We will introduce the programmable symbolic model checker TLV and illustrate its use for model checking finite-state systems.

The TLV tool, developed by Elad Shahar, is a programmable symbolic calculator over finite-state systems, based on the CMU symbolic model checker SMV.

It can be used to model check temporal properties of finite-state systems.

Examples of Model Checking

Using TLV for Model Checking

Consider the following program TRY-1 which attempts to solve the mutual exclusion problem by shared variables:

Variables $y_1$ and $y_2$ signify whether processes $P_1$ and $P_2$ are interested in entering their critical sections.

Local

$y_1$: boolean where $y_1 = y_2 = 0$
$P_1::$

$P_2::$

To check whether assertion $\phi_{\text{exclusion}}$ is an invariant of program TRY-1, we invoke the model checking tool TLV, a model checker based on the SMV tool developed in CMU by Ken McMillan and Ed Clarke.

We prepare two input files: try1.smv, which contains the SMV representation of TRY-1, and try1.pf, a proof script file. The proof script file contains some printing commands, definition of the assertion $\phi_{\text{exclusion}}$, and a command to check its invariance over the program.

We will present each of these input files.

Program Properties: Invariance

For program TRY-1, the property of mutual exclusion can be specified by requiring that the assertion

$\phi_{\text{exclusion}} : \neg(at \ (\ l_3 \land \ at \ (\ l_4 \land \ l_5)))$

be an invariant of TRY-1. This implies that no execution of TRY-1 can ever get to a state in which both processes execute their critical sections at the same time.

In LTL, this property can be specified as

$\neg(\ (l_3 \land \ at \ (\ l_4 \land \ l_5)))$
Lecture 2: Demonstrating Model Checking

A. Pnueli

File try1.smv

MODULE main
VAR y1: boolean; y2: boolean;
P[1]: process MP(y1, y2);
P[2]: process MP(y2, y1);

MODULE MP(mine, hers)
VAR loc: 0..5;
ASSIGN init(mine):=0; init(loc):=0;
next(loc):=case
loc in {0, 3, 4, 5}: (loc + 1) mod 6;
loc=1: {1, 2};
loc=2 & !hers: 3;
1: loc;
endcase;
next(mine):=case
loc=3: 1;
loc=5: 0;
1: mine;
endcase;

JUSTICE
loc != 0, ! (loc = 2 & !hers), loc != 3, loc != 4, loc != 5

Analysis of Reactive Systems, NYU, Fall, 2009

37

Lecture 2: Demonstrating Model Checking

A. Pnueli

File try1.pf

Print "Check for Mutual Exclusion"
Let exclusion := ! (P[1].loc = 4 & P[2].loc = 4);
Call Invariance (exclusion);
The call to procedure Invariance invokes the process which checks whether
any reachable state violates the assertion exclusion.
The call to procedure Invariance invokes the process which checks whether
the assertive invariant (exclusion) holds.

File try1.pf

The results of model-checking TRY-1 are

reaching the state (y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 1),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 2),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 3),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 1, P[2].loc = 3),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 2, P[2].loc = 3),
reaching the state (y1 = 0, y2 = 0, P[1].loc = 3, P[2].loc = 3),
reaching the state (y1 = 0, y2 = 1, P[1].loc = 3, P[2].loc = 4),
reaching the state (y1 = 1, y2 = 1, P[1].loc = 4, P[2].loc = 4),

Obviously, the problem is that the processes test each other's value first and
only later set their own.

File try1.pf

38

Lecture 2: Demonstrating Model Checking

A. Pnueli

Result of verifying TRY-1

Load "try1.pf"
Model checking Invariance Property
*** Property is NOT VALID ***
Counter-Example Follows:
State 1: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0,
State 2: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 1,
State 3: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 2,
State 4: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 3,
State 5: y1 = 0, y2 = 0, P[1].loc = 1, P[2].loc = 3,
State 6: y1 = 0, y2 = 0, P[1].loc = 2, P[2].loc = 3,
State 7: y1 = 0, y2 = 0, P[1].loc = 3, P[2].loc = 3,
State 8: y1 = 0, y2 = 1, P[1].loc = 3, P[2].loc = 4,
State 9: y1 = 1, y2 = 1, P[1].loc = 4, P[2].loc = 4,

Analysis of Reactive Systems, NYU, Fall, 2009

39

Lecture 2: Demonstrating Model Checking

A. Pnueli

Expressed in a more readable form

local y1; y2:
boolean where
y1 = y2 = 0
P1::

0: loop forever do
1: Non-Critical
2: await y2
3: y1 := 1
4: Critical

P2::

0: loop forever do
1: Non-Critical
2: await y1
3: y2 := 1
4: Critical

The counterexample is:

h0; m0; y1: 0; y2: 0
i;

h1; m0; y1: 0; y2: 0
i;

h1; m1; y1: 0; y2: 0
i;

h1; m2; y1: 0; y2: 0
i;

h2; m3; y1: 0; y2: 0
i;

h3; m4; y1: 0; y2: 1
i;

h4; m4; y1: 1; y2: 1
i;

reaching the state

h4; m4; y1: 1; y2: 1
i;

Chapter "Because of"?

Model checking is an invariant property.

File try1.pf

40
The counter example is:

\[
\begin{align*}
\text{Critical} : y_1 = 1, y_2 = 1, P[1].loc = 3, P[2].loc = 3, \\
\text{Non-Critical} : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 7 : y_1 = 1, y_2 = 1, P[1].loc = 3, P[2].loc = 3, \\
\text{State} 6 : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 5 : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 4 : y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 2, \\
\text{State} 3 : y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 2, \\
\text{State} 2 : y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 1, \\
\text{State} 1 : y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 0.
\end{align*}
\]

The following program is now correct:

\[
\begin{align*}
\text{Critical} : y_1 = 1, y_2 = 1, P[1].loc = 3, P[2].loc = 3, \\
\text{Non-Critical} : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 7 : y_1 = 1, y_2 = 1, P[1].loc = 3, P[2].loc = 3, \\
\text{State} 6 : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 5 : y_1 = 0, y_2 = 1, P[1].loc = 2, P[2].loc = 3, \\
\text{State} 4 : y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 2, \\
\text{State} 3 : y_1 = 0, y_2 = 0, P[1].loc = 1, P[2].loc = 2, \\
\text{State} 2 : y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 1, \\
\text{State} 1 : y_1 = 0, y_2 = 0, P[1].loc = 0, P[2].loc = 0.
\end{align*}
\]

Let us see whether the program is now correct.
The following program TRY-3 uses a variable \( u \) to indicate which process has the higher priority.

\[
\begin{align*}
\text{local } & \quad turn : [1 \rightarrow 2] \\
\text{where } & \quad turn = 1 & P_1 \quad \text{Non-Critical} & \quad & P_2 \quad \text{Critical} \\
& \quad turn := 2 & P_1 \quad \text{Critical} & \quad & P_2 \quad \text{Non-Critical} \\
\end{align*}
\]

```
loop forever do
1: Non-Critical
2: await turn = 1
3: Critical
4: turn := 2

local \( u \) : [0 \rightarrow 0]
where \( u = 0 \)
```

The counter example is:

- \( i \rightarrow 0; \) \( m \rightarrow 0; \) \( turn : 1 \) i
- \( h \rightarrow 1; \) \( m \rightarrow 0; \) \( turn : 1 \) i
- \( h \rightarrow 2; \) \( m \rightarrow 0; \) \( turn : 1 \) i
- \( h \rightarrow 3; \) \( m \rightarrow 0; \) \( turn : 1 \) i
- \( h \rightarrow 0; \) \( m \rightarrow 0; \) \( turn : 2 \) i
- \( h \rightarrow 2; \) \( m \rightarrow 0; \) \( turn : 2 \) i

We obtain the following results:

- Check for Mutual Exclusion: ***Property is VALID***
- Check for the absence of deadlock: ***Property is VALID***
- Check Accessibility for \( P_1 \): ***Property is NOT VALID***

The counter-example follows:

- State no. 1: \( P_1 \).loc = 0, \( P_2 \).loc = 0, turn = 1
- State no. 2: \( P_1 \).loc = 1, \( P_2 \).loc = 0, turn = 1
- State no. 3: \( P_1 \).loc = 2, \( P_2 \).loc = 0, turn = 1
- State no. 4: \( P_1 \).loc = 3, \( P_2 \).loc = 0, turn = 1
- State no. 5: \( P_1 \).loc = 4, \( P_2 \).loc = 0, turn = 1
- State no. 6: \( P_1 \).loc = 0, \( P_2 \).loc = 0, turn = 2
- State no. 7: \( P_1 \).loc = 1, \( P_2 \).loc = 0, turn = 2
- State no. 8: \( P_1 \).loc = 2, \( P_2 \).loc = 0, turn = 2

In a more readable form:

```
loop forever do
1: Non-Critical
2: await turn = 1
3: Critical
4: turn := 2

local turn : [1 \rightarrow 2]
where turn = 1
```

The counter-example follows:

- ***Property is NOT VALID***
- Check for the absence of deadlock: ***Property is VALID***
- Check for Mutual Exclusion: ***Property is VALID***

Exercise: Develop a counter-example to the property of accessibility for program TRY-3.
Finally a good program for mutual exclusion.

Following is a good shared variables solution to the mutual exclusion problem.

Consider the following example in smv code:

Additional Examples of SMV Code

```
ns
```

Peterson's for 2 Processes:

```
define y1, y2:
  boolean where
  y1 = y2 = 0
s:
  f1, f2
where s = 1
p1::
  loop forever do
    1:
      non-critical
    2:
      (y1, s) := (1, 1)
    3:
      await y2 = 0 _ s = 1
    4:
      critical
    5:
      y1 := 0
```

```
p2::
  loop forever do
    1:
      non-critical
    2:
      (y2, s) := (1, 2)
    3:
      await y1 = 0 _ s = 2
    4:
      critical
    5:
      y2 := 0
```

```
variables y1 and y2 signify whether processes P1 and P2 are interested in entering their critical sections. Variables s serve as a tie-breaker. It always contains the signature of the last processor to enter the waiting location (`3`, `m3`).

Model checking this program, we find that it satisfies the three properties of mutual exclusion, absence of deadlock, and accessibility.

Perspectives for 2 Processes:

Following is a good shared variables solution to the mutual exclusion problem.