NuSMV 2.0 User Manual

**Roberto Cavada, Alessandro Cimatti,
Emanuele Olivetti, Marco Pistore,
and Marco Roveri**

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

Email: `nusmv@irst.itc.it`

This document is part of the distribution package of the NuSMV model checker, available at http://nusmv.irst.itc.it.

Parts of this documents have been taken from "The SMV System - Draft", by K. McMillan, available at http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps.

1 Introduction

REMARK. This manual is outdated and does not describe all the new features of NuSMV2. An updated version of the manual will be made available as soon as possible. Chapter 6 [Getting started with BMC], page 48 contains a quick introduction to the SAT based Bounded Model Checking provided by NuSMV2.

`NuSMV` is a symbolic model checker jointly developed by Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST). The `NuSMV` project aims at the development of an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas. `NuSMV` is available at `http://nusmv.irst.itc.it`.

The main features of `NuSMV` are the following:

- **Functionalities.** `NuSMV` allows for the representation of synchronous and asynchronous finite state systems, and the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual, as well as graphical, interface.
- **Architecture.** A software architecture has been defined. The different components and functionalities of `NuSMV` have been isolated and separated in modules. Interfaces between modules have been provided. This should allow to reduce the effort needed to modify and extend `NuSMV`.
- **Quality of the implementation.** `NuSMV` is written in ANSI C, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. `NuSMV` uses the state of the art BDD package developed at Colorado University. This makes it very robust, portable, efficient. Furthermore, its code should be easy to understand and modify by other people than the developers.

This document is structured as follows.

- In Chapter 2 [Examples], page 2 we present some examples of usage of `NuSMV`.
- In Chapter 3 [Syntax], page 8 we define the syntax of the input language of `NuSMV`.
- In Chapter 4 [Running NuSMV interactively], page 20 the commands of the interaction shell are described.
- In Chapter 5 [Running NuSMV batch], page 46 we define the batch mode of `NuSMV`.
- In Chapter 6 [Getting started with BMC], page 48 we describe the SAT based Bounded Model Checking provided by `NuSMV`.

2 Examples

The input language of NuSMV is designed to allow the description of finite state machines (FSM) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily 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.

Specifications can be expressed in CTL (Computation Tree Logic), or LTL (Linear Temporal Logic). These logics allow a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in concise a syntax.

The primary purpose of the NuSMV input is to describe the transition relation of a Kripke structure. Any expression in the propositional calculus can be used to describe this 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.

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;

SPEC
  AG(request -> AF state = busy)
```

The first part defines the Kripke structure. The space of states of the Kripke structure 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 Kripke structure 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 column) if its current value is `ready` and `request` is `1` (i.e. true). Otherwise (the `1` before the column) 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.

The keyword '`SPEC`' is followed by a CTL formula, that is intended to be checked for truth in the Kripke structure defined above. The intuitive reading of the formula is that every time `request` is true, then in all possible future evolution, eventually `state` must become `busy`.

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);
 SPEC
   AG AF bit2.carry_out
```

The Kripke structure 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 into the BDD representation of the system. However, they cannot be given values 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.

The previous examples describe synchronous systems, where the assignments statements are taken into account 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).

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);

SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)
```

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. For each gate, the specification of this program states that the output of the gate oscillates (i.e. that its value is infinitely often `0`, and infinitely often `1`). In fact, this specification is false, since the system is not forced to eventually choose a given process to execute, hence 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 CTL 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, thus making the specification true.

The alternative to using processes to model an asynchronous circuit would be to have all gates execute simultaneously, but allow each gate the non-deterministic choice of evaluating its output, or keeping the same output value. Such a model of the inverter ring would look like the following:

```
MODULE main
 VAR
   gate1 : inverter(gate3.output);
   gate2 : inverter(gate2.output);
   gate3 : inverter(gate1.output);
 SPEC
   (AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
 VAR
   output : boolean;
 ASSIGN
  init(output) := 0;
  next(output) := !input union 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.

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;
 SPEC
   AG !(proc1.state = critical &  proc2.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
```

If any of the specification is false, the `NuSMV` model checker attempts to produce a counterexample, proving that the specification is false. This is not always possible, since formulas preceded by 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. In addition, some formulas require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state.

In the case of the semaphore program, suppose that the specification were changed to:

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

In other words, we specify that if `proc1` wants to enter its critical region, it eventually does. The output of the model checker in this case is shown in the figure below[1].

```
-- 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

state 1.2:
[executing process proc1]

-- loop starts here --
state 1.3:
proc1.state = entering

state 1.4:
[executing process proc2]

state 1.5:
[executing process proc2]
proc2.state = entering

state 1.6:
[executing process proc1]
semaphore = 1
proc2.state = critical

state 1.7:
[executing process proc2]

state 1.8:
[executing process proc2]
proc2.state = exiting

state 1.9:
semaphore = 0
proc2.state = idle
```

The counterexample above shows a path with `proc1` going to the `entering` state, followed by a loop in which `proc2` repeatedly enters its critical region and then returns to its `idle` state, with `proc1` only executing while `proc2` is in its critical region. This path shows that the specification is false, since `proc1` never enters its critical region. Note that in the printout of an execution sequence, only the values of variables that change are printed, to make it easier to follow the action in systems with a large number of variables.

`NuSMV` allows to specify the Kripke structure 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

[1] The output has been slightly edited in order to make it more readable.

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);
 SPEC
   (AG AF gate1.output) & (AG AF !gate1.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 Kripke structures, where the set of initial states is empty or the transition relation is not total. This may result in logical absurdities.

3 Syntax

We present now the complete syntax of the input language of NuSMV. In the following, an atom may be any sequence of characters starting with a character in the set {A-Za-z_} and followed by a possibly empty sequence of characters belonging to the set {A-Za-z0-9_\$#-}. A number is any sequence of digits. A digit belongs to the set {0-9}.

All characters and case in a name are significant. Whitespace characters are space (⟨SPACE⟩), tab (⟨TAB⟩) and newline (⟨RET⟩). Any string starting with two dashes ('--') and ending with a newline is a comment. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below. Grammar productions enclosed in square brackets ('[]') are optional.

3.1 Expressions

Expressions are constructed from variables, constants, and a collection of operators, including boolean connectives, integer arithmetic operators, case expressions and set expressions.

3.1.1 Simple Expressions

Simple expressions are expressions built only from current state variables. Simple expressions can be used to specify sets of states, e.g. the initial set of states. The syntax of simple expressions is as follows:

```
simple_expr ::
          atom                          ;; a symbolic constant
        | number                        ;; a numeric constant
        | "TRUE"                        ;; The boolean constant "TRUE"
        | "FALSE"                       ;; The boolean constant "FALSE"
        | var_id                        ;; a variable identifier
        | "(" simple_expr ")"
        | "!" simple_expr               ;; logical not
        | simple_expr "&" simple_expr   ;; logical and
        | simple_expr "|" simple_expr   ;; logical or
        | simple_expr "->" simple_expr  ;; logical implication
        | simple_expr "<->" simple_expr ;; logical equivalence
        | simple_expr "=" simple_expr   ;; equality
        | simple_expr "<" simple_expr   ;; less than
        | simple_expr ">" simple_expr   ;; greater than
        | simple_expr "<=" simple_expr  ;; less than or equal
        | simple_expr ">=" simple_expr  ;; greater than or equal
        | simple_expr "+" simple_expr   ;; integer addition
        | simple_expr "-" simple_expr   ;; integer subtraction
        | simple_expr "*" simple_expr   ;; integer multiplication
        | simple_expr "/" simple_expr   ;; integer division
        | simple_expr "mod" simple_expr ;; integer remainder
        | set_simple_expr               ;; a set simple_expression
        | case_simple_expr              ;; a case expression
```

A *var_id*, (see Section 3.2.10 [Identifiers], page 15) or identifier, is a symbol or expression which identifies an object, such as a variable or a defined symbol. Since a var_id can be an atom, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the interpreter as an error.

The order of parsing precedence for operators from high to low is:

```
*,/
+,-
mod
=,<,>,<=,>=
!
&
|
->,<->
```

Operators of equal precedence associate to the left. Parentheses may be used to group expressions.

3.1.1.1 Case Expressions

A case expression has the following syntax:

```
case_simple_expr ::
          "case"
             simple_expr ":" simple_expr ";"
             simple_expr ":" simple_expr ";"
             ...
             simple_expr ":" simple_expr ";"
          "esac"
```

A `case_simple_expr` returns the value of the first expression on the right hand side of ‘`:`’, such that the corresponding condition on the left hand side evaluates to `1`. Thus, if `simple_expr` on the left side is true, then the result is the corresponding `simple_expr` on the right side. If none of the expressions on the left hand side evaluates to `1`, the result of the `case_expression` is the numeric value `1`. It is an error for any expression on the left hand side to return a value other than the truth values `0` or `1`.

3.1.1.2 Set Expressions

A set expression has the following syntax:

```
set_elem :: atom     ;; a symbolic constant
          | number   ;; a numeric constant

set_expr ::
         "{" set_elem "," ... "," set_elem "}"  ;; set definition
         |  simple_expr "in" simple_expr        ;; set inclusion predicate
         |  simple_expr "union" simple_expr     ;; set union
```

A set can be defined by enumerating its elements inside curly braces ‘`{...}`’. The elements of the set can be numbers or symbolic constants. The inclusion operator ‘`in`’ tests a value for membership in a set. The union operator ‘`union`’ takes the union of two sets. If either argument is a number or a symbolic value instead of a set, it is coerced to a singleton set.

3.1.2 Next Expressions

While simple expressions can represent sets of states, next expressions relate current and next state variables to express transitions in the Kripke structure. The structure of next expressions is similar to the structure of simple expressions (See Section 3.1.1 [Simple Expressions], page 8). The difference is that next expression allow to refer to next state variables. The grammar is depicted below.

```
next_expr ::
          atom                          ;; a symbolic constant
        | number                        ;; a numeric constant
        | "TRUE"                        ;; The boolean constant "TRUE"
        | "FALSE"                       ;; The boolean constant "FALSE"
        | var_id                        ;; a variable identifier
        | "(" next_expr ")"
        | "next" "(" simple_expr ")"    ;; next value of an "expression"
        | "!" next_expr                 ;; logical not
        | next_expr "&" next_expr       ;; logical and
        | next_expr "|" next_expr       ;; logical or
        | next_expr "->" next_expr      ;; logical implication
        | next_expr "<->" next_expr     ;; logical equivalence
        | next_expr "=" next_expr       ;; equality
        | next_expr "<" next_expr       ;; less than
        | next_expr ">" next_expr       ;; greater than
        | next_expr "<=" next_expr      ;; less than or equal
        | next_expr ">=" next_expr      ;; greater than or equal
        | next_expr "+" next_expr       ;; integer addition
        | next_expr "-" next_expr       ;; integer subtraction
        | next_expr "*" next_expr       ;; integer multiplication
        | next_expr "/" next_expr       ;; integer division
        | next_expr "mod" next_expr     ;; integer remainder
        | set_next_expr                 ;; a set next_expression
        | case_next_expr                ;; a case expression
```

`set_next_expr` and `case_next_expr` are the same as `set_simple_expr` (see Section 3.1.1.2 [Set Expressions], page 9) and `case_simple_expr` (see Section 3.1.1.1 [Case Expressions], page 9) respectively, with the replacement of `"simple"` with `"next"`. The only additional production is `"next" "(" simple_expr ")"`, which allows to "shift" all the variables in `simple_expr` to the *next* state. The `next` operator distributes on every operator. For instance, the formula `next((A & B) | C)` is a shorthand for the formula `(next(A) & next(B)) | next(C)`. It is an error if in the scope of the `next` operator occurs another `next` operator.

3.2 Definition of the Kripke structure

3.2.1 State Variables

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation:

```
var_declaration :: "VAR"
             atom ":" type ";"
             atom ":" type ";"
             ...
```

The type associated with a variable declaration can be either a boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays).

3.2.1.1 Type Specifiers

A type specifier has the syntax:

```
type :: boolean
     |  "{" val "," val "," ... val "}"
     |  number ".." number
     |  "array" number ".." number "of" type
     |  atom [ "(" simple_expr "," simple_expr "," ...  ")" ]
     |  "process" atom [ "(" simple_expr "," ... "," simple_expr ")" ]

val  :: atom
     |  number
```

A variable of type `boolean` can take on the numerical values `0` and `1` (representing false and true, respectively). In the case of a list of values enclosed in quotes (where atoms are taken to be symbolic constants), the variable is a scalar which take any of these values. In the case of an `array` declaration, the first `simple_expr` is the lower bound on the subscript and the second `simple_expr` is the upper bound. Both of these expressions must evaluate to integer constants. Finally, an atom optionally followed by a list of expressions in parentheses indicates an instance of module atom (See Section 3.2.9 [MODULE declarations], page 13). The keyword causes the module to be instantiated as an asynchronous process (See Section 3.2.12 [Processes], page 16).

3.2.2 Input Variables

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation:

```
ivar_declaration :: "IVAR"
             atom ":" type ";"
             atom ":" type ";"
             ...
```

The type associated with a variable declaration can be either a boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays) (See Section 3.2.1 [State Variables], page 11).

3.2.3 `ASSIGN` declarations

An assignment has the form:

```
assign_declaration :: "ASSIGN"
          assign_body ";"
          assign_body ";"
          ...

assign_body ::
          atom                ":=" simple_expr     ;; normal assignment
        | "init" "(" atom ")" ":=" simple_expr     ;; init assignment
        | "next" "(" atom ")" ":=" next_expr       ;; next assignment
```

On the left hand side of the assignment, atom denotes the current value of a variable, '`init(atom)`' denotes its initial value, and '`next(atom)`' denotes its value in the next state. If the expression on the right hand side evaluates to an integer or symbolic constant, the assignment simply means that the left hand side is equal to the right hand side. On the other hand, if the expression evaluates to a set, then the assignment means that the left hand side is contained in that set. It is an error if the value of the expression is not contained in the range of the variable on the left hand side.

In order for a program to be implementable, there must be some order in which the assignments can be executed such that no variable is assigned after its value is referenced. This is not the case if there is a circular dependency among the assignments in any given process. Hence, such a condition is an error. In it is an error for a variable to be assigned a value more than once at any given time. More precisely, it is an error if:

1. the next or current value of a variable is assigned more than once in a given process, or
2. the initial value of a variable is assigned more than once in the program, or
3. the current value and the initial value of a variable are both assigned in the program, or
4. the current value and the next value of a variable are both assigned in the program.

3.2.4 `TRANS` declarations

The transition relation R of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a boolean valued expression T, introduced by the '`TRANS`' keyword. The syntax of a `TRANS` declaration is:

```
trans_declaration :: "TRANS" trans_expr

trans_expr  :: next_expr
```

It is an error for the expression to yield any value other than `0` or `1`. If there is more than one `TRANS` declaration, the transition relation is the conjunction of all of `TRANS` declarations.

3.2.5 `INIT` declarations

The set of initial states of the model is determined by a boolean expression under the '`INIT`' keyword. The syntax of a `INIT` declaration is:

```
init_declaration :: "INIT" init_expr

init_expr   :: simple_expr
```

It is an error for the expression to contain the `next()` operator, or to yield any value other than `0` or `1`. If there is more than one `INIT` declaration, the initial set is the conjunction of all of the `INIT` declarations.

3.2.6 INVAR declarations

The set of invariant states (i.e. the analogous of normal assignments, as described in Section 3.2.3 [ASSIGN declarations], page 11) can be specified using a boolean expression under the 'INVAR' keyword. The syntax of a INVAR declaration is:

```
invar_declaration        :: simple_expr
```

It is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one INVAR declaration, the invariant set is the conjunction of all of the INVAR declarations.

3.2.7 DEFINE declarations

In order to make descriptions more concise, a symbol can be associated with a commonly expression. The syntax for this kind of declaration is:

```
define_declaration :: "DEFINE"
                      atom ":=" simple_expr ";"
                      atom ":=" simple_expr ";"
                      ...
                      atom ":=" simple_expr ";"
```

Whenever an identifier referring to the symbol on the left hand side of the ':=' in a DEFINE occurs in an expression, it is replaced by the expression on the right hand side. The expression on the right hand side is always evaluated in its context, however (see Section 3.2.9 [MODULE declarations], page 13 for an explanation of contexts). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

Defined symbols cannot be given values non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not.

3.2.8 ISA declarations

There are cases in which some parts of a module could be shared among different modules, or could be used as a module themselves. In NuSMV it is possible to declare the common parts as separate modules, and then use the ISA declaration to import the common parts inside a module declaration.

The syntax of an ISA declaration is as follows:

```
isa_declaration :: "ISA" atom
```

where `atom` must be the name of a declared module. The ISA declaration can be thought as a simple macro expansion command, because the body of the module referenced by an ISA command is replaced to the ISA declaration.

3.2.9 MODULE declarations

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be so that each instance of a module can refer to different data values. A module can contain instances of other modules, allowing a structural hierarchy to be built.

The syntax of a module declaration is as follows.

```
module ::
        "MODULE" atom [ "(" atom "," atom "," ... atom ")" ]
        [ var_declaration       ]
        [ ivar_declaration      ]
        [ assign_declaration    ]
```

```
[ trans_declaration      ]
[ init_declaration       ]
[ invar_declaration      ]
[ spec_declaration       ]
[ checkinvar_declaration ]
[ ltlspec_declaration    ]
[ compute_declaration    ]
[ fairness_declaration   ]
[ define_declaration     ]
[ isa_declaration        ]
```

The *atom* immediately following the keyword "MODULE" is the name associated with the module. Module names are drawn from a separate name space from other names in the program, and hence may clash with names of variables and definitions. The optional list of atoms in parentheses are the formal parameters of the module. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated (see below).

An *instance* of a module is created using the VAR declaration (see Section 3.2.1 [State Variables], page 11). This declaration supplies a name for the instance, and also a list of actual parameters, which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error if the number of actual parameters is different from the number of formal parameters. The semantic of module instantiation is similar to call-by-reference. For example, in the following program fragment:

```
MODULE main
...
 VAR
  a : boolean;
  b : foo(a);
...
MODULE foo(x)
 ASSIGN
   x := 1;
```

the variable a is assigned the value 1. This distinguishes the call-by-reference mechanism from a call-by-value scheme.

Now consider the following program:

```
MODULE main
...
 DEFINE
   a := 0;
 VAR
   b : bar(a);
...
MODULE bar(x)
 DEFINE
   a := 1;
   y := x;
```

In this program, the value of y is 0. On the other hand, using a call-by-name mechanism, the value of y would be 1, since a would be substituted as an expression for x.

Forward references to module names are allowed, but circular references are not, and result in an error.

3.2.10 Identifiers

An `id`, or identifier, is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows.

```
id ::
        atom
        | "self"
        | id "." atom
        | id "[" simple_expr "]"
```

An *atom* identifies the object of that name as defined in a `VAR` or `DEFINE` declaration. If `a` identifies an instance of a module, then the expression '`a.b`' identifies the component object named '`b`' of instance '`a`'. This is precisely analogous to accessing a component of a structured data type. Note that an actual parameter of module '`a`' can identify another module instance '`b`', allowing '`a`' to access components of '`b`', as in the following example:

```
MODULE main
...  VAR
  a : foo(b);
  b : bar(a);
...

MODULE foo(x)
 DEFINE
   c := x.p | x.q;

MODULE bar(x)
 VAR
   p : boolean;
   q : boolean;
```

Here, the value of '`c`' is the logical or of '`p`' and '`q`'.

If '`a`' identifies an array, the expression '`a[b]`' identifies element '`b`' of array '`a`'. It is an error for the expression '`b`' to evaluate to a number outside the subscript bounds of array '`a`', or to a symbolic value.

It is possible to refer the name the current module has been instantiated to by using the `self` builtin identifier.

```
MODULE element(above, below, token)
 VAR
   Token : boolean;

 ASSIGN
   init(Token) := token;
   next(Token) := token-in;

 DEFINE
   above.token-in := Token;
   grant-out := below.grant-out;

MODULE cell
 VAR
   e2 : element(self,   e1, 0);
   e1 : element(e1   , self, 1);
```

```
    DEFINE
      e1.token-in := token-in;
      grant-out := grant-in & !e1.grant-out;

  MODULE main
    VAR c1 : cell;
```

In this example the name the `cell` module has been instantiated to is passed to the submodule `element`. In the `main` module, declaring `c1` to be an instance of module `cell` and defining `above.token-in` in module `e2`, really amounts to defining the symbol `c1.token-in`. When you, in the `cell` module, declare `e1` to be an instance of module `element`, and you define `grant-out` in module `e1` to be `below.grant-out`, you are really defining it to be the symbol `c1.grant-out`.

3.2.11 The `main` module

The syntax of a `NuSMV` program is:

```
program ::
        module_1
        module_2
        ...
        module_n
```

There must be one module with the name `main` and no formal parameters. The module `main` is the one evaluated by the interpreter.

3.2.12 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword '`process`' (see Section 3.2.1 [State Variables], page 11). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Each instance of a process has a special boolean variable associated with it called `running`. The value of this variable is `1` if and only if the process instance is currently selected for execution. A process may run only when its parent is running. In addition no two processes with the same parents may be running at the same time.

3.2.13 `FAIRNESS` declarations

A *fairness constraint* is a CTL formula which is assumed to be true infinitely often in all *fair execution paths*. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths. Fairness constraints are declared using the following syntax:

```
fairness_declaration :: "FAIRNESS" fair_expr

fair_expr   :: ctl_expr
```

A path is considered fair if and only if all constraints declared in this manner are true infinitely often.

3.3 Specifications

The specifications to be checked on the Kripke structure can be expressed in the temporal logics CTL and LTL. Furthermore, it is possible to analyze quantitative characteristics of the model by specifying computations. Specifications can be positioned within modules. The specification is processed in such a way to rename the variables according to the containing context.

Specifications are evaluated by `NuSMV`. In case of a CTL or LTL false specification, a counterexample is printed out.

3.3.1 CTL Specifications

A CTL specification is given as a formula in the temporal logic CTL, introduced by the keyword '`SPEC`'. The syntax of this declaration is:

```
spec_declaration :: "SPEC" spec_expr

spec_expr   :: ctl_expr
```

The syntax of CTL formulas recognized by the `NuSMV` parser is as follows:

```
ctl_expr ::
        simple_expr                 ;; a simple boolean expression
        | "(" ctl_expr ")"
        | "!" ctl_expr              ;; logical not
        | ctl_expr "&" ctl_expr     ;; logical and
        | ctl_expr "|" ctl_expr     ;; logical or
        | ctl_expr "->" ctl_expr    ;; logical implies
        | ctl_expr "<->" ctl_expr   ;; logical equivalence
        | "EG" ctl_expr             ;; exists globally
        | "EX" ctl_expr             ;; exists next state
        | "EF" ctl_expr             ;; exists finally
        | "AG" ctl_expr             ;; forall globally
        | "AX" ctl_expr             ;; forall next state
        | "AF" ctl_expr             ;; forall finally
        | "E" "[" ctl_expr "U" ctl_expr "]" ;; exists until
        | "A" "[" ctl_expr "U" ctl_expr "]" ;; forall until
```

It is an error for an expressions in a CTL formula to contain a '`next()`' operator or to have non-boolean components, i.e. subformulas which evaluate to a value other than `0` or `1`. Also, `simple_expr` in CTL formulas can not contain case statements.

It is also possible to specify invariants, i.e. propositional formulas which must hold invariantly in the model. The corresponding command is '`INVARSPEC`', with syntax:

```
checkinvar_declaration :: "INVARSPEC" simple_expr ";"
```

This statement corresponds to

```
SPEC  AG simple_expr ";"
```

but can be checked by a specialized algorithm during reachability analysis.

3.3.2 LTL Specifications

`NuSMV` allows for specifications expressed in LTL. Model checking of LTL specifications is based on the construction of a tableau corresponding to the LTL formula and on CTL model checking, along the lines described in *[CGH97]*. This construction is completely transparent to the user.

LTL specifications are introduced by the keyword '`LTLSPEC`'. The syntax of this declaration is:

```
ltlspec_declaration :: "LTLSPEC" ltl_expr
```

where

```
ltl_expr ::
        simple_expr                ;; a simple boolean expression
        | "(" ltl_expr ")"
        | "!" ltl_expr             ;; logical not
        | ltl_expr "&" ltl_expr    ;; logical and
        | ltl_expr "|" ltl_expr    ;; logical or
        | ltl_expr "->" ltl_expr   ;; logical implies
        | ltl_expr "<->" ltl_expr  ;; logical equivalence
        | "G" ltl_expr             ;; globally
        | "X" ltl_expr             ;; next state
        | "F" ltl_expr             ;; finally
        | ltl_expr "U" ltl_expr    ;; until
```

As for CTL formulas, `simple_expr` can not contain case statements. The counterexample generated to show the falsity of a LTL specification may contain state variables which have been introduced by the tableau construction procedure. Currently it is not possible to navigate the counterexamples for LTL formulas.

3.3.3 Real Time CTL Specifications and Computations

`NuSMV` allows for Real Time CTL specifications *[EMSS90]*. `NuSMV` assumes that each transition takes unit time for execution. RTCTL extends the syntax of CTL path expressions with the following bounded modalities:

```
rtctl_expr ::
        ctl_expr
      | "EBF" range rtctl_expr
      | "ABF" range rtctl_expr
      | "EBG" range rtctl_expr
      | "ABG" range rtctl_expr
      | "A" "[" rtctl_expr "BU" range rtctl_expr "]"
      | "E" "[" rtctl_expr "BU" range rtctl_expr "]"

range  :: number ".." number"
```

Intuitively, in the formula `E [` *a* `BU` *m*`..`*n* *b* `]` *m* (*n*, resp.) represents the minimum (maximum) number of permitted transition along a path of a structure before the eventuality holds.

Real time CTL specifications can be defined with the following syntax, which extends the syntax for CTL specifications.

```
spec_declaration :: "SPEC" rtctl_expr
```

With the '`COMPUTE`' statement, it is also possible to compute quantitative information on the Kripke structure. In particular, it is possible to compute the exact bound on the delay between two specified events, expressed as CTL formulas. The syntax is the following:

```
compute_declaration :: "COMPUTE" compute_expr
```

where

```
compute_expr ::
            "MIN" "[" rtctl_expr "," rtctl_expr "]"
          | "MAX" "[" rtctl_expr "," rtctl_expr "]"
```

`MIN [`*start* , *final*`]` computes the set of states reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach the state. If a fixed point is reached and no states intersect *final* then *infinity* is returned.

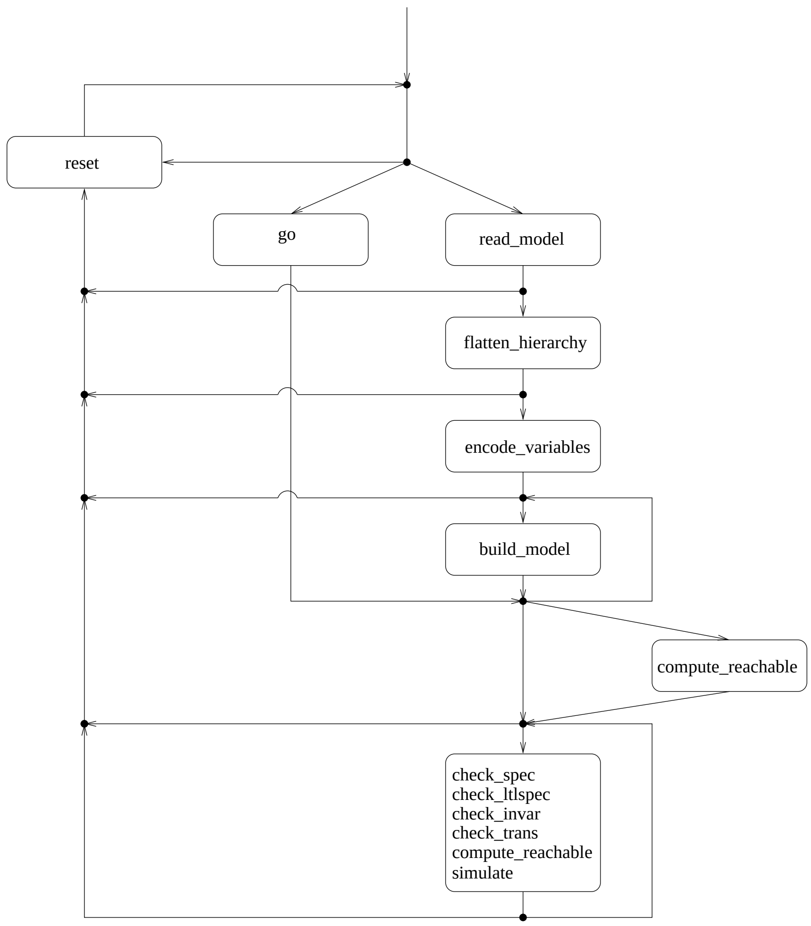
`MAX [`*start* , *final*`]` returns the length of the longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, then *infinity* is returned.

4 Running NuSMV interactively

The main interaction mode of NuSMV is through an interactive shell. In this mode NuSMV enters a read-eval-print loop. The user can activate the various NuSMV computation steps as system commands with different options. These steps can therefore be invoked separately, possibly undone or repeated under different modalities. These steps include the construction of the model under different partitioning techniques, model checking of specifications, and the configuration of the BDD package. The interactive shell of NuSMV is activated from the system prompt as follows ('NuSMV>' is the default NuSMV shell prompt):

```
system_prompt> NuSMV -int ⟨RET⟩
NuSMV release 1.1 (compiled 30-Jul-99 at 7:25 PM)
NuSMV>
```

A NuSMV command is a sequence of words. The first word specifies the command to be executed. The remaining words are arguments to the invoked command. Commands separated by a ';' are executed sequentially; the NuSMV shell waits for each command to terminate in turn. The behavior of commands can depend on environment variables, similar to "csh" environment variables.



In the following we present the possible commands followed by the related environment variables, classified in different categories. Every command answers to the option -h by printing out the command usage. When output is paged for some commands (option -m), it is piped through the program specified by the UNIX PAGER shell variable, if defined, or through UNIX command "more". Environment variables can be assigned a value with the set command.

Command sequences to `NuSMV` must obey the (partial) order specified in the figure depicted in the previous page. For instance, it is not possible to evaluate CTL expressions before the model is built.

4.1 Model Reading and Building

The following commands allow for the parsing and the compilation of the model into BDD.

verbose_level Environment Variable

Controls the verbosity of the system. Possible values are integers from 0 (no messages) to 4 (full messages). The default value is 0.

read_model - *Reads a NuSMV file into NuSMV.* Command

`read_model [-h] [-i model-file]`

Reads a NuSMV file. If the `-i` option is not specified, it reads from the file specified in the environment variable `input_file`.

Command options:

`-i model-file`
: Sets the environment variable `input_file` to `model-file`, and reads the model from the specified file.

input_file Environment Variable

Stores the name of the input file containing the model. It can be set by the '`set`' command or by the command line option '*-i*'. There is no default value.

flatten_hierarchy - *Flattens the hierarchy of modules* Command

`flatten_hierarchy [-h]`

This command is responsible of the instantiation of modules and processes. The instantiation is performed by substituting the actual parameters for the formal parameters, and then by prefixing the result via the instance name.

show_vars - *Shows model's symbolic variables and their values* Command

`show_vars [-h] [-s] [-i] [-m | -o output-file]`

Prints symbolic input and state variables of the model with their range of values (as defined in the input file).

Command Options:

`-s`
: Prints only state variables.

`-i`
: Prints only input variables.

`-m`
: Pipes the output to the program specified by the `PAGER` shell variable if defined, else through the `UNIX` command "more".

`-o output-file`
: Writes the output generated by the command to `output-file`

encode_variables - *Builds the BDD variables necessary to compile the model into BDD.* Command

`encode_variables [-h] [-i order-file]`

Generates the boolean BDD variables and the ADD needed to encode propositionally the (symbolic) variables declared in the model.
The variables are created as default in the order in which they appear in a depth first traversal of the hierarchy.

The input order file can be partial and can contain variables not declared in the model. Variables not declared in the model are simply discarded. Variables declared in the model which are not listed in the ordering input file will be created and appended at the end of the given ordering list, according to the default ordering.

Command options:

`-i order-file`
: Sets the environment variable `input_order_file` to `order-file`, and reads the variable ordering to be used from file `order-file`. This can be combined with the `write_order` command. The variable ordering is written to a file, which can be inspected and reordered by the user, and then read back in.

input_order_file Environment Variable

Indicates the file name containing the variable ordering to be used in building the model by the '`encode_variables`' command. There is no default value.

write_order - *Writes variable order to file.* Command

`write_order [-h] [(-o | -f) order-file]`

Writes the current order of BDD variables in the file specified via the -o option. If no option is specified the environment variable `output_order_file` will be considered. If the variable `output_order_file` is unset (or set to an empty value) then standard output will be used.

Command options:

`-o order-file`
: Sets the environment variable `output_order_file` to `order-file` and then dumps the ordering list into that file.

`-f order-file`
: Alias for -o option. Supplied for backward compatibility.

output_order_file Environment Variable

The file where the current variable ordering has to be written. The default value is '`temp.ord`'.

build_model - *Compiles the flattened hierarchy into BDD* Command

`build_model [-h] [-f] [-m Method]`

Compiles the flattened hierarchy into BDD (initial states, invariants, and transition relation) using the method specified in the environment variable `partition_method` for building the transition relation.

Command options:

`-m Method`
: Sets the environment variable `partition_method` to the value `Method`, and then builds the transition relation. Available methods are `Monolithic`, `Threshold` and `Iwls95CP`.

`-f`

Forces model construction. By default, only one partition method is allowed. This option allows to overcome this default, and to build the transition relation with different partitioning methods.

partition_method Environment Variable

The method to be used in building the transition relation, and to compute images and preimages. Possible values are:

- **Monolithic**. No partitioning at all.
- **Threshold**. Conjunctive partitioning, with a simple threshold heuristic. Assignments are collected in a single cluster until its size grows over the value specified in the variable `conj_part_threshold`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable.
- **Iwls95CP**. Conjunctive partitioning, with clusters generated and ordered according to the heuristic described in *[RAP+95]*. Works in conjunction with the variables `image_cluster_size`, `image_W1`, `image_W2`, `image_W3`, `image_W4`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable. It is also possible to avoid (default) preordering of clusters (see *[RAP+95]*) using `iwls95preorder` variable.

conj_part_threshold Environment Variable

The limit of the size of clusters in conjunctive partitioning. The default value is 0 BDD nodes.

image_cluster_size, image_W{1,2,3,4} Environment Variables

The parameters to configure the behavior of the *Iwls95CP* partitioning algorithm. `image_cluster_size` is used as threshold value for the clusters. The default value is 1000 BDD nodes. The other parameters attribute different weights to the different factors in the algorithm. The default values are 6, 1, 1, 2 respectively. (For a detailed description, please refer to *[RAP+95]*.)

affinity Environment Variable

Enables affinity clustering heuristic described in *[MOON00]*, possible values are 0 or 1. The default value is 1.

iwls95preorder Environment Variable

Enables cluster preordering following heuristic described in *[RAP+95]*, possible values are 0 or 1. The default value is 0. Preordering can be very slow.

image_verbosity Environment Variable

Sets the verbosity for the image method *Iwls95CP*, possible values are `0` or `1`. The default value is `0`.

print_iwls95options - *Prints the Iwls95 Options.* Command

`print_iwls95options [-h]`

This command prints out the configuration parameters of the IWLS95 clustering algorithm, i.e. `image_verbosity`, `image_cluster_size` and `image_W1,2,3,4`.

go - *Initializes the system for the verification.* Command

`go [-h]`

This command initializes the system for verification. It is equivalent to the command sequence `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`, `build_flat_model`, `build_boolean_model`. If some commands have already been executed, then only the remaining ones will be invoked.

Command options:

-h

Prints the command usage.

process_model - *Performs the batch steps and then returns control to the interactive shell.* Command

`process_model [-h] [-i model-file] [-m Method]`

Reads the model, compiles it into BDD and performs the model checking of all the specification contained in it. If the environment variable `forward_search` has been set before, then the set of reachable states is computed. If the environment variables `enable_reorder` and `reorder_method` are set, then the reordering of variables is performed accordingly. This command simulates the batch behavior of NuSMV and then returns the control to the interactive shell.

Command options:

`-i model-file`
Sets the environment variable `input_file` to file `model-file`, and reads the model from file `model-file`.

`-m Method`
Sets the environment variable `partition_method` to `Method` and uses it as partitioning method.

forward_search Environment Variable

Enables the computation of the reachable states during the `process_model` command and when used in conjunction with the `ag_only_search` environment variable enables the use of an ad hoc algorithm to verify invariants.

check_trans - *Checks the transition relation for totality.* Command

`check_trans [-h] [-m | -o output-file]`

Checks if the transition relation is total. If the transition relation is not total then a potential deadlock state is shown out.

Command options:

`-m`
Pipes the output generated by the command to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".

`-o output-file`
Writes the output generated by the command to the file `output-file`.

At the beginning reachable states are computed in order to guarantee that deadlock states are actually reachable.

check_trans Environment Variable

Controls the activation of the totality check of the transition relation during the `process_model` call. Possible values are `0` or `1`. Default value is `0`.

4.2 Commands for Checking Specifications

compute_reachable - *Computes the set of reachable states* Command

`compute_reachable [-h]`

Computes the set of reachable states. The result is then used to simplify image and preimage computations. This can result in improved performances for models with sparse state spaces. Sometimes this option may slow down the performances because the computation of reachable states may be very expensive. The environment variable `forward_search` is set during the execution of this command.

print_reachable_states - *Prints out the number of reachable states.* Command

`print_reachable_states [-h]`

Prints the number of reachable states of the given model. The reachable states are computed if needed.

check_spec - *Performs fair CTL model checking.* Command

`check_spec [-h] [-m | -o output-file] [-n number | -p "ctl-expr [IN context]"]`

Performs fair CTL model checking.

A `ctl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the SPEC formulas in the database are checked.

Command options:

`-m`
: Pipes the output generated by the command in processing `SPEC`s to the program specified by the `PAGER` shell variable if defined, else through the `UNIX` command "more".

`-o output-file`
: Writes the output generated by the command in processing `SPEC`s to the file `output-file`.

`-p "ctl-expr [IN context]"`
: A CTL formula to be checked. `context` is the module instance name which the variables in `ctl-expr` must be evaluated in.

`-n number`
: Checks the CTL property with index `number` in the property database.

If the `ag_only_search` environment variable has been set, and the set of reachable states has been already computed, then a specialized algorithm to check AG formulas is used instead of the standard model checking algorithms.

ag_only_search Environment Variable

Enables the use of an ad hoc algorithm for checking AG formulas. The algorithm, given a formula of the form *AG alpha*, computes the set of states satisfying *alpha*, and checks whether it contains the reachable states is the empty set. If this is the case, then the property is verified, else a counterexample is printed.

check_invar - *Performs model checking of invariants* Command

`check_invar [-h] [-m | -o output-file] [-n number | -p "invar-expr [IN context]"]`

Performs invariant checking on the given model. An invariant is a set of states. Checking the invariant is the process of determining that all states reachable from the initial states lie in the invariant. Invariants to be verified can be provided as simple formulas (without any temporal operators) in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` can be used for checking a particular invariant of the model. If neither `-n` nor `-p` are used, all the invariants are checked.

During checking of invariant all the fairness conditions associated with the model are ignored.

If an invariant does not hold, a proof of failure is demonstrated. This consists of a path starting from an initial state to a state lying outside the invariant. This path has the property that it is the shortest path leading to a state outside the invariant.

Command options:

`-m`
: Pipes the output generated by the program in processing `INVARSPEC`s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".

`-o output-file`
: Writes the output generated by the command in processing `INVARSPEC`s to the file `output-file`.

`-p "invar-expr [IN context]"`
: The command line specified invariant formula to be verified. `context` is the module instance name which the variables in `invar-expr` must be evaluated in.

check_ltlspec - *Performs LTL model checking* Command

`check_ltlspec [-h] [-m | -o output-file] [-n number | -p "ltl-expr [IN context]"]`

Performs model checking of LTL formulas. LTL model checking is reduced to CTL model checking as described in the paper by [CGH97].

A `ltl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the LTLSPEC formulas in the database are checked.

Command options:

`-m`
: Pipes the output generated by the command in processing `LTLSPEC`s to the program specified by the `PAGER` shell variable if defined, else through the Unix command "more".

`-o output-file`
: Writes the output generated by the command in processing `LTLSPEC`s to the file `output-file`.

`-p "ltl-expr [IN context]"`
: An LTL formula to be checked. `context` is the module instance name which the variables in `ltl_expr` must be evaluated in.

`-n number`
: Checks the LTL property with index `number` in the property database.

compute - *Performs computation of quantitative characteristics* Command

`compute [-h] [-m | -o output-file] [-n number | -p "compute-expr [IN context]"]`

This command deals with the computation of quantitative characteristics of real time systems. It is able to compute the length of the shortest (longest) path from two given set of states.

```
MAX [ alpha , beta ]
```

```
MIN [ alpha , beta ]
```

Properties of the above form can be specified in the input file via the keyword `COMPUTE` or directly at command line, using option `-p`.

Option `-n` can be used for computing a particular expression in the model. If neither `-n` nor `-p` are used, all the COMPUTE specifications are computed.

Command options:

`-m`
: Pipes the output generated by the command in processing `COMPUTE`s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".

`-o output-file`
: Writes the output generated by the command in processing `COMPUTE`s to the file `output-file`.

`-p "compute-expr [IN context]"`
: A COMPUTE formula to be checked. `context` is the module instance name which the variables in `compute-expr` must be evaluated in.

`-n number`
: Computes only the property with index `number`

add_property - *Adds a property to the list of properties* Command

`add_property [-h] [(-c | -l | -i | -q) -p "formula [IN context]"]`

Adds a property in the list of properties. It is possible to insert LTL, CTL, INVAR and quantitative (COMPUTE) properties. Every newly inserted property is initialized to unchecked. A type option must be given to properly execute the command.

Command options:

`-c`
: Adds a CTL property.

`-l`
: Adds an LTL property.

`-i`
: Adds an INVAR property.

`-q`
: Adds a quantitative (COMPUTE) property.

`-p "formula [IN context]"`
: Adds the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

4.3 Simulation Commands

To give the user the opportunity to get familiar with the behavior of the model to be checked, a simulation feature has been provided in the NuSMV system. A simulation's facility lets the user widely navigate structures and properties of the model under test through the generation of its possible executions (*traces* from now on). It also allows the user to acquire confidence with the correctness of the model before actual verification of properties. Maximum flexibility and degrees of freedom in a simulation session are achieved introducing three different trace generation policies: deterministic, random and interactive. Each of them corresponds to a different way a state is chosen from a set.
In deterministic simulation mode the first state of a set (whatever it is) is chosen, while in the random one the choice is non deterministically performed. 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).
Third simulation mode lets the user have 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. To avoid that the user easily gets confused when the number of possible choices is too high, an internal limit on the number of states that are showed step by step has been set. If, eventually, such a situation should take place, the system will ask the user to "guide" the simulation via the insertion of some further constraints (as simple expressions) over the variables. The system will continue to ask for constraints insertion until the number of future states will be under a predefined threshold: expressions entered during this particular phase are accumulated (in a logical product) in a single big constraint used only for the actual step of the simulation and discarded before the next one.
The system also makes a control over the expressions given by the user and will request for a further insertion if an inconsistency arises. Cases of inconsistency (i.e. empty set of states) may be located in:

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

It is also possible to specify some constraints as a command line parameter: these expressions will be considered in all of the simulation steps. If entered constraints were too strong an execution may not exist, simulation stops in the last found state and the trace generated until that point is stored in the system (see also command descriptions below).

Every trace generated during a NuSMV session (created either by the counterexample generator or by the simulator itself) can be explored in a second time via the `show_traces` command or the `goto_state` command (see Section 4.4 [Traces Inspection Commands], page 34).

Commands descriptions follow:

pick_state - *Picks a state from the set of initial states* Command

`pick_state [-h] [-v] [-r | -i [-a]] [-c "constraints"]`

Chooses an element from the set of initial states, and makes it the `current state` (replacing the old one). The chosen state is stored as the first state of a new trace ready to be lengthened by `steps` states by the `simulate` command. The state can be chosen according to different policies which can be specified via command line options. By default the state is chosen in a deterministic way.

Command Options:

`-v`

Verbosely prints out chosen state (all state variables, otherwise it prints out only the label `t.1` of the state chosen, where `t` is the number of the new trace, that is the number of traces so far generated plus one).

`-r`
: Randomly picks a state from the set of initial states.

`-i`
: Enables the user to interactively pick up an initial state. The user is requested to choose a state from a list of possible items (every item in the list doesn't show state variables unchanged with respect to a previous item). If the number of possible states is too high, then the user has to specify some further constraints as "simple expression".

`-a`
: Displays all state variables (changed and unchanged with respect to a previous item) in an interactive picking. This option works only if the `-i` options has been specified.

`-c "constraints"`
: Uses `constraints` to restrict the set of initial states in which the state has to be picked. `constraints` must be enclosed between double quotes `" "`.

show_traces - *Shows the traces generated in a NuSMV session* Command

`show_traces [ [-h] [-v] [-t] [-m | -o output-file] -a | trace_number ]`

Shows the traces currently stored in system memory, if any. By default it shows the last generated trace, if any.

Command Options:

`-v`
: Verbosely prints traces content (all state variables, otherwise it prints out only those variables that have changed their value from previous state).

`-t`
: Prints only the total number of currently stored traces.

`-a`
: Prints all the currently stored traces.

`-m`
: Pipes the output through the program specified by the `PAGER` shell variable if defined, else through the `UNIX` command "more".

`-o output-file`
: Writes the output generated by the command to `output-file`

`trace_number`
: The (ordinal) identifier number of the trace to be printed.

showed_states Environment Variable

Controls the maximum number of states showed during an interactive simulation session. Possible values are integers from 1 to 100. The default value is 25.

simulate - *Performs a simulation from the current selected state* Command

`simulate [-h] [-p | -v] [-r | -i [-a]] [-c "constraints"] steps`

Generates a sequence of at most `steps` states (representing a possible execution of the model), starting from the *current state*. The current state must be set via the *pick_state* or *goto_state* commands.

It is possible to run the simulation in three ways (according to different command line policies): deterministic (the default mode), random and interactive.

The resulting sequence is stored in a trace indexed with an integer number taking into account the total number of traces stored in the system. There is a different behavior in the way traces are built, according to how *current state* is set: *current state* is always put at the beginning of a new trace (so it will contain at most steps + 1 states) except when it is the last state of an existent old trace. In this case the old trace is lengthened by at most steps states.

Command Options:

`-p`
: Prints current generated trace (only those variables whose value changed from the previous state).

`-v`
: Verbosely prints current generated trace (changed and unchanged state variables).

`-r`
: Picks a state from a set of possible future states in a random way.

`-i`
: Enables the user to interactively choose every state of the trace, step by step. If the number of possible states is too high, then the user has to specify some constraints as simple expression. These constraints are used only for a single simulation step and are *forgotten* in the following ones. They are to be intended in an opposite way with respect to those constraints eventually entered with the pick_state command, or during an interactive simulation session (when the number of future states to be displayed is too high), that are *local* only to a single step of the simulation and are *forgotten* in the next one.

`-a`
: Displays all the state variables (changed and unchanged) during every step of an interactive session. This option works only if the `-i` option has been specified.

`-c "constraints"`
: Performs a simulation in which computation is restricted to states satisfying those `constraints`. The desired sequence of states could not exist if such constraints were too strong or it may happen that at some point of the simulation a future state satisfying those constraints doesn't exist: in that case a trace with a number of states less than `steps` trace is obtained. Note: `constraints` must be enclosed between double quotes `" "`.

`steps`
: Maximum length of the path according to the constraints. The length of a trace could contain less than `steps` states: this is the case in which simulation stops in an intermediate step because it may not exist any future state satisfying those constraints.

A typical execution sequence of a simulation session could be as follows:
suppose to use the model described below[1].

[1] This model (short.smv) is taken from the `NuSMV` distribution.

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

This model has now to be read into the `NuSMV` system[2] (see Section 4.1 [Model Reading and Building], page 21). A starting point (i.e. a state) is needed before the simulator could start to build any trace. By default the simulator gets the actual *current state* as a starting point of every new trace: so it will be included as the first state of a new trace except if otherwise stated; the user has been provided with the following two `NuSMV` commands to overcome this behavior:

- `goto_state`: this command (see Section 4.4 [Traces Inspection Commands], page 34) provides the user to set any state of any already existent trace as the *current state*;
- `pick_state`: this command can be used whenever one wants to choose the starting point of a new trace among the set of initial states, and it has to be used when a *current state* doesn't exist yet (that is when the system has being reset).

As at this point of the example *current state* doesn't exist, and there is no trace currently stored in the system, an item from the set of initial states has to be picked using the `NuSMV` `pick_state` command. A simulation session can be started now, using the `simulate` command. The system will generate a trace according to the options entered via command line[3]. New builded 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.

```
system_prompt> NuSMV -int short.smv
NuSMV release 1.1 (compiled 30-Jul-99 at 7:25 PM)
NuSMV > go
NuSMV > pick_state -r
NuSMV > print_current_state -v
Current state is 1.1
request = 0
state = ready
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 ###################
```

[2] It is supposed that every `NuSMV` command is followed by a ⟨RET⟩ keystroke. `NuSMV` commands are written in a **bold face** to distinguish them from system output messages.

[3] Note that the `simulate` argument *steps* refers to the number of states that the system will generate starting from the *current state* which is always stored as the first state of a new trace (except when it is the last one in an existent trace). For this reason the resulting trace contains 4 states (three state have been appended to a trace containing *current state* at the beginning).

```
-> 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
```

All that the user has to do to change the starting point of the next new trace is to use the `pick_state` or the `goto_state` commands[4]. Note that in the next example trace 1 has been lengthened as *current state* was set to the last state of an existing trace.

```
NuSMV > goto_state 1.4
request = 1
state = busy
NuSMV > simulate -r 3
*********  Starting Simulation From State  1.4  *********
NuSMV > show_traces 1
#################### Trace number: 1 ####################
-> State 1.1 <-
    request = 0
    state = ready
-> State 1.2 <-
    request = 1
    state = busy
-> State 1.3 <-
    state = ready
-> State 1.4 <-
    state = busy
-> State 1.5 <-
    state = ready
-> State 1.6 <-
    request = 0
    state = busy
-> State 1.7 <-
    request = 1
```

The user is also able to interactively choose the states of the trace he wants to build: an example of an interactive picking of an initial state is showed below:

```
NuSMV > pick_state -i -a
***************  AVAILABLE FUTURE STATES  ***************

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

[4] *current state* is deleted executing the `reset` command (see Section 4.6 [Administration Commands], page 39).

```
    request = 1
    state = ready

Choose a state from the above (0-1): 1 ⟨RET⟩

Chosen state is: 1
request = 1
state = ready
```

Furthermore, the user is able to specify some constraints to restrict sets of states from which the simulator will pick out. Constraints can be set for both the `pick_state` command and the `simulate` command. For the `simulate` command these constraints 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 therefore to be intended in an opposite way with respect to those constraints eventually entered together with the `pick_state` command, or during an interactive simulation session (when the number of future states to be displayed is too high), that are "local" only to a single step of the simulation and are "forgotten" in the next one. For example lets pick an initial state with some simple constraints:

```
NuSMV > pick_state -c "request = 1" -i
***************  AVAILABLE FUTURE STATES  ***************

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

There's only one future state. Press Return to Proceed. ⟨RET⟩

Chosen state is: 0
request = 1
state = ready
```

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).

4.4 Traces Inspection Commands

A trace is a sequence of states corresponding to a possible execution of the model. Traces are created by `NuSMV` when a formula is found to be false; they are also generated by the simulation feature (Section 4.3 [Simulation Commands], page 28). Each trace has a number, and the states are numbered within the trace. Trace n has states *n.1, n.2, n.3,* "...".

The trace inspection commands of `NuSMV` allow to navigate along the traces produced by `NuSMV`. During the navigation, there is a *current state*, and the *current trace* is the trace the *current state* belongs to. The commands are the following:

goto_state - *Goes to a given state of a trace* Command

`goto_state [-h] state`

Makes `state` the *current state*. This command is used to navigate alongs traces produced by NuSMV. During the navigation, there is a *current state*, and the *current trace* is the trace the *current state* belongs to.

print_current_state - *Prints out the current state* Command

`print_current_state [-h] [-v]`

Prints the name of the *current state* if defined.

Command options:

`-v`

Prints the value of all the state variables of the *current state*.

4.5 Interface to the DD Package

`NuSMV` uses the state of the art BDD package CUDD *[Som98]*. Control over the BDD package can very important to tune the performance of the system. In particular, the order of variables is critical to control the memory and the time required by operations over BDDs. Reordering methods can be activated to determine better variable orders, in order to reduce the size of the existing BDDs. Reordering methods can be activated either

Reordering of the variables can be triggered in two ways: by the user, or by the BDD package. In the first way, reordering is triggered by the interactive shell command `dynamic_var_ordering` with the `-f` option.

Reordering is triggered by the BDD package when the number of nodes reaches a given threshold. The threshold is initialized and automatically adjusted after each reordering by the package. This is called dynamical reordering, and can be enabled or disabled by the user. Dynamic reordering is enabled with the shell command `dynamic_var_ordering` with the option `-e`, and disabled with the `-d` option.

enable_reorder Environment Variable

Specifies whether dynamic reordering is enabled (when value is '`0`') or disabled (when value is '`1`').

reorder_method Environment Variable

Specifies the ordering method to be used when dynamic variable reordering is fired. The possible values, corresponding to the reordering methods available with the CUDD package, are listed below. The default value is `sift`.

`sift`: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.

`random`: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.

`random_pivot`:
Same as `random`, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.

`sift_converge`:
The `sift` method is iterated until no further improvement is obtained.

`symmetry_sift`:
This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.

`symmetry_sift_converge`:
The `symmetry_sift` method is iterated until no further improvement is obtained.

`window{2,3,4}`:
Permutes the variables within windows of n adjacent variables, where n can be either 2, 3 or 4, so as to minimize the overall BDD size.

`window{2,3,4}_converge`:
The window{2,3,4} method is iterated until no further improvement is obtained.

`group_sift`:
This method is similar to `symmetry_sift`, but uses more general criteria to create groups.

`group_sift_converge`:
The `group_sift` method is iterated until no further improvement is obtained.

`annealing`:
This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.

`genetic`: This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.

`exact`: This method implements a dynamic programming approach to exact reordering. It only stores a BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.

`linear`: This method is a combination of sifting and linear transformations.

`linear_conv`:
The `linear` method is iterated until no further improvement is obtained.

dynamic_var_ordering - *Deals with the dynamic variable ordering.* Command

`dynamic_var_ordering [-d] [-e <method>] [-f <method>] [-h]`

Controls the application and the modalities of (dynamic) variable ordering. Dynamic ordering is a technique to reorder the BDD variables to reduce the size of the existing BDDs. When no options are specified, the current status of dynamic ordering is displayed. At most one of the options `-e`, `-f`, and `-d` should be specified.

Dynamic ordering may be time consuming, but can often reduce the size of the BDDs dramatically. A good point to invoke dynamic ordering explicitly (using the `-f` option) is after the commands `build_model`, once the transition relation has been built. It is possible to save the ordering found using `write_order` in order to reuse it (using `build_model -i order-file`) in the future.

Command options:

`-d`
Disable dynamic ordering from triggering automatically.

`-e <method>`
Enable dynamic ordering to trigger automatically whenever a certain threshold on the overall BDD size is reached. `<method>` must be one of the following:

sift: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.

random: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.

random_pivot: Same as **random**, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.

sift_converge: The **sift** method is iterated until no further improvement is obtained.

symmetry_sift: This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.

symmetry_sift_converge: The **symmetry_sift** method is iterated until no further improvement is obtained.

window2,3,4: Permutes the variables within windows of "n" adjacent variables, where "n" can be either 2, 3 or 4, so as to minimize the overall BDD size.

window2,3,4_converge: The **window2,3,4** method is iterated until no further improvement is obtained.

group_sift: This method is similar to **symmetry_sift**, but uses more general criteria to create groups.

group_sift_converge: The **group_sift** method is iterated until no further improvement is obtained.

annealing: This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.

genetic: This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.

exact: This method implements a dynamic programming approach to exact reordering. It only stores a BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.

linear: This method is a combination of sifting and linear transformations.

linear_converge: The **linear** method is iterated until no further improvement is obtained.

`-f <method>`
: Force dynamic ordering to be invoked immediately. The values for `<method>` are the same as in option `-e`.

print_bdd_stats - *Prints out the BDD statistics and parameters* Command

`print_bdd_stats [-h]`

Prints the statistics for the BDD package. The amount of information depends on the BDD package configuration established at compilation time. The configurtion parameters are printed out too. More information about statistics and parameters can be found in the documentation of the CUDD Decision Diagram package.

set_bdd_parameters - *Creates a table with the value of all currently active NuSMV flags and change accordingly the configurable parameters of the BDD package.* Command

`set_bdd_parameters [-h] [-s]`

Applies the variables table of the NuSMV environnement to the BDD package, so the user can set specific BDD parameters to the given value. This command works in conjunction with the `print_bdd_stats` and `set` commands.

`print_bdd_stats` first prints a report of the parameters and statistics of the current bdd_manager. By using the command `set`, the user may modify the value of any of the parameters of the underlying BDD package. The way to do it is by setting a value in the variable `BDD.parameter name` where `parameter name` is the name of the parameter exactly as printed by the `print_bdd_stats` command.

Command options:

-s

Prints the BDD parameter and statistics after the modification.

4.6 Administration Commands

! *shell_command* Command

Executes a shell command. The *shell_command* is executed by calling `bin/sh -c` *shell_command*. If the command does not exists or you have not the right to execute it, then an error message is printed.

alias - *Provides an alias for a command* Command

`alias [-h] [<name> [<string>]]`

The "alias" command, if given no arguments, will print the definition of all current aliases.

Given a single argument, it will print the definition of that alias (if any). Given two arguments, the keyword `<name>` becomes an alias for the command string `<string>`, replacing any other alias with the same name.

Command options:

<*name*>

Alias

<*string*>

Command string

It is possible to create aliases that take arguments by using the history substitution mechanism. To protect the history substitution character '`%`' from immediate expansion, it must be preceded by a '\' when entering the alias.

For example:

```
NuSMV> alias read "read_model -i \%:1.smv ; set input_order_file \%:1.ord"
NuSMV> read short
```

will create an alias 'read', execute "read_model -i short.smv; set input_order_file short.ord".

And again:

```
NuSMV> alias echo2 "echo Hi ; echo \%* !"
NuSMV> echo2 happy birthday
```

will print:

```
Hi
happy birthday !
```

CAVEAT: Currently there is no check to see if there is a circular dependency in the alias definition. e.g.

```
NuSMV> alias foo "echo print_bdd_stats; foo"
```

creates an alias which refers to itself. Executing the command `foo` will result an infinite loop during which the command `print_bdd_stats` will be executed.

echo - *Merely echoes the arguments* Command

`echo [-h] <args>`

Echoes its arguments to standard output.

help - *Provides on-line information on commands* Command

`help [-a] [-h] [<command>]`

If invoked with no arguments "help" prints the list of all commands known to the command interpreter. If a command name is given, detailed information for that command will be provided.

Command options:

-a

Provides a list of all internal commands, whose names begin with the underscore character ('_') by convention.

history - *list previous commands and their event numbers* Command

`history [-h] [<num>]`

Lists previous commands and their event numbers. This is a UNIX-like history mechanism inside the NuSMV shell.

Command options:

<num>

Lists the last <num> events. Lists the last 30 events if <num> is not specified.

History Substitution:

The history substitution mechanism is a simpler version of the csh history substitution mechanism. It enables you to reuse words from previously typed commands.

The default history substitution character is the '%' ('!' is default for shell escapes, and '#' marks the beginning of a comment). This can be changed using the "set" command. In this description '%' is used as the history_char. The '%' can appear anywhere in a line. A line containing a history substitution is echoed to the screen after the substitution takes place. '%' can be preceded by a '\' in order to escape the substitution, for example, to enter a '%' into an alias or to set the prompt.

Each valid line typed at the prompt is saved. If the "history" variable is set (see help page for "set"), each line is also echoed to the history file. You can use the "history" command to list the previously typed commands.

Substitutions:

At any point in a line these history substitutions are available.

%:0 Initial word of last command.

%:n n-th argument of last command.

%$ Last argument of last command.

*%** All but initial word of last command.

%% Last command.

%stuf Last command beginning with "stuf".

%n Repeat the n-th command.

%-n Repeat the n-th previous command.

^old^new Replace "old" with "new" in previous command. Trailing spaces are significant during substitution. Initial spaces are not significant.

print_usage - *Prints processor and BDD statistics.* Command

`print_usage [-h]`

Prints a formatted dump of processor-specific usage statistics, and BDD usage statistics. For Berkeley Unix, this includes all of the information in the `getrusage()` structure.

Command options:

-h

Prints the command usage.

quit - *exits NuSMV* Command

`quit [-h] [-s]`

Stops the program. Does not save the current network before exiting.

Command options:

-s

Frees all the used memory before quitting. This is slower, and it is used for finding memory leaks.

reset - *Resets the whole system.* Command

`reset [-h]`

Resets the whole system, in order to read in another model and to perform verification on it.

Command options:

-h

Prints the command usage.

set - *Sets an environment variable* Command

`set [-h] [<name>] [<value>]`

A variable environment is maintained by the command interpreter. The "set" command sets a variable to a particular value, and the "unset" command removes the definition of a variable. If "set" is given no arguments, it prints the current value of all variables.

Command options:

-h

Prints the command usage.

<*name*>

Variable name

<*value*>

Value to be assigned to the variable.

Interpolation of variables is allowed when using the set command. The variables are referred to with the prefix of '$'. So for example, what follows can be done to check the value of a set variable:

```
NuSMV> set foo bar
NuSMV> echo $foo
bar
```

The last line "bar" will be the output produced by NuSMV.

Variables can be extended by using the character ':' to concatenate values. For example:

```
NuSMV> set foo bar
NuSMV> set foo $foo:foobar
NuSMV> echo $foo
bar:foobar
```

The variable `foo` is extended with the value `foobar` .

Whitespace characters may be present within quotes. However, variable interpolation lays the restriction that the characters ':' and '/' may not be used within quotes. This is to allow for recursive interpolation. So for example, the following is allowed

```
NuSMV> set "foo bar" this
NuSMV> echo $"foo bar"
```

```
this
```
The last line will be the output produced by NuSMV.

But in the following, the value of the variable `foo/bar` will not be interpreted correctly:

```
NuSMV> set "foo/bar" this
NuSMV> echo $"foo/bar"
foo/bar
```
If a variable is not set by the "set" command, then the variable is returned unchanged.

Different commands use environment information for different purposes. The command interpreter makes use of the following parameters:

autoexec
: Defines a command string to be automatically executed after every command processed by the command interpreter. This is useful for things like timing commands, or tracing the progress of optimization.

open_path
: "open_path" (in analogy to the shell-variable PATH) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory (.) is the first item in this list. The standard system library (typically $NuSMV_LIBRARY_PATH) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.

nusmv_stderr
: Standard error (normally stderr) can be re-directed to a file by setting the variable nusmv_stderr.

nusmv_stdout
: Standard output (normally stdout) can be re-directed to a file by setting the variable nusmv_stdout.

source - *Executes a sequence of commands from a file* Command

`source [-h] [-p] [-s] [-x] <file> [<args>]`

Reads and executes commands from a file.

Command options:

-p
: Prints a prompt before reading each command.

-s
: Silently ignores an attempt to execute commands from a nonexistent file.

-x
: Echoes each command before it is executed.

<file>
: File name

Arguments on the command line after the filename are remembered but not evaluated. Commands in the script file can then refer to these arguments using the history substitution mechanism.

EXAMPLE:

Contents of test.scr:

```
read_model -i %:2
```

```
flatten_hierarchy
build_variables
build_model
compute_fairness
```

Typing "source test.scr short.smv" on the command line will execute the sequence

```
 read_model -i short.smv
flatten_hierarchy
build_variables
build_model
compute_fairness
```

(In this case `%:0` gets "source", `%:1` gets "test.scr", and `%:2` gets "short.smv".) If you type "alias st source test.scr" and then type "st short.smv bozo", you will execute

```
 read_model -i bozo
flatten_hierarchy
build_variables
build_model
compute_fairness
```

because "bozo" was the second argument on the last command line typed. In other words, command substitution in a script file depends on how the script file was invoked. Switches passed to a command are also counted as positional parameters. Therefore, if you type "st -x short.smv bozo", you will execute

```
 read_model -i short.smv
flatten_hierarchy
build_variables
build_model
compute_fairness
```

To pass the "-x" switch (or any other switch) to "source" when the script uses positional parameters, you may define an alias. For instance, "alias srcx source -x".

time - *Provides a simple CPU elapsed time value* Command

`time [-h]`

Prints the processor time used since the last invocation of the "time" command, and the total processor time used since NuSMV was started.

unalias - *Removes the definition of an alias.* Command

`unalias [-h] <alias-names>`

Removes the definition of an alias specified via the alias command.

Command options:

<*alias-names*>
: Aliases to be removed

unset - *Unsets an environment variable* Command

`unset [-h] <variables>`

A variable environment is maintained by the command interpreter. The "set" command sets a variable to a particular value, and the "unset" command removes the definition of a variable.

Command options:

-h

Prints the command usage.

<variables>

Variables to be unset

usage - *Provides a dump of process statistics* Command

`usage [-h]`

Prints a formatted dump of processor-specific usage statistics. For Berkeley Unix, this includes all of the information in the getrusage() structure.

which - *Looks for a file called "file_name"* Command

`which [-h] <file_name>`

Looks for a file in a set of directories which includes the current directory as well as those in the NuSMV path. If it finds the specified file, it reports the found file's path. The searching path is specified through the `"set open_path"` command in `".nusmvrc"`.

Command options:

<file_name>

File to be searched

4.7 Other Environment Variables

The behavior of the system depends on the value of some environment variables. For instance, an environment variable specifies the partitioning method to be used in building the transition relation. The value of environment variables can be inspected and modified with the 'set' command. Environment variables can be either logical or utility.

autoexec Environment Variable

Defines a command string to be automatically executed after every command processed by the command interpreter. This may be useful for timing commands, or tracing the progress of optimization.

filec Environment Variable

Enables file completion a la "csh". If the system has been compiled with the "readline" library, the user is able to perform file completion by typing the ⟨TAB⟩ key (in a way similar to the file completion inside the "bash" shell). If the system has not been compiled with the "readline" library, a built-in method to perform file completion a la "csh" can be used. This method is enabled with the 'set filec' command. The "csh" file completion method can be also enabled if the "readline" library has been used. In this case the features offered by readline will be disabled.

shell_char Environment Variable

`shell_char` specifies a character to be used as shell escape. The default value of this environment variable is '!'.

history_char Environment Variable

`history_char` specifies a character to be used in history substitutions. The default value of this environment variable is '%'.

open_path Environment Variable

`open_path` (in analogy to the shell-variable `PATH`) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory ('.') is first in this list. The standard system library (`NuSMV_LIBRARY_PATH`) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.

nusmv_stderr Environment Variable

Standard error (normally `stderr`) can be re-directed to a file by setting the variable `nusmv_stderr`.

nusmv_stdout Environment Variable

Standard output (normally `stdout`) can be re-directed to a file by setting the internal variable `nusmv_stdout`.

nusmv_stdin Environment Variable

Standard input (normally `stdin`) can be re-directed to a file by setting the internal variable `nusmv_stdin`.

5 Running NuSMV batch

When the `-int` option is not specified, `NuSMV` runs as a batch program, in the style of `SMV`, performing (some of) the steps described in previous section in a fixed sequence.

`system_prompt> NuSMV [command line options]` *input-file* ⟨RET⟩

The program described in *input-file* is processed, and the corresponding finite state machine is built. Then, if *input-file* contains formulas to verify, their truth in the specified structure is evaluated. For each formula which is not true a counterexample is printed.

The batch mode can be controlled with the following command line options:

```
NuSMV [-s] [-ctt] [-lp] [-n idx] [-v vl] [-cpp] [-is] [-ic] [-ils]
      [-ii] [-r] [-f] [-int] [-h | -help] [-i iv_file] [-o ov_file]
      [-AG] [-GUI] [-load script_file] [-reorder] [-dynamic] [-m method]
      [[-mono]|[-thresh cp_t]|[-cp cp_t]|[-iwls95 cp_t]] [-coi]
      [-noaffinity] [-iwls95preorder] [-bmc] [-bmc_length k]
      [-ofm fm_file] [-obm bm_file] [input-file]
```

where the meaning of the options is described below. If *input-file* is not provided in batch mode, then the model is read from standard input.

`-s` Avoids to load the `NuSMV` commands contained in '`~/.nusmvrc`' or in '`.nusmvrc`' or in '`$${NuSMV_LIBRARY_PATH}/master.nusmvrc`'.

`-ctt` Checks whether the transition relation is total.

`-lp` Lists all properties in SMV model

`-n` *idx* Specifies which property of SMV model shuld be checked

`-v` *verbose-level*
Enables printing of additional information on the internal operations of `NuSMV`. Setting *verbose-level* to `1` gives the basic information. Using this option makes you feel better, since otherwise the program prints nothing until it finishes, and there is no evidence that it is doing anything at all. Setting the *verbose-level* higher than 1 enables printing of much extra information.

`-cpp` Runs preprocessor on SMV files

`-is` Does not check SPEC

`-ic` Does not check COMPUTE

`-ils` Does not check LTLSPEC

`-ii` Does not check INVARSPEC

`-r` Prints the number of reachable states before exiting. If the `-f` option is not used, the set of reachable states is computed.

`-f` Computes the set of reachable states before evaluating CTL expressions.

`-int` Starts interactive shell.

`-help`
`-h` Prints the command line help.

`-i` *iv_file* Reads the variable ordering from file *iv_file*.

`-o` *ov_file* Reads the variable ordering from file *ov_file*.

`-AG` Verifies only AG formulas using an ad hoc algorithm (see documentation for the `ag_only_search` environment variable).

`-GUI` Enables GUI behavior

`-load` *cmd-file*
: Interprets NuSMV commands from file *cmd-file*.

`-reorder` Enables variable reordering after having checked all the specification if any.

`-dynamic` Enables dynamic reordering of variables

`-m` *method* Uses *method* when variable ordering is enabled. Possible values for method are those allowed for the `reorder_method` environment variable (see Section 4.5 [Interface to the DD package], page 35).

`-mono` Enables monolithic transition relation

`-thresh` *cp_t*
: conjunctive partitioning with threshold of each partition set to *cp_t* (DEFAULT, with *cp_t*=1000)

`-cp` *cp_t* DEPRECATED: use -thresh instead.

`-iwls95` *cp_t*
: Enables Iwls95 conjunctive partitioning and sets the threshold of each partition to *cp_t*

`-coi` Enables cone of influence reduction

`-noaffinity`
: Disables affinity clustering

`-iwls95preoder`
: Enables iwls95 preordering

`-bmc` Enables BMC instead of BDD model checking

`-bmc` *k* Sets bmc_length variable, used by BMC

`-ofm` *fm_file*
: prints flattened model to file *fn_file*

`-obm` *bm_file*
: Prints boolean model to file *bn_file*

6 Getting started with BMC

In this chapter we give a short introduction to the use of Bounded Model Checking (BMC) in NuSMV2. The aim is to get the user start using the BMC functionality of NuSMV2 in a fast and easy way. The tutorial is based on a simple example that can be found in the file bmc_tutorial.smv, in the distribution package. No background on BMC is assumed. For a more in-depth introduction to the theory underlying BMC please refer to *[BCCZ99]*.

6.1 A simple model

Consider the following model, representing a simple, deterministic counter modulo 8:

```
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.

6.2 Checking of LTL properties

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 NuSMV2 BMC to demonstrate its falsity.

So let's start NuSMV2: open a shell, make your current directory where the tutorial file bmc_tutorial.smv is located, make sure that the NuSMV executable file can be found by the underling system, and finally launch it:

```
system_prompt> NuSMV -int bmc_tutorial.smv
*** This is NuSMV <version> (compiled ...)
*** For more information of NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv-users@irst.itc.it>.
NuSMV >
```

Now setup the BMC module in NuSMV2, using the command 'go_bmc':

```
NuSMV > go_bmc
NuSMV >
```

NuSMV2 has silently setup its BMC module and you can now work with it. It is time to try to check the false property 'G (y=4 -> X y=6)'. You can use the command 'check_ltlspec_bmc' specifying the property you are interested in checking through the option '-p':

```
NuSMV > check_ltlspec_bmc -p "G ( y=4 -> X y=6 )"
-- 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
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
NuSMV >
```

NuSMV2 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.

```
 length:  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
```

If you look at the NuSMV2 output on the command 'check_ltlspec_bmc', you can see that, before finding the counterexample of length 5, NuSMV2 also tried to construct counterexamples of lower length (i.e. with bound from 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:

```
 length:  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 NuSMV2 to say anything about the following state.

In general, the 'check_ltlspec_bmc' command tries to find a counterexample of increasing length, and immediately stops the iteration when it succeeds, declaring the formula is false. You can control the maximum number of iterations of 'check_ltlspec_bmc' by specifying a value for the bmc_length environment variable. You can see and set the values of environment variables using the 'set' command. Try what follows:

```
NuSMV > set
The user variables are:
...
bmc_length                      "10"
...
NuSMV >
```

The default value for the variable bmc_length is '10'. In this way the check_ltlspec_bmc command iterates length from 0 to 10 and then exits even if no counterexample has been previously found. For instance, if we set a smaller value NuSMV2 cannot prove the falsity of the formula:

```
NuSMV > set bmc_length 4
NuSMV > set
```

```
The user variables are:
...
bmc_length                    "4"
...
NuSMV >

NuSMV > check_ltlspec_bmc -p "G ( y=4 -> X y=6 )"
-- 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 ...
NuSMV >
```

It is possible to check a specification based on a single bound, without iterating from 0 to `bmc_length`, by using the command 'check_ltlspec_bmc_onepb' as follows:

```
NuSMV > set bmc_length 5
NuSMV > check_ltlspec_bmc_onepb -p "G ( y=4 -> X y=6 )"
-- specification  G (y = 4 ->  X y = 6)   is false
-- as demonstrated by the following execution sequence
State 4.1: y = 0
State 4.2: y = 1
State 4.3: y = 2
State 4.4: y = 3
State 4.5: y = 4
State 4.6: y = 5
NuSMV >
```

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:

```
NuSMV > set bmc_length 10
NuSMV > check_ltlspec_bmc -p "!G F y = 2"
-- 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 ...
-- no counterexample found with bound 5 for ...
-- no counterexample found with bound 6 for ...
-- specification ! G F y = 2   is false
-- as demonstrated by the following execution sequence
State 2.1: y = 0
State 2.2: y = 1
State 2.3: y = 2
State 2.4: y = 3
State 2.5: y = 4
State 2.6: y = 5
State 2.7: y = 6
State 2.8: y = 7
NuSMV >
```

In this example NuSMV2 had been increasing problem length until it found a loop around [0..7] which contains the y value 2, that shows that the property cannot be true.

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.

```
                   ---------------------
                  |                     |
                  |                     |
                  V                     |
       o--->o--->o--->o-...->o--->o--->o--->o--->
time:  0    1    2    3      k-2  k-1  k    k+1   ...
                 l
```

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). Because the loop always jumps to a previous time it is called 'loopback'. Please also note that because of the bound, state k+1 is forced to be equal to state 2. You can control these two parts parameters (k and l) by setting the environment variables `bmc_length` and `bmc_loopback` as we are going to show now.

```
NuSMV > set bmc_length 7
NuSMV > set bmc_loopback 0
NuSMV > check_ltlspec_bmc_onepb -p "!G F (y =2)"
-- specification ! G  F y = 2   is false
-- as demonstrated by the following execution sequence
State 5.1: y = 0
State 5.2: y = 1
...
State 5.7: y = 6
State 5.8: y = 7
NuSMV >
```

```
           ----------------------------------
          |                                  |
          |                                  |
          V                                  |
          o--->o--->o--->o--->o--->o--->o--->o
length:   0    1    2    3    4    5    6    7
y value:  0    1    2    3    4    5    6    7
```

As you can see by properly setting k and l NuSMV2 is able to find a counterexample for a false liveness property. Try now to change values of k and l:

```
NuSMV > set bmc_length 9
NuSMV > check_ltlspec_bmc_onepb -p "!G F (y =2)"
-- no counterexample found with bound 9 and loop at 0 for specification
   ! G  F y = 2
```

```
NuSMV > set bmc_length 7
NuSMV > set bmc_loopback 1
NuSMV > check_ltlspec_bmc_onepb -p "!G F (y =2)"
-- no counterexample found with bound 7 and loop at 1 for specification
   ! G  F y = 2
NuSMV >
```

In these two examples NuSMV2 did not find a counterexample. Look at the following figures to understand this behaviour.

Case k = 9, l = 0:

```
           ---------------------------------------------
          |                                             |
          |                                             |
          V                                             |
          o--->o--->o--->o--->o--->o--->o--->o--->o--->o
length:   0    1    2    3    4    5    6    7    8    9
y value:  0    1    2    3    4    5    6    7    0    1
```

Case k = 7, l = 1:

```
                ------------------------------
               |                              |
               |                              |
               V                              |
          o--->o--->o--->o--->o--->o--->o--->o
length:   0    1    2    3    4    5    6    7
y value:  0    1    2    3    4    5    6    7
```

Cases (k=9, l=0) and (k=7, l=1) are counterexample structures which do not match with the model of the counter, so it is not possible for NuSMV2 to find a valid counterexample if its parameters are not properly set.

In this specific case all couples of (k, l) which actually produce a counterexamples have a loop length of 7. In NuSMV2 it is possible to specify the loop in four different ways by using always the bmc_loopback environment variable:

- **The loop as a precise time-point.** Assign a natural number to the bmc_loopback environment variable.
- **The loop length.** Assign a negative number to bmc_loopback. 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.** Assign the symbol 'X' to the bmc_loopback variable. In this case NuSMV2 will not find infinite counterexamples.
- **All possible loops.** Assign the symbol '*' to the bmc_loopback variable. In this case NuSMV2 will search counterexamples for paths with any possible loopbacks in the path bounds. A counterexample with no loop will be also searched. This is the default value assigned to the bmc_loopback variable when NuSMV2 starts its execution.

We show now that by setting the loop length to 7 NuSMV2 is able to find a valid counterexample with any value of bmc_length:

```
NuSMV > set bmc_loopback -7
NuSMV > set bmc_length 12
NuSMV > check_ltlspec_bmc_onepb -p "!G F y = 2"
```

```
-- specification ! G  F y = 2   is false
-- as demonstrated by the following execution sequence
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
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:

```
                                  ----------------------------------
                                 |                                  |
                                 |                                  |
                                 V                                  |
          o--->o--->o--->o--->o--->o--->o--->o--->o--->o--->o--->o--->o
length:   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 we had specified a greater value for `bmc_length` we would have obtained again a valid counterexample whose loop would have been shifted farther to the right in the figure.

If no loopback is specified, `NuSMV2` is not able to find a counterexample for the given liveness property:

```
NuSMV > set bmc_length 12
NuSMV > set bmc_loopback X
NuSMV > check_ltlspec_bmc -p "!G F y = 2"
-- no counterexample found with bound 0 and no loop for ...
-- no counterexample found with bound 1 and no loop for ...
-- no counterexample found with bound 2 and no loop for ...
-- no counterexample found with bound 3 and no loop for ...
-- no counterexample found with bound 4 and no loop for ...
-- no counterexample found with bound 5 and no loop for ...
-- no counterexample found with bound 6 and no loop for ...
-- no counterexample found with bound 7 and no loop for ...
-- no counterexample found with bound 8 and no loop for ...
-- no counterexample found with bound 9 and no loop for ...
-- no counterexample found with bound 10 and no loop for ...
-- no counterexample found with bound 11 and no loop for ...
-- no counterexample found with bound 12 and no loop for ...
NuSMV >
```

Please note that in this example we used command 'check_ltlspec_bmc' instead of command 'check_ltlspec_bmc_onepb' because we wanted make `NuSMV2` iterating length from 0 to 12, trying to find a counterexample for several problem lengths.

6.3 Checking of invariant properties

Now let us start checking for a new kind of properties: invariants. An invariant is a propositional property which must always hold. BMC tries to prove the truth of invariants via an 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.

To check invariants with BMC you can use the check_invar_bmc command. Consider for example the formula 'y in (0..12)': we want to prove it is an invariant property by using NuSMV2 BMC:

```
NuSMV > check_invar_bmc -p "y in (0..12)"
-- cannot prove the invariant y in (0 .. 12) : the induction fails
-- as demonstrated by the following execution sequence
State 1.1: y = 12
State 1.2: y = 13

NuSMV >
```

NuSMV2 answered the given invariant cannot be proved, and it showed a state satisfying "y in (0..12)" having a successor state not satisfying "y in (0..12)". This two-steps sequence of assignments shows why the induction fails. Note that NuSMV2 did not say the given formula is really false, because the induction step can establish only if a given formula is actually true, but cannot be sure if a formula it found out to be false is really logically false. BMC-based invariants checking is not always able to provide a precise answer as demonstrated by this case.

Then we try to prove a stronger invariant, saying that 'y in (0..7)':

```
NuSMV > check_invar_bmc -p "y in (0..7)"
-- invariant y in (0 .. 7)   is true
NuSMV >
```

In this case NuSMV2 is able to rove that "y in (0..7)" is true. Indeed, sometimes it is easier to prove a stronger invariant.

Now we check a false invariant:

```
NuSMV > check_invar_bmc -p "y in (0..6)"
-- cannot prove the invariant y in (0 .. 6) : the induction fails
-- as demonstrated by the following execution sequence
State 1.1: y = 6
State 1.2: y = 7
NuSMV >
```

As for property "y in (0..12)" NuSMV2 returns a two steps sequence showing that the induction fails. The main difference is that in the former case the state 'y=12' is NOT reachable, while in the latter case the state 'y=6' can be reached. The BMC-based invariant checker does not take into account reachability. To resolve ambiguous cases you can use the BDD-based invariant checker, which is based on reachability, to obtain a more satisfying result:

```
NuSMV > go
NuSMV > check_invar -p "y in (0..12)"
-- invariant y in (0 .. 12)  is true

NuSMV > check_invar -p "y in (0..6)"
-- invariant y in (0 .. 6)  is false
-- as demonstrated by the following execution sequence
```

```
State 2.1: y = 0
State 2.2: y = 1
State 2.3: y = 2
State 2.4: y = 3
State 2.5: y = 4
State 2.6: y = 5
State 2.7: y = 6
State 2.8: y = 7
NuSMV >
```

Sometimes however SAT-based model checking can tackle models that are out of reach for BDD-based approaches.

6.4 Further information

In this tutorial we have been using essentially three BMC commands (check_ltlspec_bmc, check_ltlspec_bmc_onepb and check_invar_bmc) and ignored several others. In general, all the commands can handle a large number of options which have not been discussed in this tutorial. Please refer to the detailed command descriptions for further information.

Each of the described commands is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. After the problem is generated, `NuSMV2` internally calls the SIM propositional solver in order to find an assignment which validates the problem. It is also possible to generate the problem without calling to SIM solver. Each generated problem is dumped in DIMACS format into a file. DIMACS is the standard format used as input by most external SAT solver, so it is possible to use `NuSMV2` to generate problem only, and then invoke an external SAT solver separately. The commands which generate DIMACS problem are: gen_ltlspec_bmc, gen_ltlspec_bmc_onepb and gen_invar_bmc. The default DIMACS filename is held by the `bmc_dimacs_filename` and `bmc_invar_dimacs_filename` variables. DIMACS filenames can contain some particular symbols which can be automatically expanded by `NuSMV2`. Please refer to the detailed commands descriptions for further information about pattern expansion. It is also possible to specify the filename via the command line of the command being invoked. You can also use the commands check_ltlspec_bmc, check_ltlspec_bmc_onepb and check_invar_bmc to dump DIMACS problem and then check via the '-o' option. If this option is not activated, problem dumping will not be performed.

6.5 BMC commands details

6.5.1 Relevant environment variables

bmc_length Environment Variable

Sets the generated problem length. Possible values are any natural number, but must be compatible with the current value held by the variable *bmc_loopback*. The default value is `10`.

bmc_loopback Environment Variable

Sets the generated problem loop. Possible values are:

- Any natural number, but less or equal than the current value of the variable *bmc_length*. In this case the loop point is absolute.
- Any negative number, but greater or equal than -*bmc_length*. In this case specified loop is the loop length.

- The symbol 'X', which means "no loopback".
- The symbol '*', which means "any possible loopbacks".

The default value is `*`.

bmc_dimacs_filename Environment Variable

This is the default file name used when generating DIMACS problem dumps. This variable may be taken into account by all commands which belong to the gen_ltlspec_bmc family. DIMACS file name can contain special symbols which will be expanded to represent the actual file name. Possible symbols are:

- **@F** The currently loaded model name with full path.
- **@f** The currently loaded model name without full path.
- **@n** The numerical index of the currently processed formula in the properties database.
- **@k** The currently generated problem length.
- **@l** The currently generated problem loopback value.

The default value is "`@f_k@k_l@l_n@n.dimacs`".

bmc_invar_dimacs_filename Environment Variable

This is the default file name used when generating DIMACS invar dumps. This variable may be taken into account by the command gen_invar_bmc. DIMACS file name can contain special symbols which will be expanded to represent the actual file name. Possible symbols are:

- **@F** The currently loaded model name with full path.
- **@f** The currently loaded model name without full path.
- **@n** The numerical index of the currently processed formula in the properties database.

The default value is "`@f_invar_n@n.dimacs`".

6.5.2 BMC commands for checking of LTL specifications

check_ltlspec_bmc - *Checks the given LTL specification, or all LTL specifications if no formula is given. Checking parameters are the maximum length and the loopback values* Command

```
check_ltlspec_bmc [-h | -n idx | -p "formula" [IN context]] [-k max_length]
[-l loopback] [-o filename]
```

This command generates one or more problems, and calls SAT solver for each one. Each problem is related to a specific problem length, which increases from zero (0) to the given maximum problem length. Here "*length*" is the length of the problem that system is going to generate and/or solve.
In this context the maximum problem length is represented by the *-k* command parameter, or by its default value stored in the environment variable *bmc_length*.
The single generated problem also depends on the "*loopback*" parameter you can explicitly specify by the *-l* option, or by its default value stored in the environment variable *bmc_loopback*.
The property to be checked may be specified using the *-n idx* or the *-p "formula"* options.
If you need to generate a dimacs dump file of all generated problems, you must use the option *-o "filename"*.

Command options:

`-n` *index*
: *index* is the numeric index of a valid LTL specification formula actually located in the properties database.

`-p "formula" [IN context]`
: Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-k` *max_length*
: *max_length* is the maximum problem length used . Only natural number are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

`-l` *loopback*
: *loopback* value may be:
- a natural number in (0, *max_length*). Positive sign ('`+`') can be also used as prefix of the number. Invalid combination of length and loopback will be skipped during the generation/solving process.
- a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Invalid combination of length and loopback will be skipped during the generation/solving process.
- the symbol 'X', which means "no loopback"
- the symbol '*', which means "all possible loopback from zero to *length*"

`-o` *filename*
: *filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- `@`F: model name with path part
- `@`f: model name without path part
- `@`k: current problem length
- `@`l: current loopback value
- `@`n: index of the currently processed formula in the properties database
- `@@`: the '`@`' character

check_ltlspec_bmc_onepb - *Checks the given LTL specification, or all LTL specifications if no formula is given. Checking parameters are the single problem length and the loopback values* Command

`check_ltlspec_bmc_onepb [-h | -n idx | -p "formula" [IN context]] [-k length] [-l loopback] [-o filename]`

As command check_ltlspec_bmc but it produces only one single problem with fixed length and loopback values, with no iteration of the problem length from zero to max_length.

Command options:

`-n` *index*
: *index* is the numeric index of a valid LTL specification formula actually located in the properties database.
The validity of *index* value is checked out by the system.

`-p "formula [IN context]"`
: Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-k` *length*

length is the problem length used when generating the single problem. Only natural number are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

`-l` *loopback*

loopback value may be:
- a natural number in (0, *max_length*). Positive sign ('`+`') can be also used as prefix of the number. Invalid combination of length and loopback will be skipped during the generation/solving process.
- a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *length*. Invalid combination of length and loopback will be skipped during the generation/solving process.
- the symbol 'X', which means "no loopback"
- the symbol '*', which means "all possible loopback from zero to *length*"

`-o` *filename*

filename is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- `@`F: model name with path part
- `@`f: model name without path part
- `@`k: current problem length
- `@`l: current loopback value
- `@`n: index of the currently processed formula in the properties database
- `@@`: the '`@`' character

DIMACS files generation, BMC commands for checking of LTL specifications, Getting started with BMC

6.5.3 BMC commands for checking of invariants

check_invar_bmc - *Generates and solve the given invariant, or all invariants if no formula is given* Command

`check_invar_bmc [-h | -n idx | -p "formula" [IN context]] [-o filename]`

Command options:

`-n` *index*

index is the numeric index of a valid INVAR specification formula actually located in the properties database.
The validity of *index* value is checked out by the system.

`-p "formula [IN context]"`

Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-o` *filename*

filename is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- `@`F: model name with path part
- `@`f: model name without path part
- `@`n: index of the currently processed formula in the properties database
- `@@`: the '`@`' character

6.5.4 BMC commands for DIMACS files generation

gen_ltlspec_bmc - *Dumps into one or more dimacs files the given LTL specification, or all LTL specifications if no formula is given. Generation and dumping parameters are the maximum length and the loopback values* Command

`gen_ltlspec_bmc [-h | -n idx | -p "formula" [IN context]] [-k max_length] [-l loopback] [-o filename]`

This command generates one or more problems, and dumps each problem into a dimacs file. Each problem is related to a specific problem length, which increases from zero (0) to the given maximum problem length. In this short description "*length*" is the length of the problem that system is going to dump out.
In this context the maximum problem length is represented by the *max_length* parameter, or by its default value stored in the environment variable *bmc_length*.
Each dumped problem also depends on the loopback you can explicitly specify by the *-l* option, or by its default value stored in the environment variable *bmc_loopback*.
The property to be checked may be specified using the *-n idx* or the *-p "formula"* options.
You may specify dimacs file name by using the option *-o "filename"*, otherwise the default value stored in the environment variable *bmc_dimacs_filename* will be considered.

Command options:

`-n` *index*
: *index* is the numeric index of a valid LTL specification formula actually located in the properties database.
The validity of *index* value is checked out by the system.

`-p "formula [IN context]"`
: Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-k` *max_length*
: *max_length* is the maximum problem length used when increasing problem length starting from zero. Only natural number are valid values for this option. If no value is given the environment variable *bmc_length* value is considered instead.

`-l` *loopback*
: *loopback* value may be:
- a natural number in (0, *max_length*). Positive sign ('+') can be also used as prefix of the number. Invalid combination of length and loopback will be skipped during the generation and dumping process.
- a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Invalid combination of length and loopback will be skipped during the generation process.
- the symbol 'X', which means "no loopback"
- the symbol '*', which means "all possible loopback from zero to *length*"

`-o` *filename*
: *filename* is the name of dumped dimacs files. If this options is not specified, variable *bmc_dimacs_filename* will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- `@F`: model name with path part
- `@f`: model name without path part

- @k: current problem length
- @l: current loopback value
- @n: index of the currently processed formula in the properties database
- @@: the '@' character

gen_ltlspec_bmc_onepb - *Dumps into one dimacs file the problem generated for the given LTL specification, or for all LTL specifications if no formula is explicitly given. Generation and dumping parameters are the problem length and the loopback values* Command

`gen_ltlspec_bmc_onepb [-h | -n idx | -p "formula" [IN context]] [-k length] [-l loopback] [-o filename]`

As the *gen_ltlspec_bmc* command, but it generates and dumps only one problem given its length and loopback.

Command options:

`-n` *index*
: *index* is the numeric index of a valid LTL specification formula actually located in the properties database.
The validity of *index* value is checked out by the system.

`-p "formula [IN context]"`
: Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-k` *length*
: *length* is the single problem length used to generate and dump it. Only natural number are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

`-l` *loopback*
: *loopback* value may be:
- a natural number in (0, *length*). Positive sign ('+') can be also used as prefix of the number. Invalid combination of length and loopback will be skipped during the generation and dumping process.
- a negative number in (-1, -*length*). Invalid combination of length and loopback will be skipped during the generation process.
- the symbol 'X', which means "no loopback"
- the symbol '*', which means "all possible loopback from zero to *length*"

`-o` *filename*
: *filename* is the name of the dumped dimacs file. If this options is not specified, variable *bmc_dimacs_filename* will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- @F: model name with path part
- @f: model name without path part
- @k: current problem length
- @l: current loopback value
- @n: index of the currently processed formula in the properties database
- @@: the '@' character

gen_invar_bmc - *Generates the given invariant, or all invariants if no formula is given* Command

`gen_invar_bmc [-h | -n idx | -p "formula" [IN context]] [-o filename]`

Command options:

`-n` *index*
: *index* is the numeric index of a valid INVAR specification formula actually located in the properties database.
The validity of *index* value is checked out by the system.

`-p "formula" [IN context]`
: Checks the `formula` specified on the command-line.
`context` is the module instance name which the variables in `formula` must be evaluated in.

`-o` *filename*
: *filename* is the name of the dumped dimacs file. If you do not use this option the dimacs file name is taken from the environment variable *bmc_invar_dimacs_filename*.

File name may contain special symbols which will be macro-expanded to form the real dimacs file name. Possible symbols are:
- `@`F: model name with path part
- `@`f: model name without path part
- `@`n: index of the currently processed formula in the properties database
- `@@`: the '`@`' character

6.5.5 BMC commands for simulation

bmc_simulate - *Generates a trace of the model from 0 (zero) to k* Command

`bmc_simulate [-h]`

bmc_simulate does not require a specification to build the problem, because only the model is used to build it.

Bibliography

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

[CCGR98] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. `NuSMV`: a reimplementation of `SMV`. *Proceeding of the International Workshop on Software Tools for Technology Transfer (STTT-98)*, BRICS Notes Series, NS-98-4, pages 25–31. Also IRST Technical Report 9801-06.

[BCLM+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[CGH97] E. Clarke, O. Grumberg and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):57–71, February 1997.

[CMB90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, Berlin, June 1990. Springer.

[Dil88] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1988.

[EMSS91] E. Allen Emerson, A. K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 136–145, Berlin, Germany, June 1991. Springer.

[McMil92] K.L. McMillan. *The SMV system – DRAFT*. 1992.
Available at `http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps`.

[McMil93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[Mart85] A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs and W.H. Freeman, editors, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260, New York, 1985.

[MOON00] Moon, Hachtel, Somenzi, Border-Block Tringular Form and Conjunction Schedule in Image Computation, to appear at FMCAD00.

[RAP+95] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.

[Som98] F. Somenzi. CUDD: CU Decision Diagram package — release 2.2.0 Department of Electrical and Computer Engineering — University of Colorado at Boulder, May 1998.

[Vis96] "VIS: A system for Verification and Synthesis", The VIS Group, In the Proceedings of the 8th International Conference on Computer Aided Verification, p428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996

Appendix A Compatibility with CMU SMV

NuSMV is mostly source compatible with the original version of SMV distributed at Carnegie Mellon University from which we started. The main problem you will run into is variable names in old programs that conflicts with new reserved words. In Appendix B [New reserved words], page 65 you will find the list of all the new reserved words of NuSMV with respect to CMU SMV.

The IMPLEMENTS, INPUT, OUTPUT statements are all parsed from the input file, but they are internally ignored.

NuSMV differs from CMU SMV also in the kind of formulas accepted as input. The main differences are listed below.

- It is no longer possible to write formulas containing nested 'next'.

```
TRANS
  next(alpha & next(beta | next(gamma))) -> delta
```

- It is no longer possible to write formulas containing 'next' in the right hand side of "normal" and "init" assignments (they are allowed in the right hand side of "next" assignments), and with the statements 'INVAR' and 'INIT'.

```
INVAR
  next(alpha) & beta
INIT
  next(beta) -> alpha
ASSIGN
  delta := alpha & next(gamma);       -- normal assignments
  init(gamma) := alpha & next(delta); -- init assignments
```

- It is no longer possible to write 'SPEC', 'FAIRNESS' statements containing 'next'.

```
FAIRNESS
 next(running)
SPEC
 next(x) & y
```

- The check for circular dependencies among variables has been restricted.

  We say that variable x depends on variable y if $x := f(y)$. We say that there is a circular dependency in the definition of x if:

  - x depends on itself (e.g. $x := f(x,y)$);
  - x depends on y and y depends on x (e.g. $x := f(y)$ and $y := f(x)$ or $x := f(z)$, $z := f(y)$ and $y := f(x)$).

  In the case of circular dependencies among variables there is no fixed order in which we can compute the involved variables. Avoiding circular dependencies among variables guarantee that there exists an order in which the variables can be computed. In NuSMV circular dependencies are not allowed.

  In CMU SMV the test for circular dependencies is able to detect circular dependencies only in "normal" assignments, and not in "next" assignments. The circular dependencies check of NuSMV has been extended to detect circularities also in "next" assignments. For instance the following fragment of code is accepted by CMU SMV but discarded by NuSMV.

```
MODULE main
VAR
  y : boolean;
  x : boolean;
ASSIGN
  next(x) := x & next(y);
  next(y) := y & next(x);
```

In order to have full control on the ordering of variables the variable used to encode processes has been made explicit (in CMU SMV it is not possible to control the position in the ordering of such a variable. It is hardcoded at the top of the ordering.). Its name is `_process_selector_`.

Another difference between NuSMV and CMU SMV is in the variable order file. The variable ordering file accepted by NuSMV can be partial and can contain variables not declared in the model. Variables listed in the ordering file but not declared in the model are simply discarded. The variables declared in the model but not listed in the variable file provided in input are created at the end of the given ordering following the default ordering. All the ordering files generated by CMU SMV are accepted in input from NuSMV but the ordering files generated by NuSMV are not accepted in input to CMU SMV for free (it is necessary to remove the `_process_selector_` variable. Notice that there is no guarantee that such ordering is a good ordering for CMU SMV.).

All the NuSMV programs that do not contain keywords regarding new features (such us LTL model checking, invariant checking, etc.) are accepted by CMU SMV.

With respect to the CMU SMV version from which we started (SMV 2.4.4), we treat the invariant defined via the 'INVAR' statement.

Appendix B New reserved words

The list of the new reserved words of `NuSMV` w.r.t. CMU `SMV` is the following:

`F` It is the *sometimes* temporal operator. It is used to write LTL formulas.

`G` It is the *always* temporal operator. It is used to write LTL formulas.

`X` It is the *next time* temporal operator. It is used to write LTL formulas.

`LTLSPEC` It is used to introduce LTL specifications.

`INVARSPEC`
It is used to introduce invariant specifications.

`IVAR` It is used to introduce input variables.

Command Index

Variable Index

Index

I

L

M

N

O

P

R

S

T

V