

Model-Checking: Einführung

Andreas Koop
Universität Paderborn
Wintersemester 2004/05
Betreuer: Prof. Dr. Gregor Engels

21. Januar 2005

Inhaltsverzeichnis

1	Einleitung	1
2	Model-Checking Grundlagen	3
2.1	Modellierung	4
2.2	Spezifikation von Eigenschaften	5
2.2.1	Die temporale Logik CTL	5
2.2.2	Vertreter der temporalen Logik	7
2.2.3	Beispiele	7
2.3	Verifikation	8
2.3.1	Symbolisch	9
2.3.2	On-the-fly	9
2.3.3	Mittels Abstraktion	9
3	Model-Checking Werkzeuge	11
3.1	SMV	11
3.1.1	Modellierung	11
3.1.2	Spezifikation	13
3.1.3	Verifikation	13
3.2	SPIN	14
3.3	Model-Checker für UML-Diagramme	15
3.3.1	vUML	15
3.3.2	Hugo/RT	16
4	Zusammenfassung	18

1 Einleitung

Ein Grundproblem in der Softwareentwicklung ist, dass fertiggestellte Software nach einem längeren Entwicklungsprozess oft nicht mehr den Anforderungen des Kunden entspricht oder gar die anfangs festgelegte Spezifikation nicht erfüllt. Daraus ergibt sich im Softwareentwicklungsprozess die wichtige Aufgabe der Verifikation. So müssen z. B. in sicherheitskritischen Systemen Fehler von vornherein ausgeschlossen werden, da ein Fehler im produktiven Einsatz weitreichende und mitunter katastrophale Schäden verursachen kann. Zudem ist eine nachträgliche Fehlerkorrektur oft nur mit hohem Aufwand und Kosten verbunden.

Konventionelle Validierungsmethoden, wie das Simulieren des Systems oder Testen von einzelnen Anwendungsfällen, bieten in solchen komplexen Systemen keine akzeptable Lösung. Erst die zusätzliche formale Verifikation des Systems, z. B. durch Model-Checking, kann die gewünschte Sicherheit und Qualität gewährleisten [CW96].

Model-Checking ist ein Verfahren zur automatischen Verifikation von Software und reaktiven Systemen [BBF⁺01]. Erstmals wurde die Technik des Model-Checking 1981 von Clarke und Emerson [CE81] und unabhängig davon von Quille und Sifakis [QS81] vorgestellt. Während es damals für einzelne Anwendungsfälle in der Industrie entwickelt wurde, so ist es heute allgemein eine Technik zum Finden schwer erkennbarer Fehler in Systemen geeigneter Größe. Entstanden ist die Idee des Model-Checking aus der Erfahrung, dass die Komplexität eines Systems und somit auch die Anzahl möglicher Fehler im Entwurf exponentiell mit der Anzahl der Systemkomponenten steigt.

Beim Model-Checking wird zunächst ein Modell der zu entwickelnden Software benötigt. Nachfolgend werden die Eigenschaften spezifiziert, die das System zu erfüllen hat, und das Modell wird dahingehend überprüft, ob es diese Eigenschaften besitzt bzw. erfüllt.

Mittlerweile existieren mächtige und etablierte Werkzeuge, wie z. B. *SMV*¹ und *SPIN*², mit denen einzelne Komponenten komplexer Softwaresysteme verifiziert werden können. Heutzutage werden solche Werkzeuge besonders in Bereichen eingesetzt, in denen Fehler im System schwere wirtschaftliche Folgen haben oder sogar Menschenleben gefährden.

¹<http://www-2.cs.cmu.edu/~modelcheck/smv.html>

²<http://netlib.bell-labs.com/netlib/spin/whatispin.html>

Aufbau der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die Grundlagen des Model-Checking erläutert. Dazu gehört die Einführung in die Kripke-Strukturen (Abschnitt 2.1), die Spezifikation von Eigenschaften mittels temporaler Logik (Abschnitt 2.2) und die eigentliche Durchführung des Verifikationsprozesses (Abschnitt 2.3).

In Kapitel 3 werden einige der bekanntesten Werkzeuge vorgestellt (SMV und SPIN). Ein ausführliches Beispiel wird mit dem Model-Checker SMV in Abschnitt 3.1 behandelt. Abschnitt 3.3 ist dann den Model-Checkern für UML-Diagramme gewidmet.

Den Abschluss der Seminararbeit bildet eine Zusammenfassung in Kapitel 4.

2 Model-Checking Grundlagen

Wie eingangs erwähnt stellt *Model-Checking* ein Verfahren zur automatisierten Verifikation von Softwaresystemen dar. Dies ist notwendig, um die Korrektheit der entwickelten Software zu gewährleisten und sicherheitskritische Systemeigenschaften sicher zu stellen.

Der prinzipielle Ablauf bei der Verifikation eines Softwaresystems mittels Model-Checking ist in Abbildung 2.1 dargestellt.

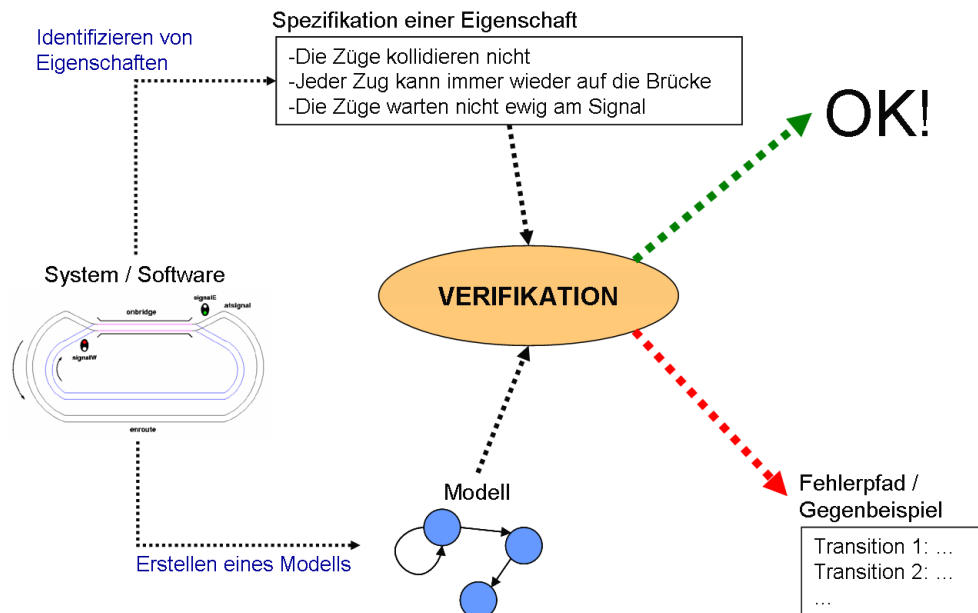


Abbildung 2.1: Verifikation mittels Model-Checking

Als Eingabe bekommt der Model-Checker ein *geeignetes Modell* des Softwaresystems sowie die zu verifizierenden Systemeigenschaften (*Spezifikationen*). Der eigentliche Verifikationsschritt besteht nun darin, zu prüfen, ob das Modell die gewünschten Eigenschaften besitzt oder nicht. Im Falle der Korrektheit gibt der Model-Checker ein „einfaches OK“ aus, ansonsten den Pfad, der zur Verletzung der zu verifizierenden Eigenschaft führte, also ein Gegenbeispiel.

Formal lässt sich Model-Checking als ein Entscheidungsproblem wie folgt definieren:

Sei M ein Modell und Φ die Spezifikation einer Eigenschaft. Erfüllt das Modell die Spezifikation? Gilt also $M \models \Phi$?

Für die Durchführung des Model-Checkings werden daher geeignete Formalismen zur Modellierung des Softwaresystems und zur Spezifikation der gewünschten Eigenschaften benötigt. In den nächsten Abschnitten wird dazu die *Kripke-Struktur* (Modell, Abschnitt 2.1) und die *CTL*¹ (Spezifikation, Abschnitt 2.2) vorgestellt. Anschließend wird die Durchführung des Model-Checkings erläutert (Abschnitt 2.3).

2.1 Modellierung

Für die Modellierung wird eine zustandsbasierte Beschreibung des zu untersuchenden Systems benötigt. Dafür eignen sich besonders die sogenannten *Kripke-Strukturen*. Sie sind mit den endlichen Automaten vergleichbar, enthalten zusätzlich an jedem Zustand jedoch noch eine Menge von Aussagen.

Formal ist eine Kripke-Struktur ein Quadrupel $M = (S, R, L, s_0)$ über den atomaren Aussagen $P = \{p_1, \dots, p_n\}$ mit

- einer endlichen *Zustandsmenge* S
- einer totalen *Übergangsrelation* $R \subseteq S \times S$
- einer *Beschriftungsfunktion* $L : S \rightarrow 2^P$, die jedem Zustand $s \in S$ eine Menge von atomaren Aussagen aus P zuordnet
- einem *Anfangszustang* $s_0 \in S$

Zur Verdeutlichung ist in Abbildung 2.2 ein Beispiel einer Kripke-Struktur für den Zugriff auf eine Ressource dargestellt. Nach einer *Anfrage* der Ressource wird der Zugriff im Zustand s_1 erlaubt (*Bestätigt*). Anschließend wird entweder direkt der Zustand s_0 angenommen oder es kann nach einem Wechsel in den Zustand s_2 beliebig lange die Aussage *Beschäftigt* gelten bevor wieder nach s_0 gewechselt wird.

Berechnungen auf einer Kripke-Struktur werden als *Pfade* bezeichnet. Ein *Pfad* π besteht dabei aus einer unendlichen Folge von Zuständen $\pi = s_0 s_1 s_2 \dots$ mit $(s_i, s_{i+1}) \in R$. Die Menge aller Pfade, ausgehend vom Anfangszustand s_0 , wird dann als *Berechnungsbaum* dargestellt.

¹Computation Tree Logic

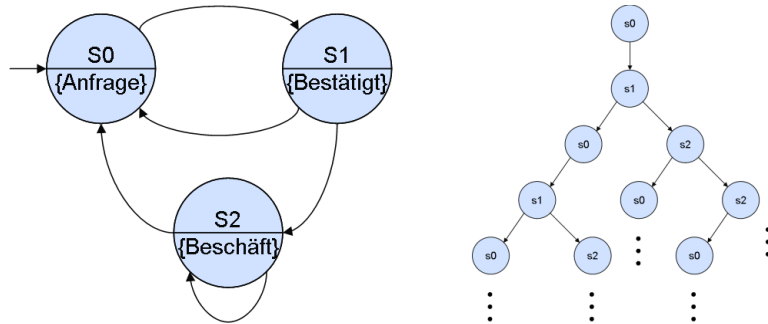


Abbildung 2.2: Beispiel einer Kripke-Struktur und dem dazugehörigen Berechnungsbaum

2.2 Spezifikation von Eigenschaften

Für die Spezifikation der gewünschten Eigenschaft Φ des Modells kommt die *temporale Logik* zum Einsatz, welche erstmals 1977 von Amir Pnuelli vorgestellt wurde [Pnu77]. Sie erweitert die klassische Aussagenlogik um zusätzliche Operatoren (Pfadquantoren, Temporaloperatoren), mit denen zeitliche Zusammenhänge spezifiziert werden können. Damit lassen sich z. B. Situationen beschreiben, in denen bestimmte Zustände irgendwann oder niemals erreicht werden.

2.2.1 Die temporale Logik CTL

Eine Variante der temporalen Logik ist die sogenannte *Computation Tree Logic (CTL)*. Sie wurde erstmalig 1982 von Clarke und Emerson [CE81] eingeführt und bietet einen ausdrucksstarken Formalismus zur Beschreibung von Berechnungsbaum-Eigenschaften. Neben den üblichen Junktoren der Aussagenlogik ist die CTL um *Pfadquantoren* und *Temporaloperatoren* erweitert. Die einzelnen CTL-Bausteine sind:

Aussagen:

Jede atomare Aussage $p_1, \dots, p_n \in P$ ist eine CTL-Formel.

Junktoren:

$\neg p_1, p_1 \vee p_2, p_1 \wedge p_2, \dots$ sind CTL-Formeln.

Pfadquantoren:

Ep (**E**xists): Aussage p gilt auf *mindestens* einem Berechnungspfad.

Ap (**A**lways): Aussage p gilt auf *allen* Berechnungspfaden – also immer.

Temporaloperatoren:

Xp (**N**eXt): Im *nächsten, direkt folgenden* Zustand gilt p .

Fp (**F**uture): Es wird *irgendwann* ein Zustand erreicht, indem p gilt.

Gp (**G**lobally): In *jedem erreichbaren* Zustand gilt p .

p_1Up_2 (**Until**): p_1 gilt in *allen* Zuständen eines Berechnungspfades bis ein Zustand erreicht wird, indem p_2 gilt.

Eine wichtige Eigenschaft von CTL ist, dass Pfadquantoren immer nur in Verbindung mit Temporaloperatoren auftreten dürfen [CGL96]. Die gängigsten Kombination dabei sind:

- **EF p**: „Es gibt *einen* Pfad, auf dem *irgendwann* **p** gilt.“
- **AF p**: „Auf *allen* Pfaden gilt *irgendwann* **p**.“
- **EG p**: „Es gibt *einen* Pfad, auf dem *immer* **p** gilt.“
- **AG p**: „Auf *allen* Pfaden gilt *immer* **p**.“

Graphisch können die eben genannten CTL-Formeln als Ausschnitt eines Berechnungsbaums wie in Abbildung 2.3 visualisiert werden.

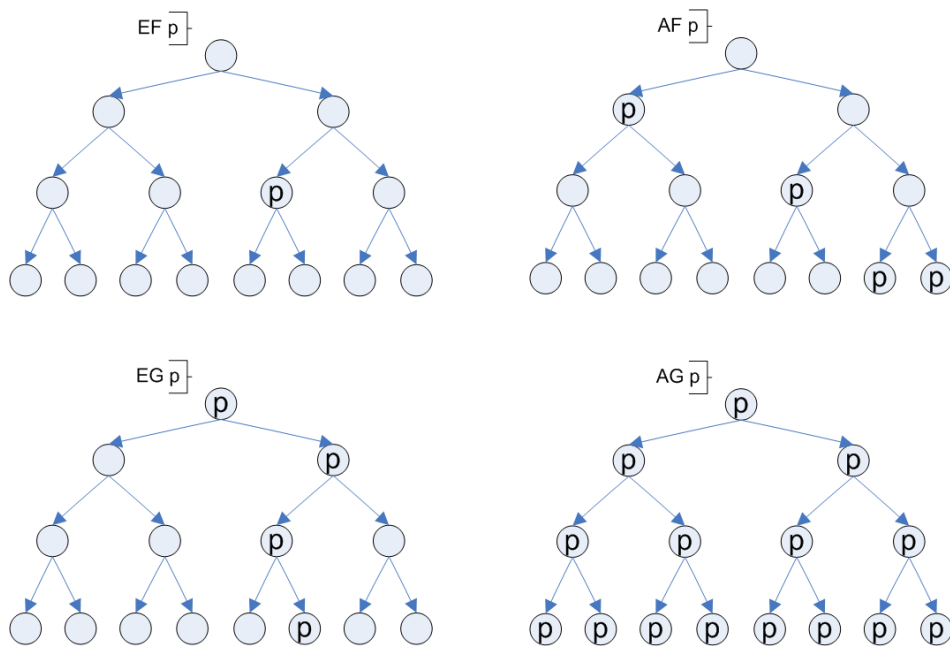


Abbildung 2.3: Grundlegende CTL-Formeln

2.2.2 Vertreter der temporalen Logik

Neben der vorgestellten CTL, gibt es weitere wichtige Varianten, die sich in der Verfügbarkeit der Temporaloperatoren und der Pfadquantoren sowie in der Semantik dieser Operatoren unterscheiden. Die wichtigsten Vertreter dabei sind

- CTL (Computation Tree Logic)
- LTL (Linear Temporal Logic)
- CTL* (eine Obermenge von CTL und LTL)

Diese Varianten unterscheiden sich vor allem in ihrer Ausdrucksmächtigkeit und kommen daher in unterschiedlichen Anwendungsgebieten zum Einsatz. Abbildung 2.4 veranschaulicht den Zusammenhang und die Ausdrucksmächtigkeit der einzelnen temporalen Logiken.

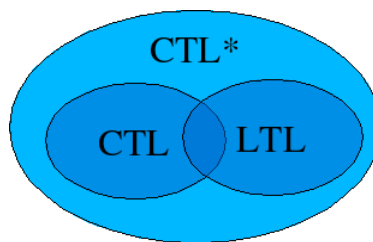


Abbildung 2.4: Ausdrucksmächtigkeit von CTL*, CTL und LTL

CTL* ist eine sehr ausdrucksstarke und komplexe Logik, von der man oft nur einen Teil benötigt, um gewisse Eigenschaften zu spezifizieren. Es ist daher vom Vorteil, sich auf eine Untermenge der temporal-logischen Ausdrücke zu beschränken. Je nach Modellgröße und zu der zu verifizierenden Eigenschaft kann der Verifikationsschritt erheblich vereinfacht werden.

Ein sehr weit verbreitete Untermenge von CTL* ist die Lineare Temporale Logik (LTL). Typisch für LTL ist der lineare Zeitbegriff – es entfallen die beiden Pfadquantoren **A** und **E**. Das Systemverhalten kann dann durch eine lineare Folge von Zuständen dargestellt werden.

2.2.3 Beispiele

Für das Beispiel aus Abbildung 2.2 können mit CTL folgende Eigenschaften formuliert werden:

- **AG**(Anfrage \Rightarrow **AF** Bestätigt): „Jede Anfrage wird *irgendwann* bestätigt.“ (ist erfüllt)
- **AG**(Beschäftigt): „Das System ist *immer* beschäftigt.“ (ist nicht erfüllt)
- **AG**(Beschäftigt \Rightarrow **A**(Beschäftigt \cup Anfrage)): „Immer wenn das System beschäftigt ist, bleibt es dort, bis die Aussage *Anfrage* wieder gilt.“ (ist erfüllt)

2.3 Verifikation

Hat man ein Modell des Systems erstellt und die gewünschten Anforderungen definiert, dann kann das eigentlich Model-Checking beginnen. Es muss also entschieden werden, ob für ein gegebenes Modell M die Spezifikation Φ erfüllbar ist.

Prinzipiell kann das Model-Checking-Problem polynomiell in der Größe des Modells M und polynomiell in der Größe der Spezifikation Φ z. B. mittels Breiten- oder Tiefensuche gelöst werden. Bei Systemen mit nebenläufigen Prozessen wächst der Berechnungsbaum jedoch exponentiell mit der Anzahl der Zustände und Variablen eines Teilsystems. Dieses Problem ist allgemein als Zustandsraumexplosion bekannt (engl. state-space-explosion). Eine „brute-force“-Methode, bei der alle möglichen Berechnungspfade überprüft werden, ist nicht effizient und folglich nur auf kleinen – nicht praxisrelevanten Modellen – anwendbar.

Als Beispiel betrachte man ein simples MUTEX-Protokoll. Es stellt sicher, dass in einem nebenläufigen System keine 2 Prozesse gleichzeitig einen kritischen Bereich betreten (z. B. schreibender Dateizugriff).

Process 0: Repeat	Process 1: Repeat
00 non-critical-section 0	00 non-critical-section 0
01 wait unless turn = 0	01 wait unless turn = 1
10 critical section 0	10 critical section 1
11 turn:= 1	11 turn:= 0

Ein Prozess besteht aus 4 Zuständen und einer Variablen `turn`, welche die Werte 0 oder 1 annehmen kann. Das Gesamtsystem, bestehend aus 2 Prozessen à 4 Zustände und einer booleschen Variablen kann insgesamt $4 \times 4 \times 2 = 32$ verschiedene Systemzustände annehmen. Bei 20 Prozessen mit jeweils 10 Zuständen hätte man bereits 10^{20} Situationen, in denen sich das Gesamtsystem aufhalten könnte.

Um diese Komplexität zu bewältigen, werden in der Praxis zur Reduktion des Zustandsraumes Verfahren mit effizienten Datenstrukturen und Algorithmen eingesetzt. In den nächsten beiden Unterabschnitten werden zwei solcher Verfahren näher erläutert: das „symbolische“ Model-Checking und das „on-the-fly“ Model-Checking.

2.3.1 Symbolisch

Im Jahre 1992 hat McMillan im Rahmen seiner Doktorarbeit [McM92b] das symbolische Model-Checking eingeführt. Hierbei wird der explizite Aufbau des Berechnungsbaums im Speicher vermieden. Die grundlegende Idee dahinter ist, die Zustände und Variablen eines Systems symbolisch durch boolesche Formeln zu beschreiben. Damit können ganze Mengen von Zuständen bezüglich einer Eigenschaft in einem Verifikationsschritt abgehandelt werden.

Bezogen auf das Beispiel des MUTEX-Protokolls ließe sich jede mögliche System-situation als Vektor von Zuständen, die ein Prozess einnehmen kann, und Variablen kodieren – also $(Zustand(P_0), Zustand(P_1), Wert(turn))$. Eine Situation, in der sich Prozess 1 im kritischen Abschnitt aufhält und uninteressant ist, in welchem Zustand sich Prozess 0 befindet und welchen Wert die Variable `turn` hat, kann z. B. durch $(*, 10, *)$ kodiert werden.

In der Praxis haben sich für die symbolische Repräsentation einer Formel die *Binary Decision Diagrams (BDDs)* (Bryan '86 [Bry86]) durchgesetzt, auf die im Rahmen dieses Seminars jedoch nicht weiter eingegangen wird. Zur weiteren Vertiefung sei an dieser Stelle auf [Bry92, JEK⁺90] verwiesen.

2.3.2 On-the-fly

Eine andere Möglichkeit den Zustandsraum zu reduzieren bietet das sogenannte „on-the-fly“ Model-Checking [Hol91]. Im Gegensatz zum symbolischen Verfahren wird der Berechnungsbaum explizit aufgebaut und gespeichert, jedoch immer nur derjenige Teil, welcher zur Verifikation einer bestimmten Systemeigenschaft notwendig ist. Um Speicherplatz zu sparen, wird auch hier eine geeignete Kodierung verwendet. Zum Durchsuchen des teilweise aufgebauten Berechnungsbaumes kommen Tiefen- bzw. Breitensuche zum Einsatz.

Mit dem „on-the-fly“ Ansatz kann z. B. effizienter festgestellt werden, ob ein Modell die geforderten Spezifikation einhält. Immer wenn sich die verletzte Eigenschaft relativ am Anfang in der Hierarchie des Berechnungsbaums befindet, wird ein „on-the-fly“ Model-Checker dies sehr schnell erkennen.

Der wohl bekannteste on-the-fly Model-Checker heißt *SPIN* [Hol97] und wurde vom Forschungs-Team von Gerard J. Holzmann entwickelt.

2.3.3 Mittels Abstraktion

Bei sehr großen und komplexen Modellen helfen die vorgestellten Techniken nicht, den Zustandsraum auf eine geeignete Größe zu reduzieren. In solchen Fällen muss das

Modell durch Abstraktion vereinfacht werden [CGL96]. Dabei ist jedoch zu beachten, dass ein vereinfachteres Modell zwar effizient verifizierbar ist, aber nicht alle Verhalten des ursprünglichen Modells besitzt. Diese Tatsache führt nicht selten zu sogenannten *false positives* bzw. *false negatives*. Ein *false positive* bedeutet, dass bei der Verifikation die zu überprüfende Eigenschaft im abstrahierten Modell als korrekt angegeben wird, obwohl sie im Originalmodell nicht zutrifft. Entsprechend bedeutet *false negative*, dass die zu überprüfende Eigenschaft als nicht zutreffend angegeben wird, obwohl sie im Originalmodell zutrifft.

3 Model-Checking Werkzeuge

In den letzten 15 Jahren sind zahlreiche Model-Checking Werkzeuge entwickelt worden, die zu meist nach den erläuterten Konzepten in Abschnitt 2.3 arbeiten. Die berühmtesten werden in diesem Kapitel nun genauer vorgestellt.

3.1 SMV

Das bekannteste symbolische Model-Checking Werkzeug ist der *Symbolic Model Verifier* (SMV [McM92a]). Es wurde von McMillan im Zusammenhang mit seiner Dissertation am Lehrstuhl von Clarke entwickelt und ist wohl das berühmteste Werkzeug auf dem Gebiet des Model-Checkings. Das Modell wird mit eigener Eingabesprache formuliert und intern meist als Binary Decision Diagrams (BDDs) verarbeitet. Die Spezifikation der Anforderungen an das Modell erfolgt in der CTL-Syntax und wird in der gleichen Datei, wo auch das Modell steht, definiert.

Die Arbeitsweise und Benutzung von SMV soll im Folgenden am bereits beschriebenen Beispiel des MUTEX-Protokolls (siehe Abschnitt 2.3) vorgeführt werden.

3.1.1 Modellierung

Gegeben seien zwei Prozesse, die jeweils die folgenden Zustände einnehmen können:

- n_i (non-critical): Prozess i befindet sich außerhalb des kritischen Bereichs.
- t_i (trying): Prozess i wartet auf Zugang zum kritischen Bereich.
- c_i (critical): Prozess i befindet sich im kritischen Abschnitt.

Jeder Prozess kann nun Zyklen der Form $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \rightarrow t_i \rightarrow c_i \rightarrow \dots$ durchlaufen, wobei das i für den jeweiligen Prozess steht, also $i \in \{1, 2\}$. Ferner sei angenommen, dass beide Prozesse unabhängig voneinander, aber nicht gleichzeitig einen Zustandsübergang ausführen (*asynchrones Interleaving*). Dann lässt sich das Gesamtsystem, wie in Abbildung 3.1 illustriert, modellieren.

Die Entsprechung in der SMV-Modellierungssprache sieht dann so aus:

```
MODULE main
VAR
```

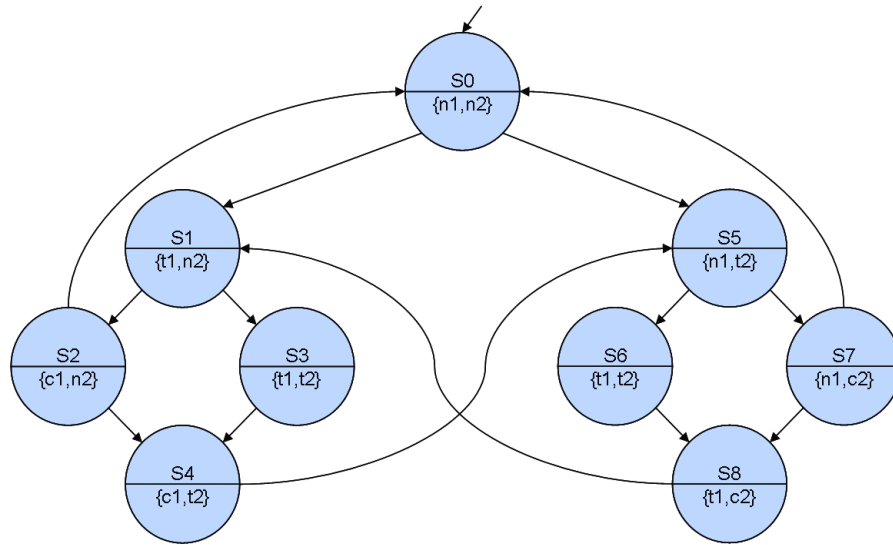


Abbildung 3.1: Modell für das MUTEX-Protokoll

```

turn : boolean;
proc1 : process proc(proc2.state,turn,0);
proc2 : process proc(proc1.state,turn,1);
ASSIGN
  init(turn) := 0;

MODULE proc(other-state,turn,myturn)
VAR
  state : {n,t,c};
ASSIGN
  init(state) := n;
  next(state) :=
    case
      (state = n) : {t,n};
      (state = t) & (other-state = n) : c;
      (state = t) & (other-state = t) & (turn = myturn) : c;
      (state = c) : {c,n};
    1 : state;
  esac;
  next(turn) :=
    case
      turn = myturn & state = c : !turn;
    1 : turn;
  esac;

```

Ein herausragendes Merkmal von SMV ist, dass komplexere Systeme aus *Modulen* zusammengesetzt werden können. Module können parametrisiert sein, es können mehrere Instanzen eines Moduls erzeugt werden und sie können Variablen anderer Module referenzieren.

3.1.2 Spezifikation

Aus Sicht des ersten Prozesses lassen sich nun die Anforderungen an ein korrektes MUTEX-Protokoll formalisieren:

Sicherheit: Zu jedem Zeitpunkt darf sich höchstens ein Prozess im kritischen Abschnitt befinden.

$$(AG\neg(c_1 \wedge c_2))$$

Lebendigkeit: Die Anforderung zum Eintritt in den kritischen Abschnitt wird irgendwann schließlich erfüllt.

$$(AG(t_1 \Rightarrow AFC_1))$$

Keine strikte Sequenzierung: Es ist keine einfach Lösung erlaubt, die z. B. den Prozessen abwechselnd Zugang gewährt. Anders formuliert: es kann vorkommen, dass der erste Prozess zweimal in Folge den kritischen Bereich betreten und wieder verlassen kann.

$$(EF(c_1 \wedge E[c_1U(\neg c_1 \wedge E[\neg c_2Uc_1])]))$$

Die entsprechende Spezifikation in SMV ist der CTL-Syntax sehr ähnlich und ist im folgenden Ausschnitt zu sehen:

```
--safety
SPEC AG!((proc1.state = c) & (proc2.state = c))

--liveness
SPEC AG((proc1.state = t) -> AF (proc1.state = c))
SPEC AG((proc2.state = t) -> AF (proc2.state = c))

--no strict sequencing
SPEC EF(proc1.state = c & E[proc1.state = c U
    (!proc1.state = c & E[! proc2.state = c U proc1.state = c ]))
```

3.1.3 Verifikation

Zur Verifikation erstellt man eine Datei (z. B. `mutex.smv`) mit dem Modell und den gewünschten Eigenschaften im SMV-Code. Anschließend startet man aus der Kommandozeile `smv mutex.smv` und erhält als Ergebnis:

```
[kopyy@localhost example]$ smv mutex.smv
-- specification AG (!(pr1.st = c & pr2.st = c)) is true
-- specification AG (pr1.st = t -> AF pr1.st = c) is true
-- specification AG (pr2.st = t -> AF pr2.st = c) is true
-- specification EF (pr1.st = c & E(pr1.st = c U (!pr1.st... is true

resources used:
user time: 0.01 s, system time: 0 s
BDD nodes allocated: 1117
Bytes allocated: 1245184
BDD nodes representing transition relation: 77 + 6
```

Das Modell des MUTEX-Protokolls mit den gewünschten Eigenschaften (Sicherheit, Lebendigkeit, keine strikte Sequenzierung) wurde damit erfolgreich verifiziert. Im Falle eines Fehlers würde man den Ausführungspfad erhalten, der zur Verletzung der Spezifikation führt, z. B.

```
-- specification AG (!(proc1.state = t & proc2.state = c)) is false
-- as demonstrated by the following execution sequence
state 1.1:
turn = 0
proc1.state = n
proc2.state = n
[stuttering]
state 1.2:
[executing process proc2]
state 1.3:
proc2.state = t
[executing process proc2]
state 1.4:
proc2.state = c
[executing process proc1]
state 1.5:
proc1.state = t
[stuttering]
```

3.2 SPIN

SPIN [Hol97] ist wohl der bekannteste „on-the-fly“ Model-Checker und wurde vom Forschungs-Team von Gerard J. Holzmann entwickelt. Es ist eines der beliebtesten und leistungsfähigsten Model-Checker für Softwaresysteme und ist in erster Linie für die Verifikation von Kommunikationsprotokollen ausgelegt. Die zu verifizierenden Eigenschaften müssen der LTL-Syntax folgen.

Die Systembeschreibungssprache von **SPIN** hat den Namen PROMELA¹ und ist für die Beschreibung asynchroner Kommunikation ausgerichtet. Von der Arbeitsweise her versucht **SPIN** den „brute-force“ Ansatz des „on-the-fly“ Model-Checkings durch die Verwendung trickreicher Verfahren zu optimieren.

Beispielsweise wird der *Depth-First Iterative Deepening* (DFID) Algorithmus zur Durchsuchung des Berechnungsbaumes eingesetzt. Der DFID-Algorithmus arbeitet nach dem Prinzip der Tiefensuche (*Depth-First-Search*, DFS), wird aber iterativ durch eine Suchtiefe beschränkt. Nach Erreichen der Tiefengrenze in jedem Iterationsschritte, setzt DFID die Suche jeweils in der Breite fort. Der Algorithmus terminiert, sobald alle Knoten im Graphen besucht sind.

Diese Strategie wird bei **SPIN** normalerweise erst nach dem Finden eines Spezifikationsfehlers durch den DFS-Algorithmus verwendet, da der Entwickler im Fall eines Fehlers ein möglichst kurzes Gegenbeispiel erhalten möchte. Durch den iterativen Ansatz des DFID-Algorithmus findet **SPIN** kurze Ausführungssequenzen.

Zum praktischen Arbeiten mit **SPIN** empfiehlt sich das graphische Frontend xSpin² oder auch die java-basierte Lösung jSpin³. Zur Spezifikation von Eigenschaften enthält xSpin sogar einen LTL-Editor. In Abbildung 3.2 ist eine Bildschirmaufnahme zu sehen.

3.3 Model-Checker für UML-Diagramme

Bei der bisherigen Betrachtung wurden zur Modellierung Kripke-Strukturen verwendet. In der Praxis ist jedoch UML die „de facto“ Standardmethode, um komplexe System zu modellieren [dMGMP02]. Damit stellt sich die Frage, wie es möglich ist, UML-Diagramme bezüglich ihres dynamischen Verhaltens zu verifizieren.

Die meisten existierenden Werkzeuge für UML Model-Checking beschränken sich dabei auf Statechart-Diagramme und übersetzen diese in die Modellierungssprache eines bekannten Model-Checkers, zumeist **SMV** oder **SPIN**. Im folgenden werden einige der am weitesten verbreiteten Werkzeuge dieser Art vorgestellt.

3.3.1 vUML

vUML [LP99] ist eines der ersten Werkzeuge, das Modelle verifiziert, bei denen das Verhalten durch UML-Statecharts beschrieben ist. Intern benutzt vUML den SPIN Model-Checker und dessen PROMELA Beschreibungssprache, jedoch bleibt dies dem

¹PRocess MEta LAnguage

²<http://spinroot.com/spin/Man/GettingStarted.html>

³<http://stwww.weizmann.ac.il/g-cs/benari/jspin/>

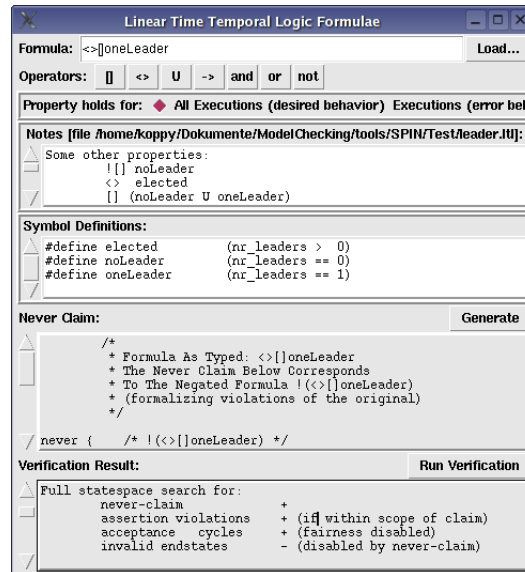


Abbildung 3.2: LTL-Editor von xSpin

Benutzer verborgen. Abbildung 3.3 illustriert die Benutzung des vUML-Werkzeugs. Nach der Erstellung eines UML-Statechart-Diagramms wird es zur Verifikation weiter an das vUML-Tool weitergereicht, wo es automatisch nach PROMELA konvertiert und anschließend mit dem SPIN Model-Checker verifiziert wird. Schlägt die Verifikation fehl, so wird ein Gegenbeispiel in Form eines UML-Sequenzdiagramms generiert. Mit Hilfe des Gegenbeispiels kann der Entwurf überarbeitet und erneut von vUML überprüft werden.

Mit vUML können Deadlocks, Erreichbarkeit ungültiger Zustände, Verletzung von Constraints auf einem Objekt, Senden eines Signals an ein terminiertes Objekt, Überläufe der Eingabequeue eines Objektes und Livelocks detektiert werden [LP]. Eine Spezifikation eigener Anforderungen an das Modell wird nicht unterstützt.

3.3.2 Hugo/RT

An der Ludwig-Maximilian-Universität München ist das Werkzeug Hugo/RT⁴ [SKM01, KM02] zur Verifikation von UML-Statechart-Diagrammen entstanden. Es kann feststellen, ob eine gewünschte Interaktionsfolge – als Kollaborations- oder Sequenzdiagramm spezifiziert – durch eine Menge von Statecharts realisierbar ist.

⁴<http://www.pst.informatik.uni-muenchen.de/projekte/hugo/>

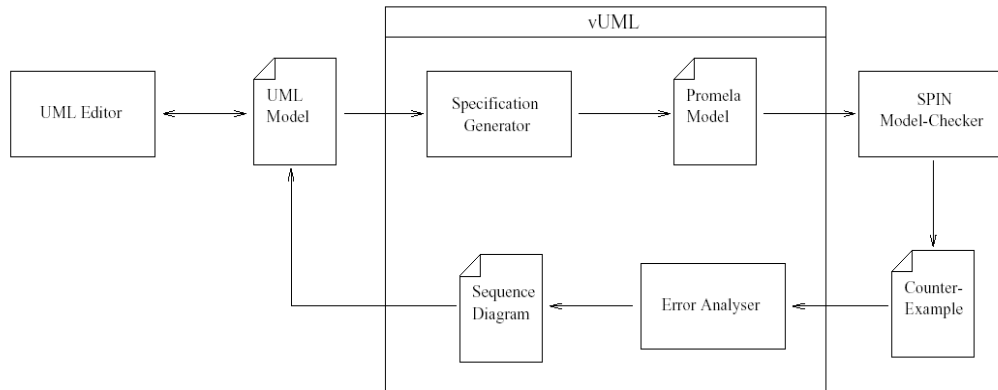


Abbildung 3.3: Benutzung von vUML (Quelle [LP99])

Als Eingabe benötigt **Hugo/RT** ein UML-Modell im XMI⁵-Format, was zur Verifikation mit **SPIN** in die PROMELA-Beschreibungssprache übersetzt wird. Ein „erfolgreicher“ Durchlauf generiert ein Gegenbeispiel, das detaillierte Informationen über Zustände und Transitionen der beteiligten Statecharts rückmeldet.

In der aktuellen Version von **Hugo/RT** ist es möglich, Statecharts, Kollaborationen, Interaktionen und OCL-Constraints in die Beschreibungssprache des Echtzeit-Model-Checkers **UPPAAL** ⁶, des „on-the-fly“ Model-Checkers **SPIN** und des Theorembeweisers **KIV** ⁷ zu übersetzen.

⁵XML Metadata Interchange

⁶<http://www.uppaal.com/>

⁷<http://i11www.ira.uka.de/~kiv>

4 Zusammenfassung

In der vorliegenden Seminararbeit ist eine Einführung in das umfassende Themengebiet des Model-Checkings gegeben worden. Zunächst sind die formalen Grundlagen vorgestellt worden: Kripke-Strukturen als mathematisches Modell, die Computation Tree Logic zur Spezifikation der gewünschten Anforderungen an das Modell und nicht zu Letzt die gängigsten Verifikationsmethoden – symbolisch bzw. on-the-fly.

Für den praktischen Einsatz existieren bereits Werkzeuge, die UML-Statecharts in die Beschreibungssprache von [SPIN](#), [SMV](#) oder [UPPAAL](#) übersetzten und mit dem jeweiligen Model-Checker überprüfen. Die Spezifikation der gewünschten Eigenschaften erfolgt zu meist entweder über ein Sequenz- oder Kollaborationsdiagramm, direkt in einer temporalen Logik, z. B. CTL, oder ist im jeweiligen Model-Checker fest vorgegeben, z. B. zur Detektion von Deadlocks, Unerreichbaren Zuständen, usw.

Zum Abschluss sei erwähnt, dass Model-Checking zwar automatisch anwendbar und daher „trivialisiert“ ist, aber bei größeren Systemen Probleme mit der Berechnungskomplexität hat. Auch wenn hierbei Abstraktion oder die Verifikation einzelner Komponenten des Systems helfen können, so gilt dennoch: Falls das reale System falsch modelliert oder abstrahiert worden ist, können die Ergebnisse der Verifikation nicht bedenkenlos auf die Realität übertragen werden.

Literaturverzeichnis

- [BBF⁺01] B. Berard, M. Bidot, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie, editors. *Systems and Software Verification*. Springer, 2001.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 8(35):677–691, 1986.
- [Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
- [CGL96] Edmund M. Clarke, O. Grumberg, and D. Long. Model checking. In *International Summer School on Deductive Program Design*, volume 152. Springer-Verlag Nato Asi, 1996.
- [CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
- [dMGMP02] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging uml designs with model checking. *Journal of Object Technology*, 1(2):101–117, 2002.
- [Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [JEK⁺90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10²⁰ States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

- [KM02] Alexander Knapp and Stephan Merz. Model checking and code generation for uml state machines and collaborations. In G. Schellhorn and W. Reif, editors, *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, Report 2002-11, Reisenburg, Germany, July 2002. Institut für Informatik, Universität Augsburg.
- [LP] Johan Lilius and Ivan Porres. The semantics of UML state machines.
- [LP99] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, 18, 1999.
- [McM92a] Kenneth Lauchlin McMillan. The smv system, <http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps>, 1992.
- [McM92b] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, 1992.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31– 1977. IEEE, IEEE Computer Society Press.
- [QS81] J.P. Quille and J. Sifakis. Spezifikation and verification of concurrent systems in cesar. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
- [SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. *Electr. Notes Theor. Comput. Sci.*, 55(3), 2001.

Glossar

BDD Binary Decision Diagram. Effiziente Datenstruktur zur Darstellung Boolescher Formeln durch einen azyklischen Graphen.

CTL Computation Tree Logic. Bei der CTL handelt es sich um eine temporale Logik mit zeitlicher Verzweigung, in der die temporalen Operatoren Aussagen über Pfade machen, die von einem gegebenen Zustand aus erreichbar sind.

Kripke-Struktur Im Kontext des Model-Checking ein verbreiteter Formalismus – dem endlichen Automaten ähnlich – zur Beschreibung von Systemmodellen. Mit Kripke-Strukturen ist es möglich, einen zustandsbasierten Transitionsgraphen zu definieren, der das Verhalten eines reaktiven Systems nachbildet.

PROMELA PRocess MEta LAnguage. Beschreibungssprache für Modelle des Model-Checkers SPIN.

SMV Symbolic Model Verifier. Einer der bekanntesten symbolischen Model-Checker, der auf BDDs basiert und in CTL formulierte Eigenschaften verifiziert.

Validierung In der Softwaretechnik ist die *Validierung* (auch Plausibilisierung) ein wichtiger Aspekt der Qualitätssicherung, mit dem überprüft wird, ob das implementierte System den vorher aufgestellten Anforderungen genügt.

Verifikation Bei der *Verifikation* handelt es sich um eine formale Beweistechnik, mit der überprüft wird, ob ein System seiner Spezifikation genügt. Es ist durchaus denkbar, dass eine Software seine Spezifikation erfüllt, nicht jedoch die Anforderungen des Kunden. Das kann passieren, wenn beim Übersetzen der informell formulierten Anforderungen in eine formale Spezifikation Fehler gemacht wurden.

Zustandsraumexplosion Bei der formalen Verifikation von Systemen, die nebenläufige Komponenten beinhalten, wächst die Anzahl der zu untersuchenden Zustände des Gesamtsystems exponentiell an und ist damit schwieriger zu behandeln.