NuSMV 1.1 User Manual

Alessandro Cimatti
cimatti@fbk.eu
IRST - Via Sommarive 18, 38055 Povo (Trento) – Italy

Marco Roveri
roveri@fbk.eu
IRST - Via Sommarive 18, 38055 Povo (Trento) – Italy
DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

This document is part of the distribution package of the NuSMV model checker, available at http://afrodite.itc.it:1024/~nusmv/ and at http://www.cs.cmu.edu/~modelcheck/. 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.

Copyright © 1998 ITC-IRST and CMU.

Table of Contents

1 Introduction 1

2 Examples 2

3 Syntax 9

3.1 Expressions 9
3.1.1 Simple Expressions 9
3.1.1.1 Case Expressions 10
3.1.1.2 Set Expressions 10
3.1.2 Next Expressions 11
3.2 Definition of the Kripke structure 12
3.2.1 State Variables 12
3.2.1.1 Type Specifiers 12
3.2.2 Input Variables 12
3.2.3 `ASSIGN` declarations 13
3.2.4 `TRANS` declarations 13
3.2.5 `INIT` declarations 14
3.2.6 `INVAR` declarations 14
3.2.7 `DEFINE` declarations 14
3.2.8 `ISA` declarations 15
3.2.9 `MODULE` declarations 15
3.2.10 Identifiers 16
3.2.11 The `main` module 18
3.2.12 Processes 18
3.2.13 `FAIRNESS` declarations 18
3.3 Specifications 19
3.3.1 CTL Specifications 19
3.3.2 LTL Specifications 20
3.3.3 Real Time CTL Specifications and Computations 20

4 Running `NuSMV` interactively 22

4.1 Model Reading and Building 23
4.2 Commands for Checking Specifications 27
4.3 Simulation Commands 29
4.4 Traces Inspection Commands 35
4.5 Model Modification Commands 36
4.6 Interface to the DD Package 37
4.7 Administration Commands 40
4.8 Other Environment Variables 44

5 Running `NuSMV` batch 45

Bibliography 47

Appendix A Compatibility with CMU `SMV` 48

Appendix B New reserved words 50

Command Index 51

Variable Index 52

Index 53

1 Introduction

NuSMV is a symbolic model checker jointly developed by Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST). The NuSMV project aims at the development of an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas. NuSMV is available at `http://afrodite.itc.it:1024/~nusmv/` and at `http://www.cs.cmu.edu/~modelcheck/`.

The main features of NuSMV are the following:

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

This document is structured as follows.

- In Chapter 2 [Examples], page 2 we present some examples of usage of NuSMV.
- In Chapter 3 [Syntax], page 9 we define the syntax of the input language of NuSMV.
- In Chapter 4 [Running NuSMV interactively], page 22 the commands of the interaction shell are described.
- In Chapter 5 [Running NuSMV batch], page 45 we define the batch mode of NuSMV.

2 Examples

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

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

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

Consider the following simple program in the NuSMV language.

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

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

The first part defines the Kripke structure. The space of states of the Kripke structure is determined by the declarations of the state variables (in the above example `request` and `state`). The variable `request` is declared to be of (predefined) type `boolean`. This means that it can assume the (integer) values `0` and `1`. The variable `state` is a scalar variable, which can take the symbolic values `ready` or `busy`.

The following assignment sets the initial value of the variable `state` to `ready`. The initial value of `request` is completely unspecified, i.e. it can be either `0` or `1`.

The transition relation of the Kripke structure is expressed by defining the value of variables in the next state (i.e. after each transition), given the value of variables in the current states (i.e. before the transition). The `case` segment sets the next value of the variable `state` to the value `busy` (after the column) if its current value is `ready` and `request` is `1` (i.e. true). Otherwise (the `1` before the column) the next value for `state` can be any in the set `{ready,busy}`.

The variable `request` is not assigned. This means that there are no constraints on its values, and thus it can assume any value. `request` is thus an unconstrained input to the system.

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

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

```
MODULE counter_cell(carry_in)
 VAR
   value : boolean;
 ASSIGN
   init(value) := 0;
   next(value) := value + carry_in mod 2;
 DEFINE
  carry_out := value & carry_in;

MODULE main
 VAR
   bit0 : counter_cell(1);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
 SPEC
   AG AF bit2.carry_out
```

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

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

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

Defined symbols do not require introducing a new variable into the BDD representation of the system. However, they cannot be given values non-deterministically. Another difference between defined symbols and variables is that while the type of variables is declared a priori, for definitions this is not the case.

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

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

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

MODULE main
 VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);

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

Among all the modules instantiated with the `process` keyword, one is nondeterministically chosen, and the assignment statements declared in that process are executed in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates. For each gate, the specification of this program states that the output of the gate oscillates (i.e. that its value is infinitely often `0`, and infinitely often `1`). In fact, this specification is false, since the system is not forced to eventually choose a given process to execute, hence the output of a given gate may remain constant, regardless of its input.

In order to force a given process to execute infinitely often, we can use a *fairness constraint*. A fairness constraint restricts the attention of the model checker to only those execution paths along which a given CTL formula is true infinitely often. Each process has a special variable called `running` which is `1` if and only if that process is currently executing.

By adding the declaration:

```
FAIRNESS
  running
```

to the module `inverter`, we can effectively force every instance of `inverter` to execute infinitely often, thus making the specification true.

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

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

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

The `union` operator (set union) coerces its arguments to singleton sets as necessary. Thus, the next `output` of each gate can be either its current `output`, or the negation of its current `input` – each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as 2^n , where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation.

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

```
MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;
 SPEC
   AG !(proc1.state = critical &  proc2.state = critical)

MODULE user(semaphore)
 VAR
   state : {idle, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
```

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

If any of the specification is false, the NuSMV model checker attempts to produce a counterexample, proving that the specification is false. This is not always possible, since formulas preceded by existential path quantifiers cannot be proved false by a showing of a single execution path. Similarly, sub-formulas preceded by universal path quantifier cannot be proved true by a showing of a single execution path. In addition, some formulas require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state.

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

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

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

```
-- specification
-- AG (proc1.state = entering -> AF proc1.state = critical)
-- is false
-- as demonstrated by the following execution sequence
state 1.1:
semaphore = 0
proc1.state = idle
proc2.state = idle

state 1.2:
[executing process proc1]

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

state 1.4:
[executing process proc2]
```

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

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

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

state 1.7:
[executing process proc2]

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

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

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

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

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

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

MODULE inverter(input)
 VAR
   output : boolean;
 INIT
   output = 0
 TRANS
   next(output) = !input | next(output) = output
```

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

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

3 Syntax

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

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

3.1 Expressions

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

3.1.1 Simple Expressions

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

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

A *var_id*, (see Section 3.2.10 [Identifiers], page 16) or identifier, is a symbol or expression which identifies an object, such as a variable or a defined symbol. Since a `var_id` can be an `atom`, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the interpreter as an error.

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

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

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

3.1.1.1 Case Expressions

A case expression has the following syntax:

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

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

3.1.1.2 Set Expressions

A set expression has the following syntax:

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

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

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

3.1.2 Next Expressions

While simple expressions can represent sets of states, next expressions relate current and next state variables to express transitions in the Kripke structure. The structure of next expressions is similar to the structure of simple expressions (See Section 3.1.1 [Simple Expressions], page 9). The difference is that next expression allow to refer to next state variables. The grammar is depicted below.

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

`set_next_expr` and `case_next_expr` are the same as `set_simple_expr` (see Section 3.1.1.2 [Set Expressions], page 10) and `case_simple_expr` (see Section 3.1.1.1 [Case Expressions], page 10) respectively, with the replacement of `"simple"` with `"next"`. The only additional production is `"next" "(" simple_expr ")"`, which allows to "shift" all the variables in `simple_expr` to the *next* state. The `next` operator distributes on every operator. For instance, the formula `next((A & B) | C)` is a shorthand for the formula `(next(A) & next(B)) | next(C)`. It is an error if in the scope of the `next` operator occurs another `next` operator.

3.2 Definition of the Kripke structure

3.2.1 State Variables

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

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

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

3.2.1.1 Type Specifiers

A type specifier has the syntax:

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

val  :: atom
     |  number
```

A variable of type `boolean` can take on the numerical values `0` and `1` (representing false and true, respectively). In the case of a list of values enclosed in quotes (where atoms are taken to be symbolic constants), the variable is a scalar which take any of these values. In the case of an `array` declaration, the first `simple_expr` is the lower bound on the subscript and the second `simple_expr` is the upper bound. Both of these expressions must evaluate to integer constants. Finally, an atom optionally followed by a list of expressions in parentheses indicates an instance of module atom (See Section 3.2.9 [MODULE declarations], page 15). The keyword causes the module to be instantiated as an asynchronous process (See Section 3.2.12 [Processes], page 18).

3.2.2 Input Variables

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

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

The type associated with a variable declaration can be either a boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays) (See Section 3.2.1 [State Variables], page 12).

3.2.3 `ASSIGN` declarations

An assignment has the form:

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

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

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

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

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

3.2.4 `TRANS` declarations

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

```
trans_declaration :: "TRANS" trans_expr

trans_expr  :: next_expr
```

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

3.2.5 INIT declarations

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

```
init_declaration :: "INIT" init_expr

init_expr   :: simple_expr
```

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

3.2.6 INVAR declarations

The set of invariant states (i.e. the analogous of normal assignments, as described in Section 3.2.3 [ASSIGN declarations], page 13) can be specified using a boolean expression under the 'INVAR' keyword. The syntax of a INVAR declaration is:

```
invar_declaration       :: simple_expr
```

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

3.2.7 DEFINE declarations

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

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

Whenever an identifier referring to the symbol on the left hand side of the ':=' in a DEFINE occurs in an expression, it is replaced by the expression on the right hand side. The expression on the right hand side is always evaluated in its context, however (see Section 3.2.9 [MODULE declarations], page 15 for an explanation of contexts). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

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

3.2.8 ISA declarations

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

The syntax of an ISA declaration is as follows:

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

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

3.2.9 MODULE declarations

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

The syntax of a module declaration is as follows.

```
module ::
        "MODULE" atom [ "(" atom "," atom "," ... atom ")" ]
        [ var_declaration        ]
        [ ivar_declaration       ]
        [ assign_declaration     ]
        [ trans_declaration      ]
        [ init_declaration       ]
        [ invar_declaration      ]
        [ spec_declaration       ]
        [ checkinvar_declaration ]
        [ ltlspec_declaration    ]
        [ compute_declaration    ]
        [ fairness_declaration   ]
        [ define_declaration     ]
        [ isa_declaration        ]
```

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

An *instance* of a module is created using the VAR declaration (see Section 3.2.1 [State Variables], page 12). This declaration supplies a name for the instance, and also a list of actual parameters,

which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error if the number of actual parameters is different from the number of formal parameters. The semantic of module instantiation is similar to call-by-reference. For example, in the following program fragment:

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

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

Now consider the following program:

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

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

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

3.2.10 Identifiers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
MODULE main
 VAR c1 : cell;
```

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

3.2.11 The `main` module

The syntax of a `NuSMV` program is:

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

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

3.2.12 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword '`process`' (see Section 3.2.1 [State Variables], page 12). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Each instance of a process has a special boolean variable associated with it called `running`. The value of this variable is `1` if and only if the process instance is currently selected for execution. A process may run only when its parent is running. In addition no two processes with the same parents may be running at the same time.

3.2.13 `FAIRNESS` declarations

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

```
fairness_declaration :: "FAIRNESS" fair_expr

fair_expr   :: ctl_expr
```

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

3.3 Specifications

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

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

3.3.1 CTL Specifications

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

```
spec_declaration :: "SPEC" spec_expr

spec_expr   :: ctl_expr
```

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

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

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

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

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

This statement corresponds to

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

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

3.3.2 LTL Specifications

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

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

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

where

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

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

3.3.3 Real Time CTL Specifications and Computations

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

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

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

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

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

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

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

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

where

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

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

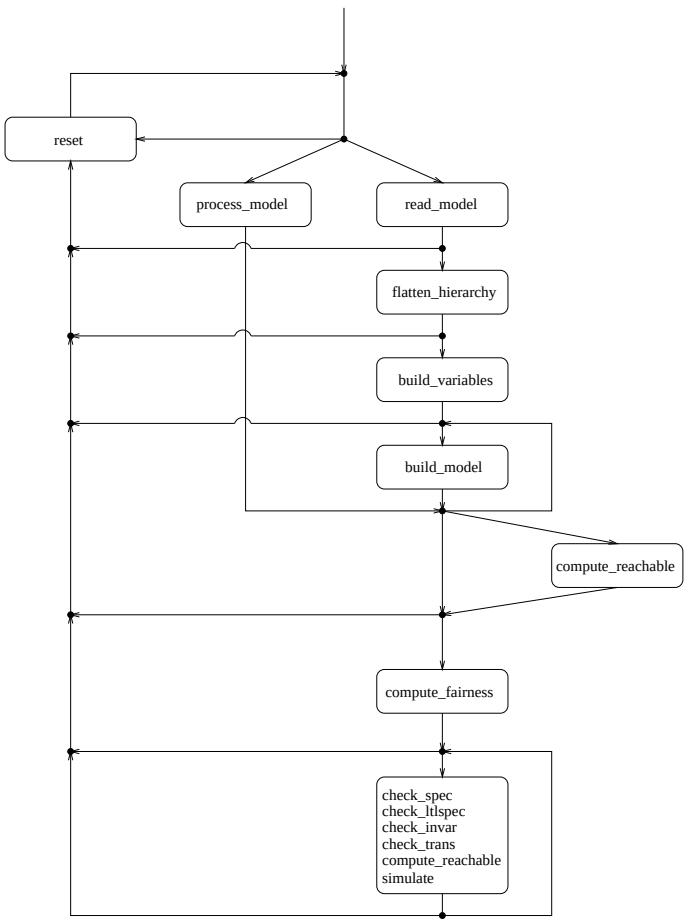
`MAX [`*start* , *final*`]` returns the length of the longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, then *infinity* is returned.

4 Running NuSMV interactively

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

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

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



In the following we present the possible commands followed by the related environment variables, classified in different categories. Every command answers to the option -h by printing out the command usage. When output is paged for some commands (option -m), it is piped through the

program specified by the UNIX PAGER shell variable, if defined, or through UNIX command "more". Environment variables can be assigned a value with the set command.

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

4.1 Model Reading and Building

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

verbose_level Environment Variable

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

read_model [*-h*] [*-i model-file*] Command

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

-i model-file
: Reads the model from file *model-file*, and sets the environment variable input_file to *model-file*.

input_file Environment Variable

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

flatten_hierarchy [*-h*] Command

Intantiates module declarations into the actual modules and processes. The instantiation substitutes the actual parameters for the formal parameters, and then prefixing the result by the instance name. This operation is performed on the textual representation, before any operation on decision diagrams is carried out.

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

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

-s Prints only state variables.

-i Prints only input variables.

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

-o output-file
: Writes the generated output to the file *output-file*.

build_variables [*-h*] [*-i order-file*] Command

Generates the boolean BDD variables and the ADD (Algebraic Decision Diagrams) needed to encode propositionally the (symbolic) variables declared in the model. *order-file* specifies the initial ordering of variables. By default the variables are created 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. The variables declared in the model not listed in the input file are created at the end of the given ordering following the default ordering.

-i order-file

Sets the environment variable `input_order_file` to *input-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 '`build_variables`' command. There is no default value.

write_order [*-h*] | [*-f order-file*] Command

Writes the current order of BDD variables in the file specified in the environment variable `output_order_file`.

-f order-file

Sets the environment variable `output_order_file` to *order-file* and then writes the ordering into it.

output_order_file Environment Variable

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

build_model [*-h*] [*-f*] [*-m method*] Command

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

-m method

Sets the environment variable `partition_method` to the value *method*, and then builds the transition relation. Available methods are '`Monolithic`', '`Disjunctive`', '`Conjunctive`' and '`Iwls95CP`'.

-f Forces the model construction using the specified method. By default, only one partition method is allowed. This option allows to overcome this default, and build the transition relation with different partitionings.

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.
- **Conjunctive**. 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`.
- **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`.
- **Disjunctive**. Disjunctive partitioning. Available only for asynchronous models (i.e. if there are processes). A partition is built for each process in the model.

conj_part_threshold Environment Variable

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

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

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

print_iwls95info [*-h*] [*-m* | *-o output-file*] Command

This command prints out the information regarding each cluster. In particular for each cluster it prints out, the cluster number, the size of the cluster (in BDD nodes), the variables occurring in it, the size of the cube that has to be quantified out relative to the cluster, and the variables to be quantified out.

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

-o output-file
Writes the output produced to file *output-file*, rather than to the standard output.

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 [*-h*] Command

This command prints out the configuration parameters of the IWLS95 clustering algorithm, i.e. `image_verbosity`, `image_cluster_size` and weights `image_W{1,2,3,4}`.

compute_fairness [*-h*] Command

For each fairness constraint, computes the corresponding set of states.

go [*-h*] Command

This command initializes the system for verification. It is equivalent to the command sequence `read_model`, `flatten_hierarchy`, `build_variables`, `build_model`,

`compute_fairness`. If some commands have already been executed, then only the remaining ones will be invoked.

process_model [*-h*] [*-i model-file*] [*-m partition-method*] Command

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`.

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

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

forward_search Environment Variable

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

check_trans [*-h*] [*-m* | *-o output-file*] Command

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

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

-o output-file
: Writes the output generated by the command to the file *output-file*.

The reachable states are computed before in order to guarantee that deadlock states are actually reachable.

check_trans Environment Variable

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

4.2 Commands for Checking Specifications

compute_reachable [*-h*] Command

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 performance because the computation of reachable states may be very expensive. The environment variable `forward_search` is set.

print_reachable_states [*-h*] Command

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

check_spec [*-h*] [*-m | -o output-file*] [*ctl-expression*] Command

Evaluates `SPEC` statements. If no *ctl-expression* is given at command line, then the command evaluates the previously parsed `SPEC` statements, if any. If a formula is false, a path witnessing its falsity is printed. Infinite paths are represented by a finite sequence of states leading to a cycle.

-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 "more" command.

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

ctl-expression
A real time CTL formula to be checked.

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

ag_only_search Environment Variable

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

check_invar [*-h*] [*-m | -o output-file*] [*invar-expression*] Command

Evaluates invariants. If no *invar-expression* is given at command line, then the command evaluates the previously parsed `INVARSPEC` statements, if any. If a formula is false, a path witnessing its falsity is printed.

-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 "more" command.

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

invar-expression
: An invariant to be verified.

check_ltlspec [*-h*] [*-m* | *-o output-file*] [*ltl-expression*] Command

Performs model checking of LTL formulas. If no *ltl-expression* is given at command line, then the previously parsed `LTLSPEC`, if any, are considered. LTL model checking is reduced to CTL model checking as described in *[CGH97]*.

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

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

ltl-expression
: A LTL formula to be checked.

compute [*-h*] [*-m* | *-o output-file*] [*compute-expression*] Command

Performs quantitative analysis. If no *compute-expression* is given at command line, then the previously parsed `COMPUTE` statements, if any, are considered.

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

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

compute-expression
: A quantitative characteristic to be computed.

4.3 Simulation Commands

To give the user the opportunity to get familiar with the behavior of the model to be checked, a simulation feature has been provided in the NuSMV system. A simulation's facility lets the user widely navigate structures and properties of the model under test through the generation of its possible executions (*traces* from now on). It also allows the user to acquire confidence with the correctness of the model before actual verification of properties. Maximum flexibility and degrees of freedom in a simulation session are achieved introducing three different trace generation policies: deterministic, random and interactive. Each of them corresponds to a different way a state is chosen from a set.
In deterministic simulation mode the first state of a set (whatever it is) is chosen, while in the random one the choice is non deterministically performed. In these two first modes traces are automatically generated by NuSMV: the user obtains the whole of the trace in a time without control over the generation itself (except for the simulation mode and the number of states entered via command line).
Third simulation mode lets the user have a complete control over traces generation by interactively building the trace. During an interactive simulation session, the system stops at every step showing a list of possible future states: the user is requested to choose one of the items. This feature is particularly useful when one wants to inspect some particular reactions of the model to be checked. To avoid that the user easily gets confused when the number of possible choices is too high, an internal limit on the number of states that are showed step by step has been set. If, eventually, such a situation should take place, the system will ask the user to "guide" the simulation via the insertion of some further constraints (as simple expressions) over the variables. The system will continue to ask for constraints insertion until the number of future states will be under a predefined threshold: expressions entered during this particular phase are accumulated (in a logical product) in a single big constraint used only for the actual step of the simulation and discarded before the next one.
The system also makes a control over the expressions given by the user and will request for a further insertion if an inconsistency arises. Cases of inconsistency (i.e. empty set of states) may be located in:

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

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

Every trace generated during a NuSMV session (created either by the counterexample generator or by the simulator itself) can be explored in a second time via the `show_traces` command or the `goto_state` command (see Section 4.4 [Traces Inspection Commands], page 35).

Commands descriptions follow:

pick_state [*-h*] [*-v*] [*-r* | *-i* [*-a*]] [*-c* "*constraints*"] Command
Chooses an element from the set of initial states, and makes it the *current state* (replacing the old one). Chosen state is stored as the first state of a new trace ready to be lengthened by *steps* states by the `simulate` command (see below). 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.

-v Verbosely prints the 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 Picks a state randomly 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 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 constraints as simple expression (see Section 3.1.1 [Simple Expressions], page 9).

-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` option has been choosen.

-c "constraints"
Uses *constraints* (a simple expression) to restrict the set of initial states in which the state has to be picked. Note: *constraints* must be enclosed between double quotes " ".

show_traces [*-h*] [*-v*] [*-t*] [*-m* | *-o output-file*] [*-a* | *trace-number*] Command
Shows the traces currently stored in system memory, if any. By default it shows the last generated trace, if any.

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

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

-a Prints all the currently stored traces.

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

-o output-file
Writes generated output to the file *output-file*.

trace-number
The (ordinal) identifier number of the trace to be printed.

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

simulate [*-h*] [*-p* | *-v*] [*-r* | *-i* [*-a*]] [*-c "constraint"*] *steps* Command
Generates a sequence of at most *steps* states (representing a possible execution of the model), starting from the *current state* (that has to be set by 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 builded, 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 one. In this case the old trace is lengthened by at most *steps* states.

-p Prints current generated trace (only those variables that have changed their value from 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 (see Section 3.1.1 [Simple Expressions], page 9). These constraints are used only for a single simulation step and are *forgotten* in the next ones. They are to be intended in an opposite way with respect to those constraints eventually entered for 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 state variables (changed and unchanged) during every step of an interactive session. This option works only if the `-i` option has been choosen.

-c "constraints"
Performs a simulation in which computation is restricted to states that satisfy those *constraints* (a simple expression). 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* 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 a future state satisfying those constraints.

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

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

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

```
                    1 : {ready,busy};
                  esac;
SPEC
  AG(request -> AF state = busy)
```

This model has now to be read into the NuSMV system[2] (see Section 4.1 [Model Reading and Building], page 23). A starting point (i.e. a state) is needed before the simulator could start to build any trace. By default the simulator gets the actual *current state* as a starting point of every new trace: so it will be included as the first state of a new trace except if otherwise stated; the user has been provided with the following two NuSMV commands to overcome this behavior:

- `goto_state`: this command (see Section 4.4 [Traces Inspection Commands], page 35) provides the user to set any state of any already existent trace as the *current state*;
- `pick_state`: this command can be used whenever one wants to choose the starting point of a new trace among the set of initial states, and it has to be used when a *current state* doesn't exist yet (that is when the system has being reset).

As at this point of the example *current state* doesn't exist, and there is no trace currently stored in the system, an item from the set of initial states has to be picked using the NuSMV `pick_state` command. A simulation session can be started now, using the `simulate` command. The system will generate a trace according to the options entered via command line[3]. New builded traces are numbered: every trace is identified by an integer number, while every state belonging to a trace is identified by a *dot notation*: for example state *1.3* is the third state of the first generated trace.

```
system_prompt> NuSMV -int short.smv
NuSMV release 1.1 (compiled 30-Jul-99 at 7:25 PM)
NuSMV > go
NuSMV > pick_state -r
NuSMV > print_current_state -v
Current state is 1.1
request = 0
state = ready
NuSMV > simulate -r 3
*********  Starting Simulation From State  1.1  *********
NuSMV > show_traces -t
There is 1 trace currently available.
NuSMV > show_traces -v
#################### Trace number: 1 ####################
-> State 1.1 <-
    request = 0
    state = ready
-> State 1.2 <-
    request = 1
    state = busy
```

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

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

```
-> State 1.3 <-
    request = 1
    state = ready
-> State 1.4 <-
    request = 1
    state = busy
```

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

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

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

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

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

[4] *current state* also changes executing the `step` command (see Section 4.4 [Traces Inspection Commands], page 35) and `reset` command (see Section 4.7 [Administration Commands], page 40) but in this last case *current state* is deleted.

```
    state = ready

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

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

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

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

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

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

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

Note how the set of possible states to choose has being restricted (in this case there is only one future state, so the system will automatically pick it, waiting for the user to press the ⟨RET⟩ key).

4.4 Traces Inspection Commands

A trace is a sequence of states corresponding to a possible execution of the model. Traces are created by NuSMV when a formula is found to be false; they are also generated by the simulation feature (Section 4.3 [Simulation Commands], page 29). Each trace has a number, and the states are numbered within the trace. Trace *n* has states *n.1, n.2, n.3, "..."*.

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

goto_state [*-h*] *state* Command

Makes *state* the *current state*.

print_current_state [*-h*] [*-v*] Command

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

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

step [*-h*] [*-v*] Command

Moves to the next state in the current trace. If the *current state* is the last in the trace, then an error message is printed out.

-v Prints the value of all state variables, otherwise it prints only the values of the state variables that have changed their value from previous state.

assign [*-h*] *variable := expression* Command

Assigns the value of *expression*, as evaluated in the *current state*, to *variable*.

variable := expression
Assigns the value of *expression*, as evaluated in the *current state*, to *variable*. This command changes the *current state*. The value of all other variables in the new *current state* remains the same as it was in the old *current state*.

eval [*-h*] *ctl-expression* Command

Evaluates *ctl-expression* in the *current state*. Since we deal with CTL formulas, the evaluation activates the model checking algorithms, and can produce a new trace starting from *current state*, to be used by later commands. If additional traces are created from state *n.i*, they are numbered *t+1*, *t+2*, *t+3*, ..., where *t* is the number corresponding to the last trace generated. Within these traces, the states are numbered *(t+i).1*, *(t+i).2*, *(t+i).3*, ...

4.5 Model Modification Commands

NuSMV allows for interactively modifying the Kripke structure. This can be done by changing the transition relation, the initial states, or adding new fairness constraints. To activate the modification mode, it is necessary to save the model with the command `dump_model`. The saved model can be restored after the modifications with the command `restore_model`.

dump_model [*-h*] Command

Initializes the system to allow interactive model modification. This command is currently available only with the '`Monolithic`' partition method.

add_init [*-h*] *simple-expression* Command

Modifies the initial states. *simple-expression* is evaluated and conjoined with the set of initial states to form the new set of initial states. This command has the same effect of an `INIT` declaration in the original input file. Before executing this command, the `dump_model` command must have been executed.

add_trans [*-h*] *trans-expression* Command

Modifies the transition relation. *trans-expression* is evaluated and then conjoined with the current transition relation. This command has the same effect of a `TRANS` statement in the original input file. Before executing this command, the `dump_model` command must have been executed.

add_fairness [*-h*] *ctl-expression* Command

Add a new fairness constraint to the existing list of fairness constraints. *ctl-expression* is evaluated and added to the current list of fairness constraints. This command has the same effect of a `FAIRNESS` statement in the original input file. Before executing this command, the `dump_model` command must have been executed.

restore_model [*-h*] Command

This command cancels the effect of all `add_fairness`, `add_init` and `add_trans` commands issued inside the model inspection session, i.e. after the `dump_model` command.

4.6 Interface to the DD Package

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

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

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

enable_reorder Environment Variable

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

reorder_method Environment Variable

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

dynamic_var_ordering [[-*d*] | [-*e method*] | [-*f method*] [-*h*]] Command

Controls the application and the modalities of (dynamic) variable ordering. 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` *file*) in the future.

-*d*
: Disables dynamic ordering.

-*f method*
: Forces reordering of variables with the specified *method*.

-*e method*
: Enables dynamic ordering to trigger automatically whenever a certain threshold on the overall BDD size is reached.

method must be one of the values allowed for the environment variable `reorder_method`.

print_bdd_stats [-*h*] Command

Prints the statistics for the BDD package. The amount of information depends on the BDD package configuration established at compilation time. The configuration

parameters are printed out too. More information about statistic and parameters can be found in the documentation of the CUDD Decision Diagram package.

set_bdd_parameters [*-h*] [*-s*] Command

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

-s Print the bdd parameter and statistics after the modification.

The command uses the global DD manager and the set of pairs (variable,value) of the NuSMV environment, the function sets specific BDD parameters to the given values. This command works in conjunction with '`print_bdd_stats`'.
'`print_bdd_stats`' first prints a report of the parameters and statistics of the DD 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.
The most important parameters are:

- *Hard limit for cache size*
  Its the maximum number of entries in the cache. Usually it is automatically determined based on the available memory.
- *Cache hit threshold for resizing*
  The hit ratio that causes the resize of the cache. The default value is `30%`.
- *Limit for fast unique table growth*
  This parameter controls the threshold beyond which no fast growth of the unique table is allowed. Usually a suitable value is determined based on the available memory.
- *Garbage collection enabled*
  Enables or disables the garbage collection. Initially the garbage collection is enabled.

These parameters can have a big impact on the performances. See the documentation of the CUDD Decision Diagram package for a more detailed explanation of the meaning of these parameters and how they can affect performances.

4.7 Administration Commands

! *shell_command* Command

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

alias [*-h*] [*name* [*string*]] Command

Defines command alias. If given no arguments, will print the definition of all current aliases. Given a single argument (*name*), it will print the definition of that alias (if any). Given two arguments, the keyword *name* becomes an alias for the command *string*, replacing any other alias with the same name.

name [string]

Defines *name* to be an alias of the command specified by *string* if given, else prints out the definition of alias *name* if any. 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 image_set set image_\%:1 \%:2
NuSMV> image_set W1 100
```

will create an alias '`image_set`', execute '`set image_W1 100`'.
Caveat: Currently there is no check to see if there is a circular dependency in the alias definition. e.g.

```
NuSMV> alias foo "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. At execution time this problem is solved by a loop detection check.

echo [*-h*] [*string*] Command

Echoes the arguments to standard output.

help [*-a*] [*-h*] [*command_name*] Command

Given no arguments, will print a list of all commands known to the command interpreter. If a *command_name* is given, detailed informations for that command will be provided if they are available.

-a Provides a list of all internal commands, which by convention begin with an underscore ('_').

history [*-h*] [*#num*] Command

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

#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. The shell escape can be changed by assigning a character to the environment variable '`shell_char`'. In this description '%' is used as the '`history_char`'. The '%' can appear anywhere in a line. A line containing a history substitution is echoed to the screen after the substitution takes place. '%' can be preceded by a '\' in order to escape the substitution, for example, to enter a '%' into an alias or to set the prompt.
Each valid line typed at the prompt is saved. If the '`history`' variable is set (see the documentation of the `set` command), each line is also echoed to the history file. You can use the `history` command to list the previously typed commands.

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

%:0	Initial word of last command.
%:n	n-th argument of last command.
%$	Last argument of last command.
*%**	All but initial word of last command.
%%	Last command.
%stuf	Last command beginning with "stuf".
%n	Repeat the n-th command.
%-n	Repeat the n-th previous command.
^old^new	Replace '`old`' with '`new`' in previous command. Trailing spaces are significant during substitution. Initial spaces are not significant.

print_usage [*-h*] Command
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 [*-h*] [*-s*] Command
Exits `NuSMV` by stopping the program.

-s	Free all the memory before quitting. This is slower, and is used for finding memory leaks.

reset [*-h*] Command
Resets the whole system, in order to read in another model and to perform verification on it.

set [-*h*] [*name* [*value*]] Command

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.

name [value]
: *name* is a variable name, and *value* is the value to be assigned to the variable.

source [-*h*] [-*p*] [-*s*] [-*x*] *script-file* [*args*] Command

Reads and executes commands from a file.

-*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.

script-file A file name containing `NuSMV` commands.

The *args* 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.

time [-*h*] Command

Prints the processor time used since the last '`time`' command, and the total processor time used since `NuSMV` was started.

unalias [-*h*] *alias-names* Command

Removes the definition of an alias.

alias-names
: Aliases to be removed

unset [-*h*] *variables* Command

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.

variables Variables to be unset.

usage [-*h*] Command

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

which [-*h*] *file-name* Command

Looks for a file in a set of directories which includes the current directory as well as those in the `open_path` environment variable. If it finds it, it reports the path. The

directories are specified through the 'set open_path' command in the '{~/}.nusmvrc' or by the shell variable NuSMV_LIBRARY_PATH.

file-name File to be searched.

4.8 Other Environment Variables

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

autoexec Environment Variable

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

filec Environment Variable

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

shell_char Environment Variable

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

history_char Environment Variable

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

open_path Environment Variable

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

nusmv_stderr Environment Variable

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

nusmv_stdout Environment Variable

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

nusmv_stdin Environment Variable

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

5 Running NuSMV batch

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

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

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

The batch interaction can be (partially) controlled with the following command line options:

```
NuSMV [-int] [-s] [-load cmd-file] [-h|-help] [-v verbose-level]
      [-r] [-f] [-AG] [-ctt] [-is] [-ils] [-ii]
      [-i order-file] [-o order-file] [-reorder] [-m method]
      [[-cp part-limit]|[-dp]|[-iwls95 part-limit]] [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.

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

`-cp` *part-limit*
Build a conjunctively partitioned transition relation.

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

`-dp` Build a disjunctively partitioned transition relation.

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

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

`-i` *order-file*
Read the variable ordering from file *order-file*.

`-ii` Do not check `INVARSPEC`s.

`-ils` Do not check `LTLSPEC`s.

`-int` Start interactive shell.

`-is` Do not check `SPEC`s and `COMPUTE`s.

`-iwls95` *part-limit*
Build a conjunctively partitioned transition relation, using *[RAP+95]* clustering heuristics, with threshold *part-limit*.

`-load` *cmd-file*
Interpret `NuSMV` commands from file *cmd-file*.

`-m` *method*
Use *method* when variable ordering is enabled. Possible values for method are those allowed for the `reorder_method` environment variable (see Section 4.6 [Interface to the DD package], page 37).

`-o` *order-file*
: Write the variable ordering to file *order-file*. No evaluation occurs when only the `-o` command option is used in batch mode.

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

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

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

`-v` *verbose-level*
: Enable 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.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A Compatibility with CMU SMV

NuSMV is mostly source compatible with the original version of SMV distributed at Carnegie Mellon University from which we started. The main problem you will run into is variable names in old programs that conflicts with new reserved words. In Appendix B [New reserved words], page 50 you will find the list of all the new reserved words of NuSMV with respect to CMU SMV.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
MODULE main
VAR
  y : boolean;
```

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

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

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

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

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

Appendix B New reserved words

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

`F`	It is the *sometimes* temporal operator. It is used to write LTL formulas.
`G`	It is the *always* temporal operator. It is used to write LTL formulas.
`X`	It is the *next time* temporal operator. It is used to write LTL formulas.
`LTLSPEC`	It is used to introduce LTL specifications.
`INVARSPEC`	It is used to introduce invariant specifications.
`IVAR`	It is used to introduce input variables.

Command Index

!

! 40

A

add_fairness 36
add_init 36
add_trans 36
alias 40
assign 35

B

build_model 24
build_variables 24

C

check_compute 27
check_invar 27
check_ltlspec 28
check_spec 27
check_trans 26
compute 28
compute_fairness 25
compute_reachable 27

D

dump_model 36
dynamic_var_ordering 38

E

echo 40
eval 35

F

flatten_hierarchy 23

G

go 25
goto_state 35

H

help 40
history 40
history substitution 40

P

pick_state 29
print_bdd_stats 38
print_current_state 35
print_iwls95info 25
print_iwls95options 25
print_reachable_states 27
print_usage 41
process_model 26

Q

quit 41

R

read_model 23
reset 41
restore_model 36

S

set 42
set_bdd_parameters 39
show_traces 30
show_vars 23
simulate 30
source 42
step 35

T

time 42

U

unalias 42
unset 42
usage 42

W

which 42
write_order 24

Variable Index

A

ag_only_search 27
autoexec 44

C

check_trans 26
conj_part_threshold 25

E

enable_reorder 37

F

filec 44
forward_search 26

H

history_char 41, 44

I

image_cluster_size, image_W{1,2,3,4} 25
image_verbosity 25
input_file 23
input_order_file 24

N

NuSMV_LIBRARY_PATH 43, 44, 46
nusmv_stderr 44
nusmv_stdin 44
nusmv_stdout 44

O

open_path 44
output_order_file 24

P

partition_method 24

R

reorder_method 37

S

shell_char 41, 44
showed_states 30

V

verbose_level 23

Index

-

-AG ... 45
-cp *part-limit* ... 45
-ctt ... 45
-dp ... 45
-f ... 45
-h ... 45
-help ... 45
-i *order-file* ... 45
-ii ... 45
-ils ... 45
-int ... 45
-is ... 45
-iwls95 *part-limit* ... 45
-load *cmd-file* ... 45
-m *method* ... 45
-o *order-file* ... 46
-r ... 46
-reorder ... 46
-s ... 46
-v *verbose-level* ... 46

.

.nusmvrc ... 43, 46

_

_process_selector_ ... 49

~

~/.nusmvrc ... 43, 46

A

administration commands ... 40

B

batch, running NuSMV ... 45

C

case expressions ... 10
comments in NuSMV language ... 9
CTL Specifications ... 19

D

DD package interface ... 37
DEFINE declaration ... 14

E

Examples ... 2
expressions ... 9

F

fair paths ... 18
fairness constraint ... 4
fairness constraints ... 18
fairness constraints declaration ... 18
FAIRNESS declarations ... 18

H

history substitution ... 40

I

identifiers ... 16
INIT declaration ... 14
input variables ... 12
interactive shell ... 22
interactive, running NuSMV ... 22
interface to DD Package ... 37
introduction ... 1
INVAR declaration ... 14
ISA declarations ... 15
IVAR ... 12

L

LTL Specifications ... 20

M

main module ... 18
master.nusmvrc ... 46
model compiling ... 23
model parsing ... 23
model reading ... 23
MODULE declarations ... 15

N

next expressions ... 11

O

options ... 45

P

process keyword ... 18
processes ... 18

R

Real Time CTL Specifications and Computations ... 20
running ... 18

S

`self` ... 17
set expressions ... 10
Shell configuration Variables ... 44
shell escape ... 41
simple expressions ... 9
Simulation Commands ... 29
state variables syntax ... 12
syntax ... 9

T

'`temp.ord`' ... 24
Traces Inspection Commands ... 35
`TRANS` declarations ... 13
type declaration ... 12
type specifiers ... 12

V

`VAR` declaration ... 12