Model Checking with NuSMV/nuXmv

Paul Jackson
Paul.Jackson@ed.ac.uk

University of Edinburgh

Formal Verification
Autumn 2023
NuSMV

NuSMV is a symbolic model checker for finite-state systems developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.

http://nusmv.fbk.eu/

- Main algorithms supported:
  - Symbolic model checking using BDDs (Binary Decision Diagrams)
  - Bounded model checking using SAT-solver reasoning engine

NuSMV is open source
nuXmv

Successor tool to NuSMV

http://nuxmv.fbk.eu/

- Algorithms for both finite-state and infinite state systems
- Uses both SAT and SMT reasoning engines
- Algorithms supported include
  - Interpolation-based invariant checking
    - Interpolants are formulas that summarise useful features of reachable state sets
  - IC3 – unbounded model checking using SAT
  - K-induction – another approach to unbounded model checking using SAT
  - CEGAR (Counter-Example-Guided Abstraction Refinement)

Not open source, but binaries freely available for academic purposes
a first SMV program

MODULE main
VAR
  b0 : boolean;
ASSIGN
  init(b0) := FALSE;
  next(b0) := !b0;
LTLSPEC
  G F (b0 & X ! 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) := 0).
  ▶ Assignments that constrain the transition relation (next(b0) := !b0).

Program followed by properties to check
Declaring state variables

SMV data types include:

**boolean:**

\[ x : \text{boolean}; \]

**enumeration:**

\[ st : \{\text{ready, busy, waiting, stopped}\}; \]

**bounded integers (intervals):**

\[ n : 1..8; \]

**arrays and bit-vectors**

\[ arr : \text{array 0..3 of \{red, green, blue\}}; \]

\[ bv : \text{signed word[8]}; \]
Assignments

initialisation:

ASSIGN
init(x) := expression ;

progression:

ASSIGN
next(x) := expression ;

immediate:

ASSIGN
y := expression ;

or

DEFINE
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 nondeterministically. i.e. it is unconstrained.
  - Unconstrained variables can be used to model nondeterministic inputs to the system.
- Immediate ASSIGN 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.
- `DEFINE` declarations are like macros in C/C++
  - LHS is *not* a declared state variable
Expressions

\[ \text{expr :: atom symbolic constant} \]
\[ \mid \text{number numeric constant} \]
\[ \mid \text{id variable identifier} \]
\[ \mid \! \text{expr logical not} \]
\[ \mid \text{expr op expr } op \text{ one of } \&, |,+,-,*,/,,=,!=,<,<=,\ldots \]
\[ \mid \text{expr [ index ] array element} \]
\[ \mid \text{next ( id ) next value} \]
\[ \mid \text{case_expr} \]
\[ \mid \text{set_expr} \]
Case Expression

case_expr :: case
   expr_a1 : expr_b1 ;
   ...
   expr_an : expr_bn ;
   TRUE : default ;
esac

- Guards are evaluated sequentially.
- The first true guard determines the resulting value
Expressions in SMV do not necessarily evaluate to one value.

- In general, they can represent a set of possible values.
  \[
  \text{init}(\text{var}) := \{a,b,c\} \cup \{x,y,z\} ;
  \]
- destination (lhs) can take any value in the set represented by the set expression (rhs)
- constant \(c\) is a syntactic abbreviation for singleton \(\{c\}\)
LTL Specifications

- LTL properties are specified with the keyword LTLSPEC:
  LTLSPEC <ltl_expression> ;
- <ltl_expression> can contain the temporal operators:
  X  F  G  U
- E.g. condition out = 0 holds until reset becomes false:
  LTLSPEC (out = 0) U (!reset)
MODULE main
VAR
    state: {welcome, enterPin, tryAgain, askAmount,
            thanksGoodbye, sorry};
    input: {cardIn, correctPin, wrongPin, ack, fundsOK,
            problem, none};
ASSIGN
    init(state) := welcome;
    next(state) := case
        state = welcome & input = cardIn : enterPin;
        state = enterPin & input = correctPin : askAmount;
        state = enterPin & input = wrongPin : tryAgain;
        state = tryAgain & input = ack : enterPin;
        state = askAmount & input = fundsOK : thanksGoodbye;
        state = askAmount & input = problem : sorry;
        TRUE : state;
    esac;
LTLSPEC F( G state = thanksGoodbye
          | G state = sorry
      );
init(state) := welcome;
next(state) := case
  state = welcome & input = cardIn : enterPin;
  state = enterPin & input = correctPin : askAmount;
  state = enterPin & input = wrongPin : tryAgain;
  state = tryAgain & input = ack : enterPin;
  state = askAmount & input = fundsOK : thanksGoodbye;
  state = askAmount & input = problem : sorry;
  TRUE : state;
esac;
Property 1

LTLSPEC NAME s1 :=
    F ( G state = thanksGoodbye
        | G state = sorry
    );

welcome  enterPin  askAmount  thanksGoodbye
          cardIn  correctPin  fundsOK
         wrongPin  ack  tryAgain
                                           problem
                                                                   sorry
Running NuSMV or nuXmv

**Batch**

% nuXmv atm.smv

**Interactive**

% nuXmv -int atm.smv
nuXmv > go
nuXmv > check_ltlspec
nuXmv > quit

- go abbreviates the sequence of commands read_model, flatten_hierarchy, encode_variables, build_model.
- For command options, use -h or look in NuSMV User Manual
nuXmv Check of Property 1

nuXmv > check_ltlspec -P s1
-- specification F ( G state = thanksGoodbye | G state = sorry)
    is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
    -> State: 1.1 <-
        state = welcome
        input = cardIn
    -- Loop starts here
    -> State: 1.2 <-
        state = enterPin
    -> State: 1.3 <-
LTL SPEC\ NAME s2 :=
\ G (\n\hspace{1cm} (state = welcome -> F \ input = \ cardIn) \&\n\hspace{1cm} (state = enterPin -> F (input = correctPin | input = wrongPin)) \&\n\hspace{1cm} (state = askAmount -> F (input = fundsOK | input = problem)) \&\n\hspace{1cm} (state = tryAgain -> F \ input = \ ack)\n\hspace{1cm})
\hspace{1cm} -> F (G state = thanksGoodbye | G state = sorry ) ;
nuXmv Check of Property 2

Trace Type: Counterexample

--> State: 2.1 <-
    state = welcome
    input = cardIn

--> State: 2.2 <-
    state = enterPin
    input = ack

--> State: 2.3 <-
    input = wrongPin

--> State: 2.4 <-
    state = tryAgain
    input = cardIn

-- Loop starts here

--> State: 2.5 <-
    input = ack

--> State: 2.6 <-
    state = enterPin
    input = wrongPin

--> State: 2.7 <-
    state = tryAgain
    input = ack
Property 3

LTLSPEC NAME s3 :=
G (  
    (state = welcome  -> F input = cardIn) \&
    (state = enterPin  -> F (input = correctPin | input = wrongPin)) \&
    (state = askAmount  -> F (input = fundsOK | input = problem)) \&
    (state = tryAgain  -> F input = ack) \&
    (state = enterPin  -> F (state = enterPin \& input = correctPin))
) 
-> F( G state = thanksGoodbye \| G state = sorry ) ;
nuXmv Check of Property 3

nuXmv > check_ltlspec -P s3
-- specification
( G ((((state = welcome -> F input = cardIn) &
    (state = enterPin ->
     F (input = correctPin | input = wrongPin))) &
    (state = askAmount ->
     F (input = fundsOK | input = problem))) &
    (state = tryAgain -> F input = ack)) &
    (state = enterPin ->
     F (state = enterPin & input = correctPin)))
-> F ( G state = thanksGoodbye | G state = sorry))
is true
Modules

MODULE counter
VAR digit : 0..9;
ASSIGN
    init(digit) := 0;
    next(digit) := (digit + 1) mod 10;

MODULE main
VAR c0 : counter;
    c1 : counter;
    sum : 0..99;
ASSIGN
    sum := c0.digit + 10 * c1.digit;

- Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- In each SMV specification there must be a module main. It is the top-most module.
- All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., c0.digit, c1.digit).
Verification of 2 Digit Counter

MODULE counter
VAR
digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := (digit + 1) mod 10;

MODULE main
VAR
c0 : counter;
c1 : counter;
sum : 0..99;
ASSIGN
  sum := c0.digit + 10* c1.digit;

LTLSPEC
  F sum = 13;

★ Is this specification satisfied by this model?
nuXmv run on 2 Digit Counter

-- specification F sum = 13 is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
c0.digit = 0
c1.digit = 0
sum = 0
-> State: 1.2 <-
c0.digit = 1
c1.digit = 1
sum = 11
-> State: 1.3 <-
c0.digit = 2
c1.digit = 2
sum = 22
-> State: 1.4 <-
c0.digit = 3
c1.digit = 3
sum = 33
...
...
MODULES WITH PARAMETERS

MODULE counter(inc)
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := inc ? (digit + 1) mod 10
                   : digit;
DEFINE top := digit = 9;

MODULE main
VAR c0 : counter(TRUE);
  c1 : counter(c0.top);
  sum : 0..99;
ASSIGN
  sum := c0.digit + 10 * c1.digit;

▶ Formal parameters (inc) are substituted with the actual parameters (TRUE, c0.top) when the module is instantiated.
nuXmv run on 2 Digit Counter Using Parameters

% nuXmv count100.smv

...

-- specification F sum = 13 is true