The NuSMV Model Checker

Program Verification - Laborator

FMI · Denisa Diaconescu · Spring 2022

NuSMV

- NuSMV (sometimes called simply SMV) is a symbolic model checker
- NuSMV stands for "New Symbolic Model Verifier"
- `http://nusmv.fbk.eu/`
- The NuSMV project aims at the development of a state-of-the-art model checker that
  - is robust, open, and customizable;
  - can be applied in technology transfer projects;
  - can be used as research tool in different domains.
- NuSMV is an Open Source product
- NuSMV User manual

NuSMV

NuSMV provides

1. A language for describing finite state models of systems
   - Reasonably expressive
   - Allows for modular construction of models
2. Model checking algorithms for checking specifications written in LTL and CTL

A first SMV program

```
MODULE main
  VAR
    b0 : boolean;
  ASSIGN
    init(b0) := FALSE;
    next(b0) := !b0;
```

An SMV program consists of:

- Declarations of state variables
  - `b0` in the example
  - these determine the state space of the model
- Assignments that constrain the valid initial states
  - `init(b0) := FALSE`
- Assignments that constrain the transition relation
  - `next(b0) := !b0`

Declaring state variables

SMV data types include:

- boolean

  `x : boolean;`

- enumeration

  `st : {ready, busy, waiting, stopped};`

- bounded integers (intervals)

  `n : 1..8;`

- arrays and bit-vectors

  ```
  arr : array 0..3 of {red, green, blue};
  bv : signed word[8];
  ```

Assignments

- initialisation:

```
ASSIGN
init(x) := expression;
```

- progression:

```
ASSIGN
next(x) := expression;
```

- immediate:

```
ASSIGN                    DEFINE
y := expression;          y := expression;
```

Assignments

- If no init() assignment is specified for a variable, then it is initialised non-deterministically.
- If no next() assignment is specified, then it evolves non-deterministically (i.e., it is unconstrained)
  - unconstrained variables can be used to model non-deterministic inputs to the system
- Immediate assignments constrain the current value of a variable in terms of the current values of other variables.
  - Immediate assignments can be used to model outputs of the system.

Expressions

$$
\begin{array}{lcll}
\textit{expr} & ::= & \text{atom} & \text{symbolic constant} \\
& | & \text{number} & \text{numeric constant} \\
& | & \text{id} & \text{variable identifier} \\
& | & !\ \textit{expr} & \text{logical not} \\
& | & \textit{expr} \heartsuit \textit{expr} & \text{binary operation} \\
& | & \textit{expr}[\textit{expr}] & \text{array lookup} \\
& | & \texttt{next}(\textit{expr}) & \text{next value} \\
& | & \textit{case\_expr} & \\
& | & \textit{set\_expr} &
\end{array}
$$

where $\heartsuit \in \{\&, |, +, -, *, /, =, != , <, <=, \ldots\}$

Case expressions

$$
\begin{array}{l}
\textit{case\_expr} \;\; ::= \\
\quad \texttt{case} \\
\qquad \textit{expr}_{a_1} \quad : \quad \textit{expr}_{b_1}; \\
\qquad\quad \ldots \\
\qquad \textit{expr}_{a_n} \quad : \quad \textit{expr}_{b_n}; \\
\quad \texttt{esac}
\end{array}
$$

- Guards are evaluated sequentially.
- The first true guard determines the resulting value.

Set expressions

Expressions in SMV do not necessarily evaluate to one value.

- In general, they can represent a set of possible values.
  `init(var) := {a,b,c} union {x,y,z};`
- Destination (lhs) can take any value in the set represented by the set expression (rhs)
- A constant `c` is a syntactic abbreviation for singleton $\{c\}$

LTL specifications

- LTL properties are specified with the keywords LTLSPEC:

```
LTLSPEC <ltl_expression>;
```

- <ltl_expression> can contain the temporal operators

```
X_  F_  G_  _U_
```

- For example, *condition out = 0 holds until reset becomes false:*

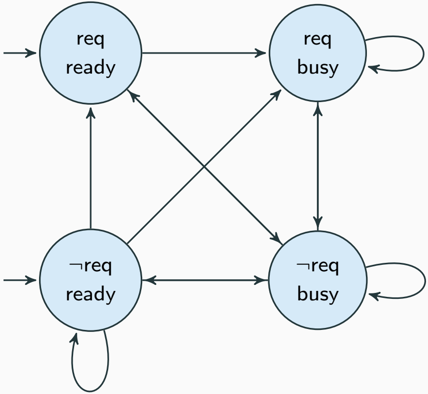
```
LTLSPEC (out = 0) U (!reset)
```

Program 2

```
MODULE main
  VAR
    request : boolean;
    status : {ready, busy};
  ASSIGN
    init(status) := ready;
    next(status) := case
                      request : busy;
                      TRUE : {ready, busy};
                    esac;
  LTLSPEC G(request -> F status = busy)
  LTLSPEC G F status = busy
```

Program 2

The model corresponding to the SMV program 2:



Note that we wrote "`busy`" as a shorthand for "`status = busy`" and "`req`" for "`request` is true".

Running NuSMV

Command Line

$ *path*/*to*/NuSMV fileName.smv

Note the results for Program 2:

- The LTL formula `G(request -> F status = busy)` is true
- The LTL formula `G F status = busy` is false and a counterexample is provided

Modules

- SMV programs consists of one or more modules
- One of the modules must be called `main`
- A module is instantiated when a variable having that module name as its type is declared

Program 3: a bit counter

- A model of a three bit binary counter circuit
- Uses three single-bit counters
- The module `counter_cell` is instantiated three times, with the names `bit0`, `bit1`, and `bit2`.
- The `counter_cell` module has one formal parameter, `carry_in`
  - in `bit0` is given the actual value `TRUE`
  - in `bit1` is `bit0.carry_out`
  - in `bit2` is `bit1.carry_out`

```
MODULE main
  VAR
    bit0 : counter_cell(TRUE);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
  LTLSPEC
    G F bit2.carry_out
```

Program 3: a bit counter

```
MODULE counter_cell(carry_in)
  VAR
    value : boolean;
  ASSIGN
    init(value) := FALSE;
    next(value) := value xor carry_in;
  DEFINE
    carry_out := value & carry_in;
```

- The effect of `DEFINE` statement could have been obtained by declaring a new variable and assigning its value.
- Defined symbols are usually preferable to variables, since they don't increase the state space by declaring new variables.
- Defined symbols cannot be assigned nondeterministically.

Program 3: a counter

Exercise: Redefine Program 3 using the variable in `counter_cell` to be

```
value : {0,1}
```

Synchronous and asynchronous composition

- By default, modules are composed synchronous
  - this means that there is a global clock and, each time it ticks, each of the modules executes in parallel
  - the bit counter example is synchronous
- By using the keyword `process`, it is possible to compose the modules asynchronous.
  - they run at different "speads", interleaving arbitrarily
  - at each tick of the clock, one of them is non-deterministically chosen and executed for one cycle.
  - asynchronous interleaving composition is useful for describing communication protocols, asynchronous circuits etc.

Program 4: The alternating bit protocol

- The alternating bit protocol (ABP) is a protocol for transmitting messages along a ”lossy line” (a line which may lose or duplicate messages).
- The protocol guarantees that, **providing that the line doesn't lose infinitely many messages**, communication between the sender and the receiver will be successful.
- We allow the line to lose or duplicate messages, but **it may not corrupt messages!**

Program 4: The alternating bit protocol

ABP works as follows.

- There are four entities:

1.	The Sender	3.	The message channel
2.	The Receiver	4.	The acknowledge channel

- The Sender transmits the first part of the message together with the control bit 0.
- If, and when, the Receiver receives a message with the control bit 0, it send 0 along the acknowledgement channel.
- When the Sender receives this acknowledgement, it sends the next packet with the control bit 1.
- If, and when, the Receiver receives this, it acknowledge by sending a 1on the acknowledgement channel.

Program 4: The alternating bit protocol

- By alternating the control bit, both receiver and sender can guard against duplicating messages and losing messages
(i.e., they ignore messages that have the unexpected control bit).
- If the Sender doesn't get the expected acknowledgement, it continually resends the message, until the acknowledge arrives.
- If the Receiver doesn't get a message with the expected control bit, it continually resends the previous acknowledgement.

Program 4: The alternating bit protocol

- Although we want to model the fact that the channel can lose messages, we want to assume that, if we send a message often enough, eventually it will arrive.
- **We assume that the channel cannot lose an infinite sequence of messages.**
- Otherwise, the channels could lose all messages and, in this case, the ABP would not work.
- We implement this assumption by introducing a fairness constraint.
  - keyword `FAIRNESS`
  - the occurrence of `FAIRNESS` φ means that, when checking a specification, we will ignore any path along which φ is not satisfied infinitely often.
  - `FAIRNESS running` restricts attention to paths along which the module in which it appears is selected for execution infinitely often.

ABP: module Sender

The module for Sender:

```
MODULE sender(ack)
VAR
  st : {sending, sent};
  message1 : {0,1};
  message2 : {0,1};
ASSIGN
   init(st) := sending;
   next(st) :=
              case
                ack = message2 & !(st = sent) : sent;
                TRUE : sending;
              esac;
```

ABP: module Sender (cont.)

```
    next(message1) :=
                        case
                          st = sent : {0,1};
                          TRUE : message1;
                        esac;
    next(message2) :=
                        case
                          st = sent : 1 - message2;
                          TRUE : message2;
                        esac;
FAIRNESS running
LTLSPEC G F st = sent
```

ABP: module Sender

- `message1` is the current bit being sent
- The new `message1` is obtained non-deterministically (i.e., from environment)
- `message2` is the control bit
- The module goes into `st=sent` only when it receives an acknowledgement corresponding to the control bit of the message it has been sending
- We impose `FAIRNESS running` meaning that the Sender must be selected to run infinitely often
- The `LTLSPEC` tests that we can always succeed in sending the current message

ABP: module Receiver

The module for Receiver is similar:

```
MODULE receiver(message1,message2)
VAR
  st : {receiving, received};
  ack : {0,1};
  expected : {0,1};
ASSIGN
   init(st) := receiving;
   next(st) :=
     case
       message2 = expected & !(st = received) : received;
       TRUE : receiving;
     esac;
```

ABP: module Receiver (cont.)

```
    next(ack) :=
                case
                  st = received : message2;
                  TRUE : ack;
                esac;
    next(expected) :=
                case
                  st = received : 1 - expected;
                  TRUE : expected;
                esac;
FAIRNESS running
LTLSPEC G F st = received
```

ABP: module one-bit-chanel

The Acknowledgement channel is an instance of the one-bit-channel:

```
MODULE one-bit-chan(input)
VAR
  forget : boolean;
  output : {0,1};
ASSIGN
   next(output) :=
       case
         forget : output;
         TRUE : input;
       esac;
FAIRNESS running
FAIRNESS (input = 1) & !forget
FAIRNESS (input = 0) & !forget
```

ABP: module one-bit-chanel

- Its lossy character is specified by the assignment to `forget`
- By the fairness constraint, we assume that the channel infinitely often transmit the message correctly.
- Note that the fairness constraint **"infinitely often `!forget`"** is not sufficient to prove the desired properties.
    - although it forces the channel to transmit infinitely often, it doesn't prevent it from (say) dropping all the 0 bits and transmitting all the 1 bits.

ABP: module two-bit-chanel

The channel for sending messages is an instance of the two-bit-channel:

```
MODULE two-bit-chan(input1,input2)
VAR
  forget : boolean;
  output1 : {0,1};
  output2 : {0,1};
ASSIGN
   next(output1) :=
       case
         forget : output1;
         TRUE : input1;
       esac;
```

ABP: module two-bit-chanel (cont.)

```
   next(output2) :=
       case
         forget : output2;
         TRUE : input2;
       esac;
FAIRNESS running
FAIRNESS (input1 = 1) & !forget
FAIRNESS (input1 = 0) & !forget
FAIRNESS (input2 = 1) & !forget
FAIRNESS (input2 = 0) & !forget
```

ABP

We tie all together in the module `main`

```
MODULE main
VAR
 s : process sender(ack_chan.output);
 r : process receiver(msg_chan.output1,msg_chan.output2);
 msg_chan : process two-bit-chan(s.message1,s.message2);
 ack_chan : process one-bit-chan(r.ack);
ASSIGN
 init(s.message2) := 0;
 init(r.expected) := 0;
 init(r.ack) := 1;
 init(msg_chan.output2) := 1;
 init(ack_chan.output) := 1;
LTLSPEC G (s.st = sent & s.message1 = 1
                           -> msg_chan.output1 = 1)
```

ABP

- Since the first control bit is 0, we initialise the Receiver to expect a 0.
- The Receiver should start off by sending 1 as its acknowledgment, so that the Sender does not think that its very first message is being acknowledged before anything has happened.
- For the same reason, the output of the channels is initialised to 1.

The specifications for ABP

Our SMV program satisfies the following specifications:

- Safety: If the message bit 1 has been sent and the correct acknowledgement has been returned, then a 1 was indeed received by the Receiver:

```
G (s.st = sent & s.message1 = 1
                  -> msg_chan.output1 = 1)
```

- Liveness: Messages get through eventually.