalten,

- die Variablen mit möglichst enge Typisierung verwenden und
- besonders in SPIN Kanäle (siehe Abschnitt 3.2 auf Seite 14) mit möglichst geringer Kapazität verwenden.

Es kann jedoch vorkommen, dass Modelle geprüft werden, die immer noch „zu groß“ für den Model-Checker sind. SPIN hat weitere sehr effiziente Mechanismen eingebaut um immer möglichst ressourcen-sparend Modelle zu prüfen.

Partial-Order-Reduction POR macht sich eine bei LTL-Formeln zugrunde liegende Beobachtung zu nutze. Nämlich, dass deren Gültigkeit oft unempfindlich ist gegenüber der Reihenfolge von Ereignissen, die nebenläufig und unabhängig sind, während der geschachtelten Tiefensuche. Dadurch kann die Anzahl der zu überprüfenden, erreichbaren Zustände stark reduziert werden (für eine vollständige Verifikation).
Es wird ein reduzierte Zustandsraum generiert, der nur Repräsentationen von Klassen von Ausführungssequenzen enthält. Diese sind ununterscheidbar für die Korrektheitsanforderung. POR erfolgt statisch (im Gegensatz zu Binary-Decision-Diagrams, BDDs) und hat dadurch keine Laufzeitverzögerungen zur Folge. Die Identifizierung der Fälle findet vor der Verifikation selber statt. Während der Verifikation findet dann nur noch die Anwendung der bereits gefundenen POR-Regeln statt. Die Speicher- und Laufzeitersparnis beträgt 10% bis 90% laut [5].

Weitere Optimierungen Neben POR finden in SPIN eine Reihe weiterer Optimierungen statt. Genauer ist es in [5] beschrieben.

- Die Zustands- (Vektor-) Kompression
  Erreicht wird eine typische Speicherersparnis von mehr als 50% bei ca. 15% Laufzeitkosten.
- Minimaler Automat
  Dabei werden die Zustandsvektoren ähnlich den BDDs verwaltet. Das spart typischerweise Speicher auf Kosten der Laufzeit.
- Bit-Zustands-Hashing
  Die verwandte Methode heißt „Supertrace“ und bring im Mittel mehr als 90% Speicherersparnis auf Kosten (ca. 1%) der erschöpfenden Graphensuche. Bei mangelndem Speicher sehr zu empfehlen aber mit Vorsicht (aufgrund eben der, wenn auch kleinen, potentiellen Gefahr, widersprüchliche Ereignisse nicht zu finden).

– Mehr Speicher für die Hash-Tabelle
  Das führt zu weniger Hashkonflikten und damit zu einer wesentlich schnelleren Zustandsraumsuche.

3 SPIN

Nachdem in den vorherigen Abschnitten das Model-Checking grundsätzlich behandelt und auf SPIN nur oberflächlich eingegangen wurde, soll in den folgenden Absätzen das SPIN-Projekt und Werkzeug vorgestellt, die Sprache Promela eingeführt und anhand des Beispiels des kritischen Abschnitts die Verifikationsmöglichkeiten mit SPIN erläutert werden.

3.1 Simple Promela Interpreter

Der Simple Promela INterpreter (SPIN) ist ein von Gerard J. Holzmann entwickeltes Werkzeug zur Modell-Verifikation, dass in der Programmiersprache C programmiert ist. Es ist während Holzmanns Tätigkeit bei den Bell-Labs – heute Lucent – entstanden und wird seitdem unter der Adresse `http://spinroot.com` als Open-Source-Software von einer Gruppe von Wissenschaftlern und Entwicklern weiterentwickelt. Die zugrunde liegende Lizenz erlaubt ausdrücklich eine wissenschaftliche/lehrende Nutzung.
2002 erhielt SPIN den Association for Computing Machinery (ACM) Software System Award, was SPIN auf eine Stufe mit Unix, TEX, Smalltalk und verschiedenen anderen herausragenden Innovationen in der Software-Entwicklung stellt. Damit wird die Relevanz des SPIN-Werkzeugs deutlich. Aber auch der erfolgreiche Einsatz in diversen Projekten weist auf die Reife des Werkzeugs hin (vgl. 2.1).
Technisch betrachtet ist SPIN ein Generator, der aus einem gegebenen Promela-Programm einen Verifizierer erzeugt. Daraus kann mit einem (beliebigen) C-Compiler ein ausführbarer Verifizierer erstellt werden, der zum einen das Modell und zum anderen Funktionalität zur Verifikation beinhaltet. Der Verifizierer stellt eine Kommandozeilen-Benutzerschnittstelle zur Verfügung über die ein Never Claim (vgl. Abschnitt 2.4) spezifiziert und daran das Modell verifiziert werden kann. Sofern eine Verifikation ein positives Ergebnis liefert, wird diese Tatsache in einem Report ausgegeben. Kann SPIN jedoch eine Verletzung des Never Claims erzeugen, wird eine Trail-Datei geschrieben, die den konfliktären

Zustandsgraphen repräsentiert. Dieser Sachverhalt ist in Abbildung 5 konzeptionell dargestellt.

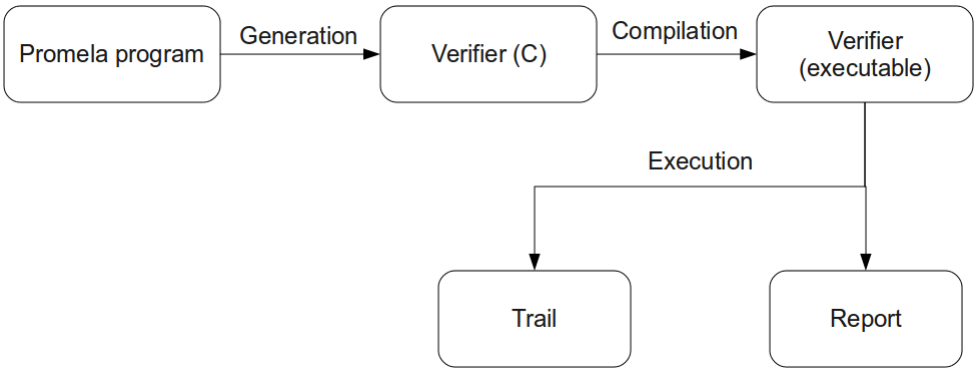


Abbildung 5. SPIN Architektur

Tools Zur komfortablen Handhabung von Promela-Programmen und Verifikationen existiert eine Reihe von grafischen Benutzerschnittstellen, die SPIN komplementieren. Die (offizielle) SPIN Distribution beinhaltet ein in TCL geschriebene und daher plattformunabhängige Oberfläche. Im Rahmen dieser Arbeit wurde jedoch das von Mordechai Ben-Ari entwickelte jSpin[1] näher betrachtet, da es zu diesem Zeitpunkt einer aktiveren Entwicklung unterlag. Darüber hinaus existiert mit [1] eine ausführliche Einführung und Dokumentation für jSpin. Funktional bietet es des Weiteren eine Integration von SpinSpider - einem Post-Prozessor von Trail-Dateien - das eine Visualisierung von Zustands-Übergangs-Diagrammen mit Hilfe der dot-Beschreibungssprache (siehe [10]) erlaubt. Diese grafische Repräsentation vereinfacht die Fehlersuche. Es sei hier nur am Rande erwähnt, dass mit EUI eine jSpin Variante für den Erigone Model Checker existiert (vgl. `http://code.google.com/p/erigone/`).
Abbildung 6 zeigt die Oberfläche von jSpin. Links unterhalb der Menü- und Werkzeugleiste findet sich das (Promela) Modell. Im rechten Teil eine textuelle Darstellung einer Verifikation, die die Zustände und die korrespondierenden Variablenbelegungen wiedergibt. Im unteren horizontal angeordneten Bereich sind

[1] jSpin kann unter `http://code.google.com/p/jspin/` frei herunter geladen werden und unterliegt der GNU General Public License.

die Kommandozeilen-Parameter für SPIN sichtbar. jSpin vereint in der grafischen Oberfläche alle Funktionalitäten von SPIN.

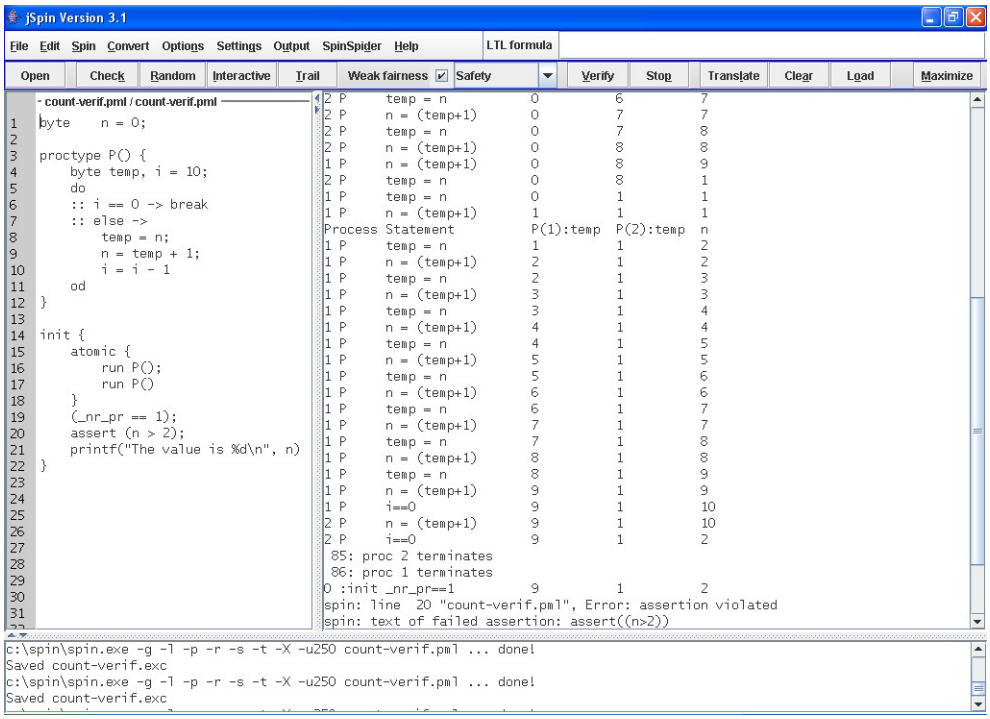


Abbildung 6. Oberfläche von jSpin

3.2 Promela

Die zur Modellierung in SPIN eingesetzte Sprache ist die imperative Process Meta Language (Promela), die eigens für SPIN entwickelt wurde. Obwohl sich ihre Syntax an C anlehnt, handelt es sich bei Promela um eine reine Modellierungssprache für Verifikationszwecke. Das bedeutet, dass sie auf Grund von diversen fehlenden Sprachkonstrukten als Implementierungssprache wie C/C++ oder Java ausscheidet. Dies ist vor dem Hintergrund des Einsatzes jedoch intendiert, da somit zur Reduktion des Zustandsraums und einer Vermeidung der Zustandsexplosion (vgl. Abschnitt 2.6) beigetragen wird. Trotzdem wird eine Einbettung von C-Programmteilen unterstützt, was der vereinfachten Gewinnung von Promela-Modellen aus konkreten Implementierung dient. Allerdings kann SPIN keinerlei Aussagen über die Korrektheit der eingebetteten Programmteile tref-

fen, da diese Code-Blöcke als ein einziger Zustand interpretiert werden.
Die Haupt-Sprachmittel, mit denen in Promela Modelle erstellt werden, sind *Typen/Variablen*, *Arrays, Operatoren, Prozesse, Atomic*-Blöcke, *Sprungmarken*, *Schleifen, Asserts* und *Kanäle*.
Tabelle 1 fasst die gültigen vordefinierten Datentypen der Sprache zusammen.

Typ	Werte	Größe (bits)
bit, bool	0,1,false,true	1
byte	0..255	8
short	-32768..32768	16
int	-2^{31}.. 2^{31}-1	32
unsigned	0..2^n -1	<= 32

Tabelle 1. Vordefinierte numerische Datentypen in Promela

Wie bereits Eingangs erwähnt, sollte eine Variablendeklaration immer den minimalen Wertebereich nutzen. Andernfalls droht eine (unnötige) Explosion des Zustandsraums. Neben den in der Tabelle genannten Typen existieren darüber hinaus noch *chan*, *pid* und *mtype*.

chan Dient der Deklaration eines Nachrichtenkanals. Abhängig von der Kapazität handelt es sich entweder um einen Datenkanal mit synchronen Eigenschaften, oder einen asynchronen Kanal.

pid Eine Prozess-ID, die zur Identifizierung einzelner Prozesse genutzt wird. SPIN ist auf 255 simultan laufende Prozesse limitiert, weshalb *pid* auf *byte* abgebildet wird. Der Zugriff auf die aktuelle Prozess-ID erfolgt mit dem Schlüsselwort *_pid*.

mtype Symbolische Namen lassen sich in Promela zum einen über die aus C hinlänglich bekannten *defines* umsetzen. Eine Alternative ist jedoch die Nutzung eines Kürzels (mnemonic type), dessen Vorteil in der Erhaltung zur Laufzeit liegt und daher im Trace sichtbar bleibt. Der *mtype* Datentyp unterstützt neben reinen symbolischen Namen zusätzlich Aufzählungen (Enumerations).

typedef Neben den bis hierhin genannten atomaren Datentypen lässt Promela auch komplexe Datentypen zu. Die Deklaration dieser Datentypen erfolgt über das Schlüsselwort *typedef*. Ein komplexer Datentyp setzt sich wiederum aus atomaren oder anderen komplexen Datentypen zusammen.

Alle Variablen werden mit 0 initialisiert. Trotzdem ist es empfohlen, Variablen explizit zu initialisieren.
Verschiedene Variablen eines identischen Datentyps können in Promela mit den in nahezu allen Programmiersprachen existierenden *Arrays* als Sequenz zusammen gefasst werden. Der Array-Zugriff, beziehungsweise die Indizierung, ist zu C identisch. Jedoch erlaubt Promela ausschließlich eindimensionale Arrays. Sollten trotzdem mehrdimensionale Arrays notwendig sein, kann mit Hilfe des *typedef* ein spezieller Datentyp angelegt werden, der ein Array kapselt.
Die meisten der in Promela vorhandenen Operatoren und Kontrollstrukturen sind identisch zur Programmiersprache C. Daher soll im Folgenden aus Gründen des Umfangs nur das jeweilige Delta genannt werden. Im Gegensatz zu C dürfen auf Operatoren beruhende Terme keinerlei Seiteneffekt haben. Dies begründet sich darin, dass SPIN Terme nutzt, um die Ausführbarkeit von Ausdrücken zu prüfen. Weiterhin kennt Promela weder den Präfix-Dekrement- noch den Inkrement-Operator. In Bezug auf Kontrollstrukturen unterstützt Promela so genannte Wächter (Guarded Commands), deren Erfindung auf E.W. Dijkstra zurück geht (vgl. [11]). Ein Guard ist ein Tupel aus einer Prämisse, die zu wahr evaluieren muss, bevor eine Anweisung ausführbar ist. Evaluiert der Guard zu falsch, wird der Kontrollfluss ohne Ausführung der Anweisung fortgeführt. Sind mehrere alternative Guards definiert und evaluieren zu wahr, führt SPIN einen Guard bzw. die korrespondierende Anweisung nicht-deterministisch aus. Guards dienen daher auch zur Modellierung von Nicht-Determinismus in SPIN.
Neben Wächtern ist es in Promela außerdem möglich, blockierende Anweisungen zu deklarieren. Erreicht der Instruction Counter (des jeweiligen Prozesses) eine blockierende Anweisung, wir der ausführende Kontext bis zum Eintritt der Anweisung angehalten. Evaluiert die Anweisung zu einem späteren Zeitpunkt zu wahr, wir der wartenden Prozess nicht-deterministisch fortgesetzt. Blockierende Anweisungen eignen sich demnach zur Synchronisation von nebenläufigen Ereignissen.
Sollen mehrere Anweisungen zusammen gefasst werden, damit im Büchi-Automat nur ein einziger Zustand für diesen Block angelegt wird, können die Anweisungen

in einem *Atomic*-Block gruppiert werden. Es ist dabei unnötig zu sagen, dass sich die atomare Ausführung dieses Blocks in der späteren Implementierung wiederfinden muss. Andernfalls trifft selbst eine erfolgreiche Verifikation mit SPIN keinerlei Aussage über dessen Korrektheit.
Das mitunter wichtigste Sprachmittel in Promela sind Prozesse. Ein Prozess wird über das Schlüsselwort *proctype* definiert und kann ähnlich zu einer Funktion, zusätzliche Parameter erwarten. Ein Prozess stellt eine Ausführungseinheit dar, die alle zugehörigen Anweisungen im Kontext des Prozesses ausführt. Die Kommunikation zwischen mehreren Prozessen (entspricht verschiedenen Kontexten) verläuft über globale Variablen ähnlich der Inter Process Communication (IPC). Soll ein Prozess direkt zu Beginn der Verifikation durch SPIN ausgeführt werden, wird dies mit dem Schlüsselwort *active* angezeigt. Es ist dann nicht mehr nötig, einen Prozess explizit durch die *init*-Funktion zu starten.
Soll nicht nur ein nebenläufiges System mit verschiedenen Prozessen modelliert werden, kommen Nachrichtenkanäle (Channels) zum Einsatz. Ein Kanal hat, abhängig von der Deklaration, eine feste Größe – eine Kapazität, wobei der Kapazität 0 die spezielle Bedeutung eines synchronen Kanals zukommt. Kanäle mit höherer Kapazität N können bis zu N Nachrichten aufnehmen, bevor auch sie blockieren. Weiterhin wird einem Kanal ein bestimmter Nachrichten-Typ zugeordnet, der entweder einem atomaren Datentyp, oder einem Komplexen entspricht. Kanäle sind in Promela immer bidirektional, beziehungsweise legen den Sender und Empfänger nicht explizit fest. Darüber sind sie folge-erhaltend (FIFO). Über einen Laufzeit-Parameter lassen sich Kanäle dahingehend beeinflussen, dass sie bei Erreichung ihrer Kapazitätsgrenze neue Nachrichten verwerfen. Damit lassen sich nicht-zuverlässige Kanäle modellieren. Hohe Kanal-Kapazitäten trägen in starkem Maße zur Zustandsexplosion bei, da sämtliche Permutationen der Kanalbelegungen einen Zustand im Büchi-Automaten darstellen.
Neben den imperativen Sprachmitteln erlaubt Promela auch den (lesenden) Zugriff auf die bereits angesprochenen Location bzw. Instruction Counter, um den aktuellen Laufzeitzustand eines Programms abzufragen. Dies erfolgt über die Einführung von einer Art wertbasierten Marken *(Ghost Variables)*, deren Zustand wahr ist, sofern der Location Counter in der aktuellen Code-Zeile steht. Da es pro Prozess verschiedene Location Counter gibt, müssen die Ghost Variables aus dem globalen Kontext über *Prozess@Marke* referenziert werden.
Das im Anhang A zu findende Promela-Programm nutzt einige der hier vorgestellten Sprachkonstrukte und ist mit dem hier erworbenen Verständnis leicht

zu verstehen. Für einen vollständigen Überblick über Promela sei hingegen auf [6] verwiesen, dass das Referenzwerk der Sprache bildet.

3.3 Verifikation mit Assertions und LTL

Wie in Abschnitt 2.2 bereits gezeigt, setzt sich das Model-Checking nicht nur aus einer Problem-Modellierung zusammen, sondern bedarf auch einer (zu erfüllenden) Anforderung – einer Spezifikation. Diese Spezifikation kann in SPIN auf zwei verschiedene Arten angegeben werden, wobei in diesem Abschnitt zuerst auf die einfacheren Zusicherungen *(Assertions)* eingegangen werden soll. Assertions sind aus anderen Programmiersprache hinlänglich bekannte Sprachmittel, die wie regulärer Programm-Code innerhalb des Programms aufgeführt und zur Laufzeit auf Gültigkeit geprüft werden. Ist die Zusicherung ungültig, beziehungsweise wird sie verletzt, wird ein Laufzeitfehler ausgelöst und das Programm gegebenenfalls beendet.
Die Einfachheit der Assertions verursacht aber auch Einschränkungen in Bezug auf die Mächtigkeit der Verifikation. Eine Assertion prüft immer nur den aktuellen Systemzustand, in dem Moment, in dem der Instruction Counter die Assertion evaluiert. Es lassen sich aber keine Anforderungen spezifizieren, die global, beziehungsweise permanent, Gültigkeit besitzen. Mit Assertions können keine globalen (System-) Invarianten ausgedrückt werden, die in jedem Systemzustand auf Gültigkeit geprüft werden.
Globale Invarianten lassen sich daher in SPIN mit LTL-Formeln beschreiben, deren Gültigkeit, beziehungsweise Verletzung (siehe 2.4), durch den Model-Checker verifiziert werden. In dieser Seminararbeit soll dies an dem im Anhang A angefügten Peterson Algorithmus verdeutlicht werden. Jedoch sollen vorher, zum besseren Verständnis, die Anforderungen an den wechselseitigen Ausschluss formuliert und die korrespondierenden LTL-Bedingungen aufgeführt werden:

1. Der wechselseitige Ausschluss dient der Synchronisation zweier oder mehrerer zeitlich verzahnter Prozesse auf gemeinsam genutzte Betriebsmittel (z. B. Datenstrukturen, Verbindungen, Geräte usw.) um Inkonsistenzen auszuschließen. Dabei darf jeweils nur ein Prozess zur gleichen Zeit auf die Betriebsmittel zugreifen. Nur ein Prozess kann sich im kritischen Abschnitt befinden. In Bezug auf die im Abschnitt 1.3 eingeführten LTL-Operatoren drückt sich diese Anforderung aus, als:
   $\Box\neg(P \wedge Q)$, wobei P und Q stellvertretend für die Zugriffe der Prozesse P und Q auf den kritischen Abschnitt stehen.

2. Der Algorithmus zum wechselseitigen Ausschluss muss Verklemmungsfreiheit garantieren. Das heißt, die um den kritischen Abschnitt konkurrierenden Prozesse dürfen nicht gegenseitig (zyklisch) aufeinander warten. Im Allgemeinen bezeichnet man diese Anforderung als (Livelock- bzw.) Deadlock-Freiheit.

   (a) Verklemmungsfreiheit wird automatisch von SPIN überprüft, indem die Location Counter sämtlicher Prozesse bei Programm-Terminierung am Ende oder einer Ende-Marke stehen müssen. Daher ist eine Abbildung auf LTL unnötig.

3. Konkurrieren zwei oder mehr Prozesse um den kritischen Abschnitt, wird garantiert, dass jeder Prozess irgendwann Zugriff auf den kritischen Abschnitt erhält. Keiner der Prozesse kann „verhungern“ (absence of starvation).
   Diese Eigenschaft wird in LTL als $(\Box\Diamond pcs) \wedge (\Box\Diamond qcs)$ ausgedrückt, wobei pcs und qcs die im Anhang A verwendeten Ghost Variablen repräsentieren.
4. Die 3. Anforderung ist verstärkt, wenn gilt, dass beide Prozesse gleichhäufig den kritischen Abschnitt betreten sollen gdw. sie den kritischen Abschnitt betreten möchten. Diese Eigenschaft bezeichnet man allgemein als Fairness. In Bezug auf das Code-Beispiel drückt sich die Fairness-Anforderung als $\Box(ptry \rightarrow (\neg qcs\,\mathcal{U}\,(qcs\,\mathcal{U}\,(\neg qcs\,\mathcal{U}\,pcs))))$ aus. Sie bedingt, dass der Prozess Q noch exakt einmal den kritischen Abschnitt betreten kann gdw. Prozess P den kritischen Abschnitt betreten möchte ($ptry == true$). Eine äquivalente Anforderung wäre für Prozess Q notwendig, wenn auch aus dessen Sicht Fairness zu gelten hat.

Die oben genannten Anforderungen lassen sich in die zwei Kategorien Sicherheitseigenschaften und Lebendigkeitseigenschaften unterteilen. Dabei entsprechen die Anforderungen 1. und 2. den Sicherheitsanforderungen, da sie verlangen, dass ein unerwünschter Zustand niemals eintritt. Die Anforderungen 3. und 4. hingegen gelten als Lebendigkeitsanforderungen, da es sich um erwünschte Zustände, bzw. Charakteristika, des Systems handelt. Diese Unterscheidung spiegelt sich in dem durch SPIN durchzuführenden Beweis wieder, um die jeweilige Anforderungskategorie abzudecken. Als ein Gegenbeispiel für Sicherheitsanforderung dient eine endliche Zustandsabfolge, in der die Sicherheitsanforderung nicht gilt. Für den Korrektheitsbeweis von Lebendigkeitsanforderungen müssen hingegen unendliche Berechnung (Zyklen) gefunden werden, in denen die gewünschte Lebendigkeitsanforderung niemals gilt. In Abbildung 7 ist der Zustandsgraph

einer verletzenden Zustandsfolge wiedergegeben, der die Sicherheitsanforderung $\Box\neg(P \wedge Q)$ verletzt, da in Zustand $S4$ sowohl P als auch Q wahr sind. SPIN muss zur Erkennung den Zustandsraum lediglich nach diesem, dem Never Claim entsprechenden Zustand, durchsuchen. Abbildung 8 zeigt das Zustandsdiagramm einer Verletzung des Lebendigkeitsanspruchs. Hier muss der Model-Checker den verletzenden Zyklus aufspüren, in dem die Lebendigkeitsanforderung nicht gilt.
SPIN bietet verschiedene Modi, die zur Verifikation von Sicherheits- und Lebendigkeitsanforderungen dienen. Im *Safety*-Modus prüft SPIN die als LTL-Formel angegebenen Sicherheitsanforderungen und sucht dabei nicht nach Zyklen. *Acceptance* Verifikation hingegen durchsucht den Zustandsraum nach Zyklen, in denen die Lebendigkeitsanforderungen verletzt werden. Als dritter Modus kann SPIN Lebendigkeitsanforderungen auch ohne Angabe von LTL-Formeln beweisen. Dieser Modus wird als *Non-Progress* bezeichnet, bedarf aber einer Kennzeichnung der Code-Teile deren Ausführung die Lebendigkeit des Programms bedeuten durch Ghost-Variablen. SPIN versucht dann wieder, Zustandszyklen zu finden, in denen die Code-Passagen unausgeführt bleiben.

4 Zusammenfassung

Fehler in der Software- (und Hardware-) Entwicklungen in der Vielzahl heutiger immer komplexer werdenden System mehren die Akzeptanz und Unterstützung der Verifikation mit Model-Checkern gerade bei kritischen Systemen. Der größte Vorteil gegenüber klassischem Testen ist der Beweis der Korrektheit. Darüber hinaus arbeiten Model-Checker völlig automatisch. Im Fehlerfall werden die verletzenden Eigenschaften ermittelt und die Beispiele dem Benutzer zur weiteren Auswertung aufgelistet. Der Model-Checker SPIN im Speziellen ist besonders gut geeignet für reaktive, nebenläufige und verteilte Systeme. Er hat die Fähigkeit, allgemeine temporallogische Eigenschaften zu testen.
Nachteile des Model-Checking und von SPIN ergeben sich aus der notwendigen Abstraktion eines Modells über ein reales System. Denn reale Programme sind im Allgemeinen Turing mächtig und damit unentscheidbar. Model-Checking betrachtet deswegen bis heute eine Unterklasse, die endlichen Automaten/Systeme. Und trotz aller Optimierungen in SPIN kann der Zustandsraum je nach Anwendungsfall immer noch sehr groß werden, eventuell zu rechenaufwändig (Zustandsexplosion). Der pragmatische Ansatz hierzu sind effiziente Algorithmen und Datenstrukturen, sowie eine „smarte“ Erstellung des Modells.

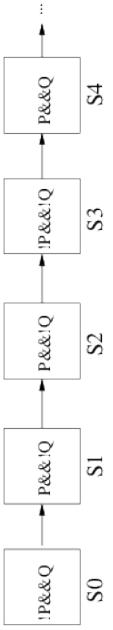
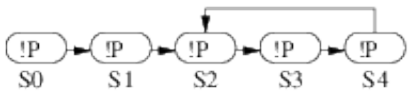


Abbildung 7. Zustandsdiagramm eines Gegenbeispiel für die Verletzung des Safety-Claims $\Box\neg(P \wedge Q)$


!P
!P
!P
!P
!P
S0
S1
S2
S3
S4

A Problem des wechselseitigen Ausschlusses (Peterson Algorithmus) modelliert in Promela

Algorithmus 2 Peterson-Algorithmus zum wechselseitigen Ausschluss

```
1  /* Copyright 2007 by Moti Ben-Ari under the GNU GPL */
2  #define ptry P@try
3  #define pcs  P@cs
4  #define qcs  Q@cs
5
6  bool wantP = false, wantQ = false;
7  byte last = 1;
8
9  active proctype P() {
10   do ::
11     wantP = true;
12     last = 1; try:
13     (wantQ == false) || (last == 2);
14 cs: wantP = false;
15   od
16 }
17
18 active proctype Q() {
19   do ::
20     wantQ = true;
21     last = 2;
22     (wantP == false) || (last == 1);
23 cs: wantQ = false;
24   od
25 }
```

Literatur

1. Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
2. René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *STTT*, 2(4):382–393, 2000.
3. Klaus Havelund. Java pathfinder a translator from java to promela. page 152. 1999.
4. Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.
5. Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
6. Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
7. Gerard J. Holzmann, Elie Najm, and Ahmed Serhrouchni. Spin model checking: An introduction. *STTT*, 2(4):321–327, 2000.
8. Moataz Kamel and Stefan Leue. Formalization and validation of the general inter-orb protocol (giop) using promela and spin. *STTT*, 2(4):394–409, 2000.
9. Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer-Verlag Berlin / Heidelberg, 1999.
10. Wikipedia. Dot (graphviz) — wikipedia, die freie enzyklopädie, 2009. [Online; Stand 2. März 2010].
11. Wikipedia. Guarded command language — wikipedia, the free encyclopedia, 2009. [Online; accessed 4-March-2010].