NuSMV 2.4 User Manual

Roberto Cavada, Alessandro Cimatti,
Charles Arthur Jochim, Gavin Keighren,
Emanuele Olivetti, Marco Pistore, Marco Roveri
and Andrei Tchaltsev

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

Email: `nusmv@irst.itc.it`

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

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

Copyright ©1998-2005 by CMU and ITC-irst.

Contents

Chapter 1

Introduction

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

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

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

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

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

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

ules. Interfaces between modules have been provided. This reduces the effort needed to modify and extend NuSMV.

- **Quality of the implementation.** NuSMV is written in ANSI C, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. NuSMV uses the state of the art BDD package developed at Colorado University, and provides a general interface for linking with state-of the-art SAT solvers. This makes NuSMV very robust, portable, efficient, and easy to understand by people other than the developers.

This document is structured as follows.

- In Chapter 2 [Input Language], page 6 we define the syntax of the input language of NuSMV.
- In Chapter 3 [Running NuSMV interactively], page 41 the commands of the interaction shell are described.
- In Chapter 4 [Running NuSMV batch], page 87 we define the batch mode of NuSMV.

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

Chapter 2

Input Language

In this chapter we present the syntax and semantics of the input language of NuSMV.

Before going into the details of the language, let us give a few general notes about the syntax. In the syntax notations used below, syntactic categories (non-terminals) are indicated by `monospace font`, and tokens and character set members (terminals) by **bold font**. Grammar productions enclosed in square brackets ('`[]`') are optional while a vertical bar ('|') is used to separate alternatives in the syntax rules. Sometimes `one of` is used at the beginning of a rule as a shorthand for choosing among several alternatives. If the characters **|**, **[** and **]** are in bold font, they lose their special meaning and become regular tokens.

In the following, an `identifier` 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_$#\-**}. All characters and case in an identifier 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 and ignored by the parser.

The syntax rule for an `identifier` is:

```
identifier ::
        identifier_first_character
      | identifier identifier_consecutive_character

identifier_first_character :: one of
        A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
        a b c d e f g h i j k l m n o p q r s t u v w x y z _

identifier_consecutive_character ::
        identifier_first_character
      | digit
      | one of $ # \ -

digit :: one of 0 1 2 3 4 5 6 7 8 9
```

An `identifier` is always distinct from the NuSMV language reserved keywords which are:

MODULE, **DEFINE**, **CONSTANTS**, **VAR**, **IVAR**, **INIT**, **TRANS**, **INVAR**, **SPEC**, **CTLSPEC**, **LTLSPEC**, **PSLSPEC COMPUTE**, **INVARSPEC**, **FAIRNESS**,

JUSTICE, **COMPASSION**, **ISA**, **ASSIGN**, **CONSTRAINT**, **SIMPWFF**, **CTLWFF**, **LTLWFF**, **PSLWFF**, **COMPWFF**, **IN**, **MIN**, **MAX**, **process**, **array**, **of**, **boolean**, **integer**, **real**, **word**, **word1**, **bool**, **EX**, **AX**, **EF**, **AF**, **EG**, **AG**, **E**, **F**, **O**, **G**, **H**, **X**, **Y**, **Z**, **A**, **U**, **S**, **V**, **T**, **BU**, **EBF**, **ABF**, **EBG**, **ABG**, **case**, **esac**, **mod**, **next**, **init**, **union**, **in**, **xor**, **xnor**, **self**, **TRUE**, **FALSE**

To represent various values we will use `integer numbers` which are any non-empty sequence of decimal digits preceded by an optional unary minus

```
integer_number ::
        - digit
      | digit
      | integer_number digit
```

and `symbolic constants` which are `identifiers`

```
  symbolic_constant :: identifier
```

Examples of `integer numbers` and `symbolic constants` are `3, -14, 007, OK, FAIL, waiting, stop`. The values of `symbolic constants` and `integer numbers` do not intersect, with the exceptions of the reserved `symbolic constants` **TRUE** and **FALSE** which are equal to the `integer numbers` `1` and `0` respectively.

2.1 Types Overview

This section provides an overview of the types that are recognised by NuSMV.

2.1.1 Boolean

The boolean type comprises two `integer numbers` `0` and `1`, or their symbolic equivalents **FALSE** and **TRUE**.

2.1.2 Integer

The domain of the integer type is simply any whole number, positive or negative. At the moment, there are implementation-dependent constraints on the this type and `integer numbers` can only be in the range $-2^{32}+1$ to $2^{32}-1$ (more accurately, these values are equivalent to the C/C++ macros `INT_MIN` and `INT_MAX`).

2.1.3 Enumeration Types

An enumeration type is a type specified by full enumerations of all the values that the type comprises. For example, the enumeration of values may be {`stopped, running, waiting, finished`}, {`2, 4, -2, 0`}, {`FAIL, 1, 3, 7, OK`}, etc. All elements of an enumeration have to be unique although the order of elements is not important.

However, in the NuSMV type system, expressions cannot be of actual enumeration types, but of their simplified and generalised versions only. Such generalised enumeration types do not contain information about the exact values constituting the types, but only the flag whether all values are `integer numbers`, `symbolic constants` or both. Below only generalised versions of enumeration types are explained.

The symbolic enum type covers enumerations containing only `symbolic constants`. For example, the enumerations {`stopped, running, waiting`} and {`FAIL, OK`} belong to the symbolic enum type.

There is also a integers-and-symbolic enum type. This type comprises enumerations which contain *both* `integer numbers` *and* `symbolic constants`, for example, {`-1, 1, waiting`}, {`0, 1, OK`}, {`running, stopped, waiting, 0`}.

Another enumeration type is integer enum. Example of enumerations of integers are {`2, 4, -2, 0`} and {`-1, 1`}. In the NuSMV type system an expression of the type integer enum is always converted to the type integer. Explaining the type of expression we will always use the type integer instead of integer enum.

The values in an enumeration may potentially contain only the boolean values, for example, {`0, 1`} or {`FALSE, TRUE`}. In this case the type will be boolean (see Section 2.1.1 [Boolean Type], page 7).

To summarise, we actually deal only with two enumeration types: symbolic enum and integers-and-symbolic enum. These types are distinguishable and have different operations allowed on them.

2.1.4 Word

The word[•] types are used to model arrays of bits (booleans) which allow bitwise logical and arithmetic operations. These types are distinguishable by their width. For example, the type word[3] represents arrays of three bits, and the type word[7] represents arrays of seven bits. Note that the width has to be greater than zero.

2.1.5 Array

Arrays are declared with a lower and upper bound for the index, and the type of the elements in the array. For example,

```
array 0..3 of boolean;
array 10..20 of {OK, y, z};
array 1..8 of array -1..2 of word[5];
```

The type `array 1..8 of array -1..2 of word[5]` means an array of 8 elements (from 1 to 8), each of which is an array of 4 elements (from -1 to 2) that are 5-bit-long words. The use of arrays in expressions are quite limited. See 2.3.11 for more information.

2.1.6 Set Types

set types are used to identify expressions representing a set of values. There are four set types : boolean set, integer set, symbolic set, integers-and-symbolic set. The set types can be used in a very limited number of ways. In particular, a variable cannot be of a set type. Only `range constant` and **`union`** operator can be used to create an expression of a set type, and only **`in`**, **`case`** and assignment[1] expressions can have imediate operands of a set type.

Every set type has a counterpart among other types. In particular,

the counterpart of a boolean set type is boolean,

the counterpart of a integer set type is integer,

the counterpart of a symbolic set type is symbolic enum,

the counterpart of a integers-and-symbolic set type is integers-and-symbolic enum.

Some types such as word[•] do not have a set type counterpart.

2.1.7 Type Order

Figure 2.1 depicts the order existing between types in NuSMV.

[1]For more information on these operators see pages 11, 17, 17, 17 and 23, respectively.

$$\begin{array}{lll}
\text{boolean} & & \text{word[1]} \\
\downarrow & & \\
\text{integer} \quad \text{symbolic enum} & & \text{word[2]} \\
\downarrow \qquad\qquad \downarrow & & \text{word[3]} \\
\text{integers-and-symbolic enum} & & \dots
\end{array}$$

$$\begin{array}{l}
\text{boolean set} \\
\downarrow \\
\text{integer set} \quad \text{symbolic set} \\
\downarrow \qquad\qquad \downarrow \\
\text{integers-and-symbolic set}
\end{array}$$

Figure 2.1: The ordering on the types in NuSMV

It means, for example, that boolean is less than integer, integer is less than integers-and-symbolic enum, etc. The word[•] types are not comparable with any other type or between each other. Any type is equal to itself.

Note that enumerations containing only `integer numbers` have the type integer (unless the only elements are `1` and `0` in which case the type is boolean).

2.2 Expressions

The previous versions of NuSMV (prior to 2.4.0) did not have the type system and as such expressions were untyped. In the current version all expressions are typed and there are constraints on the type of operands. Therefore, an expression may now potentially violate the type system, i.e. be erroneous.

To maintain backward compatibility, there is a new system variable called `backward_compatibility` (and a correponding `-old` command line option) that disables a few new features of version 2.4 to keep backward compatibility with old version of NuSMV. In particular, if this system variable is set then type violations caused by expressions of old types (i.e. enumeration type, boolean and integer) will be ignored by the type checker, instead, warnings will be printed out. See description at page 42 for further information.

If additionally, the system variable `type_checking_warning_on` is *un*set, then even these warnings will not be printed out.

2.2.1 Implicit Type Conversion

In certain expressions NuSMV can implicitly convert operands from one type to another. Such implicit conversion can be performed from a smaller type to a bigger one (in accordance with the ordering depicted in Figure 2.1). This means, for example, that word[•] types cannot be implicitly converted to other types or each other implicitly, while the type boolean can be implicitly converted to integer or integers-and-symbolic enum.

Also in some expressions operands may be converted from one type to its set type counterpart (see 2.1.6). For example, integer can be converted to integer set type.

2.2.2 Constant Expressions

A `constant` can be a boolean, integer, symbolic, word or range constant.

```
constant ::
```

```
        boolean_constant
      | integer_constant
      | symbolic_constant
      | word_constant
      | range_constant
```

Boolean Constant

A `boolean constant` is one of the `integer numbers` `0` and `1` or their symbolic equivalents **`FALSE`** and **`TRUE`**. The type of a `boolean constant` is boolean.

```
boolean_constant :: one of
        0 1 FALSE TRUE
```

Integer Constant

An `integer constant` is an `integer number` with the exception of `0` and `1` which are taken to be `boolean constants`. The type of an `integer constant` is integer.

```
integer_constant :: integer_number
```

Symbolic Constant

A `symbolic constant` is syntactically an `identifier` and indicates a unique value.

```
symbolic_constant :: identifier
```

The type of a `symbolic constant` is symbolic enum. See Section 2.3.14 [Namespaces], page 29 for more information about how `symbolic constants` are distinguished from other `identifiers`, i.e. variables, defines, etc.

Word Constant

`Word constants` begin with digit `0`, followed by one of the characters `b`/`B` (binary), `o`/`O` (octal), `d`/`D` (decimal) or `h`/`H` (hexadecimal) which gives the base that the actual constant is in. Next comes an optional decimal integer giving the number of bits, then the character _, and lastly the constant value itself. The type of a `word constant` is word[N], where `N` is the width of the constant. For example:

`0b5_10111`	has type word[5]
`0o6_37`	has type word[6]
`0d11_9`	has type word[11]
`0h12_a9`	has type word[12]

The number of bits can be skipped, in which case the width is automatically calculated from the number of digits in the constant and its base. It may be necessary to explicitly give leading zeroes to make the type correct — the following are all equivalent declarations of the integer constant `11` as a word of type word[8]:

```
0d8_11
0b8_1011
0b_00001011
0h_0b
0h8_b
```

The syntactic rule of the `word constant` is the following:

```
word_constant ::
        0 word_base [word_width] _ word_value

word_width ::
        integer_number       -- a number greater than zero

word_base ::
        b | B | o | O | d | D | h | H

word_value ::
        hex_digit
      | word_value hex_digit
      | word_value _

hex_digit :: one of
        0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

Note that

- The width of a word must be a number strictly greater than 0.
- Decimal `word constants` *must* be declared with the width specifier, since the number of bits needed for an expression like `0d_019` is unclear.
- Digits are restricted depending on the base the constant is given in.
- Digits can be separated by the underscore character ("_") to aid clarity, for example `0b_0101_1111_1100` which is equivalent to `0b_010111111100`.
- The number of bits in `word constant` has an implementation limit which for most systems is 64 bits.

Range Constant

A `range constant` specifies a set of consecutive integer numbers. For example, a constant `-1..5` indicates the set of numbers `-1, 0, 1, 2, 3, 4` and `5`. Other examples of `range constant` can be `1..10, -10..-10, 1..300`. The syntactic rule of the `range constant` is the following:

```
range_constant ::
        integer_number .. integer_number
```

with an additional constraint that the first integer number must be less than or equal to the second integer number. The type of a `range constant` is integer set.

2.2.3 Basic Expressions

A basic expression is the most common kind of expression used in NuSMV.

```
basic_expr ::
      constant                        -- a constant
    | variable_identifier             -- a variable identifier
    | define_identifier               -- a define identifier
    | ( basic_expr )
    | ! basic_expr                    -- logical or bitwise NOT
    | basic_expr & basic_expr         -- logical or bitwise AND
    | basic_expr | basic_expr         -- logical or bitwise OR
    | basic_expr xor basic_expr       -- logical or bitwise exclusive OR
```

```
    | basic_expr xnor basic_expr    -- logical or bitwise NOT exclusive OR
    | basic_expr -> basic_expr      -- logical or bitwise implication
    | basic_expr <-> basic_expr     -- logical or bitwise equivalence
    | basic_expr = basic_expr       -- equality
    | basic_expr != basic_expr      -- inequality
    | basic_expr < basic_expr       -- less than
    | basic_expr > basic_expr       -- greater than
    | basic_expr <= basic_expr      -- less than or equal
    | basic_expr >= basic_expr      -- greater than or equal
    | - basic_expr                  -- integer unary minus
    | basic_expr + basic_expr       -- integer addition
    | basic_expr - basic_expr       -- integer subtraction
    | basic_expr * basic_expr       -- integer multiplication
    | basic_expr / basic_expr       -- integer division
    | basic_expr mod basic_expr     -- integer remainder
    | basic_expr >> basic_expr      -- bit shift right
    | basic_expr << basic_expr      -- bit shift left
    | basic_expr :: basic_expr      -- word concatenation
    | basic_expr [ integer_number : integer_number ]
                                    -- word bits selection
    | word1 ( basic_expr )          -- boolean to word[1] convertion
    | bool ( basic_expr )           -- word[1] to boolean convertion
    | basic_expr union basic_expr   -- union of set expressions
    | { set_body_expr }             -- set expression
    | basic_expr in basic_expr      -- inclusion in a set expression
    | case_expr                     -- a case expression
    | basic_next_expr               -- a next expression
```

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

```
!
[ : ]
::
- (unary minus)
*   /
+   -
mod
<<   >>
union
in
=   !=   <   >   <=   >=
&
|   xor   xnor
<->
->
```

Operators of equal precedence associate to the left, except **->** that associates to the right. The constants and their types are explained in Section 2.2.2 [Constant Expressions], page 9.

Variables and Defines

A `variable_identifier` and `define_identifier` are expressions which identify a variable or a define, respectively. Their syntax rules are:

```
define_identifier :: complex_identifier
```

```
variable_identifier :: complex_identifier
```

The syntax and semantics of `complex_identifiers` are explained in Section 2.3.11 [References to Module Components], page 27. All defines and variables referenced in expressions should be declared. All identifiers (variables, defines, symbolic constants, etc) can be used prior to their definition, i.e. there is no constraint on order such as in C where a declaration of a variable should always be placed in text above the variable use. See more information about define and variable declarations in Section 2.3.2 [DEFINE Declarations], page 21 and Section 2.3.1 [Variable Declarations], page 19.

A define is a kind of macro. Every time a define is met in expressions, it is substituted by the expression associated with this define. Therefore, the type of a define is the type of the associated expression in the current context.

`variable_identifier` represents state and input variables. The type of a variable is specified in its declaration. For more information about variables, see Section 2.3 [Definition of the FSM], page 19, Section 2.3.1 [State Variables], page 20 and Section 2.3.1 [Input Variables], page 20. Since a `symbolic constant` is syntactically indistinguishable from `variable_identifiers` and `define_identifiers`, a symbol table is used to distinguish them from each other.

Parentheses

Parentheses may be used to group expressions. The type of the whole expression is the same as the type of the expression in the parentheses.

Logical and Bitwise !

The *signature* of the logical and bitwise NOT operator **!** is:

! : boolean → boolean
  : word[N] → word[N]

This means that the operation can be applied to boolean or word[•] operands. The type of the whole expression is the same as the type of the operand. If the operand is not boolean or word[•] then the expression violates the type system and NuSMV will throw an error.

Logical and Bitwise **&**, **|**, **xor**, **xnor**, **->**, **<->**

Logical and bitwise binary operators **&** (AND), **|** (OR), **xor** (exclusive OR), **xnor** (negated exclusive OR), **->** (implies) and **<->** (if and only if) are similar to the unary operator **!**, except that they take two operands. Their signature is:

&, **|**, **xor**, **xnor**, **->**, **<->** : boolean * boolean → boolean
  : word[N] * word[N] → word[N]

the operands can be of boolean or word[•] type, and the type of the whole expression is the type of the operands. Note that both word[•] operands should have the same width.

Equality (=) and Inequality (!=)

The operators **=** (equality) and **!=** (inequality) have the following signature:

=, **!=** : boolean * boolean → boolean
: integer * integer → boolean
: symbolic enum * symbolic enum → boolean
: integers-and-symbolic enum * integers-and-symbolic enum → boolean
: word[N] * word[N] → boolean
: boolean * word[1] → boolean
: word[1] * boolean → boolean

Before checking the expression for being correctly typed, implicit type conversion can be carried out on *one* of the operands. For example, in the expression

```
TRUE = 5
```

the left operand is of type boolean and the right one is of type integer. Though the signature of the operation does not have a boolean * integer rule, the expression is correct, because after implicit type conversion on the left operand the types of the operands will be integer * integer, which is a valid signature for the **=** operator.
This is also true if one of the operands is of type word[1] and the other one is of the type boolean. In this case, one of the operands is converted to the type of the other one and then the equality is checked[2].

Relational Operators >, <, >=, <=

The relational operators **>** (greater than), **<** (less than), **>=** (greater than or equal to) and **<=** (less than or equal to) have the following signature:

>, **<**, **>=**, **<=** : boolean * boolean → boolean
: integer * integer → boolean
: word[N] * word[N] → boolean
: boolean * word[1] → boolean
: word[1] * boolean → boolean

Before checking the expression for being correctly typed, implicit type conversion can be carried out on *one* of the operands.

boolean and word[•] types are implicitly converted to their integer equivalents before the result of these operations is calculated.

Arithmetic Operators +, −, *, /

The arithmetic operators **+** (addition), **−** (subtraction), ***** (multiplication) and **/** (division) have the following signature:

+, **−**, *****, **/** : boolean * boolean → integer
: integer * integer → integer
: word[N] * word[N] → word[N]
− : integer → integer
: word[N] → word[N]

Before checking the expression for being correctly typed, the implicit type conversion can be applied to *one* of the operands. The boolean operands are converted to the integer type before performing the arithmetic operation. If the operators are applied to a word[N] type, then the operations are performed modulo 2^N.

The result of the **/** operator is the quotient from the division of the first operand by the second. When integers are divided, the result of the **/** operator is the algebraic quotient with any fractional part discarded (this is often called "truncation towards zero"). If the quotient

[2] It is does not matter which operand is converted — the result will be the same.

a/b is representable, the expression (a/b)*b + (a mod b) shall equal a. If the value of the second operand is zero, the behavior is undefined and an error is thrown by NuSMV. The semantics is equivalent to the corresponding one of C/C++ languages.

In the versions of NuSMV prior to 2.4.0 the semantics of division was different. See page 15 for more detail.

Remainder Operator mod

The result of the **mod** operator is the algebraic remainder of the division. If the value of the second operand is zero, the behavior is undefined and an error is thrown by NuSMV.

The signature of the remainder operator is:

mod : integer * integer → integer
: word[N] * word[N] → word[N]
: integer * **2** → boolean

Note that when the left operand is an integer and the right one is a constant **2** then the type of the expression is Boolean. In such a way "mod 2" expressions can be used as boolean expressions to check whether the left operand is even or odd. Note also that the signature does not allow the operands to be boolean since such expressions are useless. For example, if the left operand is a boolean expression and on the right is 3 the result will always be equal to the left operand. Actually, in some cases it can be useful to allow boolean operands (for example, if text of expressions is automatically generated) therefore before applying the operation NuSMV converts boolean operand to integer but prints out a warning in this case. In all other respects, the semantics of **mod** operator is equivalent to the corresponding operator **%** of C/C++ languages. Thus if the quotient a/b is representable, the expression (a/b)*b + (a mod b) shall equal a.

Note: in older versions of NuSMV (priori 2.4.0) the semantics of quotient and remainder were different. Having the division and remainder operators / and mod be of the current, i.e. C/C++'s, semantics the older semantics of division was given by the formula:

IF (a mod b < 0) THEN (a / b $-$ 1) ELSE (a / b)

and the semantics of remainder operator was given by the formula:

IF (a mod b < 0) THEN (a mod b $+$ b) ELSE (a mod b)

Note that in both versions the equation (a/b)*b + (a mod b) = a holds. For example, in the current version of NuSMV the following holds:

7/5 = 1	7 mod 5 = 2
-7/5 = -1	-7 mod 5 = -2
7/-5=-1	7 mod -5 = 2
-7/-5=1	-7 mod -5 = -2

whereas in the older versions on NuSMV the equations were

7/5 = 1	7 mod 5 = 2
-7/5 = -2	-7 mod 5 = 3
7/-5=-1	7 mod -5 = 2
-7/-5=0	-7 mod -5 = -7

When supplied, the command line option -old_div_op switches the semantics of division and remainder to the old one.

Shift Operators <<, >>

The signature of the shift operators is:

<<, **>>** : word[N] * integer → word[N]
: word[N] * word[M] → word[N]

Before checking the expression for being correctly typed, the right operand can be implicitly converted from boolean to integer type.

Left shift **<<** and and right shift **>>** operations shift bits to the left and right respectively. A shift by N bits is equivalent to N shifts by 1 bit. A bit shifted behind the word bound is lost.

During shifting the word is padded with zeros.

For instance,

0b4_0001 **<<** 2 is equal to
0b4_0100 **<<** 1 is equal to
0b4_1000 **<<** 0 is equal to
0b4_1000

It has to be remarked that the shifting requires the right operand to be greater or equal to zero and less then the width of the word it is applied to. NUSMV raises an error if a shift is attempted that does not satisfy this restriction.

Bit Selection Operator `[ : ]`

The bit selection operator extracts consecutive bits from a word[•] expression, resulting in a new word[•] expression. This operation always decreases the width of word[•] or leaves it intact. The left expression in the brackets is the high bound and the right one is the low bound. The high bound must be greater than or equal to the low bound. The bits count from 0. The result of the operations is a word[•] value consisting of the consecutive bits beginning from the high bound of the operand down to, and including, the low bound bit. For example, 0b7_1011001[4:1] extracts bits 1 through 4 (including 1st and 4th bits) and is equal to 0b4_1100. 0b3_101[0:0] extracts bit number 0 and is equal to 0b1_1.

The signature of the bit selection operator is:

`[ : ]` : word[N] * integer$_{high}$ * integer$_{low}$ $\rightarrow$ word[integer$_{high}$ $-$ integer$_{low}$ $+ 1$]

where $0 \leq$ integer$_{low}$ $\leq$ integer$_{high}$ $<$ N

Word Concatenation Operator `::`

The concatenation operator joins two words together to create a larger word type. The operator itself is two colons (`::`), and its signature is as follows:

`::` : word[M] * word[N] $\rightarrow$ word[M+N]
: boolean * word[N] $\rightarrow$ word[N+1]
: word[N] * boolean $\rightarrow$ word[N+1]

The left-hand operand will make up the upper bits of the new word, and the right-hand operand will make up the lower bits. For example, given the two words `w1 := 0b4_1101` and `w2 := 0b2_00`, then the result of `w1::w2` is `0b6_110100`.

Boolean and word[1] Explicit Conversions

`bool` converts a word[1] to a boolean, while **`word1`** converts a boolean to a word[1].

The signatures of these conversion operators are:

`bool` : word[1] $\rightarrow$ boolean
`word1` : boolean $\rightarrow$ word[1]

The conversion obeys the following table:

`bool`(0b1_0) = 0
`bool`(0b1_1) = 1
`word1`(0) = 0b1_0
`word1`(1) = 0b1_1

Set Expressions

The set expression is an expression defining a set of boolean, integer and symbolic enum values. A set expression can be created with the **`union`** operator. For example, `1 union 0` specifies the set of values `1` and `0`. One or both of the operands of **`union`** can be sets. In this case, **`union`** returns a union of these sets. For example, expression `(1 union 0) union 3` specifies the set of values `1`, `0` and `-3`.

Note that there cannot be a set of sets in NuSMV. Sets can contain only singleton values, but not other sets.

The signature of the **`union`** operator is:

`union` : boolean set * boolean set → boolean set
: integer set * integer set → integer set
: symbolic set * symbolic set → symbolic set
: integers-and-symbolic set * integers-and-symbolic set
→ integers-and-symbolic set

Before checking the expression for being correctly typed, if it is possible, both operands are converted to their counterpart set types [3], which virtually means converting individual values to singleton sets. Then both operands are implicitly converted to a minimal type that covers both operands. If after these manipulations the operands do not satisfy the signature of **`union`** operator, an error is raised by NUSMV.

There is also another way to write a set expression by enumerating all its values between curly brackets. The syntactic rule for the values in curly brackets is:

```
set_body_expr ::
        basic_expr
      | set_body_expr , basic_expr
```

Enumerating values in curly brackets is semantically equivalent to writing them connected by **`union`** operators. For example, expression `{exp1, exp2, exp3}` is equivalent to `exp1 union exp2 union exp3`. Note that according to the semantics of **`union`** operator, expression `{{1, 2}, {3, 4}}` is equivalent to `{1, 2, 3, 4}`, i.e. there is no actually set of sets.

Set expressions can be used only as operands of **`union`** and **`in`** operations, and as the right operand of **`case`** expressions and assignments. In all other places the use of set expressions is prohibited.

Inclusion Operator `in`

The inclusion operator '**`in`**' tests the left operand for being a subset of the right operand. If either operand is a number or a symbolic value instead of a set, it is coerced to a singleton set.

The signature of the **`in`** operator is:

`in` : boolean set * boolean set → boolean
: integer set * integer set → boolean
: symbolic set * symbolic set → boolean
: integers-and-symbolic set * integers-and-symbolic set → boolean

Similar to **`union`** operation, before checking the expression for being correctly typed, if it is possible, both operands are converted to their counterpart set types [4]. Then, if required, implicit type conversion is carried out on *one* of the operands.

Case Expressions

A case expression has the following syntax:

[3] See 2.1.6 for more information about the set types and their counterpart types

[4] See 2.1.6 for more information about the set types and their counterpart types

```
case_expr :: case case_body esac

case_body ::
        basic_expr : basic_expr ;
      | case_body basic_expr : basic_expr ;
```

A `case_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` (`TRUE`). For example, the result of the expression

```
case
  left_expression_1 : right_expression_1 ;
  left_expression_2 : right_expression_2 ;
  ...
  left_expression_N : right_expression_N ;
esac
```

is `right_expression_k` such that for all i from 0 to $k-1$, `left_expression_i` is 0, and `left_expression_k` is 1. It is an error if all expressions on the left hand side evaluate to 0.

The type of expressions on the left hand side must be boolean. If one of the expression on the right is of a set type then, if it is possible, all remaining expressions on the right are converted to their counterpart set types [5]. The type of the whole expression is such a minimal type[6] that all of the expressions on the right (after possible convertion to set types) can be implicitly converted to this type. If this is not possible, NuSMV throws an error.

Basic Next Expression

`Next expressions` refer to next state variables. For example, if a variable **`v`** is a state variable, then **`next(v)`** refers to that variable **`v`** in the next time step. A **`next`** applied to a complex expression is a shorthand method of applying **`next`** to all the variables in the expressions recursively. Example: **`next`**`((1 + a) + b)` is equivalent to `(1 + `**`next`**`(a)) + `**`next`**`(b)`. Note that the **`next`** operator cannot be applied twice, i.e. **`next`**`(`**`next`**`(a))` is *not* allowed.

The syntactic rule is:

```
basic_next_expr :: next ( basic_expr )
```

A `next expression` does not change the type.

2.2.4 Simple and Next Expressions

`Simple_expressions` are expressions built only from current state variables. Therefore, the `simple_expression` cannot have a **`next`** operation inside and the syntax of `simple_expressions` is as follows:

```
simple_expr :: basic_expr
```

with the alternative `basic_next_expr` *not* allowed. `Simple_expressions` can be used to specify sets of states, for example, the initial set of states. The `next_expression` relates current and next state variables to express transitions in the FSM. The `next_expression` *can* have **`next`** operation inside, i.e.

```
next_expr :: basic_expr
```

with the alternative `basic_next_expr` allowed.

[5] See 2.1.6 for information on set types and their counterpart types

[6] See Section 2.1.7 [Type Order], page 8 for the information on the order of types.

2.3 Definition of the FSM

We consider a Finite State Machine (FSM) described in terms of *state variables* and *input variables*, which may assume different values in different *states*, of a *transition relation* describing how inputs leads from one state to possibly many different states, and of *Fairness conditions* that describe constraints on the valid paths of the execution of the FSM. In this document, we distinguish among constraints (used to constrain the behavior of a FSM, e.g. a modulo 4 counter increments its value modulo 4), and specifications (used to express properties to verify on the FSM (e.g. the counter reaches value 3).

In the following it is described how these concepts can be declared in the NuSMV language.

2.3.1 Variable Declarations

A variable can be an input or a state variable. The declaration of a variable specifies the variable's type with the help of type specifier.

Type Specifiers

A `type specifier` has the following syntax:

```
type_specifier ::
        simple_type_specifier
      | module_type_specifier

simple_type_specifier ::
        boolean
      | word [ integer_number ]
      | { enumeration_type_body }
      | integer_number .. integer_number
      | array integer_number .. integer_number
        of simple_type_specifier

enumeration_type_body ::
        enumeration_type_value
      | enumeration_type_body , enumeration_type_value

enumeration_type_value ::
        symbolic_constant
      | integer_number
```

There are two kinds of `type specifier`: a `simple type specifier` and a `module type specifier`. The `module type specifier` is explained later in Section 2.3.10 [MODULE Instantiations], page 26. The `simple type specifier` comprises boolean type, integer type, enumeration types, word[] and arrays types.

The boolean type is specified by the keyword **`boolean`**.

A enumeration type is specified by full enumeration of all the values the type comprises. For example, possible enumeration type specifiers are {`0,2,3,-1`}, {`1,0, OK`}, {`OK, FAIL, running`}. The values in the list are enclosed in curly brackets and separated by commas. The values may be `integer numbers`, `symbolic constants`, or both. All values in the list should be distinct from each other, although the order of values is not important. Note that the `symbolic constants` **`TRUE`** and **`FALSE`** are just symbolic representations of the `integer numbers` `1` and `0`, respectively.

If the list of values in the enumeration type specifier consists of just the two values `1` and `0` then the type it represents is boolean. For example, `type specifiers` {`TRUE, FALSE`} and boolean are equivalent.

Note, expressions cannot be of the actual enumeration types, but only the simplified versions of enumeration types, such as symbolic enum and integers-and-symbolic enum.

A `type specifier` can be given by two integer numbers separated by `..` (<TWO DOTS>), for example, `-1..5`. This is just a shorthand for a enumeration type containing the list of `integer numbers` from the range given in `type specifier`. For example, the `type specifiers` `-1..5` and `{-1,0,1,2,3,4,5}` are equivalent. Note that the number on the left from the two dots must be less than or equal to the number on the right.

The word[•] type is specified by the keyword **`word`** with an `integer number` supplied in square brackets. This number must be greater than zero. The purpose of the word types is to offer integer and bitwise arithmetic.

An array type is denoted by a sequence of the keyword **`array`**, an `integer number` specifying the lower bound of the array index, two dots `..`, an `integer number` specifying the upper bound of the array index, the keyword **`of`**, and the type of array's elements. The elements can themselves be arrays.

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 var_list

var_list :: identifier : type_specifier ;
          | var_list identifier : type_specifier ;
```

A `variable declaration` specifies the identifier of the variables and its type. A variable can take the values only from the domain of its type. In particular, a variable of a enumeration type may take only the values enumerated in the `type specifier` of the declaration.

Input Variables

`IVAR` s (input variables) are used to label transitions of the Finite State Machine. The difference between the syntax for the input and state variables declarations is the keyword indicating the beginning of a declaration:

```
ivar_declaration :: IVAR ivar_list
ivar_list :: identifier : simple_type_specifier ;
          | ivar_list identifier : simple_type_specifier ;
```

Another difference between input and state variables is that input variables cannot be instances of modules. The usage of input variables is more limited than the usage of state variables which can occur everywhere both in the model and specifications. Namely, input variables cannot occur in:

- Left-side of assignments. For example all these assignments are not allowed:

  ```
  IVAR i :  boolean;
  ASSIGN
  init(i) := TRUE;
  next(i) := FALSE;
  ```

- `INIT` statements. For example:

  ```
  IVAR i :  boolean;
  VAR s :  boolean;
  INIT i = s
  ```

- Scope of `next` expressions. For example:

  `IVAR i : boolean;`
  `VAR s : boolean;`
  `TRANS i -> s` – this is allowed
  `TRANS next(i -> s)` – this is NOT allowed

- Some specification kinds: `CTLSPEC`, `SPEC`, `INVARSPEC`, `COMPUTE`, `PSLSPEC`. For example:

  `IVAR i : boolean;`
  `VAR s : boolean;`
  `SPEC AF (i -> s)` – this is NOT allowed
  `LTLSPEC F (X i -> s)` – this is allowed

- Anywhere in the FSM when checking invariants with BMC and the "DUAL" algorithm. See at page 66 for further information.

Examples

Below are examples of input and state variable declarations:

```
VAR a : boolean;
VAR b : 0..1;
IVAR c : {TRUE, FALSE};
```

The variables `a`, `b` are state variables, and `c` is an input variable; all of them are of boolean type. In the following examples:

```
VAR d : {stopped, running, waiting, finished};
VAR e : {2, 4, -2, 0};
VAR f : {1, a, 3, d, q, 4};
```

the variables `d`, `e` and `f` are of enumeration types, and all their possible values are specified in the `type specifiers` of their declarations.

```
VAR g : word[3];
```

The variable `g` is of 3-bits-wide word type (i.e word[3]).

```
VAR k : array -1..1 of array {0, TRUE};
```

The variable `k` is an array of boolean elements with indexes -1, 0 and 1.

2.3.2 DEFINE Declarations

In order to make descriptions more concise, a symbol can be associated with a common expression, and a **DEFINE** declaration introduces such a symbol. The syntax for this kind of declaration is:

```
define_declaration :: DEFINE define_body

define_body :: identifier := simple_expr ;
             | define_body identifier := simple_expr ;
```

DEFINE associates an `identifier` on the left hand side of the `':='` with an expression on the right side. A define statement can be considered as a macro. Whenever a define identifier occurs in an expression, the identifier is syntactically replaced by the expression it is associated with. The associated expression is always evaluated in the context of the expression where the identifier is met (see Section 2.3.15 [Context], page 30 for an explanation of contexts). Forward references to defined symbols are allowed but circular definitions are not, and result in an error. The difference between defined symbols and variables is that while variables are statically typed, definitions are not.

2.3.3 **CONSTANTS** Declarations

CONSTANTS declarations allow the user to explicitly declare symbolic constants that might occur or not within the FSM that is being defined. **CONSTANTS** declarations are expecially useful in those conditions that require symbolic constants to occur only in **DEFINEs** body (e.g. in generated models). For an example of usage see also the command write_boolean_model. A constant is allowed to be declared multiple times, as after the first declaration any further declaration will be ignored. **CONSTANTS** declarations are an extension of the original SMV grammar, and they are supported since NuSMV 2.4. The syntax for this kind of declaration is:

```
constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier
             | constants_body  , identifier
```

2.3.4 **INIT** Constraint

The set of initial states of the model is determined by a boolean expression under the **INIT** keyword. The syntax of an INIT constraint is:

```
init_constrain :: INIT simple_expr [;]
```

Since the expression in the INIT constraint is a simple_expression, it cannot contain the **next()** operator. The expression also has to be of type boolean. If there is more than one INIT constraint, the initial set is the conjunction of all of the INIT constraints.

2.3.5 **INVAR** Constraint

The set of invariant states can be specified using a boolean expression under the **INVAR** keyword. The syntax of an INVAR constraint is:

```
invar_constraint :: INVAR simple_expr [;]
```

Since the expression in the INVAR constraint is a simple_expression, it cannot contain the **next()** operator. If there is more than one INVAR constraint, the invariant set is the conjunction of all of the INVAR constraints.

2.3.6 **TRANS** Constraint

The transition relation 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 expression, introduced by the **TRANS** keyword. The syntax of a TRANS constraint is:

```
trans_constraint :: TRANS next_expr [;]
```

It is an error for the expression to be not of the boolean type. If there is more than one TRANS constraint, the transition relation is the conjunction of all of TRANS constraints.

2.3.7 ASSIGN Constraint

An assignment has the form:

```
assign_constraint :: ASSIGN assign_list

assign_list :: assign ;
             | assign_list assign ;

assign ::
    complex_identifier          := simple_expr
  | init ( complex_identifier ) := simple_expr
  | next ( complex_identifier ) := next_expr
```

On the left hand side of the assignment, `identifier` denotes the current value of a variable, '**init(**`identifier`**)**' denotes its initial value, and '**next(**`identifier`**)**' denotes its value in the next state. If the expression on the right hand side evaluates to a not-set expression such as `integer number` 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.

Semantically assignments can be expressed using other kinds of constraints:

`ASSIGN a := exp;` is equivalent to `INVAR a in exp;`
`ASSIGN init(a) := exp;` is equivalent to `INIT a in exp;`
`ASSIGN next(a) := exp;` is equivalent to `TRANS next(a) in exp;`

Notice that, an additional constraint is forced when assignments are used with respect to their corresponding constraints counterpart: when a variable is assigned a value that it is not an element of its declared type, an error is raised.

The allowed types of the assignment operator are:

:= : boolean * boolean
: boolean * boolean set
: integer * integer
: integer * integer set
: symbolic enum * symbolic enum
: symbolic enum * symbolic set
: integers-and-symbolic enum * integers-and-symbolic enum
: integers-and-symbolic enum * integers-and-symbolic set
: word[N] * word[N]
: boolean * word[1]
: word[1] * boolean

Before checking the assignment for being correctly typed, the implicit type conversion can be applied to the *right* operand.

Rules for assignments

Assignments describe a system of equations that say how the FSM evolves through time. With an arbitrary set of equations there is no guarantee that a solution exists or that it is unique. We tackle this problem by placing certain restrictive syntactic rules on the structure of assignments, thus guaranteeing that the program is implementable.

The restriction rules for assignments are:

- **The single assignment rule** – each variable may be assigned only once.

- **The circular dependency rule** – a set of equations must not have "cycles" in its dependency graph not broken by delays.

The single assignment rule disregards conflicting definitions, and can be formulated as: one may either assign a value to a variable "`x`", or to "**`next(`**`x`**`)`**" and "**`init(`**`x`**`)`**", but not both. For instance, the following are legal assignments:

Example 1	`x := expr1 ;`
Example 2	`init( x ) := expr1 ;`
Example 3	`next( x ) := expr1 ;`
Example 4	`init( x ) := expr1 ;` `next( x ) := expr2 ;`

while the following are illegal assignments:

Example 1	`x := expr1 ;` `x := expr2 ;`
Example 2	`init( x ) := expr1 ;` `init( x ) := expr2 ;`
Example 3	`x := expr1 ;` `init( x ) := expr2 ;`
Example 4	`x := expr1 ;` `next( x ) := expr2 ;`

If we have an assignment like `x := y ;`, then we say that `x` *depends on* `y`. A *combinatorial loop* is a cycle of dependencies not broken by delays. For instance, the assignments:

```
x := y;
y := x;
```

form a combinatorial loop. Indeed, there is no fixed order in which we can compute `x` and `y`, since at each time instant the value of `x` depends on the value of `y` and vice-versa. We can introduce a "unit delay dependency" using the **`next()`** operator.

```
      x := y;
next(y) := x;
```

In this case, there is a unit delay dependency between `x` and `y`. A combinatorial loop is a cycle of dependencies whose total delay is zero. In NUSMV combinatorial loops are illegal. This guarantees that for any set of equations describing the behavior of variable, there is at least one solution. There might be multiple solutions in the case of unassigned variables or in the case of non-deterministic assignments such as in the following example,

```
    next(x) := case x=1 : 1;
                    1 : {0,1};
               esac;
```

2.3.8 `FAIRNESS` Constraints

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

NUSMV supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula `f`, which is assumed to be true infinitely often in all the fair paths. In NUSMV, justice constraints are identified by keywords

JUSTICE and, for backward compatibility, **FAIRNESS**. A compassion constraint consists of a pair of formulas `(p,q)`; if property `p` is true infinitely often in a fair path, then also formula `q` has to be true infinitely often in the fair path. In NuSMV, compassion constraints are identified by keyword **COMPASSION**. [7] If compassion constraints are used, then the model must not contain any input variables. Currently, NuSMV does not enforce this so it is the responsibility of the user to make sure that this is the case.

Fairness constraints are declared using the following syntax (all expressions are expected to be boolean):

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

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

2.3.9 MODULE Declarations

A module declaration is an encapsulated collection of declarations, constraints and specifications. A module declaration also opens a new identifier scope. Once defined, a module can be reused as many times as necessary. Modules are used in such a way that each instance of a module refers to different data structures. 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 identifier [( module_parameters )] [module_body]

module_parameters ::
          identifier
        | module_parameters , identifier

module_body ::
          module_element
        | module_body module_element

module_element ::
          var_declaration
        | ivar_declaration
        | define_declaration
        | constants_declaration
        | assign_constraint
        | trans_constraint
        | init_constraint
        | invar_constraint
        | fairness_constraint
        | ctl_specification
        | invar_specification
        | ltl_specification
        | compute_specification
        | isa_declaration
```

The `identifier` immediately following the keyword **MODULE** is the name associated with the module. Module names have a separate name space in the program, and hence may clash

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

with names of variables and definitions. The optional list of identifiers in parentheses are the formal parameters of the module.

2.3.10 **MODULE** Instantiations

An *instance* of a module is created using the **VAR** declaration (see Section 2.3.1 [State Variables], page 20) with a module type specifier (see Section 2.3.1 [Type Specifiers], page 19). The syntax of a `module type specifier` is:

```
module_type_specifier ::
       | identifier [ ( [ parameter_list ] ) ]
       | process identifier [ ( [ parameter_list ] ) ]

parameter_list ::
         simple_expr
       | parameter_list , simple_expr
```

A variable declaration with a `module type specifier` introduces a name for the module instance. The `module type specifier` provides the name of the instantiating module and also a list of actual parameters, which are assigned to the formal parameters of the module. An actual parameter can be any legal `simple expression` (see Section 2.2.4 [Simple and Next Expressions], page 18). It is an error if the number of actual parameters is different from the number of formal parameters. Whenever formal parameters occur in expressions within the module, they are replaced by the actual parameters. The semantic of module instantiation is similar to call-by-reference.[8]

Here are examples:

```
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;
```

[8] This also means that the actual parameters are analyzed in the context of the variable declaration where the module is instantiated, not in the context of the expression where the formal parameter occurs.

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.

The keyword **process** is explained in Section 2.3.12 [Processes], page 28.

2.3.11 References to Module Components (Variables and Defines) and Array Elements in Expressions

As described in Section 2.2.3 [Variables and Defines], page 12, defines and variables can be referenced in expressions as variable_identifiers and define_identifiers respectively, both of which are complex identifiers. The syntax of a complex identifier is:

```
complex_identifier ::
        identifier
      | complex_identifier . identifier
      | complex_identifier [ simple_expression ]
      | self
```

Every variable and define used in an expression should be declared. It is possible to have forward references when a variable or define identifier is used textually before the corresponding declaration.

Notations with . (<DOT>) are used to access the components of modules. For example, if m is an instance of a module (see Section 2.3.10 [MODULE Instantiations], page 26 for information about instances of modules) then the expression m.c identifies the component c of the module instance m. This is precisely analogous to accessing a component of a structured data type.

Note that actual parameters of a module can potentially be instances of other modules. Therefore, parameters of modules allow access to the components of other module instances, as in the following example:

```
MODULE main
...  VAR
  a : bar;
  m : foo(a);
...
MODULE bar
 VAR
   q : boolean;
   p : boolean;

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

Here, the value of 'm.flag' is the logical **OR** of 'a.p' and 'a.q'.

Individual elements of an array are accessed in the typical fashion with the index required given in square brackets. For example, if 'a' identifies an array, the expression 'a[N]' identifies element 'N' of array 'a'. It is an error for the expression 'N' to evaluate to a number outside the subscript bounds of array 'a', or to a symbolic value. For example, for a module definition

```
MODULE main
 VAR
  a : array -1 .. 4 of boolean;
```

```
  aa : array -1 .. 4 of array 0 .. 2 of boolean;
  b : -1..4;
```

expressions ‘a[-1]’ and ‘aa[3][0]’ are legal, whereas ‘a[5]’ and ‘a[b]’ are not.

It is possible to refer to the name that the current module has been instantiated to by using the **self** built-in identifier.

```
MODULE container(init_value1, init_value2)
  VAR c1 : counter(init_value1, self);
  VAR c2 : counter(init_value2, self);

MODULE counter(init_value, my_container)
  VAR v: 1..100;
  ASSIGN
     init(v) := init_value;
  DEFINE
     greatestCounterInContainer := v >= my_container.c1.v &
                                   v >= my_container.c2.v;

MODULE main
  VAR c : container(14, 7);
  SPEC
    c.c1.greatestCounterInContainer;
```

In this example an instance of the module container is passed to the sub-module counter. In the main module, c is declared to be an instance of the module container, which declares two instances of the module counter. Every instance of the counter module has a define greatestCounterInContainer which specifies the condition when this particular counter has the greatest value in the container it belongs to. So a counter needs access to the parent container to access all the counters in the container.

2.3.12 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword ‘**process**’ (see Section 2.3.10 [MODULE Instantiations], page 26). 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. Note that only assignments of the form

```
ASSIGN next(var_name) := ... ;
```

are influenced by processes. All other kinds of assignments and all constraints (such as TRANS, INVAR, etc) are always in force, independent of which process is selected for execution.

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. No two processes may be running at the same time.

Note that in the presence of processes NuSMV internally declares special variables running and _process_selector_. These names should NOT be used in user’s own declarations, but they can be referenced for example in the transition relation of a module.

Furthermore, if the user declares N processes, there will be N+1 processes allocated, as the module main has always its own process associated. In the following example there are three process, p1, p2 and main:

```
MODULE my_module
  -- my module definition...
```

```
MODULE main
  VAR
    p1 : process my_module;
    p2 : process my_module;
```

2.3.13 A Program and the `main` Module

The syntax of a NUSMV program is:

```
program :: module_list

module_list ::
          module
        | module_list module
```

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

2.3.14 Namespaces and Constraints on Declarations

Identifiers in the NUSMV input language may reference five different entities: modules, variables, defines, module instances, and symbolic constants.

Module identifiers have their own separate namespace. Module identifiers can be used in `module type specifiers` only, and no other kind of identifiers can be used there (see Section 2.3.10 [MODULE Instantiations], page 26). Thus, module identifiers may be equal to other kinds of identifiers without making the program ambiguous. However, no two modules should be declared with the same identifier. Modules cannot be declared in other modules, therefore they are always referenced by simple `identifiers`.

Variable, define, and module instance identifiers are introduced in a program when the module containing their declarations is instantiated. Inside this module the variables, defines and module instances may be referenced by the simple `identifiers`. Inside other modules, their simple identifiers should be preceded by the identifier of the module instance containing their declaration and `.` (`<DOT>`). Such identifiers are called `complex identifier`. The *full identifier* is a `complex identifier` which references a variable, define, or a module instance from inside the `main` module.

Let us consider the following:

```
MODULE main
  VAR a : boolean;
  VAR b : foo;
  VAR c : moo;

MODULE foo
  VAR q : boolean;
      e : moo;

MODULE moo
  DEFINE f := 0 < 1;

MODULE not_used
  VAR n : boolean;
  VAR t : used;
```

```
MODULE used
  VAR k : boolean;
```

The full identifier of the variable a is a, the full identifier of the variable q (from the module foo) is b.q, the full identifier of the module instance e (from the module foo) is b.e, the full identifiers of the define f (from the module moo) are b.e.f and c.f, because two module instances contain this define. Notice that, the variables n and k as well as the module instance t do not have full identifiers because they cannot be accessed from main (since the module not_used is not instantiated).

In the NuSMV language, variable, define, and module instances belong to one namespace, and no two full identifiers of different variable, define, or module instances should be equal. Also, none of them can be redefined.

A symbolic constant can be introduced by a variable declaration if its type specifier enumerates the symbolic constant. For example, the variable declaration

```
  VAR a : {OK, FAIL, waiting};
```

declares the variable a as well as the symbolic constants OK, FAIL and waiting. The full identifiers of the symbolic constants are equal to their simple identifiers with the additional condition – the variable whose declaration declares the symbolic constants also has a full identifier.

Symbolic constants have a separate namespace, so their identifiers may potentially be equal, for example, variable identifiers. It is an error, if the same identifier in an expression can simultaneously refer to a symbolic constant and a variable or a define. A symbolic constant may be declared an arbitrary number of times, but it must be declared at least once, if it is used in an expression.

2.3.15 Context

Every module instance has its own *context*, in which all expressions are analyzed. The context can be defined as the full identifiers of variables declared in the module without their simple identifiers. Let us consider the following example:

```
MODULE main
  VAR a : foo;
  VAR b : moo;

MODULE foo
  VAR c : moo;

MODULE moo
  VAR d : boolean;
```

The context of the module main is `' (empty)[9], the context of the module instance a (and inside the module foo) is `a.', the contexts of module moo may be `b.' (if the module instance b is analyzed) and `a.c.' (if the module instance a.c is analyzed).

2.3.16 **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:

[9] The module main is instantiated with the so called empty identifier which cannot be referenced in a program.

```
isa_declaration :: ISA identifier
```

where identifier 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.

Warning: **ISA** is a deprecated feature and will be removed from future versions of NuSMV. Therefore, avoid the use of ISA_declarations. Use module instances instead.

2.4 Specifications

The specifications to be checked on the FSM can be expressed in temporal logics like Computation Tree Logic CTL, Linear Temporal Logic LTL extended with Past Operators, and Property Specification Language (PSL) [psl03] that includes CTL and LTL with Sequencial Extended Regular Expressions (SERE), a variant of classical regular expressions. It is also possible to analyze quantitative characteristics of the FSM by specifying real-time CTL specifications. Specifications can be positioned within modules, in which case they are preprocessed to rename the variables according to their context.

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

2.4.1 CTL Specifications

A CTL specification is given as a formula in the temporal logic CTL, introduced by the keyword '**CTLSPEC**' (however, deprecated keyword '**SPEC**' can be used instead.) The syntax of this specification is:

```
ctl_specification :: CTLSPEC ctl_expr ;
                   | SPEC ctl_expr ;
```

The syntax of CTL formulas recognized by NuSMV 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 xor ctl_expr     -- logical exclusive or
    | ctl_expr -> ctl_expr      -- logical implies
    | ctl_expr <-> ctl_expr     -- logical equivalence
    | EG ctl_expr               -- exists globally
    | EX ctl_expr               -- exists next state
    | EF ctl_expr               -- exists finally
    | AG ctl_expr               -- forall globally
    | AX ctl_expr               -- forall next state
    | AF ctl_expr               -- forall finally
    | E [ ctl_expr U ctl_expr ] -- exists until
    | A [ ctl_expr U ctl_expr ] -- forall until
```

Since simple_expr cannot contain the **next** operator, ctl_expr cannot contain it either. The ctl_expr should also be a boolean expression.

Intuitively the semantics of CTL operators is as follows:

- **EX** p is true in a state s if *there exists* a state s' such that a transition goes from s to s' and p is true in s'.

- **AX** p is true in a state s if *for all* states s' where there is a transition from s to s', p is true in s'.
- **EF** p is true in a state s_0 if *there exists* a series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\dots$, $s_{n-1} \rightarrow s_n$ such that p is true in s_n.
- **AF** p is true in a state s_0 if *for all* series of transitions $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots, s_{n-1} \rightarrow s_n$ p is true in s_n.
- **EG** p is true in a state s_0 if *there exists* an infinite series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\dots$ such that p is true in *every* s_i.
- **AG** p is true in a state s_0 if *for all* infinite series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2, \dots$ p is true in *every* s_i.
- **E[**p **U** q**]** is true in a state s_0 if *there exists* a series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\dots$, $s_{n-1} \rightarrow s_n$ such that p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.
- **A[**p **U** q**]** is true in a state s_0 if *for all* series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\dots$, $s_{n-1} \rightarrow s_n$ p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.

A CTL formula is true if it is true in *all* initial states.

For a detailed description about the semantics of *PSL* operators, please see [psl03].

2.4.2 Invariant Specifications

It is also possible to specify invariant specifications with special constructs. Invariants are propositional formulas which must hold invariantly in the model. The corresponding command is **INVARSPEC**, with syntax:

```
invar_specification :: INVARSPEC simple_expr ;
```

This statement is equivalent to

```
SPEC  AG simple_expr ;
```

but can be checked by a specialised algorithm during reachability analysis. Fairness constraints are not taken into account during invariant checking.

2.4.3 LTL Specifications

LTL specifications are introduced by the keyword **LTLSPEC**. The syntax of this specification is:

```
ltl_specification :: LTLSPEC ltl_expr [;]
```

The syntax of LTL formulas recognized by NuSMV is as follows:

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

```
  | ltl_expr V ltl_expr    -- releases
  -- PAST
  | Y ltl_expr             -- previous state
  | Z ltl_expr             -- not previous state not
  | H ltl_expr             -- historically
  | O ltl_expr             -- once
  | ltl_expr S ltl_expr    -- since
  | ltl_expr T ltl_expr    -- triggered
```

Intuitively the semantics of LTL operators is as follows:

- **X** p is true at time t if p is true at time $t+1$.
- **F** p is true at time t if p is true at *some* time $t' \geq t$.
- **G** p is true at time t if p is true at *all* times $t' \geq t$.
- p **U** q is true at time t if q is true at *some* time $t' \geq t$, and *for all* time t'' (such that $t \leq t'' < t'$) p is true.
- p **V** q is true at time t if q holds at *all* time steps $t' \geq t$ up to and including the time step t'' where p also holds. Alternatively, it may be the case that p *never* holds in which case q must hold in *all* time steps $t' \geq t$.
- **Y** p is true at time $t > 0$ if p holds at time $t-1$. **Y** p is *false* at time t_0.
- **Z** p is equivalent to **Y** p with the exception that the expression is *true* at time t_0.
- **H** p is true at time t if p holds in *all* previous time steps $t' \leq t$.
- **O** p is true at time t if p held in *at least one* of the previous time steps $t' \leq t$.
- p **S** q is true at time t if q held at time $t' \leq t$ and p holds in *all* time steps from t' to t inclusive.
- p **T** q is true at time t if p held at time $t' \leq t$ and q holds in *all* time steps from t' to t inclusive. Alternatively, if p has *never* been true, then q must hold in all time steps from t_0 to t.

An LTL formula is true if it is true at the initial time $t = 0$.

In NuSMV, LTL specifications can be analyzed both by means of BDD-based reasoning, or by means of SAT-based bounded model checking. In the case of BDD-based reasoning, NuSMV proceeds according to [CGH97]. For each LTL specification, a tableau of the behaviors falsifying the property is constructed, and then synchronously composed with the model. With respect to [CGH97], the approach is fully integrated within NuSMV, and allows full treatment of past temporal operators. Note that the counterexample is generated in such a way to show that the falsity of a LTL specification may contain state variables which have been introduced by the tableau construction procedure.

In the case of SAT-based reasoning, a similar tableau construction is carried out to encode the paths of limited length, violating the property. NuSMV generates a propositional satisfiability problem, that is then tackled by means of an efficient SAT solver [BCCZ99].

In both cases, the tableau constructions are completely transparent to the user.

2.4.4 Real Time CTL Specifications and Computations

NuSMV allows for Real Time CTL specifications [EMSS91]. NuSMV assumes that each transition takes unit time for execution. RTCTL extends the syntax of CTL path expressions with the following bounded modalities:

```
rtctl_expr ::
        ctl_expr
      | EBF range rtctl_expr
      | ABF range rtctl_expr
```

```
       | EBG range rtctl_expr
       | ABG range rtctl_expr
       | A [ rtctl_expr BU range rtctl_expr ]
       | E [ rtctl_expr BU range rtctl_expr ]
range  :: integer_number .. integer_number
```

Intuitively, the semantics of the RTCTL operators is as follows:

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

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

```
rtctl_specification :: SPEC rtctl_expr [;]
```

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

```
compute_specification :: COMPUTE compute_expr [;]
```

where

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

MIN [*start* , *final*] returns the length of the shortest path from a state in *start* to a state in *final*. For this, the set of states reachable from *start* is computed. 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 computed 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. If any of the initial or final states is empty, then *undefined* is returned.

It is important to remark here that if the FSM is not total (i.e. it contains deadlock states) **COMPUTE** may produce wrong results. It is possible to check the FSM against deadlock states by calling the command `check_fsm`.

2.4.5 PSL Specifications

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

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

where

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

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

```
psl_primary_expr ::
   number                              ;; a numeric constant
  | boolean                            ;; a boolean constant
  | var_id                             ;; a variable identifier
  | { psl_expr , ... , psl_expr }
  | { psl_expr "{" psl_expr , ... , "psl_expr" }}
  | ( psl_expr )
psl_unary_expr ::
   + psl_primary_expr
  | - psl_primary_expr
  | ! psl_primary_expr
psl_binary_expr ::
   psl_expr + psl_expr
  | psl_expr union psl_expr
  | psl_expr in psl_expr
  | psl_expr - psl_expr
  | psl_expr * psl_expr
  | psl_expr / psl_expr
  | psl_expr % psl_expr
  | psl_expr == psl_expr
  | psl_expr != psl_expr
  | psl_expr < psl_expr
  | psl_expr <= psl_expr
  | psl_expr > psl_expr
  | psl_expr >= psl_expr
  | psl_expr & psl_expr
  | psl_expr | psl_expr
  | psl_expr xor psl_expr
psl_conditional_expr ::
 psl_expr ? psl_expr : psl_expr
psl_case_expr ::
 case
     psl_expr : psl_expr ;
     ...
     psl_expr : psl_expr ;
 endcase
```

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

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

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

```
FL_property ::
 ;; PRIMITIVE LTL OPERATORS
   X FL_property
 | X! FL_property
 | F FL_property
 | G FL_property
 | [ FL_property U FL_property ]
 | [ FL_property W FL_property ]
 ;; SIMPLE TEMPORAL OPERATORS
 | always FL_property
 | never FL_property
 | next FL_property
 | next! FL_property
 | eventually! FL_property
 | FL_property until! FL_property
 | FL_property until FL_property
 | FL_property until!_ FL_property
 | FL_property until_ FL_property
 | FL_property before! FL_property
 | FL_property before FL_property
 | FL_property before!_ FL_property
```

```
 | FL_property before_ FL_property
 ;; EXTENDED NEXT OPERATORS
 | X [number] ( FL_property )
 | X! [number] ( FL_property )
 | next [number] ( FL_property )
 | next! [number] ( FL_property )
 ;;
 | next_a [range] ( FL_property )
 | next_a! [range] ( FL_property )
 | next_e [range] ( FL_property )
 | next_e! [range] ( FL_property )
 ;;
 | next_event! ( psl_bexpr ) ( FL_property )
 | next_event ( psl_bexpr ) ( FL_property )
 | next_event! ( psl_bexpr ) [ number ]  ( FL_property )
 | next_event ( psl_bexpr ) [ number ]  ( FL_property )
 ;;
 | next_event_a! ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_a ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_e! ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_e ( psl_bexpr ) [psl_expr]  ( FL_property )
 ;; OPERATORS ON SEREs
 | sequence ( FL_property )
 | sequence |-> sequence [!]
 | sequence |=> sequence [!]
 ;;
 | always sequence
 | G sequence
 | never sequence
 | eventually! sequence
 ;;
 | within! ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within!_ ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within_ ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 ;;
 | whilenot! ( psl_bexpr ) sequence
 | whilenot ( psl_bexpr ) sequence
 | whilenot!_ ( psl_bexpr ) sequence
 | whilenot_ ( psl_bexpr ) sequence
sequence_or_psl_bexpr ::
   sequence
 | psl_bexpr
```

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

```
sequence ::
   { SERE }
SERE ::
   sequence
 | psl_bexpr
 ;; COMPOSITION OPERATORS
 | SERE ; SERE
 | SERE : SERE
 | SERE & SERE
```

```
 | SERE && SERE
 | SERE | SERE
 ;; RegExp QUALIFIERS
 | SERE [* [count] ]
 | [* [count] ]
 | SERE [+]
 | [+]
 ;;
 | psl_bexpr [= count ]
 | psl_bexpr [-> count ]
count ::
   number
 | range
```

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

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

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

2.5 Variable Order Input

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

2.5.1 Input File Syntax

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

```
vars_list :: EMPTY
         | var_list_item vars_list

var_list_item :: complex_identifier
             | complex_identifier . integer_number
```

Where *EMPTY* means parsing nothing.

This grammar allows for parsing a list of variable names of the following forms:

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

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

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

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

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

2.5.2 Scalar Variables

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

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

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

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

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

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

The variable representing the i^{th} bit is located at the position given in the variable ordering and the remainder are located where the scalar variable name is declared. In this case, the remaining bit variables will not be grouped together.

This is just a short-hand way of writing each individual specific-bit variable in the ordering file. The following are equivalent:

```
...                       ...
Complete_Var_Name.0       Complete_Var_Name.0
Complete_Var_Name.1       Complete_Var_Name
⋮                         ...
Complete_Var_Name.N-1
...
```

where the scalar variable `Complete_Var_Name` requires N boolean variables to encode all the possible values that it may take. It is still possible to then specify other specific-bit variables at later points in the ordering file as before.

2.5.3 Array Variables

When declaring array variables in the ordering file, each individual element must be specified separately. It is not permitted to specify just the name of the array. The reason for this is that the actual definition of an array in the model file is essentially a shorthand method of defining a list of variables that all have the same type. Nothing is gained by declaring it as an array over declaring each of the elements individually, and there is no difference in terms of the internal representation of the variables.

2.6 Clusters Ordering

When NuSMV builds a clusterized BDD-based FSM during model construction, an initial simple clusters list is roughly constructed by iterating through a *list of variables*, and by constructing the clusters by picking the transition relation associated to each variable in the list. Later, the clusters list will be refined and improved by applying the clustering alghorithm that the user previoulsy selected (see partitioning methods at page 3.1 for further information).

In [WJKWLvdBR06], Wendy Johnston and others from University of Queensland, showed that choosing a good ordering for the initial list of variables that is used to build the clusters list may lead to a dramatic improvement of performances. They did experiments in a modified version of NuSMV, by allowing the user to specify a variable ordering to be used when constructing the initial clusters list. The prototype code has been included in version 2.4.1, that offers the new option `trans_order_file` to specify a file containing a variable ordering (see at page 44 for further information).

Grammar of the clusters ordering file is the same of variable ordering file presented in section 2.5 at page 38.

Chapter 3

Running NuSMV interactively

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

```
system_prompt> NuSMV -int <RET>
NuSMV>
```

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

It is also possible to make NuSMV read and execute a sequence of commands from a file, through the command line option **-load**:

```
system_prompt> NuSMV -int -load cmd_file <RET>
```

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

In the following we present the possible commands followed by the related environment variables, classified in different categories. Every command answers to the option `-h` by printing out the command usage. When output is paged for some commands (option `-m`), it is piped through the program specified by the UNIX `PAGER` shell variable, if defined, or through the UNIX command "more". Environment variables can be assigned a value with the "set" command. Command sequences to NuSMV must obey the (partial) order specified in the Figure 3.10 depicted at page 86. For instance, it is not possible to evaluate CTL expressions before the model is built.

A number of commands and environment variables, like those dealing with file names, accept arbitrary strings. There are a few reserved characters which must be escaped if they are to be used literally in such situations. See the section describing the `history` command, on page 79, for more information.

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

verbose_level	Environment Variable

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

3.1 Model Reading and Building

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

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

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

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

Command Options:

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

input_file	Environment Variable

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

pp_list	Environment Variable

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

flatten_hierarchy - *Flattens the hierarchy of modules*	Command

`flatten_hierarchy [-h]`

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

backward_compatibility	Environment Variable

It is used to enable or disable type checking and other features provided by NuSMV 2.4. If set to `1` then the type checking is turned off, and NuSMV behaves as the old versions w.r.t. type checking and other features like writing of flattened and booleanized SMV files. If set to `0` then the type checking is turned on, and whenever a type error is encountered while compiling a NuSMV program the user is informed and the execution stopped. As default it set to `0`.

type_checking_warning_on	Environment Variable

Enables notification of warning messages generated by the type checking. If set to 0, then messages are disregarded, otherwise if set to 1 they are notified to the user. As default it set to 1.

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

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

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

Command Options:

`-s`	Prints only state variables.
`-i`	Prints only input variables.
`-m`	Pipes the output to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command to `output-file`.

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

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

Generates the boolean BDD variables and the ADD needed to encode propositionally the (symbolic) variables declared in the model. The variables are created as default in the order in which they appear in a depth first traversal of the hierarchy.
The input order file can be partial and can contain variables not declared in the model. Variables not declared in the model are simply discarded. Variables declared in the model which are not listed in the ordering input file will be created and appended at the end of the given ordering list, according to the default ordering.

Command Options:

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

input_order_file — Environment Variable

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

write_order_dumps_bits — Environment Variable

Changes the behaviour of the command `write_order`.

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

write_order - *Writes variable order to file.* Command

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

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

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

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

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

Command Options:

`-b`	Dumps bits of scalar variables instead of the single scalar variables. See also the variable `write_order_dumps_bits`.
`-o order-file`	Sets the environment variable `output_order_file` to `order-file` and then dumps the ordering list into that file.
`-f order-file`	Alias for the `-o` option. Supplied for backward compatibility.

output_order_file Environment Variable

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

vars_order_type Environment Variable

Controls the manner variables are ordered by default, when a variable ordering is not specified.

- **inputs_before**. Input variables are forced to be ordered *before* state variables (default).
- **inputs_after**. Input variables are forced to be ordered *after* state variables.
- **lexicographic**. Input and state variables will be ordered as they are declared in the input smv file, in a lexicographic order.

build_model - *Compiles the flattened hierarchy into a BDD* Command

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

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

Command Options:

`-m Method`	Sets the environment variable `partition_method` to the value `Method`, and then builds the transition relation. Available methods are `Monolithic`, `Threshold` and `Iwls95CP`.
`-f`	Forces model construction. By default, only one partition method is allowed. This option allows to overcome this default, and to build the transition relation with different partitioning methods.

partition_method — Environment Variable

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

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

conj_part_threshold — Environment Variable

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

affinity — Environment Variable

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

trans_order_file — Environment Variable

Reads the a variables list from file *tv_file*, to be used when clustering the transition relation. This feature has been provided by Wendy Johnston, University of Queensland. The results of Johnston's research have been presented at FM 2006 in Hamilton, Canada. See [WJKWLvdBR06].

image_cluster_size — Environment Variable

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

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

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

iwls95preorder Environment Variable

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

image_verbosity Environment Variable

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

print_iwls95options - *Prints the Iwls95 Options.* Command

```
print_iwls95options [-h]
```

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

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

```
go [-h] [-f]
```

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

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

Command Options:

`-f`	Forces model construction even when Cone Of Influence is enabled.

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

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

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

Command Options:

`-f`	Forces the model construction even when Cone Of Influence is enabled.
`-r`	Forces a variable reordering at the end of the computation, and dumps the new variables ordering to the default ordering file. This options acts like the command line option `-reorder`.

`-i model-file`	Sets the environment variable `input_file` to file `model-file`, and reads the model from file `model-file`.
`-m Method`	Sets the environment variable `partition_method` to `Method` and uses it as partitioning method.

write_flat_model - *Writes a flat model to a file* — Command

`write_flat_model [-h] [-o filename]`

Writes the currently loaded SMV model in the specified file, after having flattened it. Processes are eliminated and a corresponding equivalent model is printed out.

If no file is specified, the file specified via the environment variable `output_flatten_model_file` is used if any, otherwise standard output is used.

Command Options:

`-o filename`	Attempts to write the flat SMV model in `filename`

output_flatten_model_file — Environment Variable

The file where the flattened model has to be written. The default value is '`stdout`'.

write_boolean_model - *Writes a flat and boolean model to a file* — Command

`write_boolean_model [-h] [-o filename]`

Writes the currently loaded SMV model in the specified file, after having flattened and booleanized it. Processes are eliminated and a corresponding equivalent model is printed out.

If no file is specified, the file specified via the environment variable `output_boolean_model_file` is used if any, otherwise standard output is used.

Command Options:

`-o filename`	Attempts to write the flat and boolean SMV model in `filename`

In NuSMV 2.4 scalar variables are dumped as **`DEFINEs`** whose body is their boolean encoding.

This allows the user to still express and see parts of the generated boolean model in terms of the original model's scalar variables names and values, and still keeping the generated model purely boolean.

Also, symbolic constants are dumped within a **`CONSTANTS`** statement to declare the values of the original scalar variables' for future reading of the generated file.

When NuSMV detects that there were triggered one or more dynamic reorderings in the BDD engine, the command `write_boolean_model` also dumps the current variables ordering, if the option `output_order_file` is set.

The dumped variables ordering will contain single bits or scalar variables depending on the current value of the option `write_order_dumps_bits`. See command `write_order` for further information about variables ordering.

output_boolean_model_file — Environment Variable

The file where the flattened and booleanized model has to be written. The default value is ‘stdout’.

output_word_format Environment Variable

This variable sets in which base word[•] constants are outputted (during traces, counterexamples, etc, printing). Possible values are 2, 8, 10 and 16. Note that if a part of an input file is outputted (for example, if a specification expression is outputted) then the word[•] constants remain in same format as they were written in the input file.

3.2 Commands for Checking Specifications

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

compute_reachable - *Computes the set of reachable states* Command

```
compute_reachable [-h]
```

Computes the set of reachable states. The result is then used to simplify image and preimage computations. This can result in improved performances for models with sparse state spaces. Sometimes this option may slow down the performances because the computation of reachable states may be very expensive. The environment variable `forward_search` is set during the execution of this command. Since version 2.4.0, the computation of the reachable states is automatically performed as the variable `forward_search` is set by default.

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

```
print_reachable_states [-h] [-v]
```

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

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

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

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

Command Options:

`-m`	Pipes the output generated by the command to the program specified by the `PAGER` shell variable if defined, else through the UNIX command “more”.
`-o output-file`	Writes the output generated by the command to the file `output-file`.

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

check_fsm Environment Variable

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

print_fsm_stats - *Prints out information about the fsm and clustering.* Command

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

This command prints out information regarding the fsm and 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.

Command Options:

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

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

`print_fair_states [-h] [-v]`

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

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

`print_fair_transitions [-h] [-v]`

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

check_ctlspec - *Performs fair CTL model checking.* Command

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

Performs fair CTL model checking.

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

Command Options:

`-m`	Pipes the output generated by the command in processing `SPEC` s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command in processing `SPEC` s to the file `output-file`.
`-p "ctl-expr [IN context]"`	A CTL formula to be checked. `context` is the module instance name which the variables in `ctl-expr` must be evaluated in.

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

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

Since version 2.4.1 this command substitutes `check_spec` that is *deprecated*.

check_spec - *Performs fair CTL model checking.* Command

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

Performs fair CTL model checking.

Since version 2.4.1 this command is *deprecated* but still provided for backward compatibility reasons. Use `check_ctlspec` instead.

ag_only_search Environment Variable

Enables the use of an ad hoc algorithm for checking AG formulas. Given a formula of the form *AG alpha*, the algorithm computes the set of states satisfying *alpha*, and checks whether it contains the set of reachable states. If this is not the case, the formula is proved to be false.

forward_search Environment Variable

Enables the computation of the reachable states during the `process_model` command and when used in conjunction with the `ag_only_search` environment variable enables the use of an ad hoc algorithm to verify invariants. Since version 2.4.0, this option is set by default.

ltl_tableau_forward_search Environment Variable

Forces the computation of the set of reachable states for the tableau resulting from BDD-based LTL model checking, performed by command `check_ltlspec`. If the variable `ltl_tableau_forward_search` is not set (default), the resulting tableau will inherit the computation of the reachable states from the model, if enabled. If the variable is set, the reachable states set will be calculated for the model *and* for the tableau resulting from LTL model checking. This might improve performances of the command `check_ltlspec`, but may also lead to a dramatic slowing down. This variable has effect only when the calculation of reachable states for the model is enabled (see `forward_search`).

check_invar - *Performs model checking of invariants* Command

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

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

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

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

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

Command Options:

`-m`	Pipes the output generated by the program in processing `INVARSPEC` s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command in processing `INVARSPEC` s to the file `output-file`.
`-p "invar-expr [IN context]"`	The command line specified invariant formula to be verified. `context` is the module instance name which the variables in `invar-expr` must be evaluated in.

check_ltlspec - *Performs LTL model checking* Command

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

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

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

Command Options:

`-m`	Pipes the output generated by the command in processing `LTLSPEC`s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command in processing `LTLSPEC`s to the file `output-file`.
`-p "ltl-expr [IN context]"`	An LTL formula to be checked. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-n number`	Checks the LTL property with index `number` in the property database.

compute - *Performs computation of quantitative characteristics* Command

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

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

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

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

If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, then *infinity* is returned. If any of the initial or final states is empty, then *undefined* is returned.

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

It is important to remark here that if the FSM is not total (i.e. it contains deadlock states) **`COMPUTE`** may produce wrong results. It is possible to check the FSM against deadlock states by calling the command `check_fsm`.

Command Options:

`-m`	Pipes the output generated by the command in processing `COMPUTE`s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command in processing `COMPUTE`s to the file `output-file`.
`-p "compute-expr [IN context]"`	A COMPUTE formula to be checked. `context` is the module instance name which the variables in `compute-expr` must be evaluated in.
`-n number`	Computes only the property with index `number`.

check_property - *Checks a property into the current list of properties, or a newly specified property* Command

`check_property [-h] [-n number] | [(-c | -l | -i | -s | -q ) [-p "formula [IN context]"]]`

Checks the specified property taken from the property list, or adds the new specified property and checks it. It is possible to check `LTL, CTL, INVAR, PSL` and quantitative (`COMPUTE`) properties. Every newly inserted property is inserted and checked.

Command Options:

`-c`	Checks all the `CTL` properties not already checked. If -p is used, the given formula is expected to be a `CTL` formula.
`-l`	Checks all the `LTL` properties not already checked. If -p is used, the given formula is expected to be a `LTL` formula.
`-i`	Checks all the `INVAR` properties not already checked. If -p is used, the given formula is expected to be a `INVAR` formula.
`-s`	Checks all the `PSL` properties not already checked. If `-p` is used, the given formula is expected to be a `PSL` formula.
`-q`	Checks all the `COMPUTE` properties not already checked. If -p is used, the given formula is expected to be a `COMPUTE` formula.
`-p "formula [IN context]"`	Checks the formula specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.

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

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

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

Command Options:

`-c`	Adds a `CTL` property.
`-l`	Adds an `LTL` property.
`-i`	Adds an `INVAR` property.
`-s`	Adds a `PSL` property.
`-q`	Adds a quantitative (`COMPUTE`) property.
`-p "formula [IN context]"`	Adds the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.

3.3 Commands for Bounded Model Checking

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

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

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

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

`bmc_setup [-h]`

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

go_bmc - *Initializes the system for the BMC verification.* Command

`go_bmc [-h] [-f]`

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

Command Options:

`-f`	Forces model construction even when Cone Of Influence is enabled.

sexp_inlining Environment Variable

This variable enables the Sexp inlining when the boolean model is built. Sexp inlining is performed in a similar way to RBC inlining (see system variable `rbc_inlining`) but the underlying structures and kind of problem are different, because inlining is applied at the Sexp level instead of the RBC level.

Inlining is applied to initial states, invariants and transition relations. By default, Sexp inlining is disabled.

rbc_inlining Environment Variable

When set, this variable makes BMC perform the RBC inlining before committing any problem to the SAT solver. Depending on the problem structure and length, the inlining may either make SAT solving much faster, or slow it down dramatically. Experiments showed an average improvement in time of SAT solving when RBC inlining is enabled. RBC inlining is enabled by default.

The idea about inlining was taken from [ABE00] by Parosh Aziz Abdulla, Per Bjesse and Niklas Eén.

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

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

This command generates one or more problems, and calls SAT solver for each one. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and solve. In this context the maximum problem bound is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`. The single generated problem also depends on the `loopback` parameter you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be checked may be specified using the `-n idx` or the `-p "formula"` options. If you need to generate a DIMACS dump file of all generated problems, you must use the option `-o "filename"`.

Command Options:

`-n` *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process. • a negative number in (-1, *-bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopbacks from zero to *length-1*".
`-o` *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part. • @f: model name without path part. • @k: current problem bound. • @l: current loopback value. • @n: index of the currently processed formula in the property database. • @@: the '@' character.

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

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

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

Command Options:

`-n` *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database. The validity of *index* value is checked out by the system.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *length*	*length* is the problem bound used when generating the single problem. Only natural numbers are valid values for this option. If no value is given the environment variable `bmc_length` is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.

- a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *length*. Any invalid combination of length and loopback will be skipped during the generation/solving process.
- the symbol 'X', which means "no loopback" .
- the symbol '*', which means "all possible loopback from zero to *length-1*".

`-o` *filename* — *filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:

- @F: model name with path part.
- @f: model name without path part.
- @k: current problem bound.
- @l: current loopback value.
- @n: index of the currently processed formula in the property database.
- @@: the '@' character.

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

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

This command generates one or more problems, and dumps each problem into a dimacs file. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem bound. In this short description `length` is the bound of the problem that system is going to dump out.

In this context the maximum problem bound is represented by the *max_length* parameter, or by its default value stored in the environment variable `bmc_length`.

Each dumped problem also depends on the loopback you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be checked may be specified using the `-n idx` or the `-p "formula "` options.

You may specify dimacs file name by using the option `-o filename` , otherwise the default value stored in the environment variable `bmc_dimacs_filename` will be considered.

Command Options:

`-n` *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database. The validity of `index` value is checked out by the system.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound used when increasing problem bound starting from zero. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* value is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of bound and loopback will be skipped during the generation and dumping process. • a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of bound and loopback will be skipped during the generation process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopback from zero to *length-1*".
`-o` *filename*	*filename* is the name of dumped dimacs files. If this options is not specified, variable *bmc_dimacs_filename* will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part. • @f: model name without path part. • @k: current problem bound. • @l: current loopback value . • @n: index of the currently processed formula in the property database. • @@: the '@' character.

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

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

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

Command Options:

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

`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *length*	*length* is the single problem bound used to generate and dump it. Only natural numbers are valid values for this option. If no value is given the environment variable `bmc_length` is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation and dumping process. • negative number in (-1, *-length*). Any invalid combination of length and loopback will be skipped during the generation process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopback from zero to *length-1*".
`-o` *filename*	*filename* is the name of the dumped dimacs file. If this options is not specified, variable `bmc_dimacs_filename` will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part • @f: model name without path part • @k: current problem bound • @l: current loopback value • @n: index of the currently processed formula in the property database • @@: the '@' character

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

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

For each problem this command incrementally generates many satisfiability subproblems and calls the SAT solver on each one of them. The incremental algorithm exploits the fact that subproblems have common subparts, so information obtained during a previous call to the SAT solver can be used in the consecutive ones. Logically, this command does the same thing as `check_ltlspec_bmc` (see the description on page 55) but usually runs considerably quicker. A SAT solver with an incremental interface is required by this command, therefore if no such SAT solver is provided then this command will be unavailable.

Command Options:

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

`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound must be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process. • a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopback from zero to *length-1*" .

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

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

This command generates one or more problems, and calls SAT solver for each one. The BMC encoding used is the one by of Latvala, Biere, Heljanko and Junttila as described in [LBHJ05]. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and solve. In this context the maximum problem bound is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`. The single generated problem also depends on the `loopback` parameter you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be checked may be specified using the `-n idx` or the `-p "formula"` options. If you need to generate a DIMACS dump file of all generated problems, you must use the option `-o "filename"`.

Command Options:

`-n` *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process. • a negative number in (-1, *-bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopbacks from zero to *length-1*" .
`-o` *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part. • @f: model name without path part. • @k: current problem bound. • @l: current loopback value. • @n: index of the currently processed formula in the property database. • @@: the '@' character.

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

`check_ltlspec_sbmc_inc [-h | -n idx | -p "formula [IN context]"] [-k max_length] [-o filename] [-N] [-c]`

This command generates one or more problems, and calls SAT solver for each one. The Incremental BMC encoding used is the one by of Heljanko, Junttila and Latvala, as described in [KHL05]. For each problem this command incrementally generates many satisfiability subproblems and calls the SAT solver on each one of them. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and solve. In this context the maximum problem bound is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`.

The property to be checked may be specified using the `-n idx` or the `-p "formula"` options. If you need to generate a DIMACS dump file of all generated problems, you must use the option `-o "filename"`.

Command Options:

`-n` *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.

-o *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part. • @f: model name without path part. • @k: current problem bound. • @l: current loopback value. • @n: index of the currently processed formula in the property database. • @@: the '@' character.
-N	Does not perform virtual unrolling.
-c	Performs completeness check.

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

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

This command generates one or more problems, and dumps each problem into a dimacs file. The BMC encoding used is the one by of Latvala, Biere, Heljanko and Junttila as described in [LBHJ05]. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and dump. In this context the maximum problem bound is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`. The single generated problem also depends on the `loopback` parameter you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be used for tghe problem dumping may be specified using the `-n idx` or the `-p "formula"` options. You may specify dimacs file name by using the option `-o "filename"`, otherwise the default value stored in the environment variable `bmc_dimacs_filename` will be considered.

Command Options:

-n *index*	*index* is the numeric index of a valid LTL specification formula actually located in the properties database.
-p "formula [IN context]"	Dumps the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
-k *max_length*	*max_length* is the maximum problem bound to be generated. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
-l *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.

- a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process.
- the symbol '`X`', which means "no loopback".
- the symbol '`*`', which means "all possible loopbacks from zero to *length-1*" .

`-o` *filename* — *filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:

- @F: model name with path part.
- @f: model name without path part.
- @k: current problem bound.
- @l: current loopback value.
- @n: index of the currently processed formula in the property database.
- @@: the '@' character.

bmc_length — Environment Variable

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

bmc_loopback — Environment Variable

Sets the generated problem loop. Possible values are:

- Any natural number, but less than the current value of the variable *bmc_length*. In this case the loop point is absolute.
- Any negative number, but greater than or equal to -*bmc_length*. In this case specified loop is the loop length.
- The symbol '`X`', which means "no loopback".
- The symbol '`*`', which means "any possible loopbacks".

The default value is `*`.

bmc_dimacs_filename — Environment Variable

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

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

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

bmc_sbmc_gf_fg_opt	Environment Variable

Controls whether the system exploits an optimization when performing SBMC on formulae in the form FGp or GFp. The default value is `1` (active).

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

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

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

Also, notice that during checking of invariants all the fairness conditions associated with the model are ignored.

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of *index* value is checked out by the system.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k` *max_length*	*max_length* is the maximum problem bound that can be reached. Only natural numbers are valid values for this option. Use this option only if the "een-sorensson" algorithm is selected. If no value is given the environment variable *bmc_length* is considered instead.
`-a` *alg*	*alg* specifies the algorithm. The value can be `classic` or `een-sorensson`. If no value is given the environment variable *bmc_invar_alg* is considered instead.
`-o` *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • **@F**: model name with path part • **@f**: model name without path part • **@n**: index of the currently processed formula in the properties database • @@: the '@' character

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

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

At the moment, the invariants are generated using "classic" algorithm only (see the description of `check_invar_bmc` on page 64).

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of *index* value is checked out by the system.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-o` *filename*	*filename* is the name of the dumped dimacs file. If you do not use this option the dimacs file name is taken from the environment variable `bmc_invar_dimacs_filename`. File name may contain special symbols which will be macro-expanded to form the real dimacs file name. Possible symbols are: • **@F**: model name with path part • **@f**: model name without path part • **@n**: index of the currently processed formula in the properties database • @@: the '@' character

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

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

This command does the same thing as `check_invar_bmc` (see the description on page 64) but uses an incremental algorithm and therefore usually runs considerably quicker. The incremental algorithms exploit the fact that satisfiability problems generated for a particular invariant have common subparts, so information obtained during solving of one problem can be used in solving another one. A SAT solver with an incremental interface is required by this command. If no such SAT solver is provided then this command will be unavailable.

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

Also, notice that during checking of invariants all the fairness conditions associated with the model are ignored.

Command Options:

`-n` *index*	*index* is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of *index* value is checked out by the system.
`-p "formula [IN context]"`	Checks the `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.

`-k` *max_length*	*max_length* is the maximum problem bound that can be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-a` *alg*	*alg* specifies the algorithm to use. The value can be `dual` or `zigzag`. If no value is given the environment variable *bmc_inc_invar_alg* is considered instead.

bmc_invar_alg — Environment Variable

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

bmc_inc_invar_alg — Environment Variable

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

bmc_invar_dimacs_filename — Environment Variable

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

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

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

sat_solver — Environment Variable

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

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

```
bmc_simulate [-h | -k ]
```

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

Command Options:

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

3.4 Commands for checking PSL specifications

The following command allow for model checking of PSL specifications.

check_pslspec - *Performs PSL model checking*	Command

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

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

A `psl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the PSLSPEC formulas in the database are checked. If option `-b` is used, LTL bounded model checking is applied, otherwise bdd-based model checking is applied. For LTL bounded model checking, options `-k` and `-l` can be used to define the maximum problem bound, and the value of the loopback for the single generated problems respectively; their values can be stored in the environment variables *bmc_lenght* and *bmc_loopback*. Single problems can be generated by using option `-1`. By using option `-i` the incremental version of bounded model checking is activated. Bounded model checking problems can be generated and dumped in a file by using option `-g`.

Command Options:

`-m`	Pipes the output generated by the command in processing `PSLSPEC`s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o` *output-file*	Writes the output generated by the command in processing `PSLSPEC` s to the file *output-file*
`-p "psl-expr [IN context]"`	A PSL formula to be checked. `context` is the module instance name which the variables in `psl-expr` must be evaluated in.
`-n number`	Checks the PSL property with index `number` in the property database.
`-b`	Applies SAT-based bounded model checking. The SAT solver to be used will be chosen according to the current value of the system variable `sat_solver`.
`-i`	Applies incremental SAT-bounded model checking if available, i.e. if an incremental SAT solver has been linked to NuSMV. This option can be used only in combination with the option `-b`.
`-g`	Dumps DIMACS version of bounded model checking problem into a file whose name depends on the system variable `bmc_dimacs_filename`. This feature is not allowed in combination of the option `-i`.

`-1`	Generates a single bounded model checking problem with fixed bound and loopback values, it does not iterate incrementing the value of the problem bound.
`-k` *bmc_length*	*bmc_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process. • a negative number in (-1, -*bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process. • the symbol '`X`', which means "no loopback". • the symbol '`*`', which means "all possible loopbacks from zero to *length-1*" If no value is given the environment variable *bmc_loopback* is considered instead..

3.5 Simulation Commands

In this section we describe the commands that allow to simulate a NuSMV specification. See also the section Section 3.6 [Traces], page 70 that describes the commands available for manipulating traces.

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

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

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

Command Options:

`-v`	Verbosely prints out chosen state (all state variables, otherwise it prints out only the label `t.1` of the state chosen, where `t` is the number of the new trace, that is the number of traces so far generated plus one).
`-r`	Randomly picks a state from the set of initial states.
`-i`	Enables the user to interactively pick up an initial state. The user is requested to choose a state from a list of possible items (every item in the list doesn't show state variables unchanged with respect to a previous item). If the number of possible states is too high, then the user has to specify some further constraints as "simple expression".

`-a`	Displays all state variables (changed and unchanged with respect to a previous item) in an interactive picking. This option works only if the `-i` options has been specified.
`-c "constraints"`	Uses `constraints` to restrict the set of initial states in which the state has to be picked. `constraints` must be enclosed between double quotes `" "`.

showed_states Environment Variable

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

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

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

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

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

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

Command Options:

`-p`	Prints current generated trace (only those variables whose value changed from the previous state).
`-v`	Verbosely prints current generated trace (changed and unchanged state variables).
`-r`	Picks a state from a set of possible future states in a random way.
`-i`	Enables the user to interactively choose every state of the trace, step by step. If the number of possible states is too high, then the user has to specify some constraints as simple expression. These constraints are used only for a single simulation step and are *forgotten* in the following ones. They are to be intended in an opposite way with respect to those constraints eventually entered with the `pick_state` command, or during an interactive simulation session (when the number of future states to be displayed is too high), that are *local* only to a single step of the simulation and are *forgotten* in the next one. To improve readability of the list of the states which the user must pick one from, each state is presented in terms of difference with respect of the previous one.
`-a`	Displays all the state variables (changed and unchanged) during every step of an interactive session. This option works only if the `-i` option has been specified.
`-c "constraints"`	Performs a simulation in which computation is restricted to states satisfying those `constraints`. The desired sequence of states could not exist if such constraints were too strong or it may happen that at some point of the simulation a future state satisfying those constraints doesn't exist: in that case a trace with a number of states less than `steps` trace is obtained. Note: `constraints` must be enclosed between double quotes `" "`.
`steps`	Maximum length of the path according to the constraints. The length of a trace could contain less than `steps` states: this is the case in which simulation stops in an intermediate step because it may not exist any future state satisfying those constraints.

3.6 Traces

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

Traces are created by NuSMV when a formula is found to be false; they are also generated as a result of a simulation (Section 3.5 [Simulation Commands], page 68). Each trace has a number, and the states-inputs pairs are numbered within the trace. Trace *n* has states/inputs *n.1, n.2, n.3, "..."* where *n.1* represents the initial state.

3.6.1 Inspecting Traces

The trace inspection commands of NuSMV allow for navigation along the labelled states-inputs pairs of the traces produced. During the navigation, there is a *current state*, and the *current trace* is the trace the *current state* belongs to. The commands are the following:

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

`goto_state [-h] state_label`

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

print_current_state - *Prints out the current state* Command

`print_current_state [-h] [-v]`

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

Command Options:

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

3.6.2 Displaying Traces

NuSMV comes with three trace plugins (see Section 3.7 [Trace Plugins], page 73) which can be used to display traces in the system. Once a trace has been generated by NuSMV it is printed to `stdout` using the trace explanation plugin which has been set as the current default. The command `show_traces` (see Section 3.5 [Simulation Commands], page 68) can then be used to print out one or more traces using a different trace plugin, as well as allowing for output to a file.

3.6.3 Trace Plugin Commands

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

show_plugins - *Shows the available trace explanation plugins* Command

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

Command Options:

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

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

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

default_trace_plugin — Environment Variable

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

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

```
show_traces [-h] [-v] [-t] [-m | -o output-file] [-p
plugin-no]
[-a | trace_number]
```

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

Command Options:

`-v`	Verbosely prints traces content (all state variables, otherwise it prints out only those variables that have changed their value from previous state). This option only applies when the Basic Trace Explainer plugin is used to display the trace.
`-t`	Prints only the total number of currently stored traces.
`-a`	Prints all the currently stored traces.
`-m`	Pipes the output through the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command to `output-file`.
`-p plugin-no`	Uses the specified trace plugin to display the trace.
`trace_number`	The (ordinal) identifier number of the trace to be printed. This must be the last argument of the command. Omitting the trace number causes the most recently generated trace to be printed.

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

read_trace - *Loads a previously saved trace* — Command

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

Command Options:

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

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

3.7 Trace Plugins

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

Basic Trace Explainer
States/Variables Table
XML Format Printer

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

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

3.7.1 Basic Trace Explainer

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

`trace_number.state_number`

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

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

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

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

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

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

3.7.2 States/Variables Table

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

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

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

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

3.7.3 XML Format Printer

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

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

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

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

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

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

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

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

    </input>
  </node>

</counter-example>
```

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

3.7.4 XML Format Reader

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

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

3.8 Interface to the DD Package

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

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

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

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.

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

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

`window2:`
`window3:`
`window4:`
	Permutes the variables within windows of *n* adjacent variables, where *n* can be either 2, 3 or 4, so as to minimize the overall BDD size.

`window2_converge:`
`window3_converge:`
`window4_converge:`
	The `window{2,3,4}` method is iterated until no further improvement is obtained.

`group_sift:`	This method is similar to `symmetry_sift`, but uses more general criteria to create groups.
`group_sift_converge:`	The `group_sift` method is iterated until no further improvement is obtained.
`annealing:`	This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.
`genetic:`	This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.
`exact:`	This method implements a dynamic programming approach to exact reordering. It only stores one BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.
`linear:`	This method is a combination of sifting and linear transformations.
`linear_conv:`	The `linear` method is iterated until no further improvement is obtained.

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

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

Controls the application and the modalities of (dynamic) variable ordering. Dynamic ordering is a technique to reorder the BDD variables to reduce the size of the existing BDDs. When no options are specified, the current status of dynamic ordering is displayed. At most one of the options `-e`, `-f`, and `-d` should be specified. Dynamic ordering may be time consuming, but can often reduce the size of the BDDs dramatically. A good point to invoke dynamic ordering explicitly (using the `-f` option) is after the commands `build_model`, once the transition relation has been built. It is possible to save the ordering found using `write_order` in order to reuse it (using `build_model -i order-file`) in the future.

Command Options:

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

- **sift**: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.
- **random**: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.
- **random_pivot**: Same as **random**, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.
- **sift_converge**: The **sift** method is iterated until no further improvement is obtained.
- **symmetry_sift**: This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.
- **symmetry_sift_converge**: The **symmetry_sift** method is iterated until no further improvement is obtained.
- **window{2,3,4}**: Permutes the variables within windows of "n" adjacent variables, where "n" can be either 2, 3 or 4, so as to minimize the overall BDD size.
- **window{2,3,4}_converge**: The **window{2,3,4}** method is iterated until no further improvement is obtained.
- **group_sift**: This method is similar to **symmetry_sift**, but uses more general criteria to create groups.
- **group_sift_converge**: The **group_sift** method is iterated until no further improvement is obtained.
- **annealing**: This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.
- **genetic**: This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.
- **exact**: This method implements a dynamic programming approach to exact reordering. It only stores a BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.

- **linear**: This method is a combination of sifting and linear transformations.
- **linear_converge**: The **linear** method is iterated until no further improvement is obtained.

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

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

`print_bdd_stats [-h]`

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

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

`set_bdd_parameters [-h] [-s]`

Applies the variables table of the NUSMV environnement to the BDD package, so the user can set specific BDD parameters to the given value. This command works in conjunction with the `print_bdd_stats` and `set` commands. `print_bdd_stats` first prints a report of the parameters and statistics of the current bdd_manager. By using the command `set`, the user may modify the value of any of the parameters of the underlying BDD package. The way to do it is by setting a value in the variable `BDD.parameter name` where `parameter name` is the name of the parameter exactly as printed by the `print_bdd_stats` command.

Command Options:

`-s`	Prints the BDD parameter and statistics after the modification.

3.9 Administration Commands

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

! - *shell_command* — Command

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

alias - *Provides an alias for a command* — Command

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

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

Command Options:

`<name>`	Alias
`<string>`	Command string

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

For example:

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

will create an alias ‘read’, execute ”read_model -i short.smv; set input_order_file short.ord”. And again:

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

will print:

```
Hi
happy birthday !
```

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

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

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

echo - *Merely echoes the arguments* Command

```
echo [-h] [-o filename [-a]] <string>
```

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

Command Options:

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

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

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

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

Command Options:

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

history - *list previous commands and their event numbers* Command `history [-h] [<num>]`

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

Command Options:

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

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

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

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

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

Command Options:

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

print_usage - *Prints processor and BDD statistics.* Command

`print_usage [-h]`

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

quit - *exits NuSMV* Command

`quit [-h] [-s]`

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

Command Options:

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

reset - *Resets the whole system.* Command

`reset [-h]`

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

set - *Sets an environment variable* Command

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

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

Command Options:

`<name>`	Variable name
`<value>`	Value to be assigned to the variable.

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

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

The last line "bar" will be the output produced by NUSMV. Variables can be extended by using the character ':' to concatenate values. For example:

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

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

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

The last line will be the output produced by NUSMV.

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

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

If a variable is not set by the `set` command, then the variable is returned unchanged. Different commands use environment information for different purposes. The command interpreter makes use of the following parameters:

Command Options:

`autoexec`	Defines a command string to be automatically executed after every command processed by the command interpreter. This is useful for things like timing commands, or tracing the progress of optimization.
`open_path`	"open_path" (in analogy to the shell-variable PATH) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory (.) is the first item in this list. The standard system library (typically `NuSMV_LIBRARY_PATH`) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.
`nusmv_stderr`	Standard error (normally (stderr)) can be re-directed to a file by setting the variable `nusmv_stderr`.
`nusmv_stdout`	Standard output (normally (stdout)) can be re-directed to a file by setting the variable `nusmv_stdout`.

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

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

Reads and executes commands from a file.

Command Options:

`-p`	Prints a prompt before reading each command.
`-s`	Silently ignores an attempt to execute commands from a nonexistent file.
`-x`	Echoes each command before it is executed.
`<file>`	File name.

Arguments on the command line after the filename are remembered but not evaluated. Commands in the script file can then refer to these arguments using the history substitution mechanism. EXAMPLE:
Contents of `test.scr`:

```
read_model -i %:2
flatten_hierarchy
build_variables
build_model
compute_fairness
```

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

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

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

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

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

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

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

See the variable `on_failure_script_quits` for further information.

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

```
time [-h]
```

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

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

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

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

Command Options:

`<alias-names>`	Aliases to be removed

unset - *Unsets an environment variable* Command

`unset [-h] <variables>`

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

Command Options:

`<variables>`	Variables to be unset.

usage - *Provides a dump of process statistics* Command

`usage [-h]`

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

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

`which [-h] <file_name>`

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

Command Options:

`<file_name>`	File to be searched

3.10 Other Environment Variables

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

autoexec Environment Variable

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

on_failure_script_quits Environment Variable

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

filec Environment Variable

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

shell_char Environment Variable

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

history_char Environment Variable

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

open_path Environment Variable

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

nusmv_stderr Environment Variable

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

nusmv_stdout Environment Variable

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`.

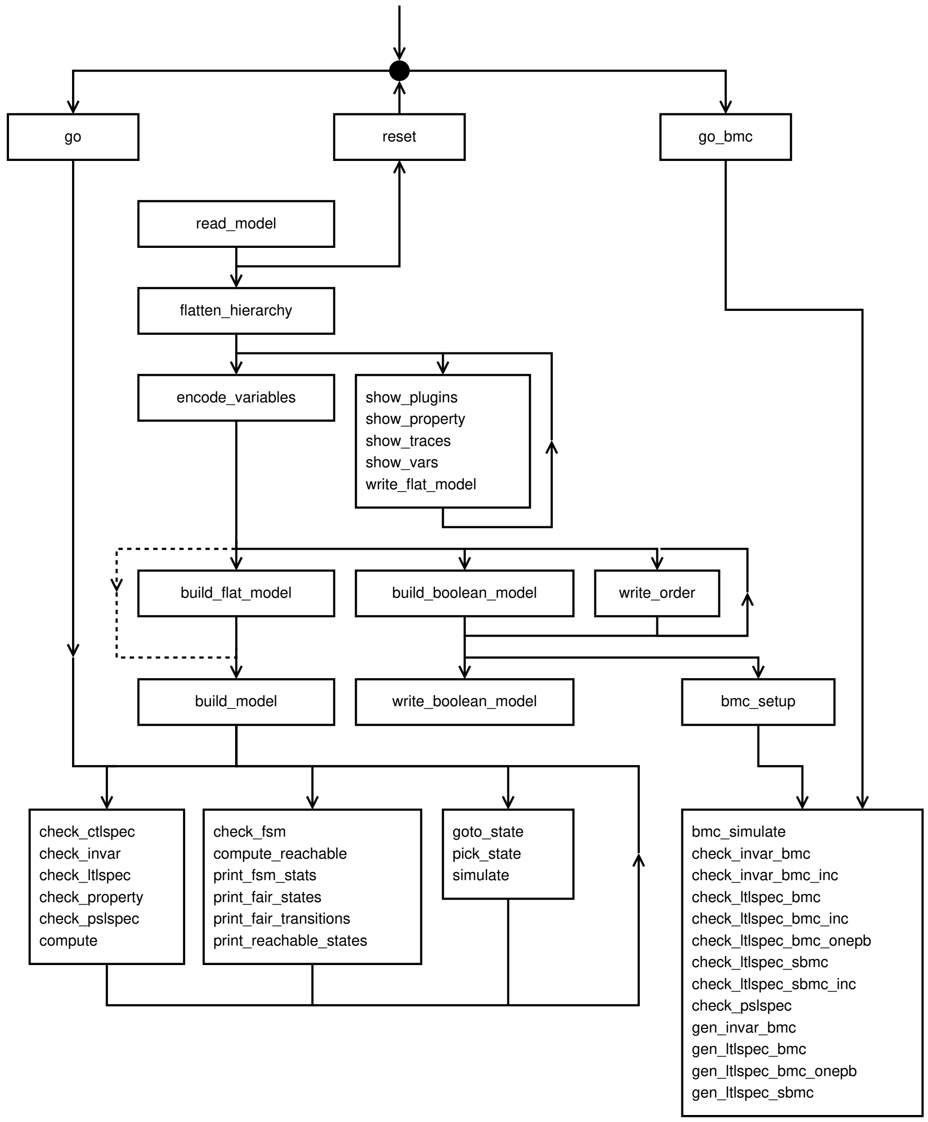

Figure 3.1: The dependency among NuSMV commands.

Chapter 4

Running NuSMV batch

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

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

The program described in *input-file* is processed, and the corresponding finite state machine is built. Then, if *input-file* contains formulas to verify, their truth in the specified structure is evaluated. For each formula which is not true a counterexample is printed.
The batch mode can be controlled with the following command line options:

```
NuSMV [-h | -help] [-v vl]
       [-s] [-old] [-old_div_op] [-dcx]
       [-cpp] [-pre pps] [-ofm fm_file] [-obm bm_file]
       [-lp] [-n idx] [-is] [-ic] [-ils] [-ips] [-ii]
       [-ctt] [[-f] [-r]]|[-df] [-flt] [-AG]  [-coi]
       [-i iv_file] [-o ov_file] [-t tv_file] [-reorder] [-dynamic] [-m method]
       [[-mono]|[-thresh cp_t]|[-cp cp_t]|[-iwls95 cp_t]]
       [-noaffinity] [-iwls95preorder]
       [-bmc] [-bmc_length k] [-sat_solver name]
       [-sin on|off] [-rin on|off]
       [ input-file]
```

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

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

`-s`	Avoids to load the NUSMV commands contained in `~/.nusmvrc` or in `.nusmvrc` or in `${NuSMV_LIBRARY_PATH }/master.nusmvrc`.
`-old`	Keeps backward compatibility with older versions of NuSMV. This option disables some new features like type checking and dumping of new extension to SMV files.
`-old_div_op`	Enables the old semantics of "`/`" and "`mod`" operations (from NUSMV 2.3.0) instead of ANSI C semantics.
`-cpp`	Runs preprocessor on SMV files before any of those specified with the -pre option.
`-pre` *pps*	Specifies a list of pre-processors to run (in the order given) on the input file before it is parsed by NUSMV. Note that if the `-cpp` command is used, then the pre-processors specified by this command will be run after the input file has been pre-processed by that pre-processor. *pps* is either one single pre-processor name (with or without double quotes) or it is a space-seperated list of pre-processor names contained within double quotes.
`-ofm` *fm_file*	prints flattened model to file *fn_file*
`-obm` *bm_file*	Prints boolean model to file *bn_file*
`-lp`	Lists all properties in SMV model
`-n` *idx*	Specifies which property of SMV model should be checked
`-is`	Does not check `SPEC`
`-ic`	Does not check `COMPUTE`
`-ils`	Does not check `LTLSPEC`
`-ips`	Does not check `PSLSPEC`
`-ii`	Does not check `INVARSPEC`
`-ctt`	Checks whether the transition relation is total.
`-f`	Computes the set of reachable states before evaluating CTL expressions. Since NuSMV-2.4.0 this option is set by default, and it is provided for backward compatibility only. See also option -df.
`-r`	Prints the number of reachable states before exiting. If the `-f` option is not used, the set of reachable states is computed.
`-df`	Disable the computation of the set of reachable states. This option is provided since NuSMV-2.4.0 to prevent the computation of reachable states that are otherwise computed by default.

`-flt`	Forces the computation of the set of reachable states for the tableau resulting from BDD-based LTL model checking (command `check_ltlspec`). If the option `-flt` is not specified (default), the resulting tableau will inherit the computation of the reachable states from the model, if enabled. If the option `-flt` is specified, the reachable states set will be calculated for the model *and* for the tableau resulting from LTL model checking. This might improve performances of the command `check_ltlspec`, but may also lead to a dramatic slowing down. This options has effect only when the calculation of reachable states is enabled (see `-f`).
`-AG`	Verifies only AG formulas using an ad hoc algorithm (see documentation for the `ag_only_search` environment variable).
`-coi`	Enables cone of influence reduction
`-i` *iv_file*	Reads the variable ordering from file *iv_file*.
`-o` *ov_file*	Reads the variable ordering from file *ov_file*.
`-t` *tv_file*	Reads a variable list from file *tv_file*. This list defines the order for clustering the transition relation. This feature has been provided by Wendy Johnston, University of Queensland. The results of Johnston's et al. research have been presented at FM 2006 in Hamilton, Canada. See [WJKWLvdBR06].
`-reorder`	Enables variable reordering after having checked all the specification if any.
`-dynamic`	Enables dynamic reordering of variables
`-m` *method*	Uses *method* when variable ordering is enabled. Possible values for method are those allowed for the `reorder_method` environment variable (see Section 3.8 [Interface to DD package], page 75).
`-mono`	Enables monolithic transition relation
`-thresh` *cp_t*	conjunctive partitioning with threshold of each partition set to *cp_t* (DEFAULT, with *cp_t*=1000)
`-cp` *cp_t*	DEPRECATED: use `thresh` instead.
`-iwls95` *cp_t*	Enables Iwls95 conjunctive partitioning and sets the threshold of each partition to *cp_t*
`-noaffinity`	Disables affinity clustering
`-iwls95preoder`	Enables *Iwls95CP* preordering
`-bmc`	Enables BMC instead of BDD model checking (works only for LTL properties and PSL properties that can be translated into LTL)
`-bmc_length` *k*	Sets `bmc_length` variable, used by BMC
`-sat_solver` *name*	Sets `sat_solver` variable, used by BMC so select the sat solver to be used.

`-sin` *on,off*	Enables (on) or disables (off) Sexp inlining, by setting system variable `sexp_inlining`. Default value is `off`.
`-rin` *on,off*	Enables (on) or disables (off) RBC inlining, by setting system variable `rbc_inlining`. Default value is `on`. The idea about inlining was taken from [ABE00] by Parosh Aziz Abdulla, Per Bjesse and Niklas Eén.

Bibliography

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on sat-solvers. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2000.

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

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

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *In J. Sifakis, editor, Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, volume 407 of LNCS, pages 365–373, Berlin*, June 1990.

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

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

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

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In *ACM Distinguished Dissertations. MIT Press*, 1988.

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

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

[KHL05] T. Junttila K. Heljanko and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference CAV 2005*, number 3576 in Lecture Notes in Computer Science, pages 98–111. Springer, 2005.

[LBHJ05] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference VMCAI 2005*, number 3385 in Lecture Notes in Computer Science, pages 380–395. Springer, 2005.

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

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

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

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

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

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

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

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

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

[WJKWLvdBR06] P. A. Strooper W. Johnston K. Winter L. van den Berg and P. Robinson. Model-based variable and transition orderings for efficient symbolic model checking. In *FM 2006: Formal Methods*, number 4085 in Lecture Notes in Computer Science, pages 524–540. Springer Berlin, 2006.

Appendix A

Compatibility with CMU SMV

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

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

`F, G, X, U, V, W, H, O, Y, Z, S, T, B`	These names are reserved for the LTL temporal operators.
`CTLSPEC`	It is used to introduce CTL specifications.
`LTLSPEC`	It is used to introduce LTL specifications.
`INVARSPEC`	It is used to introduce invariant specifications.
`PSLSPEC`	It is used to introduce PSL specifications.
`IVAR`	It is used to introduce input variables.
`JUSTICE`	It is used to introduce "justice" fairness constraints.
`COMPASSION`	It is used to introduce "compassion" fairness constraints.
`CONSTANT`	It is used to force declaration of constants.
`word`	It is used to declare word type variables.
`word1`	It is used to cast boolean expressions to word type.
`bool`	It is used to cast word1 expressions to boolean type.

The `IMPLEMENTS`, `INPUT`, `OUTPUT` statements are not no longer supported by NuSMV.

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

In particular:

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

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

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

  ```
  INVAR
  ```

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

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

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

- The check for circular dependencies among variables has been done more restrictive. We say that variable x depends on variable y if $x := f(y)$. We say that there is a circular dependency in the definition of x if:

  – x depends on itself (e.g. $x := f(x,y)$);

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

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

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

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

Another difference between NuSMV and CMU SMV is in the variable order file. The variable ordering file accepted by NuSMV can be partial and can contain variables not declared in the model. Variables listed in the ordering file but not declared in the model are simply discarded. The variables declared in the model but not listed in the variable file provided in input are created at the end of the given ordering following the default ordering. All the ordering files generated by CMU SMV are accepted in input from NuSMV but the ordering files generated by NuSMV may be not accepted by CMU SMV. Notice that there is no guarantee that a good ordering for CMU SMV is also a good ordering for NuSMV. In the ordering files for NuSMV, identifier `_process_selector_` can be used to control the position of the variable that encodes process selection. In CMU SMV it is not possible to control the position of this variable in the ordering; it is hard-coded at the top of the ordering. A further difference about variable ordering consists in the fact that in NuSMV it is allowed to specify single bits of scalar variables. In the example:

```
VAR x : 0..7;
```

NuSMV will create three variables `x.0`, `x.1` and `x.2` that can be explicitly mentioned in the variable ordering file to fine control their ordering.

Appendix B

Typing Rules

This appendix gives the explicit formal typing rules for NuSMV's input language, as well as notes on implicit conversion and casting.

In the following, an atomic constant is defined as being any sequence of characters starting with a character in the set {`A-Za-z_`} and followed by a possible empty sequence of characters from the set {`A-Za-z0-9_$#-\`}. An integer is any whole number, positive or negative.

B.1 Types

The main types recognised by NuSMV are as follows:

- boolean
- integer
- symbolic enum
- integers-and-symbolic enum
- boolean set
- integer set
- symbolic set
- integers-and-symbolic set
- word[N] (where `N` is any whole number ≥ 1)

For more detalied description of existing types see Section 2.1 [Types], page 7.

B.2 Implicit Conversion

In certain situations NuSMV is able to carry out implicit conversion of types. There are two kind of implicit convertion. The first one converts expression of one type to a greater type. The order to types is given in Figure B.1. For more information on type ordering see Section 2.2.1 [Implicit Type Conversion], page 9.

Another kind of implicit type convertions changes the type of an expression to its counterpart set type. The Figure B.2 shows the direction of such convertions. For more information on set types and their counterpart types see Section 2.1.6 [Set Types], page 8.

boolean → integer → integers-and-symbolic enum

symbolic enum → integers-and-symbolic enum

word[1]

word[2]

word[3]

...

boolean set → integer set → integers-and-symbolic set

symbolic set → integers-and-symbolic set

Figure B.1: The ordering on the types in NuSMV

boolean → boolean set
integer → integer set
symbolic enum → symbolic set
integers-and-symbolic enum → integers-and-symbolic set

Figure B.2: Implicit convertion to counterpart set types

B.3 Type Rules

The type rules are presented below with the operators on the left and the signatures of the rules on the right. To save space, more than one operator may be on the left-hand side, and it is also the case that an individual operator may have more than one signature. For more information on these expressions and their type rules see Section 2.2 [Expressions], page 9.

Constants

boolean_constant : boolean
integer_constant : integer
symbolic_constant : symbolic enum
word_constant : word[N] (where `N` is the number of bits required)
range_constant : integer set

Variable and Define

variable_identifier : `Type` (where `Type` is the type of the variable)
define_identifier : `Type` (where `Type` is the type of the define's expression)

Arithmetic Operators

`-`	: boolean $\rightarrow$ integer
	: integer $\rightarrow$ integer
	: word[N] $\rightarrow$ word[N]

The implicit type conversion can be applied to the operand.

`+`, `-`, `/`, `*`	: boolean * boolean $\rightarrow$ integer
	: integer * integer $\rightarrow$ integer
	: word[N] * word[N] $\rightarrow$ word[N]

The implicit type conversion can be applied to *one* of the operands.

`mod`	: integer * **2** $\rightarrow$ boolean
	: integer * integer $\rightarrow$ integer
	: word[N] * word[N] $\rightarrow$ word[N]

For operations on words, the result is taken modulo 2^N

`>`, `<`, `>=`, `<=`	: boolean * boolean $\rightarrow$ boolean
	: integer * integer $\rightarrow$ boolean
	: word[N] * word[N] $\rightarrow$ boolean
	: boolean * word[1] $\rightarrow$ boolean
	: word[1] * boolean $\rightarrow$ boolean

The implicit type conversion can be applied to *one* of the operands.

Logic Operators

`!` (negation)	: boolean $\rightarrow$ boolean
	: word[N] $\rightarrow$ word[N]
`&`, `|`, `->`, `<->`, `xor`, `xnor`	: boolean * boolean $\rightarrow$ boolean
	: word[N] * word[N] $\rightarrow$ word[N]
`=`, `!=`	: boolean * boolean $\rightarrow$ boolean
	: integer * integer $\rightarrow$ boolean
	: symbolic enum * symbolic enum $\rightarrow$ boolean
	: integers-and-symbolic enum * integers-and-symbolic enum $\rightarrow$ boolean
	: word[N] * word[N] $\rightarrow$ boolean
	: boolean * word[1] $\rightarrow$ boolean
	: word[1] * boolean $\rightarrow$ boolean

The implicit type conversion can be applied to *one* of the operands.

Bit-Wise Operators

`::` (concatenation)	: word[N] * word[M] $\rightarrow$ word[N+M]
	: boolean * word[N] $\rightarrow$ word[N+1]
	: word[N] * boolean $\rightarrow$ word[N+1]
exp_1`[`exp_2`,` exp_3`]`	: word[N] * integer * integer $\rightarrow$ word[$exp_3 - exp_2 + 1$]

exressions exp_2 and exp_3 must evaluate to integers such that $0 \leq exp_2 \leq exp_3 < \mathtt{N}$

`<<`, `>>` (shift)	: word[N] * integer $\rightarrow$ word[N]
	: word[N] * boolean $\rightarrow$ word[N]

Set Operators

$\{exp_1, exp_2, \ldots, exp_n\}$	: equivalent to consecutive `union` operations
`union`	: boolean set * boolean set → boolean set
	: integer set * integer set → integer set
	: symbolic set * symbolic set → symbolic set
	: integers-and-symbolic set * integers-and-symbolic set → integers-and-symbolic set

At first, if it is possible, the operands are converted to their set counterpart types, then both operands are implicitly converted to a minimal common type

`in`	: boolean set * boolean set → boolean set
	: integer set * integer set → integer set
	: symbolic set * symbolic set → symbolic set
	: integers-and-symbolic set * integers-and-symbolic set → integers-and-symbolic set

At first, if it is possible, the operands are converted to their set counterpart types, then implicit convertion is performed on one of the operands

Case Expression

```
case  cond1 : result1;
      cond2 : result2;
      ...
      condn : resultn;
esac
```

$cond_i$ must be of type boolean. If one of $result_i$ is of a set type then all other $result_k$ are converted to their counterpart set types. The overall type of the expression is such a minimal type that each $result_i$ can be implicitly converted to.

Formula Operators

`EX`, `AX`, `EF`, `AF`, `EG`, `AG`, `X`, `Y`, `Z`, `G`, `H`, `F`, `O`	: boolean → boolean
`A-U`, `E-U`, `U`, `S`	: boolean * boolean → boolean
`A-BU`, `E-BU`	: boolean * integer * integer * boolean → boolean
`EBF`, `ABF`, `EBG`, `ABG`	: integer * integer * boolean → boolean

Miscellaneous Operators

Integer..Integer	: integer_number * integer_number → integer
bool	: word[1] → boolean
word1	: boolean → word[1]
next, init	: any type → the same type
()	: any type → the same type
:=	: boolean * boolean → no type
	: boolean * boolean set → no type
	: integer * integer → no type
	: integer * integer set → no type
	: symbolic enum * symbolic enum → no type
	: symbolic enum * symbolic set → no type
	: integers-and-symbolic enum * integers-and-symbolic enum → no type
	: integers-and-symbolic enum * integers-and-symbolic set → no type
	: word[N] * word[N] → no type
	: boolean * word[1] → no type
	: word[1] * boolean → no type

Implicit type conversion is performed on the right operand only

Appendix C

Production Rules

This appendix contains the syntactic production rules for writing a NuSMV program.

Identifiers

```
identifier ::
        identifier_first_character
      | identifier identifier_consecutive_character

identifier_first_character :: one of
        A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
        a b c d e f g h i j k l m n o p q r s t u v w x y z _

identifier_consecutive_character ::
        identifier_first_character
      | digit
      | one of $ # \ -

digit :: one of 0 1 2 3 4 5 6 7 8 9
```

Note that there are certain reserved keyword which cannot be used as identifiers (see page 6).

Variable and DEFINE Identifiers

```
define_identifier :: complex_identifier

variable_identifier :: complex_identifier
```

Complex Identifiers

```
complex_identifier ::
        identifier
      | complex_identifier . identifier
      | complex_identifier [ simple_expression ]
      | self
```

Integer Numbers

```
integer_number ::
```

```
        - digit
      | digit
      | integer_number digit
```

Constants

```
constant ::
        boolean_constant
      | integer_constant
      | symbolic_constant
      | word_constant
      | range_constant

boolean_constant :: one of
        0 1 FALSE TRUE

integer_constant :: integer_number

symbolic_constant :: identifier

word_constant :: [word_width] word_base ' word_value

word_width :: integer_number (>0)

word_base :: b | B | o | O | d | D | h |  H

word_value ::
        hex_digit
      | word_value hex_digit
      | word_value _

hex_digit :: one of
        0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

Note that there are some additional restrictions on the exact format of word constants (see page 11).

```
range_constant ::
        integer_number .. integer_number
```

Basic Expressions

```
basic_expr ::
        constant                    -- a constant
      | variable_identifier         -- a variable identifier
      | define_identifier           -- a define identifier
      | ( basic_expr )
      | ! basic_expr                -- logical/bitwise NOT
      | basic_expr & basic_expr     -- logical/bitwise AND
      | basic_expr | basic_expr     -- logical/bitwise OR
      | basic_expr xor basic_expr   -- logical/bitwise exclusive OR
      | basic_expr xnor basic_expr  -- logical/bitwise NOT xor
      | basic_expr -> basic_expr    -- logical/bitwise implication
      | basic_expr <-> basic_expr   -- logical/bitwise equivalence
      | basic_expr = basic_expr     -- equality
      | basic_expr != basic_expr    -- inequality
      | basic_expr < basic_expr     -- less than
```

```
      | basic_expr > basic_expr     -- greater than
      | basic_expr <= basic_expr    -- less than or equal
      | basic_expr >= basic_expr    -- greater than or equal
      | basic_expr + basic_expr     -- integer addition
      | basic_expr - basic_expr     -- integer subtraction
      | basic_expr * basic_expr     -- integer multiplication
      | basic_expr / basic_expr     -- integer division
      | basic_expr mod basic_expr   -- integer remainder
      | basic_expr >> basic_expr    -- bit shift right
      | basic_expr << basic_expr    -- bit shift left
      | basic_expr :: basic_expr    -- word concatenation
      | basic_expr [ integer_number : integer_number ]
                                    -- word bits selection
      | word1 ( basic_expr ) -- boolean to word[1] convertion
      | bool ( basic_expr )  -- word[1] to boolean convertion
      | basic_expr union basic_expr -- union of set expressions
      | { set_body_expr }            -- set expression
      | basic_expr in basic_expr     -- inclusion expression
      | case_expr                    -- a case expression
      | next ( basic_expr )          -- a next expression

set_body_expr ::
        basic_expr
      | set_body_expr , basic_expr
```

Case Expression

```
case_expr :: case case_body esac

case_body ::
        basic_expr : basic_expr ;
      | case_body basic_expr : basic_expr ;
```

Simple Expression

```
simple_expr :: basic_expr
```

Note that simple expressions *cannot* contain **next** operators.

Next Expression

```
next_expr :: basic_expr
```

Type Specifier

```
type_specifier ::
        simple_type_specifier
      | module_type_spicifier

simple_type_specifier ::
        boolean
      | word [ integer_number ]
      | { enumeration_type_body }
      | integer_number .. integer_number
      | array integer_number .. integer_number
                of simple_type_specifier

enumeration_type_body ::
```

```
        enumeration_type_value
      | enumeration_type_body , enumeration_type_value

enumeration_type_value ::
        symbolic_constant
      | integer_number
```

Input Variable

```
ivar_declaration :: IVAR var_list
```

DEFINE Declaration

```
define_declaration :: DEFINE define_body

define_body :: identifier := simple_expr ;
             | define_body identifier := simple_expr ;
```

CONSTANTS Declaration

```
constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier
             | constants_body  , identifier
```

ASSIGN Declaration

```
assign_constraint :: ASSIGN assign_list

assign_list :: assign ;
             | assign_list assign ;

assign ::
    complex_identifier          := simple_expr
  | init ( complex_identifier ) := simple_expr
  | next ( complex_identifier ) := next_expr
```

TRANS Statement

```
trans_constraint :: TRANS next_expr [;]
```

INIT Statement

```
init_constrain :: INIT simple_expr [;]
```

INVAR Statement

```
invar_constraint :: INVAR simple_expr [;]
```

Module Declarations

```
module :: MODULE identifier [(module_parameters)] [module_body]

module_parameters ::
          identifier
```

```
        | module_parameters , identifier

module_body ::
          module_element
        | module_body module_element

module_element ::
          var_declaration
        | ivar_declaration
        | define_declaration
        | constants_declaration
        | assign_constraint
        | trans_constraint
        | init_constraint
        | invar_constraint
        | fairness_constraint
        | ctl_specification
        | invar_specification
        | ltl_specification
        | compute_specification
        | isa_declaration
```

Module Type Specifier

```
module_type_specifier ::
      | identifier [ ( [ parameter_list ] ) ]
      | process identifier [ ( [ parameter_list ] ) ]

parameter_list ::
        simple_expr
      | parameter_list , simple_expr
```

ISA Declaration

```
isa_declaration :: ISA identifier
```

Warning: this is a deprecated feature and will eventually be removed from NuSMV. Use module instances instead.

CTL Specification

```
ctl_specification :: SPEC ctl_expr ;

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 xor ctl_expr     -- logical exclusive or
    | ctl_expr -> ctl_expr      -- logical implies
    | ctl_expr <-> ctl_expr     -- logical equivalence
    | EG ctl_expr               -- exists globally
    | EX ctl_expr               -- exists next state
    | EF ctl_expr               -- exists finally
```

```
    | AG ctl_expr               -- forall globally
    | AX ctl_expr               -- forall next state
    | AF ctl_expr               -- forall finally
    | E [ ctl_expr U ctl_expr ] -- exists until
    | A [ ctl_expr U ctl_expr ] -- forall until
```

INVAR Specification

```
invar_specification :: INVARSPEC simple_expr ;
```

This is equivalent to

```
SPEC  AG simple_expr ;
```

but is checked by a specialised algorithm during reachability analysis.

LTL Specification

```
ltl_specification :: LTLSPEC ltl_expr [;]

ltl_expr ::
    simple_expr              -- a simple boolean expression
    | ( ltl_expr )
    | ! ltl_expr             -- logical not
    | ltl_expr & ltl_expr    -- logical and
    | ltl_expr | ltl_expr    -- logical or
    | ltl_expr xor ltl_expr  -- logical exclusive or
    | ltl_expr -> ltl_expr   -- logical implies
    | ltl_expr <-> ltl_expr  -- logical equivalence
    -- FUTURE
    | X ltl_expr             -- next state
    | G ltl_expr             -- globally
    | F ltl_expr             -- finally
    | ltl_expr U ltl_expr    -- until
    | ltl_expr V ltl_expr    -- releases
    -- PAST
    | Y ltl_expr             -- previous state
    | Z ltl_expr             -- not previous state not
    | H ltl_expr             -- historically
    | O ltl_expr             -- once
    | ltl_expr S ltl_expr    -- since
    | ltl_expr T ltl_expr    -- triggered
```

Real Time CTL Specification

```
rtctl_specification :: SPEC rtctl_expr [;]

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  :: integer_number .. integer_number
```

It is also possible to compute quantative information for the FSM:

```
compute_specification :: COMPUTE compute_expr [;]

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

PSL Specification

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

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

psl_primary_expr ::
   number                               ;; a numeric constant
 | boolean                              ;; a boolean constant
 | var_id                               ;; a variable identifier
 | { psl_expr , ... , psl_expr }
 | { psl_expr "{" psl_expr , ... , "psl_expr" }}
 | ( psl_expr )
psl_unary_expr ::
   + psl_primary_expr
 | - psl_primary_expr
 | ! psl_primary_expr
psl_binary_expr ::
   psl_expr + psl_expr
 | psl_expr union psl_expr
 | psl_expr in psl_expr
 | psl_expr - psl_expr
 | psl_expr * psl_expr
 | psl_expr / psl_expr
 | psl_expr % psl_expr
 | psl_expr == psl_expr
 | psl_expr != psl_expr
 | psl_expr < psl_expr
 | psl_expr <= psl_expr
 | psl_expr > psl_expr
 | psl_expr >= psl_expr
 | psl_expr & psl_expr
 | psl_expr | psl_expr
 | psl_expr xor psl_expr
psl_conditional_expr ::
 psl_expr ? psl_expr : psl_expr
psl_case_expr ::
 case
    psl_expr : psl_expr ;
    ...
    psl_expr : psl_expr ;
 endcase
```

Among the subclasses of psl_expr we depict the class psl_bexpr that will be used in the following to identify purely boolean, i.e. not temporal, expressions.

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

FL_property ::
 ;; PRIMITIVE LTL OPERATORS
   X FL_property
 | X! FL_property
 | F FL_property
 | G FL_property
 | [ FL_property U FL_property ]
 | [ FL_property W FL_property ]
 ;; SIMPLE TEMPORAL OPERATORS
 | always FL_property
 | never FL_property
 | next FL_property
 | next! FL_property
 | eventually! FL_property
 | FL_property until! FL_property
 | FL_property until FL_property
 | FL_property until!_ FL_property
 | FL_property until_ FL_property
 | FL_property before! FL_property
 | FL_property before FL_property
 | FL_property before!_ FL_property
 | FL_property before_ FL_property
 ;; EXTENDED NEXT OPERATORS
 | X [number] ( FL_property )
 | X! [number] ( FL_property )
```

```
 | next [number] ( FL_property )
 | next! [number] ( FL_property )
 ;;
 | next_a [range] ( FL_property )
 | next_a! [range] ( FL_property )
 | next_e [range] ( FL_property )
 | next_e! [range] ( FL_property )
 ;;
 | next_event! ( psl_bexpr ) ( FL_property )
 | next_event ( psl_bexpr ) ( FL_property )
 | next_event! ( psl_bexpr ) [ number ]  ( FL_property )
 | next_event ( psl_bexpr ) [ number ]  ( FL_property )
 ;;
 | next_event_a! ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_a ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_e! ( psl_bexpr ) [psl_expr]  ( FL_property )
 | next_event_e ( psl_bexpr ) [psl_expr]  ( FL_property )
 ;; OPERATORS ON SEREs
 | sequence ( FL_property )
 | sequence |-> sequence [!]
 | sequence |=> sequence [!]
 ;;
 | always sequence
 | G sequence
 | never sequence
 | eventually! sequence
 ;;
 | within! ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within!_ ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 | within_ ( sequence_or_psl_bexpr , psl_bexpr ) sequence
 ;;
 | whilenot! ( psl_bexpr ) sequence
 | whilenot ( psl_bexpr ) sequence
 | whilenot!_ ( psl_bexpr ) sequence
 | whilenot_ ( psl_bexpr ) sequence
sequence_or_psl_bexpr ::
   sequence
 | psl_bexpr

sequence ::
   { SERE }
SERE ::
   sequence
 | psl_bexpr
 ;; COMPOSITION OPERATORS
 | SERE ; SERE
 | SERE : SERE
 | SERE & SERE
 | SERE && SERE
 | SERE | SERE
 ;; RegExp QUALIFIERS
 | SERE [* [count] ]
 | [* [count] ]
 | SERE [+]
```

```
 | [+]
 ;;
 | psl_bexpr [= count ]
 | psl_bexpr [-> count ]
count ::
   number
 | range

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

Command Index

Variable Index

Index