NuSMV 2.3 User Manual

Roberto Cavada, Alessandro Cimatti,
Emanuele Olivetti, Gavin Keighren,
Marco Pistore, Marco Roveri,
Simone Semprini and Andrey Tchaltsev

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`.

Contents

Chapter 1

Introduction

NuSMV is a symbolic model checker originated from the reengineering, reimplementation and extension of CMU SMV, the original BDD-based model checker developed at CMU [McM93]. The NuSMV project aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer projects: it is a well structured, open, flexible and documented platform for model checking, and is robust and close to industrial systems standards [CCGR00].

Version 1 of NuSMV basically implements BDD-based symbolic model checking. Version 2 of NuSMV (NuSMV2 in the following) inherits all the functionalities of the previous version, and extend them in several directions [CCG+02]. The main novelty in NuSMV2 is the integration of model checking techniques based on propositional satisfiability (SAT) [BCCZ99]. SAT-based model checking is currently enjoying a substantial success in several industrial fields, and opens up new research directions. BDD-based and SAT-based model checking are often able to solve different classes of problems, and can therefore be seen as complementary techniques.

Starting from NuSMV2, we are also adopting a new development and license model. NuSMV2 is distributed with an OpenSource license[1], that allows anyone interested to freely use the tool and to participate in its development. The aim of the NuSMV OpenSource project is to provide to the model checking community a common platform for the research, the implementation, and the comparison of new symbolic model checking techniques. Since the release of NuSMV2, the NuSMV team has received code contributions for different parts of the system. Several research institutes and commercial companies have express interest in collaborating to the development of NuSMV. The main features of NuSMV are the following:

- **Functionalities.** NuSMV allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual interface, as well as in batch mode.

- **Architecture.** A software architecture has been defined. The different components and functionalities of NuSMV have been isolated and separated in mod-

[1](see `http://www.opensource.org`)

ules. Interfaces between modules have been provided. This reduces 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, and provides a general interface for linking with state-of the-art SAT solvers. This makes NuSMV very robust, portable, efficient, and easy to understand by other people than the developers.

This document is structured as follows.

- In Chapter 2 [Syntax], page 6 we define the syntax of the input language of NuSMV.
- In Chapter 3 [Running NuSMV interactively], page 25 the commands of the interaction shell are described.
- In Chapter 4 [Running NuSMV batch], page 68 we define the batch mode of NuSMV.

NuSMV is available at `http://nusmv.irst.itc.it`.

Chapter 2

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.

2.1 Expressions

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

2.1.1 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 1
        | "FALSE"                       ;; The boolean constant 0
        | 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 "xor" simple_expr ;; logical exclusive or
        | simple_expr "->" simple_expr  ;; logical implication
        | simple_expr "<->" simple_expr ;; logical equivalence
```

```
          | simple_expr "=" simple_expr    ;; equality
          | simple_expr "!=" simple_expr   ;; inequality
          | 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 2.2.10 [Identifiers], page 13) 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
=,!=,<,>,<=,>=
|,xor
<->
->
```

Operators of equal precedence associate to the left, except `->` that associates to the right. Parentheses may be used to group expressions.

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`.

Set Expressions

A set expression has the following syntax:

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

A set can be defined by enumerating its elements inside curly braces '{...}'. 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.

2.1.2 Next Expressions

While simple expressions can represent sets of states, next expressions relate current and next state variables to express transitions in the FSM. The structure of next expressions is similar to the structure of simple expressions (See Section 2.1.1 [simple expressions], page 6). 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 1
        | "FALSE"                     ;; The boolean constant 0
        | 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 "xor" next_expr   ;; logical exclusive or
        | next_expr "->" next_expr    ;; logical implication
        | next_expr "<->" next_expr   ;; logical equivalence
        | next_expr "=" next_expr     ;; equality
        | next_expr "!=" next_expr    ;; inequality
        | 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 2.1.1 [set expressions], page 7) and `case_simple_expr` (see Section 2.1.1 [case expressions], page 7) 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.

2.2 Definition of the FSM

2.2.1 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).

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 2.2.9 [MODULE declarations], page 12). The keyword causes the module to be instantiated as an asynchronous process (see Section 2.2.12 [processes], page 14).

2.2.2 Input Variables

IVAR (input variables) are used to label transitions of the Finite State Machine. The syntax for the declaration of input variables is the following:

```
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 2.2.1 [state variables], page 9).

2.2.3 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. It is also an error for a variable to be assigned more than once at any given time. More precisely, it is an error if any of the following occur:

- the next or current value of a variable is assigned more than once in a given process
- the initial value of a variable is assigned more than once in the program
- the current value and the initial value of a variable are both assigned in the program
- the current value and the next value of a variable are both assigned in the program

2.2.4 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.

2.2.5 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.

2.2.6 INVAR declarations

The set of invariant states (i.e. the analogous of normal assignments, as described in Section 2.2.3 [ASSIGN declarations], page 10) can be specified using a boolean expression under the '`INVAR`' keyword. The syntax of a `INVAR` declaration is:

```
invar_declaration :: "INVAR " invar_expr [";"]
invar_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 `INVAR` declaration, the invariant set is the conjunction of all of the `INVAR` declarations.

2.2.7 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 (see Section 2.2.9 [MODULE declarations], page 12 for an explanation of contexts). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

It is not possible to assign values to defined symbols non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not.

2.2.8 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.

2.2.9 **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 2.2.1 [state variables], page 9). 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.

2.2.10 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
    e2.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`.

2.2.11 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.

2.2.12 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword '`process`' (see Section 2.2.1 [state variables], page 9). 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.

2.2.13 `FAIRNESS` declarations

A fairness constraint restricts the attention only to *fair execution paths*. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths.

NuSMV supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula `f` which is assumed to be true infinitely often in all the fair paths. In NuSMV justice constraints are identified by keywords `JUSTICE` and, for backward compatibility, `FAIRNESS`. A compassion constraint consists of a pair of formulas `(p,q)`; if property `p` is true infinitely often in a fair path, then also formula `q` has to be true infinitely often in the fair path. In NuSMV compassion constraints are identified by keyword `COMPASSION`.[1] If compassion constraints are used then the model must not contain any input variables. Currently, NuSMV does not enforce this so it is the responsibility of the user to make sure that this is the case.

Fairness constraints are declared using the following syntax:

```
fairness_declaration ::
       "FAIRNESS " simple_expr [";"]
     | "JUSTICE " simple_expr [";"]
     | "COMPASSION " "(" simple_expr "," simple_expr ")" [";"]
```

A path is considered fair if and only if it satisfies all the constraints declared in this manner.

2.3 Specifications

The specifications to be checked on the FSM can be expressed in two different temporal logics: the Computation Tree Logic CTL, and the Linear Temporal Logic LTL extended with Past Operators. It is also possible to analyze quantitative characteristics of the FSM by specifying real-time CTL specifications. Specifications can be positioned within modules, in which case they are preprocessed to rename the variables according to the containing context.

CTL and LTL specifications are evaluated by NuSMV in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, NuSMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property.

2.3.1 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
```

[1]In the current version of NuSMV, compassion constraints are supported only for BDD-based LTL model checking. We plan to add support for compassion constraints also for CTL specifications and in Bounded Model Checking in the next releases of NuSMV.

```
      | ctl_expr "xor" ctl_expr   ;; logical exclusive 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`.

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.

2.3.2 LTL Specifications

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 "xor" ltl_expr ;; logical exclusive or
     | ltl_expr "->" ltl_expr  ;; logical implies
     | ltl_expr "<->" ltl_expr ;; logical equivalence
    ;; FUTURE
     | "X" ltl_expr            ;; next state
     | "G" ltl_expr            ;; globally
     | "F" ltl_expr            ;; finally
     | ltl_expr "U" ltl_expr   ;; until
     | ltl_expr "V" ltl_expr   ;; releases
    ;; PAST
     | "Y" ltl_expr            ;; previous state
     | "Z" ltl_expr            ;; not previous state not
     | "H" ltl_expr            ;; historically
     | "O" ltl_expr            ;; once
     | ltl_expr "S" ltl_expr   ;; since
     | ltl_expr "T" ltl_expr   ;; triggered
```

In NuSMV, LTL specifications can be analyzed both by means of BDD-based reasoning, or by means of SAT-based bounded model checking. In the first case, NuSMV proceeds along the lines described in [CGH97]. For each LTL specification, a tableau able to recognize the behaviors falsifying the property is constructed, and then synchronously composed with the model. With respect to [CGH97], the approach is fully integrated within NuSMV, and allows for full treatment of past temporal operators. In the case of BDD-based reasoning, the counterexample generated to show the falsity of a LTL specification may contain state variables which have been introduced by the tableau construction procedure. In the second case, a similar tableau construction is carried out to encode the existence of a path of limited length violating the property. NuSMV generates a propositional satisfiability problem, that is then tackled by means of an efficient SAT solver [BCCZ99]. In both cases, the tableau constructions are completely transparent to the user.

2.3.3 Real Time CTL Specifications and Computations

NuSMV allows for Real Time CTL specifications [EMSS91]. 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, the semantics of the RTCTL operators is as follows:

- `EBF` *m*`..`*n* *p* requires that there exists a path starting from a state, such that property *p* holds in a future time instant *i*, with $m \leq i \leq n$
- `ABF` *m*`..`*n* *p* requires that for all paths starting from a state, property *p* holds in a future time instant *i*, with $m \leq i \leq n$
- `EBG` *m*`..`*n* *p* requires that there exists a path starting from a state, such that property *p* holds in all future time instants *i*, with $m \leq i \leq n$
- `ABG` *m*`..`*n* *p* requires that for all paths starting from a state, property *p* holds in all future time instants *i*, with $m \leq i \leq n$
- `E [` *p* `BU` *m*`..`*n* *q* `]` requires that there exists a path starting from a state, such that property *q* holds in a future time instant *i*, with $m \leq i \leq n$, and property *p* holds in all future time instants *j*, with $m \leq j < i$
- `A [` *p* `BU` *m*`..`*n* *q* `]`, requires that for all paths starting from a state, property *q* holds in a future time instant *i*, with $m \leq i \leq n$, and property *p* holds in all future time instants *j*, with $m \leq j < i$

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 FSM. 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.

2.3.4 PSL Specifications

NuSMV allows for PSL specifications as from version 1.01 of PSL Language Reference Manual [psl03]. PSL specifications are introduced by the keyword “PSLSPEC”. The syntax of this declaration (as from the PSL parser distributed by IBM, [PSL]) is:

```
pslspec_declaration :: "PSLSPEC " psl_expr ";"
```

where

```
psl_expr ::
   psl_primary_expr
 | psl_unary_expr
 | psl_binary_expr
 | psl_conditional_expr
 | psl_case_expr
 | psl_property
```

The six classes above define the building blocks for `psl_property` and provide means of combining instances of that class; they are defined as follows:

```
psl_primary_expr ::
   number                              ;; a numeric constant
 | boolean                             ;; a boolean constant
 | var_id                              ;; a variable identifier
 | "{" psl_expr "," ... "," psl_expr "}"
 | "{" psl_expr "{" psl_expr "," ... "," "psl_expr" "}" "}"
 | "(" psl_expr ")"
psl_unary_expr ::
   "+" psl_primary_expr
 | "-" psl_primary_expr
 | "!" psl_primary_expr
psl_binary_expr ::
   psl_expr "+" psl_expr
 | psl_expr "union" psl_expr
```

```
 | psl_expr "in" psl_expr
 | psl_expr "-" psl_expr
 | psl_expr "*" psl_expr
 | psl_expr "/" psl_expr
 | psl_expr "%" psl_expr
 | psl_expr "==" psl_expr
 | psl_expr "!=" psl_expr
 | psl_expr "<" psl_expr
 | psl_expr "<=" psl_expr
 | psl_expr ">" psl_expr
 | psl_expr ">=" psl_expr
 | psl_expr "&" psl_expr
 | psl_expr "|" psl_expr
 | psl_expr "xor" psl_expr
psl_conditional_expr ::
 psl_expr "?" psl_expr ":" psl_expr
psl_case_expr ::
 "case"
     psl_expr ":" psl_expr ";"
     ...
     psl_expr ":" psl_expr ";"
 "esac"
```

Among the subclasses of `psl_expr` we depict the class `psl_bexpr` that will be used in the following to identify purely propositional, i.e. not temporal, expressions. The class of PSL properties `psl_property` is defined as follows:

```
psl_property ::
   replicator psl_expr                 ;; a replicated property
 | FL_property "abort" psl_bexpr
 | psl_expr "<->" psl_expr
 | psl_expr "->" psl_expr
 | FL_property
 | OBE_property
replicator ::
   "forall" var_id [index_range] "in" value_set ":"
index_range ::
   "[" range "]"
range ::
   low_bound ":" high_bound
low_bound ::
   number
 | identifier
high_bound ::
   number
 | identifier
 | "inf"                               ;; infite high bound
value_set ::
   "{" value_range "," ... "," value_range "}"
 | "boolean"
value_range ::
   psl_expr
 | range
```

The instances of `FL_property` are temporal properties built using LTL operators and SEREs operators, and are defined as follows:

```
FL_property ::
 ;; PRIMITIVE LTL OPERATORS
   "X" FL_property
 | "X!" FL_property
 | "F" FL_property
 | "G" FL_property
 | "[" FL_property "U" FL_property "]"
 | "[" FL_property "W" FL_property "]"
 ;; SIMPLE TEMPORAL OPERATORS
 | "always" FL_property
 | "never" FL_property
 | "next" FL_property
 | "next!" FL_property
 | "eventually!" FL_property
 | FL_property "until!" FL_property
 | FL_property "until" FL_property
 | FL_property "until!_" FL_property
 | FL_property "until_" FL_property
 | FL_property "before!" FL_property
 | FL_property "before" FL_property
 | FL_property "before!_" FL_property
 | FL_property "before_" FL_property
 ;; EXTENDED NEXT OPERATORS
 | "X" [number] "(" FL_property ")"
 | "X!" [number] "(" FL_property ")"
 | "next" [number] "(" FL_property ")"
 | "next!" [number] "(" FL_property ")"
 ;;
 | "next_a" [range] "(" FL_property ")"
 | "next_a!" [range] "(" FL_property ")"
 | "next_e" [range] "(" FL_property ")"
 | "next_e!" [range] "(" FL_property ")"
 ;;
 | "next_event!" "(" psl_bexpr ")" "(" FL_property ")"
 | "next_event" "(" psl_bexpr ")" "(" FL_property ")"
 | "next_event!" "(" psl_bexpr ")" "[" number "]"  "(" FL_property ")"
 | "next_event" "(" psl_bexpr ")" "[" number "]"  "(" FL_property ")"
 ;;
 | "next_event_a!" "(" psl_bexpr ")" "["psl_expr"]"  "(" FL_property ")"
 | "next_event_a" "(" psl_bexpr ")" "["psl_expr"]"  "(" FL_property ")"
 | "next_event_e!" "(" psl_bexpr ")" "["psl_expr"]"  "(" FL_property ")"
 | "next_event_e" "(" psl_bexpr ")" "["psl_expr"]"  "(" FL_property ")"
 ;; OPERATORS ON SEREs
 | sequence "(" FL_property ")"
 | sequence "|->" sequence ["!"]
 | sequence "|=>" sequence ["!"]
 ;;
 | "always" sequence
 | "G" sequence
 | "never" sequence
 | "eventually!" sequence
```

```
 ;;
 | "within!" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
 | "within" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
 | "within!_" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
 | "within_" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
 ;;
 | "whilenot!" "(" psl_bexpr ")" sequence
 | "whilenot" "(" psl_bexpr ")" sequence
 | "whilenot!_" "(" psl_bexpr ")" sequence
 | "whilenot_" "(" psl_bexpr ")" sequence
sequence_or_psl_bexpr ::
   sequence
 | psl_bexpr
```

Sequences, i.e. istances of class `sequence`, are defined as follows:

```
sequence ::
   "{" SERE "}"
SERE ::
   sequence
 | psl_bexpr
 ;; COMPOSITION OPERATORS
 | SERE ";" SERE
 | SERE ":" SERE
 | SERE "&" SERE
 | SERE "&&" SERE
 | SERE "|" SERE
 ;; RegExp QUALIFIERS
 | SERE "[*" [count] "]"
 | "[*" [count] "]"
 | SERE "[+]"
 | "[+]"
 ;;
 | psl_bexpr "[=" count "]"
 | psl_bexpr "[->" count "]"
count ::
   number
 | range
```

Istances of `OBE_property` are CTL properties in the PSL style and are defined as follows:

```
OBE_property ::
  "AX" OBE_property
 | "AG" OBE_property
 | "AF" OBE_property
 | "A" "[" OBE_property "U" OBE_property "]"
 | "EX" OBE_property
 | "EG" OBE_property
 | "EF" OBE_property
 | "E" "[" OBE_property "U" OBE_property "]"
```

The NuSMV parser allows to input any specification based on the grammar above, but, currently verification of PSL specifications is supported only for the OBE subset, and for a subset of PSL for which it is possible to define a translation into LTL. For the specifications that belongs to this subsets, it is possible to apply all the verification techniques that can be applied to LTL and CTL Specifications.

2.4 Variable Order Input

It is possible to specify the order in which variables should appear in the BDD's generated by NuSMV. The file which gives the desired order can be read in using the `-i` option in batch mode or by setting the `input_order_file` environment variable in interactive mode.

2.4.1 Input File Syntax

The syntax for input files describing the desired variable ordering is as follows, where the file can be considered as a list of variable names, each of which must be on a separate line:

```
vars_list :: EMPTY
           | var_list_item vars_list

var_list_item :: var_main_id
               | var_main_id.NUMBER

var_main_id :: ATOM
             | var_main_id[NUMBER]
             | var_main_id.var_id

var_id :: ATOM
        | var_id[NUMBER]
        | var_id.var_id
```

That is, a list of variable names of the following forms:

```
Complete_Var_Name        - to specify an ordinary variable
Complete_Var_Name[index] - to specify an array variable element
Complete_Var_Name.NUMBER - to specify a specific bit of an encoded
                           scalar variable
```

where `Complete_Var_Name` is just the name of the variable if it appears in the module `MAIN`, otherwise it has the module name(s) prepended to the start, for example:

```
mod1.mod2...modN.varname
```

where `varname` is a variable in `modN`, and `modN.varname` is a variable in `modN-1`, and so on. Note that the module name `main` is implicitely prepended to every variable name and therefore must not be included in their declarations.
Any variable which appears in the model file, but not the ordering file is placed after all the others in the ordering. Variables which appear in the ordering file but not the model file are ignored. In both cases NuSMV displays a warning message stating these actions.

Comments can be included by using the same syntax as regular NuSMV files. That is, by starting the line with `--`.

2.4.2 Scalar Variables

A variable which has a finite range of values that it can take is encoded as a set of boolean variables. These boolean variables represent the binary equivalents of all the possible values for the scalar variable. Thus, a scalar variable that can take values from 0 to 7 would require three boolean variables to represent it.

It is possible to not only declare the position of a scalar variable in the ordering file, but each of the boolean variables which represent it.
If only the scalar variable itself is named then all the boolean variables which are actually used to encode it are grouped together in the BDD.
Variables which are grouped together will always remain next to each other in the BDD and in the same order. When dynamic variable re-ordering is carried out, the group of variables are treated as one entity and moved as such.
If a scalar variable is omitted from the ordering file then it will be added at the end of the variable order and the specific-bit variables that represent it will be grouped together. However, if any specific-bit variables have been declared in the ordering file (see below) then these will not be grouped with the remaining ones.
It is also possible to specify that specific-bit variables are placed elsewhere in the ordering. This is achieved by first specifying the scalar variable name in the desired location, then simply specifying `Complete_Var_Name.i` at the position where you want that bit variable to appear:

```
...
Complete_Var_Name
...
Complete_Var_Name.i
...
```

The result of doing this is that the variable representing the i^{th} bit is located in a different position to the remainder of the variables representing the rest of the bits. The specific-bit variables *varname.0, ..., varname.i-1, varname.i+1, ..., varname.N* are grouped together as before.

If any one bit occurs before the variable it belongs to, the remaining specific-bit variables are not grouped together:

```
...
Complete_Var_Name.i
...
Complete_Var_Name
...
```

The variable representing the i^{th} bit is located at the position given in the variable ordering and the remainder are located where the scalar variable name is declared. In this case, the remaining bit variables will not be grouped together.
This is just a short-hand way of writing each individual specific-bit variable in the ordering file. The following are equivalent:

```
...                        ...
Complete_Var_Name.0        Complete_Var_Name.0
Complete_Var_Name.1        Complete_Var_Name
:                          ...
Complete_Var_Name.N-1
...
```

where the scalar variable `Complete_Var_Name` requires N boolean variables to encode all the possible values that it may take.

It is still possible to then specify other specific-bit variables at later points in the ordering file as before.

2.4.3 Array Variables

When declaring array variables in the ordering file, each individual element must be specified separately. It is not permitted to specify just the name of the array.

The reason for this is that the actual definition of an array in the model file is essentially a shorthand method of defining a list of variables that all have the same type. Nothing is gained by declaring it as an array over declaring each of the elements individually, and there is no difference in terms of the internal representation of the variables.

Chapter 3

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>
```

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 the 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 on page 67. For instance, it is not possible to evaluate CTL expressions before the model is built.

A number of commands and environment variables, like those dealing with file names, accept arbitrary strings. There are a few reserved characters which must be escaped if they are to be used literally in such situations. See the section describing the `history` command, on page 59, for more information.

The verbosity of NuSMV is controlled by the following environment variable.

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`.

3.1 Model Reading and Building

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

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.

pp_list Environment Variable

Stores the list of pre-processors to be run on the input file before it is parsed by NuSMV. The pre-processors are executed in the order specified by this variable. The argument must either be the empty string (specifying that no pre-processors are to be run on the input file), one single pre-processor name or a space seperated list of pre-processor names inside double quotes. Any invalid names are ignored. The default is none.

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 a 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_dumps_bits Environment Variable

Changes the behaviour of the command `write_order`.

When this variable is set, `write_order` will dump the bits constituting the boolean encoding of each scalar variable, instead of the scalar variable itself. This helps to work at bits level in the variable ordering file. See the command `write_order` for further information. The default value is `0`.

write_order - *Writes variable order to file.* Command `write_order [-h] [-b] [(-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.

By default, the bits constituting the scalar variables encoding are not dumped. When a variable bit should be dumped, the scalar variable which the bit belongs to is dumped instead if not previously dumped. The result is a variable ordering containing only scalar and boolean model variables.

To dump single bits instead of the corresponding scalar variables, either the option `-b` can be specified, or the environment variable `write_order_dumps_bits` must be previously set.

When the boolean variable dumping is enabled, the single bits will occur within the resulting ordering file in the same position that they occur at BDD level.

Command Options:

`-b`	Dumps bits of scalar variables instead of the single scalar variables. See also the variable `write_order_dumps_bits`.
`-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 the `-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 a BDD* — Command

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

Compiles the flattened hierarchy into a 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]) by setting the `iwls95preorder` variable appropriately.

conj_part_threshold Environment Variable

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

affinity Environment Variable

Enables affinity clustering heuristic described in [MHS00], possible values are `0` or `1`. The default value is `1`.

image_cluster_size Environment Variable

One of the parameters to configure the behaviour of the *Iwls95CP* partitioning algorithm. `image_cluster_size` is used as threshold value for the clusters. The default value is `1000` BDD nodes.

image_W{1,2,3,4} Environment Variable

The other parameters for the *Iwls95CP* partitioning algorithm. These attribute different weights to the different factors in the algorithm. The default values are `6`, `1`, `1`, `6` respectively. (For a detailed description, please refer to [RAP+95].)

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_W{1,2,3,4}`.

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.

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.

3.2 Commands for Checking Specifications

The following commands allow for the BDD-based model checking of a NuSMV model.

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] [-v]
```

Prints the number of reachable states of the given model. In verbose mode, prints also the list of all reachable states. The reachable states are computed if needed.

check_fsm - *Checks the transition relation for totality.* — Command

```
check_fsm [-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.

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_fsm — 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`.

print_fair_states - *Prints out the number of fair states* — Command

```
print_fair_states [-h] [-v]
```

Prints the number of fair states of the given model. In verbose mode, prints also the list of all fair states.

print_fair_transitions - *Prints out the number of fair states* Command

`print_fair_transitions [-h] [-v]`

Prints the number of fair transitions of the given model. In verbose mode, prints also the list of all fair transitions. The transitions are displayed as state-input pairs.

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, 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. Given a formula of the form *AG alpha*, the algorithm computes the set of states satisfying *alpha*, and checks whether it contains the set of reachable states. If this is not the case, the formula is proved to be false.

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_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.

3.3 Commands for Bounded Model Checking

In this section we describe in detail the commands for doing and controlling Bounded Model Checking in NuSMV. Bounded Model Checking is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. After the problem is generated, NuSMV internally calls a propositional SAT solver in order to find an assignment which satisfies the problem. Currently NuSMV supplies three SAT solvers: SIM, Zchaff and MiniSat. Notice that Zchaff and MiniSat are for non-commercial purposes only. They are therefore not included in the source code distribution or in some of the binary distributions of NuSMV.

Some commands for Bounded Model Checking use incremental algorithms. These algorithms exploit the fact that satisfiability problems generated for a particular bounded model checking problem often share common subparts. So information obtained during solving of one satisfiability problem can be used in solving of another one. The incremental algorithms usually run quicker then non-incremental ones but require a SAT solver with incremental interface. At the moment, only Zchaff and MiniSat offer such an interface. If none of these solvers are linked to NuSMV, then the commands which make use of the incremental algorithms will not be available.

It is also possible to generate the satisfiability problem without calling the SAT solver. Each generated problem is dumped in DIMACS format to a file. DIMACS is the standard format used as input by most SAT solvers, so it is possible to use NuSMV with a separate external SAT solver. At the moment, the DIMACS files can be generated only by commands which do not use incremental algorithms.

bmc_setup - *Builds the model in a Boolean Epression format.* Command

`bmc_setup [-h]`

You must call this command before use any other bmc-related command. Only one call per session is required.

go_bmc - *Initializes the system for the BMC verification.* Command `go_bmc [-h]`

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

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 value* — 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 bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and solve. In this context the maximum problem bound 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 bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any 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*. Any 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 loopbacks from zero to *length-1*” .

`-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 bound. • @l: current loopback value. • @n: index of the currently processed formula in the property 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 bound and the loopback value* — 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 bound and loopback values, with no iteration of the problem bound 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 bound used when generating the single problem. Only natural numbers are valid values for this option. If no value is given the environment variable `bmc_length` is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign (’+’) can be also used as prefix of the number. Any 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*. Any 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-1*".
`-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 bound. • @l: current loopback value. • @n: index of the currently processed formula in the property database. • @@: the '@' character.

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 bound and the loopback value* 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 bound, which increases from zero (0) to the given maximum problem bound. In this short description `length` is the bound of the problem that system is going to dump out.

In this context the maximum problem bound 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 bound used when increasing problem bound starting from zero. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* value is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of bound 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*. Any invalid combination of bound 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-1*".
`-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 bound. • @l: current loopback value . • @n: index of the currently processed formula in the property 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 bound and the loopback value* — 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 bound 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 bound used to generate and dump it. Only natural numbers are valid values for this option. If no value is given the environment variable `bmc_length` is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation and dumping process. • negative number in (-1, -*length*). Any 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-1*".
`-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 bound • @l: current loopback value • @n: index of the currently processed formula in the property database • @@: the '@' character

check_ltlspec_bmc_inc - *Checks the given LTL specification, or all LTL specifications if no formula is given, using an incremental algorithm. Checking parameters are the maximum length and the loopback value* Command

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

For each problem this command incrementally generates many satisfiability subproblems and calls the SAT solver on each one of them. The incremental algorithm exploits the fact that subproblems have common subparts, so information obtained during a previous call to the SAT solver can be used in the consecutive ones. Logically, this command does the same thing as `check_ltlspec_bmc` (see the description on page 36) but usually runs considerably quicker. A SAT solver with an incremental interface is required by this command, therefore if no such SAT solver is provided then this command will be unavailable.

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 bound must be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any 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*. Any 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-1*" .

bmc_length Environment Variable

Sets the generated problem bound. 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 than the current value of the variable *bmc_length*. In this case the loop point is absolute.
- Any negative number, but greater than or equal to -*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 path part.
- **@n** The numerical index of the currently processed formula in the property database.
- **@k** The currently generated problem length.
- **@l** The currently generated problem loopback value.
- @@ The '@' character.

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

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

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

In Bounded Model Checking, invariants are proved using induction. For this, satisfiability problems for the base and induction step are generated and a SAT solver is invoked on each of them. At the moment, two algorithms can be used to prove invariants. In one algorithm, which we call "classic", the base and induction steps are built on one state and one transition, respectively. Another algorithm, which we call "een-sorensson" [ES04], can build the base and induction steps on many states and transitions. As a result, the second algorithm is more powerful.

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property 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 bound that can be reached. Only natural numbers are valid values for this option. Use this option only if the "een-sorensson" algorithm is selected. If no value is given the environment variable *bmc_length* is considered instead.
`-a` *alg*	*alg* specifies the algorithm. The value can be `classic` or `een-sorensson`. If no value is given the environment variable *bmc_invar_alg* is considered instead.
`-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

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]`

At the moment, the invariants are generated using "classic" algorithm only (see the description of `check_invar_bmc` on page 42).

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property 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

check_invar_bmc_inc - *Generates and solves the given invariant, or all invariants if no formula is given, using incremental algorithms* — Command

`check_invar_bmc_inc [-h | -n idx | -p "formula" [IN context]] [-a algorithm]`

This command does the same thing as `check_invar_bmc` (see the description on page 42) but uses an incremental algorithm and therefore usually runs considerably quicker. The incremental algorithms exploit the fact that satisfiability problems generated for a particular invariant have common subparts, so information obtained during solving of one problem can be used in solving another one. A SAT solver with an incremental interface is required by this command. If no such SAT solver is provided then this command will be unavailable.

There are two incremental algorithms which can be used: “Dual” and “ZigZag”. Both algorithms are equally powerful, but may show different performance depending on a SAT solver used and an invariant being proved. At the moment, the “Dual” algorithm cannot be used if there are input variables in a given model. For additional information about algorithms, consider [ES04].

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property 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 bound that can be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-a` *alg*	*alg* specifies the algorithm to use. The value can be `dual` or `zigzag`. If no value is given the environment variable *bmc_inc_invar_alg* is considered instead.

bmc_invar_alg — Environment Variable

Sets the default algorithm used by the command `check_invar_bmc`. Possible values are `classic` and `een-sorensson`. The default value is `classic`.

bmc_inc_invar_alg — Environment Variable

Sets the default algorithm used by the command `check_invar_bmc_inc`. Possible values are `dual` and `zigzag`. The default value is `dual`.

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 path part.
- **@n** The numerical index of the currently processed formula in the properties database.
- **@@** The ‘@’ character.

The default value is “`@f_invar_n@n`”.

sat_solver — Environment Variable

The SAT solver's name actually to be used. Default SAT solver is SIM. Depending on the NuSMV configuration, also the Zchaff and MiniSat SAT solvers can be available or not. Notice that Zchaff and MiniSat are for non-commercial purposes only.

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

`bmc_simulate [-h | -k ]`

`bmc_simulate` does not require a specification to build the problem, because only the model is used to build it. The problem length is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`.

Command Options:

`-k` *length*	*length* is the length of the generated simulation.

3.4 Commands for checking PSL specifications

The following command allow for model checking of PSL specifications.

check_pslspec - *Performs PSL model checking* Command

```
check_pslspec [-h] [-m | -o output-file] [-n number | -p
"psl-expr  [IN context]"] [-b [-i] [-g] [-1] [-k
bmc_lenght] [-l loopback]]
```

Depending on the characteristics of the PSL property and on the options, the commands applies CTL-based model checking, or LTL-based, possibily bounded model checking.

A `psl-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 PSLSPEC formulas in the database are checked. If option `-b` is used, LTL bounded model checking is applied, otherwise bdd-based model checking is applied. For LTL bounded model checking, options `-k` and `-l` can be used to define the maximum problem bound, and the value of the loopback for the single generated problems respectively; their values can be stored in the environment variables *bmc_lenght* and *bmc_loopback*. Single problems can be generated by using option `-1`. By using option `-i` the incremental version of bounded model checking is activated. Bounded model checking problems can be generated and dumped in a file by using option `-g`.

Command Options:

`-m`	Pipes the output generated by the command in processing `PSLSPEC`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 `PSLSPEC` s to the file *output-file*
`-p "psl-expr [IN context]"`	A PSL formula to be checked. `context` is the module instance name which the variables in `psl-expr` must be evaluated in.
`-n number`	Checks the PSL property with index `number` in the property database.
`-b`	Applies SAT-based bounded model checking. The SAT solver to be used will be chosen according to the current value of the system variable `sat_solver`.
`-i`	Applies incremental SAT-bounded model checking if available, i.e. if an incremental SAT solver has been linked to NuSMV. This option can be used only in combination with the option `-b`.
`-g`	Dumps DIMACS version of bounded model checking problem into a file whose name depends on the system variable `bmc_dimacs_filename`. This feature is not allowed in combination of the option `-i`.
`-1`	Generates a single bounded model checking problem with fixed bound and loopback values, it does not iterate incrementing the value of the problem bound.
`-k` *bmc_length*	*bmc_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any 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*. Any 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 loopbacks from zero to *length-1*" If no value is given the environment variable *bmc_loopback* is considered instead..

3.5 Simulation Commands

In this section we describe the commands that allow to simulate a NuSMV specification. See also the section Section 3.6 [Traces], page 49 that describes the commands available for manipulating traces.

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 `" "`.

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.

3.6 Traces

A trace is a sequence of states-inputs pairs corresponding to a possible execution of the model. Each pair contains the inputs that caused the transition to the new state, and the new state itself. The initial state has no such input values defined as it does not depend on the values of any of the inputs. The values of any constants declared in `DEFINE` sections are also part of a trace. If the value of a constant depends only on state variables then it will be treated as if it is a state variable too. If it depends only on input variables then it will be treated as if it is an input variable. If however, a constant depends upon both input and state variables, then it gets displayed in a seperate "combinatorial" section. Since the values of any such constants depend on one or more inputs, the initial state does not contain this section either.

Traces are created by NuSMV when a formula is found to be false; they are also generated as a result of a simulation (Section 3.5 [Simulation Commands], page 47). Each trace has a number, and the states-inputs pairs are numbered within the trace. Trace *n* has states/inputs *n.1, n.2, n.3,* "..." where *n.1* represents the initial state.

3.6.1 Inspecting Traces

The trace inspection commands of NuSMV allow for navigation along the labelled states-inputs pairs of the traces produced. 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_label`

Makes `state_label` the *current state*. This command is used to navigate along 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*.

3.6.2 Displaying Traces

NuSMV comes with three trace plugins (see Section 3.7 [Trace Plugins], page 51) which can be used to display traces in the system. Once a trace has been generated by NuSMV it is printed to `stdout` using the trace explanation plugin which has been set as the current default. The command `show_traces` (see Section 3.5 [Simulation Commands], page 47) can then be used to print out one or more traces using a different trace plugin, as well as allowing for output to a file.

3.6.3 Trace Plugin Commands

The following commands relate to the plugins which are available in NuSMV.

show_plugins - *Shows the available trace explanation plugins* — Command

`show_plugins [-h] [-n plugin-no | -a]`

Command Options:

`-n plugin-no`	Shows the plugin with the index number equal to `plugin-no`.
`-a`	Shows all the available plugins.

Shows the available plugins that can be used to display a trace which has been generated by NuSMV, or that has been loaded with the `read_trace` command. The plugin that is used to read in a trace is also shown. The current default plugin is marked with "`[D]`".

All the available plugins are displayed by default if no command options are given.

default_trace_plugin — Environment Variable

This determines which trace plugin will be used by default when traces that are generated by NuSMV are to be shown. The values that this variable can take depend on which trace plugins are installed. Use the command `show_plugins` to see which ones are available. The default value is `0`.

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

`show_traces [-h] [-v] [-t] [-m | -o output-file] [-p plugin-no] [-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). This option only applies when the Basic Trace Explainer plugin is used to display the trace.
`-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`.
`-p plugin-no`	Uses the specified trace plugin to display the trace.
`trace_number`	The (ordinal) identifier number of the trace to be printed. This must be the last argument of the command. Omitting the trace number causes the most recently generated trace to be printed.

If the XML Format Output plugin is being used to save generated traces to a file with the intent of reading them back in again at a later date, then only one trace should be saved per file. This is because the trace reader does not currently support multiple traces in one file.

read_trace - *Loads a previously saved trace* Command

`read_trace [-h | -i file-name]`

Command Options:

`-i file-name`	Reads in a trace from the specified file. Note that the file must only contain one trace.

Loads a trace which has been previously output to a file with the XML Format Output plugin. The model from which the trace was originally generated must be loaded and built using the command "`go`" first.
Please note that this command is only available on systems that have the Expat XML parser library installed.

3.7 Trace Plugins

NuSMV comes with three plugins which can be used to diaplay a trace that has been generated:

Basic Trace Explainer
States/Variables Table
XML Format Printer

There is also a plugin which can read in any trace which has been output to a file by the XML Format Printer. Note however that this reader is only available on systems that have the Expat XML parser library installed.

Once a trace has been generated it is output to `stdout` using the currently selected plugin. The command `show_traces` can be used to output any previuosly generated, or loaded, trace to a specific file.

3.7.1 Basic Trace Explainer

This plugin prints out each state (the current values of the variables) in the trace, one after the other. The initial state contains all the state variables and their initial values. States are numbered in the following fasion:

```
trace_number.state_number
```

There is the option of printing out the value of every variable in each state, or just those which have changed from the previous one. The one that is used can be chosen by selecting the appropriate trace plugin. The values of any constants which depend on both input and state variables are printed next. It then prints the set of inputs which cause the transition to a new state (if the model contains inputs), before actually printing the new state itself. The set of inputs and the subsequent state have the same number associated to them.

In the case of a looping trace, if the next state to be printed is the same as the last state in the trace, a line is printed stating that this is the point where the loop begins.

With the exception of the initial state, for which no input values are printed, the output syntax for each state is as follows:

```
-> Input: TRACE_NO.STATE_NO <-
     /* for each input var (being printed), i: */
     INPUT_VARi = VALUE
-> State: TRACE_NO.STATE_NO <-
     /* for each state var (being printed), j: */
     STATE_VARj = VALUE
     /* for each combinatorial constant (being printed), k: */
     CONSTANTk = VALUE
```

where `INPUT_VAR`, `STATE_VAR` and `CONSTANT` have the relevant module names prepended to them (seperated by a period) with the exception of the module “`main`” .

The version of this plugin which only prints out those variables whose values have changed is the initial default plugin used by NuSMV.

3.7.2 States/Variables Table

This trace plugin prints out the trace as a table, either with the states on each row, or in each column. The entries along the state axis are:

```
S0 C1 I1 S1 ... Cn In Sn
```

where `S0` is the initial state, and I_i gives the values of the input variables which caused the transition from state S_{i-1} to state S_i. C_i gives the values of any combinatorial constants, where the value depends on the values of the state variables in state S_{i-1} and the values of input variables in state S_i.

The variables in the model are placed along the other axis. Only the values of state variables are displayed in the State row/column, only the values of input variables are displayed in the Input row/column and only the values of combinatorial constants are displayed in the Constants row/column. All remaining cells have '−' displayed.

3.7.3 XML Format Printer

This plugin prints out the trace either to `stdout` or to a specified file using the command `show_traces`. If traces are to be output to a file with the intention of them being loaded again at a later date, then each trace must be saved in a separate file. This is because the XML Reader plugin does not currently support multiple traces per file. The format of a dumped XML trace file is as follows:

```
<?XML_VERSION_STRING?>
<counter-example type=TRACE_TYPE desc=TRACE_DESC>

  /* for each state, i: */
  <node>
    <state id=i>

      /* for each state var, j: */
      <value variable=j>VALUE</value>

    </state>
    <combinatorial id=i+1>

      /* for each combinatorial constant, k: */
      <value variable=k>VALUE</value>

    </combinatorial>
    <input id=i+1>

      /* for each input var, l: */
      <value variable=l>VALUE</value>

    </input>
  </node>

</counter-example>
```

Note that for the last state in the trace, there is no input section in the node tags. This is because the inputs section gives the new input values which cause the transition to the next state in the trace. There is also no combinatorial section as this depends on the values of the inputs and are therefore undefined when there are no inputs.

3.7.4 XML Format Reader

This plugin makes use of the Expat XML parser library and as such can only be used on systems where this library is available. Previously generated traces for a given model can be loaded using this plugin provided that the original model file[1] has been loaded, and built using the command `go`.

When a trace is loaded, it is given the smallest available trace number to identify it. It can then be manipulated in the same way as any generated trace.

3.8 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 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 dynamic 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.

[1] To be exact, $M_1 \subseteq M_2$, where M_1 is the model from which the trace was generated, and M_2 is the currently loaded, and built, model. Note however, that this may mean that the trace is not valid for the model M_2.

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 one 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.
- **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_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.

3.9 Administration Commands

This section describes the administrative commands offered by the interactive shell of NuSMV.

! - *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] [-o filename [-a]] <string>`

Echoes the specified string either to standard output, or to `filename` if the option `-o` is specified.

Command Options:

`-o filename`	Echoes to the specified filename instead of to standard output. If the option `-a` is not specified, the file `filename` will be overwritten if it already exists.
`-a`	Appends the output to the file specified by option `-o`, instead of overwritting it. Use only with the option `-o`.

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 'ín 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.

Command Options:

`%: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.

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.

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:

`<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:

Command Options:

`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`.

See the variable `on_failure_script_quits` for further information.

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:

`<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

3.10 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.

on_failure_script_quits Environment Variable

When a non-fatal error occurs during the interactive mode, the interactive interpreter simply stops the currently executed command, prints the reason of the problem, and prompts for a new command. When set, this variables makes the command interpreter quit when an error occur, and then quit NuSMV. This behaviour might be useful when the command `source` is controlled by either a system pipe or a shell script. Under these conditions a mistake within the script interpreted by `source` or any unexpected error might hang the controlling script or pipe, as by default the interpreter would simply give up the current execution, and wait for further commands. The default value of this environment variable is `0`.

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

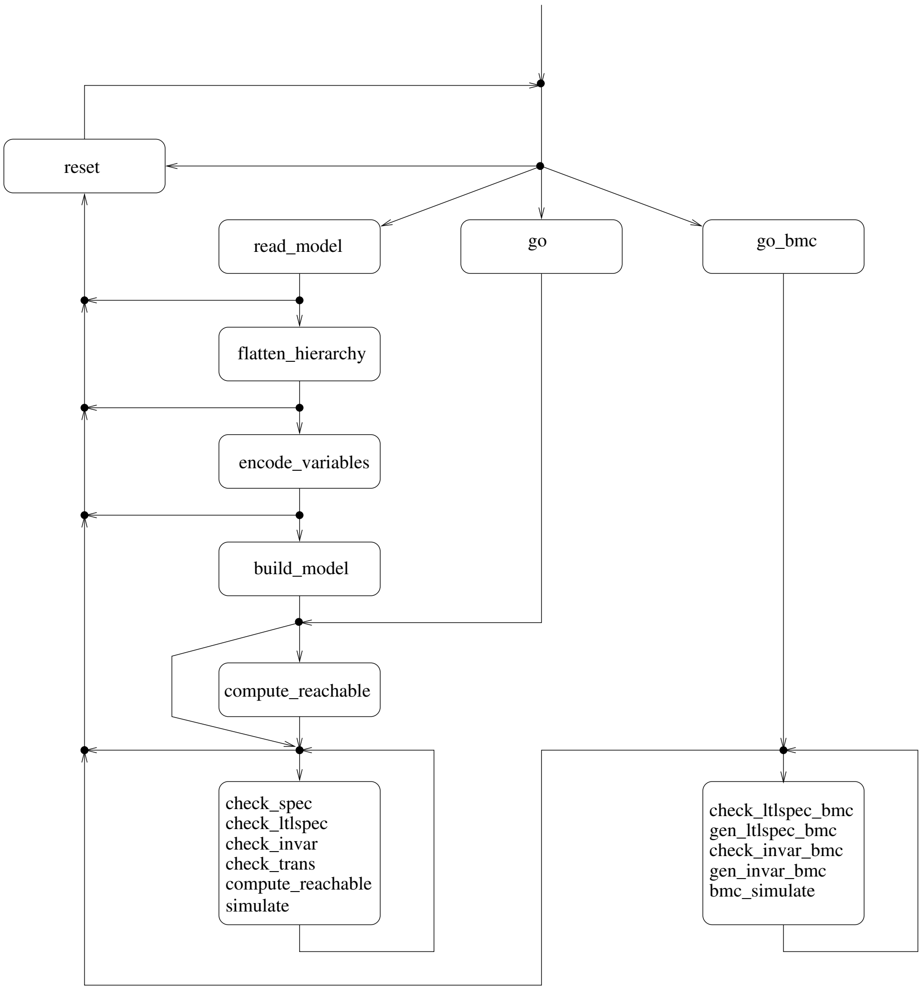

Figure 3.1: The dependency among NuSMV commands.

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`.

Chapter 4

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

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

`-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.

`-int` Starts interactive shell.

`-load` *cmd-file* Starts the interactive shell and then executes NuSMV commands from file *cmd-file*. If an error occurs during a command execution, commands that follow will not be executed. See also the variable `on_failure_script_quits`.

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

`-cpp` Runs preprocessor on SMV files before any of those specified with the -pre option.

`-pre` *pps* Specifies a list of pre-processors to run (in the order given) on the input file before it is parsed by NuSMV. Note that if the `-cpp` command is used, then the pre-processors specified by this command will be run after the input file has been pre-processed by that pre-processor. *pps* is either one single pre-processor name (with or without double quotes) or it is a space-seperated list of pre-processor names contained within double quotes.

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

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

`-lp` Lists all properties in SMV model

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

`-is` Does not check `SPEC`

`-ic` Does not check `COMPUTE`

`-ils` Does not check `LTLSPEC`

`-ips` Does not check `PSLSPEC`

`-ii` Does not check `INVARSPEC`

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

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

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

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

`-coi` Enables cone of influence reduction

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

`-o` *ov_file* Reads the variable ordering from file *ov_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 3.8 [Interface to DD package], page 54).

`-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*

`-noaffinity`
: Disables affinity clustering

`-iwls95preoder`
: Enables *Iwls95CP* preordering

`-bmc`
: Enables BMC instead of BDD model checking (works only for LTL properties and PSL properties that can be translated into LTL)

`-sat_solver` *name*
: Sets the default SAT Solver to be used with BMC

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

Bibliography

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

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(4):401–424*, April 1994.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *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.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*, 2002.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. In *International Journal on Software Tools for Technology Transfer (STTT), 2(4)*, March 2000.

[CGH97] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In *Formal Methods in System Design, 10(1):57–71*, February 1997.

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In *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. Krushan, editors, Proceedings of Computer-Aided Verification (CAV'90), volume 531 of LNCS, pages 136-145, Berlin, Germany*, June 1991.

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

[Mar85] A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *In H. Fuchs and W.H. Freeman, editors,* Proceedings of the 1985 Chapel Hill Conference on VLSI, *pages 245–260, New York*, 1985.

[McM92] K.L. McMillan. The smv system – draft. In *Available at* `http://www.cs.cmu.edu/ modelcheck/smv/smvmanual.r2.2.ps`, 1992.

[McM93] K.L. McMillan. Symbolic model checking. In *Kluwer Academic Publ.*, 1993.

[MHS00] Moon, Hachtel, and Somenzi. Border-block tringular form and conjunction schedule in image computation. In *FMCAD*, 2000.

[PSL] Language Front-End for Sugar Foundation Language. http://www.haifa.il.ibm.com/projects/verification/sugar/parser.html.

[psl03] Accellera, Property Specification Language - Reference Manual - Version 1.01. http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf, April 2003.

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

[sfVS96] ”VIS: A system for Verification and The VIS Group Synthesis”. Proceedings of the 8th international conference on computer aided verification, p428-432. In *Springer Lecture Notes in Computer Science, 1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ*, 1996.

[Som98] F. Somenzi. Cudd: Cu decision diagram package — release 2.2.0. In *Department of Electrical and Computer Engineering — University of Colorado at Boulder*, May 1998.

Appendix A

Compatibility with CMU SMV

The NuSMV language is mostly source compatible with the original version of SMV distributed at Carnegie Mellon University from which we started. In this appendix we describe the most common problems that can be encountered when trying to use old CMU SMV programs with NuSMV.

The main problem is variable names in old programs that conflicts with new reserved words. The list of the new reserved words of NuSMV w.r.t. CMU SMV is the following:

`F, G, X, U, V, W, H, O, Y, Z, S, T, B`	These names are reserved for the LTL temporal operators.
`LTLSPEC`	It is used to introduce LTL specifications.
`INVARSPEC`	It is used to introduce invariant specifications.
`IVAR`	It is used to introduce input variables.
`JUSTICE`	It is used to introduce "justice" fairness constraints.
`COMPASSION`	It is used to introduce "compassion" fairness constraints.

The `IMPLEMENTS`, `INPUT`, `OUTPUT` statements are not supported by NuSMV. They are parsed from the input file, but are internally ignored.

NuSMV differs from CMU SMV also in the controls that are performed on the input formulas. Several formulas that are valid for CMU SMV, but that have no clear semantics, are not accepted by NuSMV. In particular:

- 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 done more restrictive. 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);
  ```

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 may be not accepted by CMU SMV. Notice that there is no guarantee that a good ordering for CMU SMV is also a good ordering for NuSMV. In the ordering files for NuSMV, identifier `_process_selector_` can be used to control the position of the variable that encodes process selection. In CMU SMV it is not possible to control the position of this variable in the ordering; it is hard-coded at the top of the ordering.

Command Index

Variable Index

Index