nuXmv debugging models

Enrico Magnago

University of Trento,
Fondazione Bruno Kessler

Debugging models

Debugging a model: inspect finite state transition system

check_fsm
check if there are deadlocks in the fsm.

```
MODULE main
  VAR
    x : 0..2;
    y : {a, b, c};

  TRANS x = 2 -> next(x) = 0;
  TRANS y = a -> next(x) = 1;
```

Q: Is there a dead-lock? Where?

Deadlock? Ask nuXmv

```
nuXmv > go
nuXmv > check_fsm

############################################################
The transition relation is not total. A state without
successors is:
x = 2
y = a
The transition relation is not deadlock-free.
A deadlock state is:
x = 2
y = a
############################################################
```

What about this mutual exclusion protocol?

```
MODULE Mutex(signal, shared, id)
  VAR
    pc : {l0, l1, l2};

  ASSIGN
    init(pc) := l0;
    next(pc) :=
      case
        pc = l0 & signal & shared = 0 : l1;
        pc = l1 & !signal & shared = id : l2;
        pc = l2 : l0;
        TRUE : pc;
      esac;

  TRANS pc = l0 & next(pc) = l1 -> next(shared) = id;
  TRANS pc = l2 & next(pc) = l0 -> next(shared) = 0;

MODULE main
  VAR
    signal : boolean;
    shared : 0..2;
    m0 : Mutex(signal, shared, 1);
    m1 : Mutex(signal, shared, 2);

  INVARSPEC NAME SAFE := m0.pc != l2 | m1.pc != l2;
```

Q: is there a dead-lock?

Guess what, ask nuXmv

```
The transition relation is not total. A state without
successors is:
signal = TRUE
shared = 0
m0.pc = l2
m1.pc = l0
The transition relation is not deadlock-free.
A deadlock state is:
signal = TRUE
shared = 0
m0.pc = l0
m1.pc = l0
```

Q: what's the difference between the two states?

Guess what, ask nuXmv

```
The transition relation is not total. A state without
successors is:
signal = TRUE
shared = 0
m0.pc = l2
m1.pc = l0
The transition relation is not deadlock-free.
A deadlock state is:
signal = TRUE
shared = 0
m0.pc = l0
m1.pc = l0
```

Q: what's the difference between the two states? The first one is not reachable.
Q: why?

Get reachable states

print_reachable_states
prints number of reachable states and total number of states.

```
system diameter: 1
reachable states: 6 (2^2.58496) out of 54 (2^5.75489)
```

Only 6 states are reachable, this is fishy.
Can we list them?

print_reachable_states -v

```
------- State    1 ------
  signal = TRUE
  shared = 2
  m0.pc = l0
  m1.pc = l0
  ------- State    2 ------
  signal = FALSE
  shared = 2
  m0.pc = l0
  m1.pc = l0
  ------- State    3 ------
  signal = TRUE
  shared = 0
  m0.pc = l0
  m1.pc = l0
```

```
------- State    4 ------
signal = FALSE
shared = 0
m0.pc = l0
m1.pc = l0
------- State    5 ------
signal = TRUE
shared = 1
m0.pc = l0
m1.pc = l0
------- State    6 ------
signal = FALSE
shared = 1
m0.pc = l0
m1.pc = l0
-------------------------
```

print_reachable_states -f

```
((m0.pc = l0 & m1.pc = l0) & (shared = 2 | (shared = 1 | shared = 0)))
```

Fix it

Let's try and fix the model

Q: What are the issues of this mutual exclusion model?
Q: Can we fix them?
Q: How?

Correctness of the model

Is the model correct?
We hope so, but we cannot be sure.

We can increase our confidence by trying to verify/falsify other properties.

Distance between states
We might want to compute the number of steps required to go from one set of states to another.

COMPUTE

`COMPUTE MIN [start, end];`
minimum number of steps required to reach a state in `end`, starting from a state in `start`. `infinity` if unreachable.

`COMPUTE MAX [start, end];`
maximum number of steps required to reach a state in `end`, starting from a state in `start`. `infinity` if unbounded, `undefined` if unreachable.

- `start` and `end` can be CTL formulae.
- `check_compute` command tells NUXMV to evaluate the `COMPUTE` statements.

COMPUTE example

Recall our simple mutual exclusion protocol

- **Q** What's the value of
  `COMPUTE MIN [m0.pc = l0, m0.pc = l2];` ?
- **Q** What's the value of
  `COMPUTE MAX [m0.pc = l0, m0.pc = l2];` ?

Finally, TEST!

We have seen some automated ways to increase our confidence in the correctness of the model.
All these techniques help us if we know what to look for.
Otherwise we can always perform simulations and look at what is happening.

- `pick_state`, `simulate`: we have already seen these ones.
- `read_trace`: load a trace from a file.
- `execute_traces`: checks whether all stored traces are in the language of the model.
- `execute_partial_traces`: tries to complete the trace such that it is a valid execution of the model.