Lecture 11

NuSMV: Model checking in practice

October 3, 2005
Overview

- A model of a mutual exclusion protocol.
- Checking properties of the mutual exclusion protocol.
- Problem solving using model checking.
- Coding in NuSMV
Develop a model.
  
  - Usually derived from a program.

Formally specify a property.

If the model and the property don’t match, one is wrong.
Modeling mutual exclusion

Two processes, accessing a shared resource.

Process states:

- \( n \): not trying to access the resource.
- \( t \): trying to access the resource.
- \( c \): in the process’s critical section.

Allowable transitions:

\[
\begin{align*}
n & \rightarrow t \\
t & \rightarrow c \\
c & \rightarrow n \\
n & \rightarrow t \\
t & \rightarrow c \\
c & \rightarrow \ldots
\end{align*}
\]
A possible model

\[ s_0 : n_1 n_2 \]

Desired properties: safety, liveness, non-blocking, no strict sequencing.
A possible model

\[ s_0 : n_1 n_2 \]

\[ s_1 : t_1 n_2 \]

\[ s_5 : n_1 t_2 \]

Desired properties: safety, liveness, non-blocking, no strict sequencing.
A possible model

Desired properties: safety, liveness, non-blocking, no strict sequencing.
A possible model

Desired properties: safety, liveness, non-blocking, no strict sequencing.
A possible model

Desired properties: safety, liveness, non-blocking, no strict sequencing.
Safety

Only one process can access the resource at a time.

\[ \text{AG[ ...]} \]

Check all states:
Safety

Only one process can access the resource at a time.

\[ AG [\neg (c_1 \land c_2)] \]

Check all states:
Liveness

Every request is eventually serviced.

Equivalently: if \( t_1 \) then always eventually \( c_1 \).

\[
A G [
\]

Check \( t_1 \) states:
Liveness

Every request is eventually serviced.

Equivalently: If $t_1$ then always eventually $c_1$.

$$A \ G[t_1 \rightarrow A \ F \ c_1]$$

Check $t_1$ states:
Non-blocking

A process can request the resource at any time.
“Allowed to” = possibly, not necessarily:

\[ A G[ \quad ] \]

Check \( n_1 \) states:

\[ s_0 : n_1 n_2 \]
\[ s_1 : t_1 n_2 \]
\[ s_2 : c_1 n_2 \]
\[ s_3 : t_1 t_2 \]
\[ s_4 : c_1 t_2 \]
\[ s_5 : n_1 t_2 \]
\[ s_6 : n_1 c_2 \]
\[ s_7 : t_1 c_2 \]
Non-blocking

A process can request the resource at any time.
“Allowed to” = possibly, not necessarily:

\[ \text{AG}[n_1 \rightarrow \text{EX} t_1] \]

Check \( n_1 \) states:
No strict sequencing

\( c_1 \) and \( c_2 \) should not be required to alternate.

Desired scenario:

- Process 1 accesses the resource for a while.
- Process 1 releases the resource.
- Process 2 does not have the resource (for a while).
- Process 1 re-obtains the resource.
No strict sequencing

\[
\text{EF}[\quad]
\]

\[
\begin{align*}
& s_0 : n_1 n_2 \\
& s_1 : t_1 n_2 \\
& s_2 : c_1 n_2 \\
& s_3 : t_1 t_2 \\
& s_4 : c_1 t_2 \\
& s_5 : n_1 t_2 \\
& s_6 : n_1 c_2 \\
& s_7 : t_1 c_2
\end{align*}
\]
No strict sequencing

\[ EF [c_1 \land E[c_1 \cup (\neg c_1 \land E[\neg c_2 \cup c_1])]] \]
Liveness problem

(Requesting process waits forever while the other process repeatedly gets the resource.)
Properties

- Safety: $\mathcal{A} G \neg (c_1 \land c_2)$
- Liveness: $\mathcal{A} G [t_1 \rightarrow A F c_1]$
- Non-blocking: $\mathcal{A} G [n_1 \rightarrow E X t_1]$
- No strict sequencing: $E F [c_1 \land E [c_1 U (\neg c_1 \land E [\neg c_2 U c_1])]]$
NuSMV

- Model checker for LTL and CTL.
- Synchronous or asynchronous processes.
- Each iteration of a process is atomic.
- Processes communicate via shared variables.

http://nusmv.irst.itc.it/
NuSMV model specification

Collection of modules for processes and an entry point.

```plaintext
MODULE  name[(param, ...)]
VAR
  id:type
  ...
ASSIGN
  init(id) := exp
  ...
  next(id) := exp
  ...
SPEC  CTL formula
  ...
LTLSPEC  LTL formula
  ...
FAIRNESS  CTL formula
  ...
```
Mutual exclusion example
Implementation outline

- Both processes behave the same, so only one module.
- Processes have a state: n, t, or c.
- Processes either change state or stay in the n or c state.
- Process action depends on its state and the state of the other process.

Outline:

```plaintext
MODULE prc(other-st)
VAR st : {n, t, c};
ASSIGN
  init(st) := n;
  next(st) := ...;
```
STATE TRANSITIONS

MODULE prc(other-st)

VAR st : {n, t, c};

ASSIGN

init(st) := n;

next(st) :=

case
(st = n) : {t,n};
(st = t) & (other-st = n) : c;
(st = t) & (other-st = t) : c;
(st = c) : {c,n};
1 : st;
esac;
Entry point module

- Instantiate the processes.
- Pass to each the other’s state (by reference).

```plaintext
MODULE main
VAR
  pr1 : process prc(pr2.st);
  pr2 : process prc(pr1.st);
```
Coding properties

- Safety: $\text{AG } \neg(c_1 \land c_2)$

- Liveness: $\text{AG}(t_1 \rightarrow \text{AF } c_1), \text{AG}(t_2 \rightarrow \text{AF } c_2)$

MODULE main
VAR
  pr1 : process prc(pr2.st);
  pr2 : process prc(pr1.st);
-- safety
SPEC $\text{AG } !((pr1.st = c) \land (pr2.st = c))$
-- liveness
SPEC $\text{AG}((pr1.st = t) \rightarrow \text{AF } (pr1.st = c))$
SPEC $\text{AG}((pr2.st = t) \rightarrow \text{AF } (pr2.st = c))$
Counterexample trace

-- specification AG (pr1.st = t -> AF pr1.st = c) is false
-- as demonstrated by the following execution sequence
Trace Description:  CTL Counterexample
Trace Type:  Counterexample

-> State:  1.1 <- pr1.st = n, pr2.st = n
-> Input:  1.2 <-  _process_selector_ = pr1
-- Loop starts here
-> State:  1.2 <- pr1.st = t
-> Input:  1.3 <- _process_selector_ = pr2
-> State:  1.3 <-

Execution model:

► At every clocktick some process runs or no process runs.

Here, process 2 runs forever.

Solution:  FAIRNESS running in process module.
A second counterexample

Process 2 continually requests and gets the critical section. Revised model:
Changes to the entry point

Each process accesses a global turn variable and its own identity.

MODULE main
VAR
    pr1 : process prc(pr2.st, turn, 0);
    pr2 : process prc(pr1.st, turn, 1);
    turn : boolean;
ASSIGN
    init(turn) := 0;

No changes to the properties.
Changes to the process module

- Only enter the critical section if turn = myturn.
- Update turn.

\[ \text{next}(st) := \]
\begin{align*}
\text{case} \\
(st = n) & : t, n; \\
(st = t) & (\text{other-st} = n) : c; \\
(st = t) & (\text{other-st} = t) & (\text{turn} = \text{myturn}) : c; \\
(st = c) & : c, n; \\
1 & : st; \\
esac;
\]

\[ \text{next}(\text{turn}) := \]
\begin{align*}
\text{case} \\
(\text{next}(st) = c) & (\text{other-st} = t) & (\text{turn} = \text{myturn}) : !\text{turn}; \\
1 & : \text{turn}; \\
esac; \]
A third counterexample

\[ \text{pr1} = \text{n} \quad \text{pr1} = \text{t} \]
\[ \text{pr2} = \text{n} \quad \rightarrow \quad \text{pr2} = \text{t} \quad \rightarrow \quad \text{pr2} = \text{c} \]

Process 2 in its critical section forever.  
(Also a problem with the original model.)

Solution: **FAIRNESS** ! (st = c)
Model checking for problem solving

Players: Ferryman, wolf, goat, cabbage.

Problem:
- All start on one side of the river.
- All should end up on the other side of the river.
- The ferryman can only take one at a time.
- If left alone without the ferryman:
  - The wolf eats the goat.
  - The goat eats the cabbage.

Strategy:
- Create a model describing all possible behaviors.
- Try to prove that the undesired behavior always happens.
- The counterexample is a solution to the problem.
Variables and initialization

MODULE main

VAR
  ferryman : {initial,destination};
  goat     : {initial,destination};
  cabbage  : {initial,destination};
  wolf     : {initial,destination};
  carry    : {g,c,w,none};

ASSIGN
  init(ferryman) := initial;
  init(goat)    := initial;
  init(cabbage) := initial;
  init(wolf)    := initial;
  init(carry)   := none;
Updating the ferryman

\[ \text{next(ferryman)} := \{\text{initial}, \text{destination}\}; \]

\[ \text{next(carry)} := \]
\[ \text{case} \]
\[ \quad \text{ferryman} = \text{goat} : g; \]
\[ \quad 1 : \text{none}; \]
\[ \text{esac union} \]

\[ \text{case} \]
\[ \quad \text{ferryman} = \text{cabbage} : c; \]
\[ \quad 1 : \text{none}; \]
\[ \text{esac union} \]

\[ \text{case} \]
\[ \quad \text{ferryman} = \text{wolf} : w; \]
\[ \quad 1 : \text{none}; \]
\[ \text{esac union none;} \]
Updating the others

next(goat) := case
    ferryman = goat & next(carry) = g : next(ferryman);
    1 : goat;
esac;

next(cabbage) := case
    ferryman = cabbage & next(carry) = c : next(ferryman);
    1 : cabbage;
esac;

next(wolf) := case
    ferryman = wolf & next(carry) = w : next(ferryman);
    1 : wolf;
esac;
Property and result

\[
\text{LTLSPEC} \quad \neg \left( (\neg (\text{goat} = \text{cabbage} \lor \text{goat} = \text{wolf}) \rightarrow \text{goat} = \text{ferryman}) \land (\text{goat} = \text{destination} \land \text{cabbage} = \text{destination} \land \text{wolf} = \text{destination} \land \text{ferryman} = \text{destination}) \right)
\]

\[
\begin{align*}
\text{F,G,C,W} & \xrightarrow{G} \text{F,G} \\
\text{F,C,W} & \xleftarrow{C} \\
\text{F,G,C} & \xrightarrow{C} \\
\text{F,G,W} & \xleftarrow{G} \\
\text{F,C,W} & \xrightarrow{W} \\
\text{F,G} & \xleftarrow{G} \\
\text{F,G} & \xrightarrow{G} \text{F,G,C,W} \\
\text{F,G} & \xleftarrow{G} \\
& \ldots
\end{align*}
\]
Property and result

\[
\text{LTLSPEC } !\left( (\text{goat} = \text{cabbage} \lor \text{goat} = \text{wolf}) \rightarrow \text{goat} = \text{ferryman} \right) \\
U (\text{goat} = \text{destination} \land \text{cabbage} = \text{destination} \land \\
\text{wolf} = \text{destination} \land \text{ferryman} = \text{destination})
\]

\[
\begin{array}{c}
F,G,C,W \\ \\
F,C,W \\ \\
F,G,W \\ \\
F,G
\end{array}
\xrightarrow{G}
\begin{array}{c}
F,G \\ \\
F,G,C \\ \\
F,C,W \\ \\
F,G,C,W \\ \\
F,G
\end{array}
\xrightarrow{G}
\begin{array}{c}
F,G,C,W
\end{array}
\xrightarrow{G}

\text{Solution:}

G(\text{goat} = \text{destination} \land \ldots)

or bounded model checking
Another algorithm

Peterson and Fischer algorithm:

Process 1:
\[
y_1 := \text{case } y_2 \text{ of } \perp \rightarrow T | T \rightarrow T | F \rightarrow F; \\
y_1 := \text{case } y_2 \text{ of } \perp \rightarrow y_1 | T \rightarrow T | F \rightarrow F;
\]
wait until \(y_1 \neq y_2\);
// critical section
\[
y_1 := \perp;
\]
goto start;

Process 2:
\[
y_2 := \text{case } y_1 \text{ of } \perp \rightarrow T | T \rightarrow F | F \rightarrow T; \\
y_2 := \text{case } y_1 \text{ of } \perp \rightarrow y_2 | T \rightarrow F | F \rightarrow T;
\]
wait until \(y_1 = \perp \text{ or } y_1 = y_2\);
// critical section
\[
y_2 := \perp;
\]
goto start;

Problems: Multiple assignments to \(y_1\) and \(y_2\), loops.
Goal: create a model systematically.
Model encoding

- One module per process.
- Use a program counter label in each process.
- Variable updating depends on label.

Process 1:
1. \( y_1 := \text{case } y_2 \text{ of } \bot \rightarrow T | T \rightarrow T | F \rightarrow F; \)
2. \( y_1 := \text{case } y_2 \text{ of } \bot \rightarrow y_1 | T \rightarrow T | F \rightarrow F; \)
3. \( \text{wait until } y_1 \neq y_2; \)
   \( // \text{ critical section} \)
4. \( y_1 := \bot; \)
5. \( \text{goto start;} \)

Model code:

\begin{verbatim}
VAR label : \{l1,l2,l3,l4,l5\};
ASSIGN
  init(label) := l1;
  next(label) := case
    label = l1 : l2;
    label = l2 : l3;
    label = l3 & !(y1 = y2) : l4;
    label = l3 : l3;
    label = l4 : l5;
    label = l5 : l1;
  esac;
\end{verbatim}
Variable updates

Process 1:
1. \( y_1 := \text{case } y_2 \text{ of } \bot \rightarrow T \mid T \rightarrow T \mid F \rightarrow F; \)
2. \( y_1 := \text{case } y_2 \text{ of } \bot \rightarrow y_1 \mid T \rightarrow T \mid F \rightarrow F; \)
3. wait until \( y_1 \neq y_2; \)
   
   // critical section
4. \( y_1 := \bot; \)
5. goto start;

- Three possible values on lines 1 and 2.
- One possible value on line 4.

\[
\text{next}(y_1) :=
\begin{array}{ll}
\text{case} & \\
\text{label} = l_1 \land y_2 = \text{bottom} & : \text{true}; \\
\text{label} = l_1 \land y_2 = \text{true} & : \text{true}; \\
\text{label} = l_1 \land y_2 = \text{false} & : \text{false}; \\
\text{label} = l_2 \land y_2 = \text{bottom} & : y_1; \\
\text{label} = l_2 \land y_2 = \text{true} & : \text{true}; \\
\text{label} = l_2 \land y_2 = \text{false} & : \text{false}; \\
\text{label} = l_4 & : \text{bottom}; \\
1 & : y_1;
\end{array}
\]

Multiple variables can be handled similarly.
Fairness, properties to check

Process 1:
  y1 := case y2 of ⊥ -> T | T -> T | F -> F;
  y1 := case y2 of ⊥ -> y1 | T -> T | F -> F;
  wait until y1 != y2;
  // critical section
  y1 := ⊥;
  goto start;

Fairness: FAIRNESS running, in each module.

Properties:
  ▶ Safety:
    AG !(pr1.label = 14 & pr2.label = 14)
  ▶ Liveness:
    AG(label in {11,12,13} -> AF label in 14)
MODULE P1(y1,y2)
VAR label : \{l1,l2,l3,l4,l5\};
ASSIGN
  init(label) := l1;
  next(label) :=
    case
      label = l1 & y2 = bottom : true;
      label = l1 & y2 = true : true;
      label = l1 & y2 = false : false;
      label = l3 & !(y1 = y2) : l4;
      label = l2 & y2 = bottom : y1;
      label = l3 : l3;
      label = l2 & y2 = true : true;
      label = l4 : l5;
      label = l2 & y2 = false : false;
      label = l5 : l1;
      label = l4 : bottom;
    esac; 1 : y1;
  esac;