NuSMV 2.2 Tutorial

Roberto Cavada, Alessandro Cimatti,
Gavin Keighren, Emanuele Olivetti,
Marco Pistore and Marco Roveri

ITC-irst - Via Sommarive 18, 38055 Povo (Trento) – Italy

Email: `nusmv@irst.itc.it`

Contents

Chapter 1

Introduction

In this tutorial we give a short introduction to the usage of the main functionalities of NuSMV. In Chapter 2 [Examples], page 3 we describe the input language of NuSMV by presenting some examples of NuSMV models. Chapter 3 [Simulation], page 8 shows how the user can get familiar with the behavior of a NuSMV model by exploring its possible executions. Chapter 4 [CTL Model Checking], page 13 and Chapter 5 [LTL Model Checking], page 17 give an overview of BDD-based model checking, while Chapter 6 [Bounded Model Checking], page 20 presents SAT-based model checking in NuSMV.

Chapter 2

Examples

In this section we describe the input language of NuSMV by presenting some examples of NuSMV models. A complete description of the NuSMV language can be found in the NuSMV 2.2 User Manual. Also, all mentioned example files can be found in the distributed archive of NuSMV 2.2, or can be individually downloaded from the NuSMV web pages, at the URL `<http://nusmv.irst.itc.it/examples/examples.html>`.

The input language of NuSMV is designed to allow for the description of Finite State Machines (FSMs from now on) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can specify a system as a synchronous Mealy machine, or as an asynchronous network of nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones – booleans, scalars and fixed arrays. Static data types can also be constructed.

The primary purpose of the NuSMV input is to describe the transition relation of the FSM; this relation describes the valid evolutions of the state of the FSM. In general, any propositional expression in the propositional calculus can be used to define the transition relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The NuSMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in NuSMV is similar to that of single assignment data flow language. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a description language, or a programming language.

2.1 Synchronous Systems

2.1.1 Single Process Example

Consider the following simple program in the NuSMV language:

```
MODULE main
VAR
  request : boolean;
  state   : {ready, busy};
```

```
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request = 1 : busy;
                   1                           : {ready, busy};
                 esac;
```

The space of states of the FSM is determined by the declarations of the state variables (in the above example `request` and `state`). The variable `request` is declared to be of (predefined) type `boolean`. This means that it can assume the (integer) values `0` and `1`. The variable `state` is a scalar variable, which can take the symbolic values `ready` or `busy`. The following assignment sets the initial value of the variable `state` to `ready`. The initial value of `request` is completely unspecified, i.e. it can be either `0` or `1`. The transition relation of the FSM is expressed by defining the value of variables in the next state (i.e. after each transition), given the value of variables in the current states (i.e. before the transition). The `case` segment sets the next value of the variable `state` to the value `busy` (after the colon) if its current value is `ready` and `request` is `1` (i.e. true). Otherwise (the `1` before the colon) the next value for `state` can be any in the set {`ready, busy`}. The variable `request` is not assigned. This means that there are no constraints on its values, and thus it can assume any value. `request` is thus an unconstrained input to the system.

2.1.2 Binary Counter

The following program illustrates the definition of reusable modules and expressions. It is a model of a three bit binary counter circuit. The order of module definitions in the input file is not relevant.

```
MODULE counter_cell(carry_in)
 VAR
   value : boolean;
 ASSIGN
   init(value) := 0;
   next(value) := (value + carry_in) mod 2;
 DEFINE
  carry_out := value & carry_in;
MODULE main
 VAR
   bit0 : counter_cell(1);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
```

The FSM is defined by instantiating three times the module type `counter_cell` in the module `main`, with the names `bit0`, `bit1` and `bit2` respectively. The `counter_cell` module has one formal parameter `carry_in`. In the instance `bit0`, this parameter is given the actual value `1`. In the instance `bit1`, `carry_in` is given the value of the expression `bit0.carry_out`. This expression is evaluated in the context of the `main` module. However, an expression of the form '$a.b$' denotes component 'b' of module 'a', just as if the module 'a' were a data structure in a standard programming language. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`.

The keyword 'DEFINE' is used to assign the expression `value & carry_in` to the symbol `carry_out`. A definition can be thought of as a variable with value (functionally) depending on the current values of other variables. The same effect could have been obtained as follows (notice that the *current* value of the variable is assigned, rather than the *next* value.):

```
VAR
  carry_out : boolean;
ASSIGN
  carry_out := value & carry_in;
```

Defined symbols do not require introducing a new variable, and hence do not increase the state space of the FSM. On the other hand, it is not possible to assign to a defined symbol a value non-deterministically. Another difference between defined symbols and variables is that while the type of variables is declared a priori, for definitions this is not the case.

2.2 Asynchronous Systems

The previous examples describe synchronous systems, where the assignments statements are taken into account in parallel and simultaneously. NuSMV allows to model asynchronous systems. It is possible to define a collection of parallel processes, whose actions are interleaved, following an asynchronous model of concurrency. This is useful for describing communication protocols, or asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock region).

2.2.1 Inverter Ring

The following program represents a ring of three asynchronous inverting gates.

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
   init(output) := 0;
   next(output) := !input;
MODULE main
 VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);
```

Among all the modules instantiated with the `process` keyword, one is nondeterministically chosen, and the assignment statements declared in that process are executed in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates.

We remark that the system is not forced to eventually choose a given process to execute. As a consequence the output of a given gate may remain constant, regardless of its input. In order to force a given process to execute infinitely often, we can use a fairness constraint. A fairness constraint restricts the attention of the model checker to only those execution paths along which a given formula is true infinitely often. Each process has a special variable called `running` which is `1` if and only if that process is currently executing.
By adding the declaration:

```
FAIRNESS
  running
```

to the module `inverter`, we can effectively force every instance of `inverter` to execute infinitely often.
An alternative to using processes to model an asynchronous circuit is to allow all gates to execute simultaneously, but to allow each gate to choose non-deterministically to re-evaluate its output or to keep the same output value. Such a model of the inverter ring would look like the following:

```
MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
  init(output) := 0;
  next(output) := (!input) union output;
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate1.output);
   gate3 : inverter(gate2.output);
```

The `union` operator (set union) coerces its arguments to singleton sets as necessary. Thus, the next `output` of each gate can be either its current `output`, or the negation of its current `input` – each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as 2^n, where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation. We remark that in this case we cannot force the inverters to be effectively active infinitely often using a fairness declaration. In fact, a valid scenario for the synchronous model is the one where all the inverters are idle and assign to the next `output` the current value of `output`.

2.2.2 Mutual Exclusion

The following program is another example of asynchronous model. It uses a variable semaphore to implement mutual exclusion between two asynchronous processes. Each process has four states: `idle`, `entering`, `critical` and `exiting`. The `entering` state indicates that the process wants to enter its critical region. If the variable `semaphore` is `0`, it goes to the `critical` state, and sets `semaphore` to `1`. On exiting its critical region, the process sets `semaphore` to `0` again.

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
MODULE user(semaphore)
 VAR
   state : {idle, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                  : {idle, entering};
```

```
        state = entering & !semaphore : critical;
        state = critical              : {critical, exiting};
        state = exiting               : idle;
        1                             : state;
      esac;
    next(semaphore) :=
      case
        state = entering : 1;
        state = exiting  : 0;
        1                : semaphore;
      esac;
  FAIRNESS
    running
```

2.3 Direct Specification

NUSMV allows to specify the FSM directly in terms of propositional formulas. The set of possible initial states is specified as a formula in the current state variables. A state is initial if it satisfies the formula. The transition relation is directly specified as a propositional formula in terms of the *current* and *next* values of the state variables. Any current state/next state pair is in the transition relation if and only if it satisfies the formula.

These two functions are accomplished by the '`INIT`' and '`TRANS`' keywords. As an example, here is a description of the three inverter ring using only `TRANS` and `INIT`:

```
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate1.output);
   gate3 : inverter(gate2.output);
MODULE inverter(input)
 VAR
   output : boolean;
 INIT
   output = 0
 TRANS
   next(output) = !input | next(output) = output
```

According to the `TRANS` declaration, for each inverter, the next value of the `output` is equal either to the negation of the `input`, or to the current value of the `output`. Thus, in effect, each gate can choose non-deterministically whether or not to delay.

Using `TRANS` and `INIT` it is possible to specify inadmissible FSMs, where the set of initial states is empty or the transition relation is not total. This may result in logical absurdities.

Chapter 3

Simulation

Simulation offers to the user the possibility of exploring the possible executions (*traces* from now on) of a NUSMV model. In this way, the user can get familiar with a model and can acquire confidence with its correctness before the actual verification of properties. This section describes the basic features of simulation in NUSMV. Further details on the simulation commands can be found in the NuSMV 2.2 User Manual.

3.1 Trace Strategies

In order to achieve maximum flexibility and degrees of freedom in a simulation session, NUSMV permits three different trace generation strategies: deterministic, random and interactive. Each of them corresponds to a different way a state is picked from a set of possible choices. In deterministic simulation mode the first state of a set (whatever it is) is chosen, while in the random one the choice is performed nondeterministically. In these two first modes traces are automatically generated by NUSMV: the user obtains the whole of the trace in a time without control over the generation itself (except for the simulation mode and the number of states entered via command line).

In the third simulation mode, the user has a complete control over traces generation by interactively building the trace. During an interactive simulation session, the system stops at every step, showing a list of possible future states: the user is requested to choose one of the items. This feature is particularly useful when one wants to inspect some particular reactions of the model to be checked. When the number of possible future states exceeds an internal limit, rather than "confusing" the user with a choice from a high number of possible evolutions, the system asks the user to "guide" the simulation via the insertion of some further constraints over the possible future states. The system will continue to ask for constraints insertion until the number of future states will be under the predefined threshold. The constraints entered during this phase are accumulated (in a logical product) in a single big constraint. This constraint is used only for the current step of the simulation and is discarded before the next step. The system checks the expressions entered by the user and does not accept them whenever an inconsistency arises. Cases of inconsistency (i.e. empty set of states) may be caused by:

- the entered expressions (i.e. `a & ~ a`);
- the result of the entered expressions conjoined with previous accumulated ones;
- the result of accumulated constraints conjoined with the set of possible future states.

3.2 Interactive Mode

A typical execution sequence of a simulation session could be as follows. Suppose we use the model described below.

```
MODULE main
VAR
  request : boolean;
  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request : busy;
                   1                       : {ready,busy};
                 esac;
```

As a preliminary step, this model has to read into the NuSMV system. This can be obtained by executing the following commands (we assume that the model is saved in file `short.smv`): [1]

```
system_prompt> NuSMV -int short.smv
NuSMV> go
NuSMV>
```

3.2.1 Choosing an Initial State

In order to start the simulation, an initial state has to be chosen. This can be done in three ways:

- by default, the simulator uses the *current state* as a starting point of every new simulation; this behavior if possible only if a current state is defined (e.g., if we are exploring a trace);
- if command `goto_state` is used, the user can select any state of an already existent trace as the *current state*;
- if `pick_state` is used, then the user can choose the starting state of the simulation among the initial states of the model; this command has to be used when a *current state* does not exist yet (that is when the model has not yet been processed or when the system has been reset).

At this point of the example *current state* does not exist, and there is no trace currently stored in the system. Therefore, an item from the set of initial states has to be picked using command `pick_state`. A simulation session can be started now, using the `simulate` command. Consider for instance the following simulation session:

```
system_prompt> NuSMV -int short.smv
NuSMV> go
NuSMV> pick_state -r
NuSMV> print_current_state -v
Current state is 1.1
request = 0
state = ready
```

[1]We assume that every NuSMV command is followed by a `<RET>` keystroke. In the following examples, NuSMV commands are written **`like this`** to distinguish them from system output messages.

```
NuSMV> simulate -r 3
*********  Starting Simulation From State  1.1  *********
NuSMV> show_traces -t
There is 1 trace currently available.
NuSMV> show_traces -v
#################### Trace number: 1 ####################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
    request = 0
    state = ready
-> State: 1.2 <-
    request = 1
    state = busy
-> State: 1.3 <-
    request = 1
    state = ready
-> State: 1.4 <-
    request = 1
    state = busy
```

Command **pick_state -r** requires to pick the starting state of the simulation *randomly* from the set of initial states of the model. Command **simulate -r 3** asks to build a three-steps simulation by picking randomly the next states of the steps. As shown by command **show_traces -v**, the resulting trace contains 4 states (the initial one, and the three ones that have been added by the random simulation). We remark that the generated traces are numbered: every trace is identified by an integer number, while every state belonging to a trace is identified by a *dot notation*: for example state 1.3 is the third state of the first generated trace.

3.2.2 Starting a New Simulation

Now the user can start a new simulation by choosing a new starting state. In the next example, for instance, the user extends trace 1 by first choosing state 1.4 as the *current state* and by then running a random simulation of length 3.

```
NuSMV> goto_state 1.4
The starting state for new trace is:
-> State 2.4 <-
    request = 1
    state = busy
NuSMV> simulate -r 3
********  Simulation Starting From State  2.4  ********
NuSMV> show_traces 2
################### Trace number: 2 ###################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 2.1 <-
    request = 1
    state = ready
-> State: 2.2 <-
    state = busy
-> State: 2.3 <-
    request = 0
-> State: 2.4 <-
```

```
    request = 1
-> State: 2.5 <-
    request = 0
-> State: 2.6 <-
    state = ready
-> State: 2.7 <-
NuSMV>
```

As the reader can see from the previous example, the new trace is stored as trace 2. The user is also able to interactively choose the states of the trace he wants to build: an example of an interactive simulation is shown below:

```
NuSMV> pick_state -i

***************  AVAILABLE STATES  *************

================= State =================
0) -------------------------
    request = 1
    state = ready

================= State =================
1) -------------------------
    request = 0
    state = ready


Choose a state from the above (0-1): 1 <RET>

Chosen state is: 1
NuSMV> simulate -i 1
********  Simulation Starting From State  3.1  ********

***************  AVAILABLE FUTURE STATES  *************

================= State =================
0) -------------------------
    request = 1
    state = ready

================= State =================
1) -------------------------
    request = 1
    state = busy

================= State =================
2) -------------------------
    request = 0
    state = ready

================= State =================
3) -------------------------
    request = 0
    state = busy
```

```
Choose a state from the above (0-3): 0 <RET>

Chosen state is: 0
NuSMV> show_traces 3
################### Trace number: 3 ###################
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 3.1 <-
    request = 0
    state = ready
-> State: 3.2 <-
    request = 1
```

3.2.3 Specifying Constraints

The user can also specify some constraints to restrict the set of states from which the simulator will pick out. Constraints can be set for both the `pick_state` command and the `simulate` command using option `-c`. For example the following command picks an initial state by defining a simple constraint:

```
NuSMV> pick_state -c "request = 1" -i

***************  AVAILABLE STATES  ***************

================= State =================
0) -------------------------
    request = 1
    state = ready


There's only one future state. Press Return to Proceed. <RET>

Chosen state is: 0
NuSMV> quit
system_prompt>
```

Note how the set of possible states to choose has being restricted (in this case there is only one future state, so the system will automatically pick it, waiting for the user to press the `<RET>` key). We remark that, in the case of command `simulate`, the constraints defined using option `-c` are "global" for the actual trace to be generated, in the sense that they are always included in every step of the simulation. They are hence complementary to the constraints entered with the `pick_state` command, or during an interactive simulation session when the number of future states to be displayed is too high, since these are "local" only to a single simulation step and are "forgotten" in the next one.

Chapter 4

CTL Model Checking

The main purpose of a model checker is to verify that a model satisfies a set of desired properties specified by the user. In NuSMV, the specifications to be checked can be expressed in two different temporal logics: the Computation Tree Logic CTL, and the Linear Temporal Logic LTL extended with Past Operators. CTL and LTL specifications are evaluated by NuSMV in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, NuSMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property. In this section we will describe model checking of specifications expressed in CTL, while the next section we consider the case of LTL specifications.

4.1 Computation Tree Logic

CTL is a *branching-time* logic: its formulas allow for specifying properties that take into account the non-deterministic, branching evolution of a FSM. More precisely, the evolution of a FSM from a given state can be described as an infinite tree, where the nodes are the states of the FSM and the branching in due to the non-determinism in the transition relation. The paths in the tree that start in a given state are the possible alternative evolutions of the FSM from that state. In CTL one can express properties that should hold for *all the paths* that start in a state, as well as for properties that should hold just for *some of the paths*.

Consider for instance CTL formula `AF p`. It expresses the condition that, for *all* the paths (`A`) stating from a state, *eventually in the future* (`F`) condition `p` must hold. That is, all the possible evolutions of the system will eventually reach a state satisfying condition `p`. CTL formula `EF p`, on the other hand, requires than there *exists* some path (`E`) that eventually in the future satisfies `p`.

Similarly, formula `AG p` requires that condition `p` is always, or *globally*, true in all the states of all the possible paths, while formula `EG p` requires that there is some path along which condition `p` is continuously true.

Other CTL operators are:

- `A [p U q]` and `E [p U q]`, requiring condition `p` to be true *until* a state is reached that satisfies condition `q`;
- `AX p` and `EX p`, requiring that condition `p` is true in all or in some of the next states reachable from the current state.

CTL operators can be nested in an arbitrary way and can be combined using logic operators (`!`, `&`, `|`, `->`, `<->`, ...). Typical examples of CTL formulas are `AG ! p` ("condition `p` is absent in all the evolutions"), `AG EF p` ("it is always possible to reach a state where `p` holds"), and `AG (p -> AF q)` ("each occurrence of condition `p` is followed by an occurrence of condition `q`").

In NUSMV a CTL specification is given as CTL formula introduced by the keyword "SPEC". Whenever a CTL specification is processed, NUSMV checks whether the CTL formula is true in all the initial states of the model. If this is not a case, then NUSMV generates a counter-example, that is, a (finite or infinite) trace that exhibits a valid behavior of the model that does not satisfy the specification. Traces are very useful for identifying the error in the specification that leads to the wrong behavior. We remark that the generation of a counter-example trace is not always possible for CTL specifications. Temporal operators corresponding to existential path quantifiers cannot be proved false by a showing of a single execution path. Similarly, sub-formulas preceded by universal path quantifier cannot be proved true by a showing of a single execution path.

4.2 Semaphore Example

Consider the case of the semaphore program described in Chapter 2 [Examples], page 3. A desired property for this program is that it should never be the case that the two processes `proc1` and `proc2` are at the same time in the `critical` state (this is an example of a "safety" property). This property can be expressed by the following CTL formula:

```
AG ! (proc1.state = critical & proc2.state = critical)
```

Another desired property is that, if `proc1` wants to enter its critical state, it eventually does (this is an example of a "liveness" property). This property can be expressed by the following CTL formula:

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

In order to verify the two formulas on the semaphore model, we add the two corresponding CTL specification to the program, as follows:

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
 SPEC AG ! (proc1.state = critical & proc2.state = critical)
 SPEC AG (proc1.state = entering -> AF proc1.state = critical)
MODULE user(semaphore)
 VAR
   state : {idle, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                  : {idle, entering};
       state = entering & !semaphore : critical;
       state = critical              : {critical, exiting};
       state = exiting               : idle;
       1                             : state;
     esac;
   next(semaphore) :=
     case
       state = entering : 1;
       state = exiting  : 0;
```

```
      1                : semaphore;
    esac;
FAIRNESS
  running
```

By running NUSMV with the command

```
system_prompt> NuSMV semaphore.smv
```

we obtain the following output:

```
-- specification AG (!(proc1.state = critical & proc2.state = critical))
-- is true
-- specification AG (proc1.state = entering -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following execution sequence
-> State: 1.1 <-
    semaphore = 0
    proc1.state = idle
    proc2.state = idle
-> Input: 1.2 <-
    _process_selector_ = proc1
-- Loop starts here
-> State: 1.2 <-
    proc1.state = entering
-> Input: 1.3 <-
    _process_selector_ = proc2
-> State: 1.3 <-
    proc2.state = entering
-> Input: 1.4 <-
    _process_selector_ = proc2
-> State: 1.4 <-
    semaphore = 1
    proc2.state = critical
-> Input: 1.5 <-
    _process_selector_ = proc1
-> State: 1.5 <-
-> Input: 1.6 <-
    _process_selector_ = proc2
-> State 1.6 <-
    proc2.state = exiting
-> Input: 1.7 <-
    _process_selector_ = proc2
-> State 1.7 <-
    semaphore = 0
    proc2.state = idle
```

Note that `_process_selector_` is a special variable which contains the name of the process that will execute to cause a transition to the next state. The '`Input`' section displays the values of variables that the model has no control over, that is it cannot change their value. Since processes are chosen nondeterministically in this model, it has no control over the value of `_process_selector_`.
NUSMV tells us that the first CTL specification is true: it is never the case that the two processes will be at the same time in the critical region. On the other hand, the second specification is false. NUSMV produces a counter-example path where initially `proc1` goes to state `entering` (state 1.2), and then a loop starts in which `proc2` repeatedly enters its

critical region (state 1.4) and then returns to its `idle` state (state 1.7); in the loop, `proc1` is activated only when `proc2` is in the critical region (input 1.5), and is therefore not able to enter its critical region (state 1.5). This path not only shows that the specification is false, it also points out why can it happen that `proc1` never enters its critical region.

Note that in the printout of a cyclic, infinite counter-example the starting point of the loop is marked by `-- loop starts here`. Moreover, in order to make it easier to follow the action in systems with a large number of variables, only the values of variables that have changed in the last step are printed in the states of the trace.

Chapter 5

LTL Model Checking

5.1 Linear Temporal Logic

NuSMV allows for specifications expressed in LTL. Intuitively, while CTL specifications express properties over the computation tree of the FSM (branching-time approach), LTL characterizes each linear path induced by the FSM (linear-time approach). The two logics have in general different expressive power, but also share a significant intersection that includes most of the common properties used in practice. Typical LTL operators are:

- `F p` (read "in the future `p`"), stating that a certain condition `p` holds in one of the future time instants;
- `G p` (read "globally `p`"), stating that a certain condition `p` holds in all future time instants;
- `p U q` (read "`p` until `q`"), stating that condition `p` holds until a state is reached where condition `q` holds;
- `X p` (read "next `p`"), stating that condition `p` is true in the next state.

We remark that, differently from CTL, LTL temporal operators do not have path quantifiers. In fact, LTL formulas are evaluated on linear paths, and a formula is considered true in a given state if it is true for all the paths starting in that state.

5.2 Semaphore Example

Consider the case of the semaphore program and of the safety and liveness properties already described in Chapter 4 [CTL Model Checking], page 13. These properties correspond to LTL formulas

```
G ! (proc1.state = critical & proc2.state = critical)
```

expressing that the two processes cannot be in the critical region at the same time, and

```
G (proc1.state = entering -> F proc1.state = critical)
```

expressing that whenever a process wants to enter its critical session, it eventually does.

If we add the two corresponding LTL specification to the program, as follows:[1]

[1] In NuSMV a LTL specification are introduced by the keyword "`LTLSPEC`".

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
 LTLSPEC G ! (proc1.state = critical & proc2.state = critical)
 LTLSPEC G (proc1.state = entering -> F proc1.state = critical)
MODULE user(semaphore)
 VAR
   state : {idle, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                  : {idle, entering};
       state = entering & !semaphore : critical;
       state = critical              : {critical, exiting};
       state = exiting               : idle;
       1                             : state;
     esac;
   next(semaphore) :=
     case
       state = entering : 1;
       state = exiting  : 0;
       1                : semaphore;
     esac;
 FAIRNESS
   running
```

NuSMV produces the following output:

```
-- specification  G (!(proc1.state = critical & proc2.state = critical))
-- is true
-- specification  G (proc1.state = entering ->  F proc1.state = critical)
-- is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
    semaphore = 0
    proc1.state = idle
    proc2.state = idle
-> Input: 1.2 <-
    _process_selector_ = proc2
-- Loop starts here
-> State 1.2 <-
[...]
```

That is, the first specification is true, while the second is false and a counter-example path is generated.

5.3 Past Temporal Operators

In NUSMV, LTL properties can also include *past* temporal operators. Differently from standard temporal operators, that allow to express properties over the future evolution of the FSM, past temporal operators allow to characterize properties of the path that lead to the current situation. The typical past operators are:

- `O p` (read "once `p` "), stating that a certain condition `p` holds in one of the past time instants;
- `H p` (read "historically `p` "), stating that a certain condition `p` holds in all previous time instants;
- `p S q` (read "`p` since `q` "), stating that condition `p` holds since a previous state where condition `q` holds;
- `Y p` (read "yesterday `p` "), stating that condition `p` holds in the previous time instant.

Past temporal operators can be combined with future temporal operators, and allow for the compact characterization of complex properties.

A detailed description of the syntax of LTL formulas can be found in the NuSMV 2.2 User Manual.

Chapter 6

Bounded Model Checking

In this section we give a short introduction to the use of Bounded Model Checking (BMC) in NUSMV. For a more in-depth introduction to the theory underlying BMC please refer to [BCCZ99].

Consider the following model, representing a simple, deterministic counter modulo 8 (we assume that the following specification is contained in file `bmc_tutorial.smv`):

```
MODULE main
VAR
  y : 0..15;
ASSIGN
  init(y) := 0;
TRANS
  case
     y = 7 :  next(y) = 0;
     1     :  next(y) = ((y + 1) mod 16);
  esac
```

This slightly artificial model has only the state variable `y`, ranging from 0 to 15. The values of `y` are limited by the transition relation to the [0, 7] interval. The counter starts from `0`, deterministically increments by one the value of `y` at each transition up to 7, and then restarts from zero.

6.1 Checking LTL Specifications

We would like to check with BMC the LTL specification `G ( y=4 -> X y=6 )` expressing that "each time the counter value is 4, the next counter value will be 6". This specification is obviously false, and our first step is to use NUSMV BMC to demonstrate its falsity. To this purpose, we add the following specification to file `bmc_tutorial.smv`:

```
LTLSPEC G ( y=4 -> X y=6 )
```

and we instruct NUSMV to run in BMC by using command-line option `-bmc`:

```
system_prompt> NuSMV -bmc bmc_tutorial.smv
-- no counterexample found with bound 0 for specification
   G(y = 4 -> X y = 6)
-- no counterexample found with bound 1 for ...
-- no counterexample found with bound 2 for ...
-- no counterexample found with bound 3 for ...
-- no counterexample found with bound 4 for ...
```

```
-- specification  G (y = 4 ->  X y = 6)   is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
system_prompt>
```

NuSMV has found that the specification is false, and is showing us a counterexample, i.e. a trace where the value of y becomes 4 (at time 4) and at the next step is not 6.

```
 bound:    0    1    2    3    4    5
           o--->o--->o--->o--->o--->o
 state:  y=0  y=1  y=2  y=3  y=4  y=5
```

The output produced by NuSMV shows that, before the counterexample of length 5 is found, NuSMV also tried to finds counterexamples of lengths 0 to 4. However, there are no such counterexamples. For instance, in the case of bound 4, the traces of the model have the following form:

```
 bound:    0    1    2    3    4
           o--->o--->o--->o--->o
 state:  y=0  y=1  y=2  y=3  y=4
```

In this situation, y gets the value 4, but it is impossible for NuSMV to say anything about the following state.

In general, in BMC mode NuSMV tries to find a counterexample of increasing length, and immediately stops when it succeeds, declaring that the formula is false. The maximum number of iterations can be controlled by using command-line option -bmc_length. The default value is 10. If the maximum number of iterations is reached and no counter-example is found, then NuSMV exits, and the truth of the formula is not decided. We remark that in this case we cannot conclude that the formula is true, but only that any counter-example should be longer than the maximum length.

```
system_prompt> NuSMV -bmc -bmc_length 4 bmc_tutorial.smv
-- no counterexample found with bound 0 for ...
-- no counterexample found with bound 1 for ...
-- no counterexample found with bound 2 for ...
-- no counterexample found with bound 3 for ...
-- no counterexample found with bound 4 for ...
system_prompt>
```

Let us consider now another property, !G F (y = 2), stating that y gets the value 2 only a finite number of times. Again, this is a false property due to the cyclic nature of the model. Let us modify the specification of file bmc_tutorial.smv as follows:

```
LTLSPEC !G F (y = 2)
```

and let us run NuSMV in BMC mode:

```
system_prompt> NuSMV -bmc bmc_tutorial.smv
-- no counterexample found with bound 0 for specification ! G  F y = 2
-- no counterexample found with bound 1 for ...
-- no counterexample found with bound 2 for ...
-- no counterexample found with bound 3 for ...
-- no counterexample found with bound 4 for ...
-- no counterexample found with bound 5 for ...
-- no counterexample found with bound 6 for ...
-- no counterexample found with bound 7 for ...
-- specification ! G  F y = 2   is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
-> State: 1.7 <-
  y = 6
-> State: 1.8 <-
  y = 7
-> State: 1.9 <-
  y = 0
system_prompt>
```

In this example NuSMV has increased the problem bound until a cyclic behavior of length 8 is found that contains a state where y value is 2. Since the behavior is cyclic, state 1.3 is entered infinitely often and the property is false.

```
                         =
         +---------------------------------------+
         |                                       |
         |                                       |
         |                                       |
         o--->o--->o--->o--->o--->o--->o--->o--->o
bound:   0    1    2    3    4    5    6    7    8
y value: 0    1    2    3    4    5    6    7    0
```

6.2 Finding Counterexamples

In general, BMC can find two kinds of counterexamples, depending on the property being analyzed. For safety properties (e.g. like the first one used in this tutorial), a counterexample is a finite sequence of transitions through different states. For liveness properties, counterexamples are infinite but periodic sequences, and can be represented in a bounded setting as a finite prefix followed by a loop, i.e. a finite sequence of states ending with a loop back to some previous state. So a counterexample which demonstrates the falsity of a liveness property as "`! G F p`" cannot be a finite sequence of transitions. It must contain a loop which makes the infinite sequence of transitions as well as we expected.

```
                                  =
                        +---------------------+
                        |                     |
                        |                     |
                        |                     |
       o--->o-...->o--->o--->o-...->o--->o--->o--->o--->
time:  0    1      l-1  l    l+1    k-2  k-1  k    k+1    ...
```

Consider the above figure. It represents an examples of a generic infinite counterexample, with its two parts: the prefix part (times from 0 to $l-1$), and the loop part (indefinitely from l to $k-1$). Because the loop always jumps to a previous time it is called *loopback*. The loopback condition requires that state k is identical to state l. As a consequence, state $k+1$ is forced to be equal to state $l+1$, state $k+2$ to be equal to state $l+2$, and so on.

A fine-grained control of the length and of the loopback condition for the counter-example can be specified by using command `check_ltlspec_bmc_onepb` in interactive mode. This command accepts options `-k`, that specifies the length of the counter-example we are looking for, and `-l`, that defines the loopback condition. Consider the following interactive session:

```
system_prompt> NuSMV -int bmc_tutorial.smv
NuSMV> go_bmc
NuSMV> check_ltlspec_bmc_onepb -k 9 -l 0
-- no counterexample found with bound 9 and loop at 0 for specification
   ! G  F y = 2
NuSMV> check_ltlspec_bmc_onepb -k 8 -l 1
-- no counterexample found with bound 8 and loop at 1 for specification
   ! G  F y = 2
NuSMV> check_ltlspec_bmc_onepb -k 9 -l 1
-- specification ! G  F y = 2   is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 0
-- Loop starts here
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
-> State: 1.7 <-
  y = 6
-> State: 1.8 <-
  y = 7
```

```
-> State: 1.9 <-
  y = 0
-> State: 1.10 <-
  y = 1
NuSMV> quit
system_prompt>
```

NUSMV did not find a counterexample for cases ($k = 9, l = 0$) and ($k = 8, l = 1$). The following figures show that these case look for counterexamples that do not match with the model of the counter, so it is not possible for NUSMV to satisfy them.

```
k = 9, l = 0:
                                 =
         +---------------------------------------------+
         |                                             |
         |                                             |
         |                                             |
         o--->o--->o--->o--->o--->o--->o--->o--->o--->o
bound:   0    1    2    3    4    5    6    7    8    9
y value: 0    1    2    3    4    5    6    7    0    1


k = 8, l = 1:
                              =
              +----------------------------------+
              |                                  |
              |                                  |
              |                                  |
         o--->o--->o--->o--->o--->o--->o--->o--->o
bound:   0    1    2    3    4    5    6    7    8
y value: 0    1    2    3    4    5    6    7    0
```

Case ($k = 9, l = 1$), instead allows for a counter-example:

```
k = 9, l = 1:
                                 =
              +----------------------------------------+
              |                                        |
              |                                        |
              |                                        |
         o--->o--->o--->o--->o--->o--->o--->o--->o--->o
bound:   0    1    2    3    4    5    6    7    8    9
y value: 0    1    2    3    4    5    6    7    0    1
```

In NUSMV it is possible to specify the loopback condition in four different ways:

- **The loop as a precise time-point.** Use a natural number as the argument of option `-l`.
- **The loop length.** Use a negative number as the argument of option `-l`. The negative number is the loop length, and you can also imagine it as a precise time-point loop relative to the path bound.
- **No loopback.** Use symbol 'X' as the argument of option `-l`. In this case NUSMV will not find infinite counterexamples.

- **All possible loops.** Use symbol '*' as the argument of option `-l`. In this case NuSMV will search counterexamples for paths with any possible loopback structure. A counterexample with no loop will be also searched. This is the default value for option `-l`.

In the following example we look for a counter-example of length 12 with a loop of length 8:

```
system_prompt> NuSMV -int bmc_tutorial.smv
NuSMV> go_bmc
NuSMV> check_ltlspec_bmc_onepb -k 12 -l -8
-- specification ! G  F y = 2   is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-- Loop starts here
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
-> State: 1.7 <-
  y = 6
-> State: 1.8 <-
  y = 7
-> State: 1.9 <-
  y = 0
-> State: 1.10 <-
  y = 1
-> State: 1.11 <-
  y = 2
-> State: 1.12 <-
  y = 3
-> State: 1.13 <-
  y = 4
NuSMV>
```

This picture illustrates the produced counterexample in a more effective way:

```
                                    =
                         +--------------------------------+
                         |                                |
                         |                                |
                         |                                |
         o-->o-->o-->o-->o-->o-->o-->o-->o-->o-->o-->o-->o
bound:   0   1   2   3   4   5   6   7   8   9   10  11  12
y value: 0   1   2   3   4   5   6   7   0   1   2   3   4
```

If no loopback is specified, NUSMV is not able to find a counterexample for the given liveness property:

```
system_prompt> NuSMV -int bmc_tutorial.smv
NuSMV> go_bmc
NuSMV> check_ltlspec_bmc_onepb -k 12 -l X
-- no counterexample found with bound 12 and no loop for ...
NuSMV>
```

6.3 Checking Invariants

Ahead of version 2.2.2, NUSMV supported only the 2-step inductive reasoning algorithm for invariant checking. As will become clear from this tutorial, this algorithm is not complete, so in certain cases it cannot be used to state whether an invariant specification is actually true or false.

Since version 2.2.2, NUSMV supports total inductive reasoning, which might be heavier than the 2-step approach but can make invariant specifications provable even when the latter fails.

Please refer to [ES04] for a more in-depth explanation of the theory underlying the algorithms for total temporal induction.

6.3.1 2-Step Inductive Reasoning

Bounded Model Checking in NUSMV can be used not only for checking LTL specification, but also for checking invariants. An invariant is a propositional property which must always hold. BMC tries to prove the truth of invariants via a process of inductive reasoning, by checking if (i) the property holds in every initial state, and (ii) if it holds in any state reachable from any state where it holds.

Let us modify file `bmc_tutorial.smv` by replacing the LTL specification with the following invariant specification:

```
INVARSPEC y in (0..12)
```

and let us run NUSMV in BMC mode:

```
system_prompt> NuSMV -bmc bmc_tutorial.smv
-- cannot prove the invariant y in (0 .. 12) : the induction fails
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 12
-> State: 1.2 <-
  y = 13
system_prompt>
```

NUSMV reports that the given invariant cannot be proved, and it shows a state satisfying "`y` in (0..12)" that has a successor state not satisfying "`y` in (0..12)". This two-steps sequence of assignments shows why the induction fails. Note that NUSMV does not state the given formula is really false, but only that it cannot be proven to be true using the 2-step inductive reasoning described previously.

If we try to prove the stronger invariant `y in (0..7)` we obtain:

```
system_prompt> NuSMV -bmc bmc_tutorial.smv
-- invariant y in (0 .. 7)   is true
system_prompt>
```

In this case NuSMV is able to prove that `y in (0..7)` is true. As a consequence, also the weaker invariant `y in (0..12)` is true, even if NuSMV is not able to prove it in BMC mode. On the other hand, the returned counter-example can be used to *strengthen* the invariant, until NuSMV is able to prove it.

Now we check the false invariant `y in (0..6)`:

```
-- cannot prove the invariant y in (0 .. 6) : the induction fails
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 6
-> State: 1.2 <-
  y = 7
NuSMV>
```

As for property `y in (0..12)`, NuSMV returns a two steps sequence showing that the induction fails. The difference is that, in the former case state 'y=12' is NOT reachable, while in the latter case the state 'y=6' can be reached.

6.3.2 Complete Invariant Checking

Since version 2.2.2, complete invariant checking can be obtained by running the command `check_invar_bmc` in interactive mode, and specifying the algorithm ``` ``een-sorensson'' ``` using the option `-a`. If an incremental sat solver is available, the command `check_invar_bmc_inc` may also be used.

The classic 2-step algorithm was not able to prove directly the truth of the invariant `y in (0..12)`. This invariant can now be easily checked by the complete invariant checking algorithm.

```
system_prompt> NuSMV -int bmc_tutorial.smv
NuSMV> go_bmc
NuSMV> check_invar_bmc -a een-sorensson -p "y in (0..12)"
-- no proof or counterexample found with bound 0 ...
-- no proof or counterexample found with bound 1 ...
-- no proof or counterexample found with bound 2 ...
-- no proof or counterexample found with bound 3 ...
-- no proof or counterexample found with bound 4 ...
-- no proof or counterexample found with bound 5 ...
-- invariant y in (0 .. 12)   is true
NuSMV>
```

As can be inferred from this example, NuSMV proved that the invariant actually holds, requiring a length of 6 to prove it.

Complete invariant checking can also prove that an invariant does not hold, and provide a convincing counter-example for it. For example property `y in (0..6)` that the ``` ``classic'' ``` algorithm failed to check is now proved to be false:

```
NuSMV> check_invar_bmc -a een-sorensson -p "y in (0..6)"
-- no proof or counterexample found with bound 0 ...
-- no proof or counterexample found with bound 1 ...
-- no proof or counterexample found with bound 2 ...
-- no proof or counterexample found with bound 3 ...
-- no proof or counterexample found with bound 4 ...
-- no proof or counterexample found with bound 5 ...
```

```
-- no proof or counterexample found with bound 6 ...
-- invariant y in (0 .. 6)   is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
-> State: 1.7 <-
  y = 6
-> State: 1.8 <-
  y = 7
NuSMV>
```

The provided counter-example shows that `y` actually can reach a value out of the set `(0..6)`.

Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, In TACAS'99*, March 1999.

[ES04] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. In Ofer Strichman and Armin Biere, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2004.

Command Index

Index