Institut für Informatik
Ludwig Maximilians Universität München

Model Checking

Grundlagen

Vortrag: Patrick Nepper

Proseminar „Grundlagen höherer Programmiersprachen“
(Kröger, Rauschmayer)
WiSe 02/03

06.02.2003

1 Model Checking – Was ist das? 3

2 Einführungsbeispiel: Einfacher Deadlock 3

2.1 PROMELA Modell 4

2.2 Problemlauf 5

2.3 Model Check 6

3 Übergangssysteme 6

3.1 Variablen 6

3.2 Übergangsrelationen 6

3.3 Anfangszustände 7

4 Temporale Logik 7

4.1 Propositional Temporal Logic (PTL) 7

4.2 Computation Tree Logic (CTL) 8

5 Model Checking Algorithmen 8

5.1 Lokales PTL Model Checking 9

5.2 Globales CTL Model Checking 10

6 Bandera: Model Checking in der Praxis 11

6.1 Komponenten 11

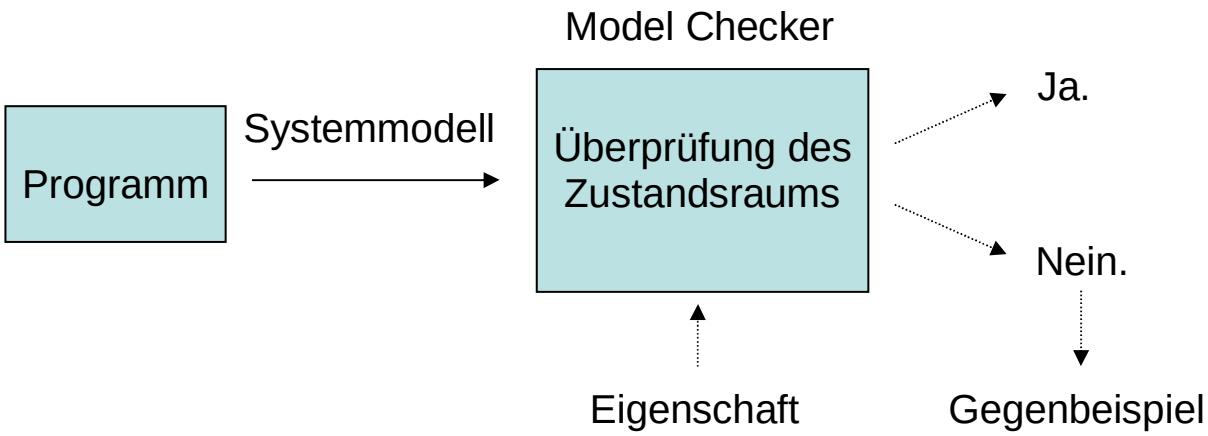
6.2 Benutzerinterface 11

6.3 Durchführung eines Model Checks 12

7 Quellenverzeichnis 12

1 Model Checking – Was ist das?

Model Checking ist eine Technik um Modelle (verteilter) **reaktiver Systeme** auf gewisse (unerwünschte) Eigenschaften zu überprüfen – dabei verstehen wir unter reaktiven Systemen derartige Systeme, die in ständigem Austausch mit ihrer Umwelt stehen, z.B. Kontrollsysteme oder Kommunikationssysteme. Ein Model Checker erhält als Eingabe ein **formales Modell** eines konkreten Programms und die zu überprüfende(n) Eigenschaft(en), ausgedrückt in **temporaler Logik**. Nach der Prüfung des Modells antwortet der Model Checker entweder mit einer Bestätigung oder einem Gegenbeispiel.



2 Einführungsbeispiel: Einfacher Deadlock

Als Einführungsbeispiel soll uns ein einfaches Javaprogramm dienen, das mit einer gewissen Wahrscheinlichkeit einen Deadlock hervorruft:

```
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++;
                    }
                }
            }
        }
    }
```

Beim Starten der Methode `Deadlock.main()` werden zwei Objekte erstellt: `lock1` und `lock2`, außerdem werden zwei Thread-Objekte `p1` und `p2` erstellt und gestartet. Diese beiden Objekte versuchen nun abwechselnd und ununterbrochen (vgl. while-Schleife) durch Synchronisation eine Sperre (Lock) auf `lock1` bzw. `lock2` zu erhalten um dann den Wert der Variablen `state` um eins zu erhöhen.

2.1 PROMELA Modell

Da ein Model Checker ein abstraktes Modell des Programms zur Bearbeitung benötigt, müssen wir das Javaprogramm vor einem Model Check in eine passende Modellsprache übersetzen, z.B. PROMELA („protocol meta language“), der Eingabesprache des Model Checkers „SPIN“. Wir verwenden hierfür die Übersetzung der Model Checking Suite Bandera (Abschnitt 6):

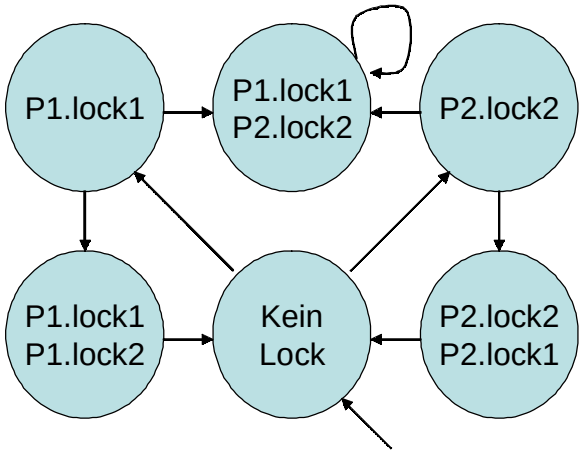
```
proctype Deadlock() {
  int _temp_;
loc_1:
   atomic {
      _allocate(Object_col,1,3,1,0,1);
      TEMP_0 = _temp_; _temp_ = 0;
      Object_const_TEMP_0 = TEMP_0;
      lock1 = TEMP_0;
      goto loc_4;
   }
loc_4:
   atomic {
      _allocate(Object_col_0,2,3,4,0,1);
      TEMP_2 = _temp_; _temp_ = 0;
      Object_const_TEMP_0 = TEMP_2;
      lock2 = TEMP_2;
      goto loc_7;
   }
loc_7:
   atomic {
      _allocate(Process1_col,3,3,7,0,1);
```

```
            p1 = _temp_; _temp_ = 0;
            Process1__this = p1;
            Process1_const_TEMP_0 = p1;
            _allocate(Process2_col,4,3,10,0,1);
            p2 = _temp_; _temp_ = 0;
            Process2__this = p2;
            Process2_const_TEMP_0 = p2;
            _startThread_1;
            goto loc_14;
        }
    loc_14:
        atomic {
            _startThread_2;
            p2 = 0;
        }
    }
```

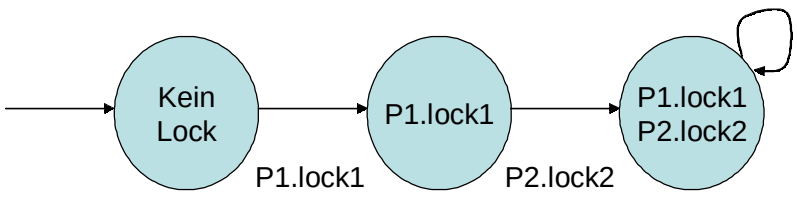
Wie man an diesem kleinen Modelausschnitt zur Klasse `Deadlock` sehen kann, wird die `Deadlock.main()`-Methode nicht einfach durch eine andere Syntax, sondern durch ein umfangreiches abstraktes Modell ausgedrückt, das auch dem Ablauf der Objektgenerierung – in unserem Beispiel der Objekte `lock1, lock2, Process1, Process2` – Rechnung trägt. Zum besseren Verständnis wurden die Abschnitte, die man in etwa direkt dem Javaquellcode zuordnen kann farblich hinterlegt.

2.2 Problemlauf

Der Java Programmcode unseres Beispiels lässt sich auch durch ein visuelles **Zustandsmodell** darstellen, indem man mögliche Zustände des Programms durch Blasen und mögliche Zustandsübergänge durch Pfeile symbolisiert:



Betrachtet man das Diagramm genau, so entdeckt man mögliche Programmabläufe, die in einem bestimmten Zustand „hängen“ bleiben. Ein solcher Problemlauf könnte z.B. wie folgt aussehen:



2.3 Model Check

Das vorliegende Thread-Programm soll also nicht in den im vorherigen Abschnitt beschriebenen Zustand {Process1.lock1 ∧ Process2.lock2} gelangen, da dies einen Deadlock produzieren würde.

Diese Eigenschaften lässt sich auch mit Hilfe **temporaler Logik** (Abschnitt 3) darstellen:

- **G**(*process1.lock1* ⇒ ¬*process2.lock2*)
- **G**(*process2.lock2* ⇒ ¬*process1.lock1*)

Der Buchstabe **G** steht hierbei für „Globally“ und bedeutet, dass die in Klammern stehende Eigenschaft global, d.h. für alle Zustände des Systems, gelten soll.
Diese Eigenschaften können nun von einem Model Checker automatisch geprüft werden. Erhält der **Model Checker SPIN** das **PROMELA** Modell aus Abschnitt 2.1 und die erste Formel **G**(*process1.lock1* ⇒ ¬*process2.lock2*) als Eingabe, so erklärt SPIN sofort, dass diese Eigenschaft verletzt wird und gibt als Ausgabe ein Gegenbeispiel (wie das in Abschnitt 2.2 besprochene) an.

3 Übergangssysteme

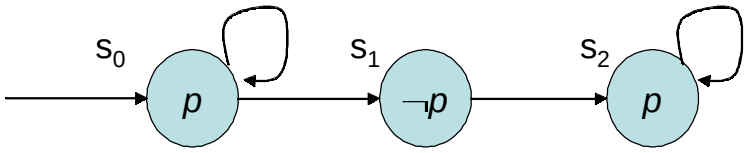


Abbildung 3-1 Ein Übergangssystem *T* mit *T* |= **FG** p

Da sich grundsätzlich alle Arten von reaktiven Systemen und Automaten mehr oder weniger gut als Übergangssysteme darstellen lassen, sind Übergangssysteme die Modell-Grundlage für Model Checking Algorithmen. Ein Übergangssystem *(transition system) T* hat drei Komponenten *T=(V, R, I)*: **Zustandsvariablen *V*, Übergangsrelationen *R*** und **Anfangszustände *I***.
Ein **Weg** π (auch „Verhalten” genannt) ist eine (un)endliche Folge $\pi = (s_1,s_2,...,s_i,...)$ von Zuständen, so dass (s_i,s_{i+1}) ein gültiger Übergang ist (vgl. Abbildung 3-1).

3.1 Variablen

V ist die Menge der **Zustandsvariablen** $v_1,\ldots,v_n$ des Systems *T*. Jede Variable v_i hat einen **Wertebereich D_i**. Der gesamte **Zustandsraum *(state space) S*** von *T* 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

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. Wenn wir in Abbildung 3-1 die

Zustandsübergänge zusätzlich zu den Pfeilen mit Symbolen α, β, γ kennzeichnen erhalten wir folgendes erweitertes Diagramm:

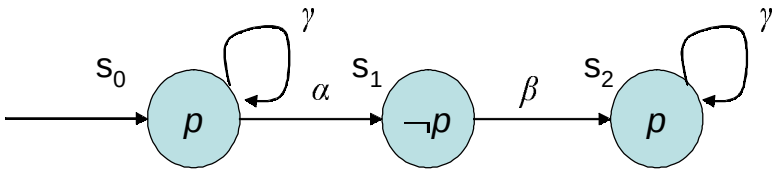


Abbildung 3-2 Ein Übergangssystem *T* mit *T* |= **FG** p und Relationssymbolen

Die durch dieses Übergangssystem gegebene Übergangsrelation lässt sich dann formal angeben: $R := \{ (s_0,\gamma,s_0), (s_0,\alpha,s_1), (s_1,\beta,s_2), (s_2,\gamma,s_2) \}$.

3.3 Anfangszustände

$I \subseteq S$ ist die Menge der Anfangszustände *(initial states)* des Systems. Das System kann in jedem dieser Zustände starten. In Abbildung 3-2 ist s_0 ein Anfangszustand.

4 Temporale Logik

Sei ein Übergangssystem *T* gegeben. Für einen Model Check sind beispielsweise folgende Fragen von Bedeutung:

- 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)

Induktive Definition:

- Jede atomare Proposition $v \in V$ ist eine Formel
- Boolsche Kombinationen von Formeln sind Formeln
- Seien φ und ψ Formeln, dann sind auch **X** φ („next φ") und φ **U** ψ („φ until ψ") Formeln

Bedeutung:

X φ („next φ")	Ein Zustand erfüllt diese Formel, wenn der folgende Zustand φ erfüllt.
φ **U** ψ („φ until ψ")	Ein Verhalten erfüllt diese Formel, wenn ein Zustand im Weg ψ erfüllt und alle Zustände davor φ erfüllen.

Übliche Relationen:

π |= φ	Verhalten π erfüllt φ.
F φ	≡ **true U** φ. "finally φ", "eventually φ". Es gibt einen Zustand der φ erfüllt.
G φ	≡ ¬**F** ¬φ. "globally φ", "always φ". Alle Zustände erfüllen φ.
φ **W** ψ	≡ (φ **U** ψ) ∨ **G** φ. "φ waits for ψ", "φ unless ψ". Entweder erfüllen alle Zuständer φ oder es gibt einen Zustand der ψ erfüllt und alle Zustände davor erfüllen φ.

Mit **PTL** läßt sich die **Korrektheit** von Eigenschaften für ganze Systeme ausdrücken. D.h. PTL Formeln gelten für jedes mögliche Verhalten des Systems. PTL kann somit keine Existenzaussage für eine Eigenschaft treffen, die nicht von jedem Verhalten des gegebenen Systems erfüllt wird.

4.2 Computation Tree Logic (CTL)

CTL führt Verhaltensquantoren ein: E (Existenzquantor), A (Allquantor). Diese Quantoren sagen aus ob eine Eigenschaft für jedes (A) Verhalten gilt oder ob es mindestens ein (E) Verhalten geben soll, dass eine Eigenschaft erfüllt.

Induktive Definition:

- Jede atomare Proposition $v \in V$ ist eine Formel
- Boolsche 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 somit im Gegensatz zu PTL die **mögliche Existenz** von Eigenschaften eines Systems ausdrücken.

5 Model Checking Algorithmen

Model Checking Algorithmen lassen sich im Allgemeinen einer dieser beiden Hauptkategorien zurechnen:

1. **Lokales Model Checking**
   Konzept: Rekursion über die Elemente der Menge der in temporaler Logik gegeben Eigenschaften. Analyse eines gewissen Ausschnitts des Zustandsraums. Die abschnittsweise Analyse des Zustandsraums ermöglicht es Algorithmen, die lokales Model Checking verwenden, u.U. auch Systeme mit unendlich vielen Zuständen (infinite-state systems) zu prüfen. Typischerweise PTL-basierend.
2. **Globales Model Checking**
   Konzept: Rekursion wie bei lokalem Model Checking. Aber Analyse des gesamten Zustandsraums. Daher sind Algorithmen dieses Typs nur für finite-state systems geeignet. Typischerweise CTL-basierend.

Sämtliche Model Checking Algorithmen teilen sich jedoch ein **Grundsatzproblem**, das die Einsetzbarkeit von Model Checkern im professionellen Umfeld unter Umständen einschränkt - die sog. **„State-Space Explosion“**. Schon bei einfachen Übergangssystemen kann der Zustandsraum exponentiell wachsen.

5.1 Lokales PTL Model Checking

Im Folgenden wird ein einfacher Model Checking Algorithmus vorgestellt, der das lokale Model Checking implementiert. Die Idee hinter diesem Algorithmus ist, einen Automaten *B* zu entwerfen, der die unerwünschten Eigenschaften des gegebenen Systems *T* erlaubt. Nun kann man den Algorithmus nach einem Weg suchen lassen, von einem Paar von Anfangszuständen beider Systeme zu einem in beiden Systemen erreichbaren unerwünschten Zustand. Die Suche ist somit auf den Zustandsraum begrenzt der durch *B* beschrieben wird.

Sei ein System *T* und ein Büchi Automat *B* gegeben, der alle unerwünschten Verhaltensweisen von *T* erlaubt. Der folgende einfache Algorithmus sucht für alle möglichen Paare ($p \in I_T$ und $q \in I_B$) einen Zyklus:

```
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)
}
```

Legende:

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

Der Algorithmus durchläuft erst die Hinrichtung zu einem unerwünschten Zustand (search-cycle=false), dann die Rückrichtung (search-cycle=true) zurück zum Paar von Anfangszuständen.

5.2 Globales CTL Model Checking

In diesem Abschnitt wird der Model Checking Algorithmus vorgestellt, den der Model Checker SMV verwendet. Dieser Algorithmus prüft auf Grundlage der gegebenen Anfangszustände und der gegebenen Übergangsrelation, ob alle erreichbaren Zustände die zu prüfenden Eigenschaften erfüllen. Dieser Algorithmus prüft somit im schlimmsten Fall den gesamten Zustandsraum.
Ein zu prüfendes System *T* wird durch seine Eigenschaften *V*, *R*, und *I* beschrieben. Der Algorithmus prüft jede Wertekombination aus *V* mit Hilfe der gegebenen Elemente von *R* und *I*.

```
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)
```

Legende:

VAR:	verwendete Variablen entsprechend *V*.
ASSIGN:	Zuweisungsbereich entsprechend *I*.
TRANS:	entspricht *R*.
SPEC:	entspricht den zu prüfenden Eigenschaften.

Wird ein eine Eigenschaft (SPEC) verletzt, gibt SMV ein Gegenbeispiel aus.

6 Bandera: Model Checking in der Praxis

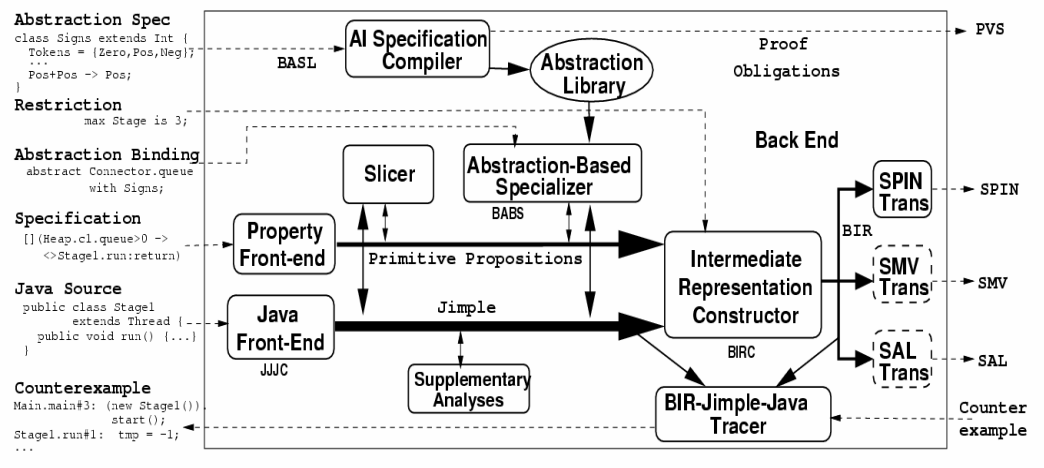


Abbildung 6-1 Bandera Organigramm

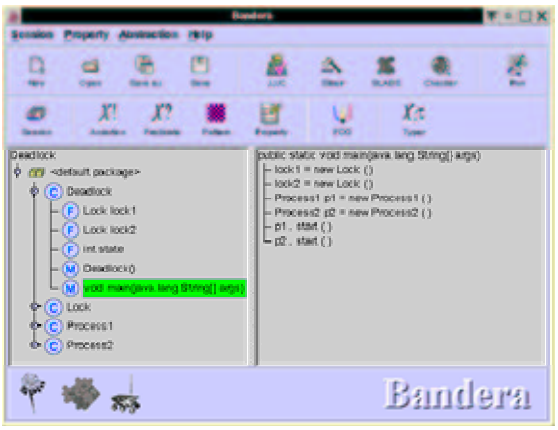
Bandera extrahiert aus einem gegebenen Javaprogramm ein Modell, das mit einem angeschlossenen Model Checker (wie SPIN) überprüft werden kann. Als Eingabe erhält Bandera den zu prüfenden Java-Code, sowie die zu prüfenden Eigenschaften in der zum angeschlossenen Model Checker passenden Modellsprache. Die Ausgabe besteht aus der Antwort des Model Checkers und der Zuordnung der Antwort zum gegebenen 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.

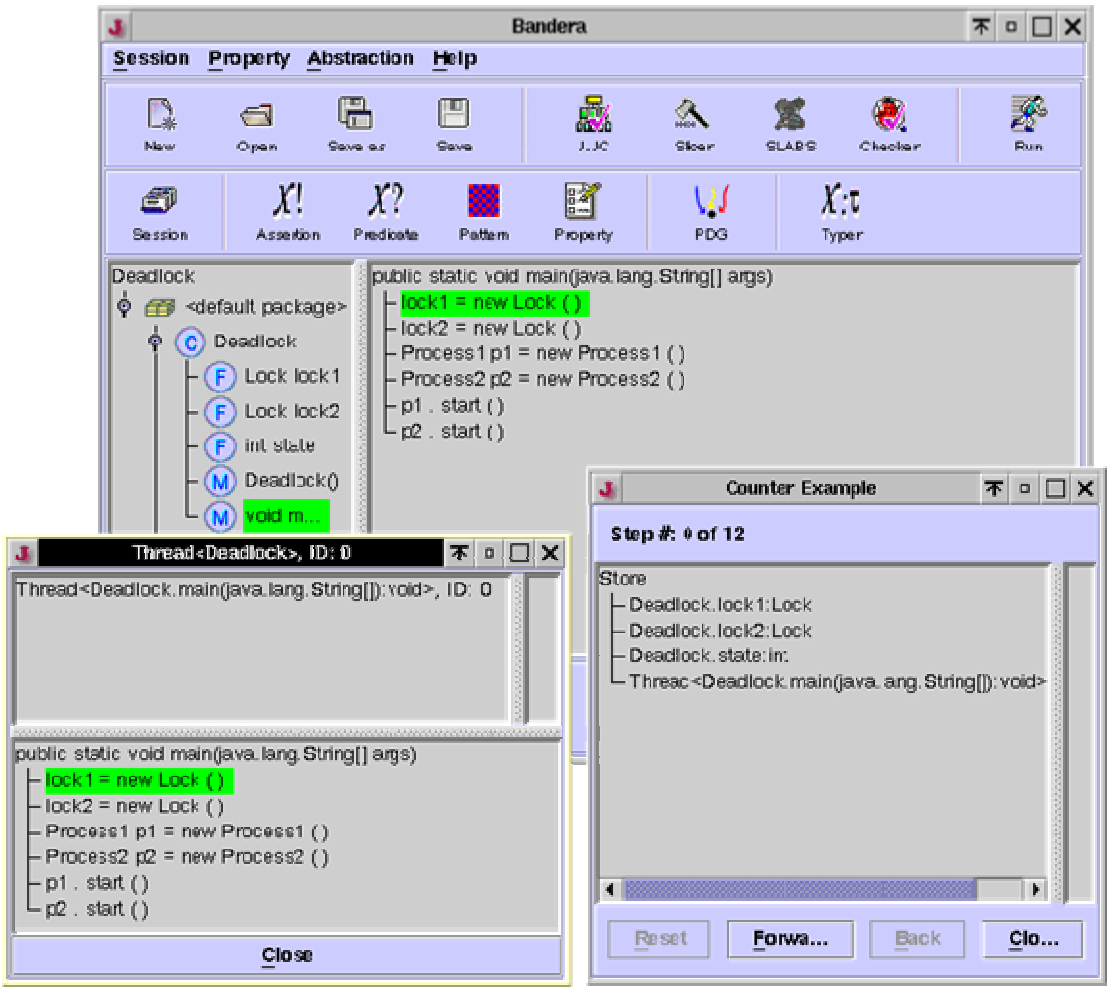
6.2 Benutzerinterface



Das Bandera Benutzerinterface 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



Um die Durchführung eines Model Checks unter der Verwendung von Bandera zu demonstrieren dient uns wieder das einfache Deadlock-Programm als Beispiel. Im Hauptfenster ist der Quellcode der Methode `Deadlock.main()` dargestellt.
Nachdem Bandera den Model Check initialisiert hat und der Model Checker Spin sein Ergebnis an Bandera übergeben hat, präsentiert Bandera ein Gegenbeispiel (Deadlock) im Fenster „Counter Example“. Es wird der aktuelle Heap dargestellt, während dem Benutzer die Möglichkeit zur Bewegung zwischen den Zuständen gegeben wird. Im ebenfalls neu erschienenen Fenster „Thread“ wird die als nächstes auszuführende Codezeile des aktuellen Threads markiert.
Wie man an diesem Beispiel sehen kann bietet Bandera Softwareentwicklern eine komfortable Schnittstelle zur Verwendung verschiedener Model Checker an – leider ändert dies aber nichts an den grundlegenden Problemen die den Einsatz von Model Checkern für reaktive Systeme einschränken.

7 Quellenverzeichnis

[1] J. C. Corbett et alteri. Bandera: Extracting Finite-State Models from Java Source Code, Department of Computing and Information Sciences, Kansas State University.
[2] S. Jha. Model Checking Basics. Computer Science Department, University of Wisconsin.
[3] W. Kahl. Spezifikationstechniken HT 2000, 2000.
[4] S. Merz. Model Checking: A Tutorial Overview. Institut für Informatik, Universität München.