Model Checking
Grundlagen

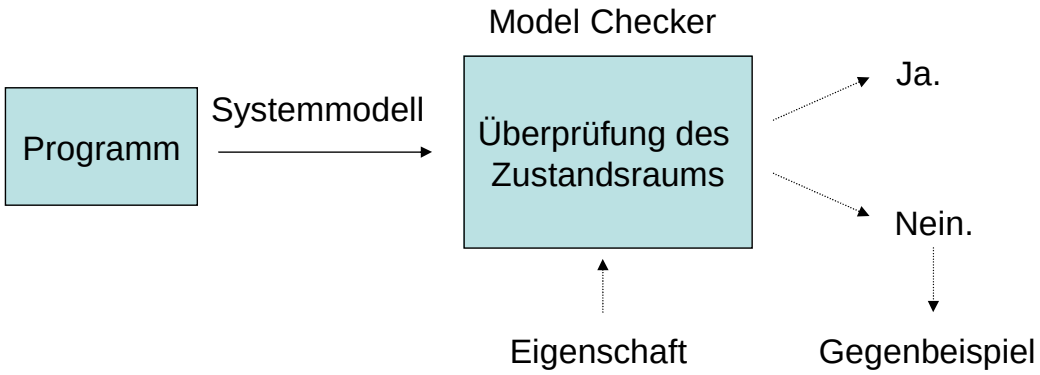
Vortrag: Patrick Nepper - 06.02.2003

Proseminar „Grundlagen höherer Programmiersprachen“
(Kröger, Rauschmayer)
WiSe 02/03
Institut für Informatik, LFE Programmierung und Softwaretechnik,
Ludwig-Maximilians-Universität München

Überblick

1 Model Checking – Was ist das?

- Technik um Modelle (verteilter) reaktiver Systeme auf gewisse (unerwünschte) Eigenschaften zu überprüfen
- Eingabe: Formales Modell eines Systems, zu überprüfende Eigenschaft (ausgedrückt in temporaler Logik)
- Ausgabe: Bestätigung oder Gegenbeispiel



2 Einführungsbeispiel: Einfacher Deadlock

```
public class Deadlock {
    static Object lock1;
    static Object lock2;
    static int  state;

    public static void main(String[] args) {
        lock1 = new Object();
        lock2 = new Object();
        Process1 p1 = new Process1();
        Process2 p2 = new Process2();
        p1.start();
        p2.start();
    }
}
```

```
class Process1 extends Thread {
    public void run() {
        while (true) {
            synchronized (Deadlock.lock1) {
                synchronized (Deadlock.lock2) {
                    Deadlock.state++;
} } }}   }

class Process2 extends Thread {
    public void run() {
        while (true) {
            synchronized (Deadlock.lock2) {
                synchronized (Deadlock.lock1) {
                    Deadlock.state++;
} } }}   }
```

PROMELA Modell

2.1 PROMELA Modell

Ein Model Checker benötigt ein abstraktes Modell des Programms.
PROMELA:

„protocol meta language“, Eingabesprache des Model Checkers „SPIN“

```
proctype Deadlock() {
  int _temp_;
loc_1:   atomic {
      _allocate(Object_col,1,3,1,0,1);
      TEMP_0 = _temp_; _temp_ = 0;
      Object_const_TEMP_0 = TEMP_0;
      lock1 = TEMP_0;
      goto loc_4;
   }
loc_4:   atomic {
      _allocate(Object_col_0,2,3,4,0,1);
      TEMP_2 = _temp_; _temp_ = 0;
      Object_const_TEMP_0 = TEMP_2;
      lock2 = TEMP_2;
      goto loc_7;
   }
```

static int state;

lock1 = new Object();

lock2 = new Object();

```
loc_7:   atomic {
      _allocate(Process1_col,3,3,7,0,1);
      p1
      Process1_const_TEMP_0 = p1;
      _allocate(Process2_col,4,3,10,0,1);
      p2
      Process2_const_TEMP_0 = p2;
      _startThread_1;
      goto loc_14;
   }
loc_14:   atomic {
      _startThread_2;
      p2 = 0;
   }
}
```

Process1 p1 = new Process1();

Process2 p2 = new Process2();

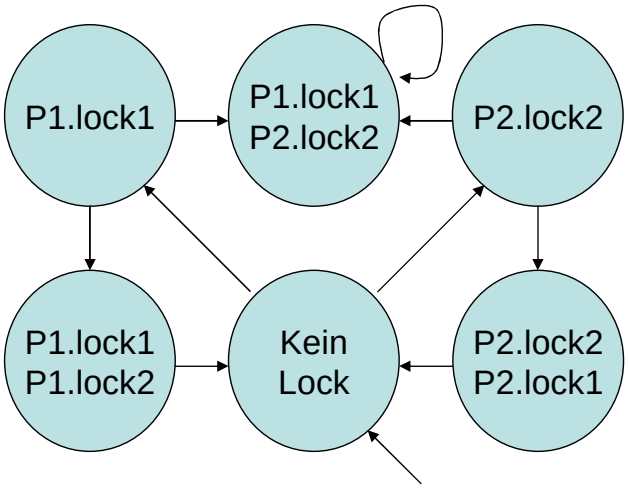
p1.start();

p2.start();

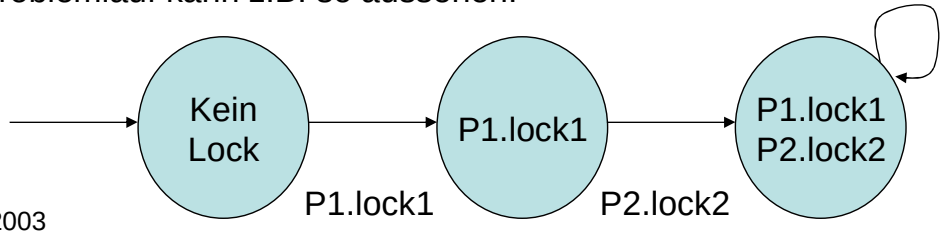
Abb 2.1 – **Bsp.:** PROMELA Code für Klasse Deadlock (Ausschnitt).

2.2 Problemlauf

Der Java Programmcode unseres Beispiels lässt sich auch in einem **Zustands-modell** darstellen:



Ein Problemlauf kann z.B. so aussehen:



2.3 Model Check

Das vorliegende Thread-Programm soll nicht in den Zustand Process1.lock1 $\wedge$ Process2.lock2 gelangen.

Diese Eigenschaften dargestellt in **temporaler Logik** (Abschnitt 3):

- **G**(*process1.lock1* $\Rightarrow \neg$*process2.lock2*)
- **G**(*process2.lock2* $\Rightarrow \neg$*process1.lock1*)

Der **Model Checker SPIN** erhält das **PROMELA** Protokoll und die erste Formel als Eingabe.

SPIN erklärt sofort, dass diese Eigenschaft verletzt wird und gibt als Ausgabe ein Gegenbeispiel an.

3 Übergangssysteme

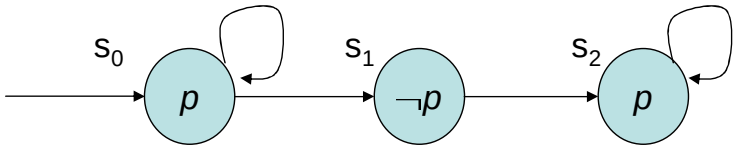


Abb 3 – Ein Übergangssystem mit T mit T |= **FG** p

Ein Übergangssystem T *(transition system)* ist das Modell eines reaktiven Systems. T hat drei Komponenten (V, R, I):

- **Zustandsvariablen**
- **Übergangsrelationen**
- **Anfangszustände**

3.1 Variablen

- **Variablen**

  $\boldsymbol{V}$ ist die Menge der **Zustandsvariablen** $v_1,\ldots,v_n$. Jede Variable v_i hat einen **Wertebereich** $\boldsymbol{D_i}$.

  Der gesamte **Zustandsraum** ***(state space) S*** ist das kartesische Produkt $\Pi^n_{i=1}D_i$.

  Ein **Zustand** ist eine Zuweisung von Werten der Geltungsbereiche zu den zugehörigen Variablen der Menge V.

  Beispiel:
  Sei $V=\{x,y\}$, $D_x=\{a,b,c\}$ und $D_y=\{0,1\}$.
  Dann ist $s=(x=b,y=1)$ ein Zustand des Modells.

3.2 Übergangsrelationen, Anfangszustände

- **Übergangsrelationen**
  R ist die Übergangsrelation mit $R:S\rightarrow S$, $s\mapsto s'$. $(s,s') \in R$ bedeutet, dass es dem System möglich ist von Zustand s in Zustand s' überzugehen.

- **Anfangszustände**
  $\boldsymbol{I\subseteq S}$ ist die Menge der Anfangszustände *(initial states)* des Systems. Das System kann in jedem dieser Zustände starten.

Ein **Weg** π (auch „Verhalten" genannt) ist eine (un)endliche Folge $\pi= (s_1, s_2,\ldots,s_i,\ldots)$ von Zuständen, so dass (s_i,s_{i+1}) ein gültiger Übergang ist.

4 Temporale Logik

Sei ein Übergangssystem T gegeben.

Fragen:

- Sind unerwünschte Zustände möglich (z.B. Deadlocks)?
- Kann es sein, dass erwünschte Zustände niemals erreicht werden können (z.B. wegen Endlosschleifen)?
- Kann es sein, dass ein Anfangszustand von einem beliebigen Zustand aus erreicht werden kann (d.h. kann das System zurückgesetzt werden)?

Temporale Logik ermöglicht es uns derartige Fragen formal auszudrücken.

4.1 Propositional Temporal Logic (PTL)

PTL: propositional temporal logic

Induktive Definition:

- Jede atomare Proposition $v \in V$ ist eine Formel
- Boolesche Kombinationen von Formeln sind Formeln
- Seien φ und ψ Formeln, dann sind auch $\mathbf{X}\ \varphi$ („next φ") und $\varphi\ \mathbf{U}\ \psi$ („φ until ψ") Formeln

Übliche Relationen:

$\pi \models \varphi$	Verhalten π erfüllt φ.
$\mathbf{F}\ \varphi$	$\equiv$ **true** $\mathbf{U}\ \varphi$. "finally φ", "eventually φ".
$\mathbf{G}\ \varphi$	$\equiv \neg\mathbf{F}\ \neg\varphi$. "globally φ", "always φ".
$\varphi\ \mathbf{W}\ \psi$	$\equiv (\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\ \varphi$. "$\varphi$ waits for ψ", "φ unless ψ".

Mit **PTL** läßt sich die **Korrektheit** von Eigenschaften eines Systems ausdrücken, nicht deren Existenz.

4.2 Computation Tree Logic

CTL: computation tree logic

CTL führt Quantoren ein: E (Existenzquantor), A (Allquantor“)

Induktive Definition:

- Jede atomare Proposition $v \in V$ ist eine Formel
- Boolesche Kombinationen von Formeln sind Formeln
- Seien φ und ψ Formeln, dann sind auch **EX** φ, **EG** φ und φ **EU** ψ Formeln

Übliche Relationen:

$T \models \varphi$	Übergangssystem T erfüllt φ (d.h. alle Anfangszustände erlauben Wege, die φ erfüllen).
EF φ	≡ **true EU** φ.
AX φ	≡ ¬**EX** ¬φ.
AG φ	≡ ¬**EF** ¬φ.

Mit **CTL** läßt sich die **mögliche Existenz** von Eigenschaften eines Systems ausdrücken.

5 Model Checking Algorithmen

Model Checking Algorithmen lassen sich i.A. einer der beiden Hauptkategorien zurechnen:

1. **Lokales Model Checking**
   Rekursion über die Elemente der Menge der in temporaler Logik gegeben Eigenschaften. Analyse eines gewissen Ausschnitts des Zustandsraums. Typischerweise PTL-basierend.

2. **Globales Model Checking**
   Rekursion wie bei lokalem MC. Aber Analyse des gesamten Zustandsraums. Typischerweise CTL-basierend.

Grundsatzproblem: „State-Space Explosion“

Schon bei einfachen Übergangssystemen wächst der Zustandsraum exponentiell.

5.1 Lokales PTL Model Checking

```
dfs(boolean search_cycle) {
     p = top(stack);
     foreach (q in successors(p)) {
          if (search_cycle and (q == seed))
               report acceptance cycle and
               exit;
          if ((q, search_cycle) not in visited)
          {
               push q onto stack;
               enter (q, search_cycle) into
               visited;
               dfs(search_cycle);
               if (not search_cycle and (q is
               accepting)) {
                    seed = q; dfs(true);
     } } }
     pop(stack);
}
// initialization
stack = emptystack(); visited = emptyset();
seed = nil;
foreach initial pair p {
     push p onto stack;
     enter (p, false) into visited;
     dfs(false)
}
```

Abb 5.1 – On-the-fly **PTL** Algorithmus (Pseudo-Code).

Ansatz:
Gegeben System *T* und Büchi Automat *B*, der alle unerwünschten Verhaltensweisen von T erlaubt.

Algorithmus sucht für alle möglichen Paare ($p \in I_T$ und $q \in I_B$) einen Zyklus.

dfs: depth first search
stack: Paare, die noch besucht werden müssen
visited: Paare, die bereits besucht wurden.

Erst Hinrichtung (search-cycle=false) dann Rückrichtung (search-cycle=true).

5.2 Globales CTL Model Checking

```
MODULE main
VAR
  request: boolean;
  status: {ready, busy};
ASSIGN
  init(status) := ready;
TRANS
  next(status) :=
    case
      request : busy;
      1       : {ready, busy};
    esac;
SPEC
  AG(request -> AF status = busy)
```

Abb 5.2 – **SMV CTL** Eingabesprache (Beispiel).

Ansatz:
Ein zu prüfendes System *T* wird beschrieben durch folgende Eigenschaften V, R, I.

Algorithmus prüft jedes Element von V mit Hilfe der gegebenen Elemente von R und I.

VAR: verwendete Variablen
ASSIGN: Zuweisungsbereich entsprechend I.
TRANS: entspricht R.
SPEC: entspricht V.

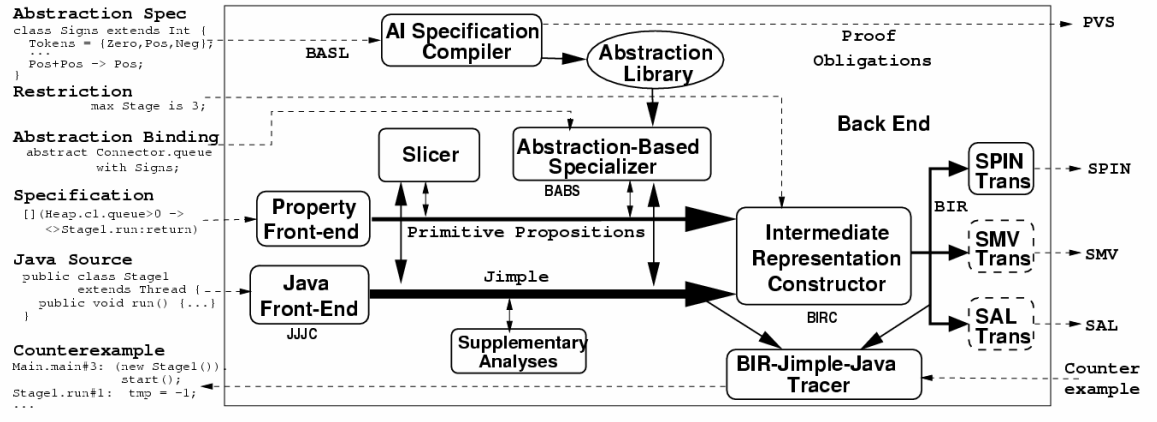
Wird ein eine SPECifikation verletzt, gibt SMV ein Gegenbeispiel aus.

6 Bandera – Model Checking in der Praxis

Bandera: extrahiert aus gegebenem Java-Code ein Modell, das mit einem angeschlossenen Model Checker (wie SPIN) gecheckt werden kann

Eingabe: Java-Code, Modell Sprache

Ausgabe: Antwort des Model Checkers, Zuordnung der Antwort zum Java-Code



6.1 Komponenten

Die Hauptkomponenten des Model Extractors Bandera:

- **Slicer**
  Der Slicer komprimiert Wege im Javaprogramm, indem er Variablen, Datenstrukturen und Kontrollpunkte, die für den Modelcheck einer gewissen Eigenschaft unerheblich sind, entfernt.
- **Abstraction Engine**
  Diese Engine unterstützt den Benutzer bei der Vereinfachung von Datentypen, die mit Variablen verbunden sind.
- **Back End**
  Das Back End generiert eine BIR, Bandera Intermediate Representation – die Grundlage für die Übersetzung in eine gewöhnliche Model Checker Eingabesprache. Das Back End enthält einen Übersetzer für jeden unterstützten Model Checker.
- **User Interface**
  Das UI bietet neben der Systeminteraktion auch die Ausgabe von Gegenbeispielen.

6.2 Benutzerinterface

Beispiel: Einfacher Deadlock

Abb 6.2 – Bandera GUI

Das Bandera Benutzer-interface gibt dem Benutzer die Möglichkeit

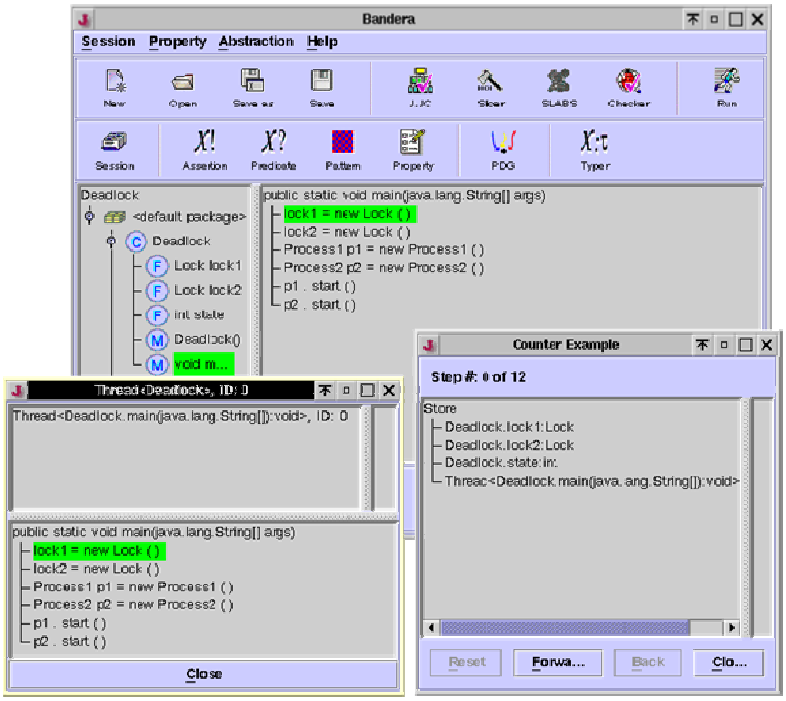
- den SLICER, Model Checker, usw. ein- oder auszuschalten,

- Laufzeitoptionen für den Model Checker auszuwählen,

- Prädikate und Formeln zu definieren, ….

6.3 Durchführung eines Model Checks

Beispiel: Einfacher Deadlock



Nachdem der MC Spin sein Ergebnis an Bandera übergeben hat präsentiert Bandera ein Gegenbeispiel (Deadlock).

Fenster Counter Example: Aktueller Heap und Möglichkeit zur Bewegung zwischen den Zuständen.

Fenster Thread: Markiert wird die als nächstes auszuführende Codezeile.