NuSMV 2.6 User Manual

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

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

Email: nusmv@fbk.eu
This document is part of the distribution package of the NuSMV model checker, available at [http://nusmv.fbk.eu](http://nusmv.fbk.eu).

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

Copyright ©1998-2005 by CMU and ITC-irst.
Copyright ©2010 by FBK-irst.
Contents

1 Introduction 4

2 Input Language 6
  2.1 Types Overview ......................... 7
    2.1.1 Boolean .......................... 7
    2.1.2 Integer .......................... 7
    2.1.3 Enumeration Types ................. 7
    2.1.4 Word ............................. 8
    2.1.5 Array ............................. 8
    2.1.6 Set Types ........................ 8
    2.1.7 Type Order ....................... 9
  2.2 Expressions ......................... 9
    2.2.1 Implicit Type Conversion .......... 10
    2.2.2 Constant Expressions ............. 10
    2.2.3 Basic Expressions ................. 12
    2.2.4 Simple and Next Expressions ..... 21
    2.2.5 Type conversion operators ...... 21
  2.3 Definition of the FSM ............... 23
    2.3.1 Variable Declarations .......... 23
    2.3.2 DEFINE Declarations ............ 27
    2.3.3 Array Define Declarations ....... 27
    2.3.4 CONSTANTS Declarations ......... 27
    2.3.5 INIT Constraint .................. 28
    2.3.6 INVAR Constraint ................ 28
    2.3.7 TRANS Constraint ................. 28
    2.3.8 ASSIGN Constraint ............... 28
    2.3.9 FAIRNESS Constraints ............. 30
    2.3.10 MODULE Declarations .......... 31
    2.3.11 MODULE Instantiations ......... 31
    2.3.12 References to Module Components (Variables and Defines) 32
    2.3.13 Processes ...................... 34
    2.3.14 A Program and the main Module .. 34
    2.3.15 Namespaces and Constraints on Declarations 34
    2.3.16 Context ....................... 36
    2.3.17 ISA Declarations ............... 36
  2.4 Specifications ..................... 36
    2.4.1 CTL Specifications ............... 36
    2.4.2 Invariant Specifications ........ 38
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 [CCG02]. 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[2] 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.

---

1(see http://www.opensource.org)

2However, asynchronous processes are deprecated in version 2.5.0 and later, and may be no longer supported in future versions.
• **Architecture.** A software architecture has been defined. The different components and functionalities of **NuSMV** have been isolated and separated in modules. Interfaces between modules have been provided. This 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 48 the commands of the interaction shell are described.

• In Chapter 4 [Running **NuSMV** batch], page 114 we define the batch mode of **NuSMV**.

**NuSMV** is available at [http://nusmv.fbk.eu](http://nusmv.fbk.eu)
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 char-
acters 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 new-
line (<RET>). Any string starting with two dashes (‘--’) and ending with a newline
is a comment and ignored by the parser. The multiline comment starts with ‘/--’ and
ends with ‘--/’.

The syntax rule for an identifier is:

identifier :::
    identifier_first_character
    | identifier identifier_consecutive_character

identifier_first_character :: one of
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
    abcdefghijklmnopqrstuvwxyz

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

\[
\text{integer\_number} ::
| \text{digit}
| \text{digit}\text{, integer\_number digit}
\]

and *symbolic constants* which are identifiers

\[
\text{symbolic\_constant} :: \text{identifier}
\]

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.*

### 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 symbolic values *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 this type and *integer numbers* can only be in the range \(-2^{31} + 1\) to \(2^{31} - 1\) (more accurately, these values are equivalent to the C/C++ macros \texttt{INT\_MIN}+1 and \texttt{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 an 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.

Enumerations cannot contain any boolean value (i.e. \{FALSE, TRUE\}). boolean type must be declared as boolean.

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 unsigned \texttt{word[\ast]} and signed \texttt{word[\ast]} types are used to model vector of bits (booleans) which allow bitwise logical and arithmetic operations (unsigned and signed, respectively). These types are distinguishable by their width. For example, type \texttt{unsigned word[3]} represents vector of three bits, which allows unsigned operations, and type \texttt{signed word[7]} represents vector of seven bits, which allows signed operations.

When values of \texttt{unsigned word[N]} are interpreted as integer numbers the bit representation used is the most popular one, i.e. each bit represents a successive power of 2 between 0 (bit number 0) and \(2^{N-1}\) (bit number \(N-1\)). Thus \texttt{unsigned word[N]} is able to represent values from 0 to \(2^N - 1\).

The bit representation of \texttt{signed word[N]} type is “two’s complement”, i.e. it is the same as for \texttt{unsigned word[N]} except that the highest bit (number \(N-1\)) has value \(-2^{N-1}\). Thus the possible value for \texttt{signed word[N]} are from \(-2^{N-1}\) to \(2^{N-1} - 1\).

### 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,

\begin{verbatim}
array 0..3 of boolean
array 10..20 of \{OK, y, z\}
array 1..8 of array -1..2 of unsigned word[5]
\end{verbatim}

The type \texttt{array 1..8 of array -1..2 of unsigned 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 unsigned words.

Array subtype is the immediate subtype of an array type. For example, subtype of \texttt{array 1..8 of array -1..2 of unsigned word[5]} is \texttt{array -1..2 of unsigned word[5]} which has its own subtype \texttt{unsigned word[5]}.

Array types are incompatible with set type, i.e. array elements cannot be of set type.

Expression of array type can be constructed with array \texttt{DEFINE} (see 2.3.3) or variables of array type (see 2.3.1).

### 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 expressions can have immediate 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 unsigned word[•] and signed word[•] do not have a set type counterpart.

2.1.7 Type Order

Figure 2.1 depicts the order existing between types in NuSMV.

```
boolean integer symbolic enum             unsigned word[1]
   ↓  ↓                                 ↑     ↓
integers-and-symbolic enum               unsigned word[2]
                           ↓  ↓
signed word[1]                           integers-and-symbolic enum
                          ↓  ↓
signed word[2]                          signed word[2]
                          ↓  ↓
signed word[3]                          signed word[3]
                          ↓  ↓...
array N1..M1 of subtype1
   ↓                  N1=N2   subtype1
array N2..M2 of subtype2
   ↓                  if and only if
     M1=M2, and     subtype2
```

Figure 2.1: The ordering on the types in NuSMV

It means, for example, that integer is less than integers-and-symbolic enum, symbolic enum
is less than integers-and-symbolic enum, etc. The unsigned word[•] and signed word[•]
any other type or between each other. Any type is equal to itself.

Note that enumerations containing only integer numbers have the type integer.

For 2 arrays types array N1..M1 of subtype1 and array N2..M2 of
subtype2 the first type is less than the second one if and only if N1=N2, M1=M2 and type
subtype1 is less than subtype2.

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

\[1\]For more information on these operators see pages 12, 18, 19, 20 and 28 respectively.
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 corresponding `-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 50 for further information. If additionally, the system variable `type_checking_warning_on` is `unset`, then even these warnings will not be printed out.

2.2.1 Implicit Type Conversion

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.

**Note:** Prior to version 2.5.1, implicit type conversion from `integer` to `boolean` (and vice versa) was performed. Since version 2.5.1, implicit `integer` to `boolean` type conversion is no longer supported, and explicit cast operators have to be used.

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 symbolic values `FALSE` and `TRUE`. The type of a boolean constant is `boolean`.

```
boolean_constant :: one of FALSE TRUE
```

### Integer Constant

An integer constant is an integer number. 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 2.3.15 [Namespaces], page 34 for more information about how symbolic constants are distinguished from other identifiers, i.e. variables, defines, etc.
**Word Constant**

Word constant begins with digit 0, followed by optional character u (unsigned) or s (signed) and one of the characters bB (binary), oO (octal), dD (decimal) or hH (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. Assuming N is the width of the constant the type of a word constant is signed word[N] if character s is provided, and unsigned word[N] otherwise. For example:

- `0sb5_10111` has type signed word[5]
- `0uo6_37` has type unsigned word[6]
- `0d11_9` has type unsigned word[11]
- `0sh12_a9` has type signed 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 unsigned word[8]:

- `0ud8_11`
- `0ub8_1011`
- `0b_00001011`
- `0h_0b`
- `0h8_0b`

The syntactic rule of the word constant is the following:

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

word_sign_specifier :: one of
    u s

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_01011111100`. 
• For a given width $N$ the value of a constant has to be in range $0 \ldots 2^N - 1$. For decimal signed words (both $s$ and $d$ are provided) the value of a constant has to be in range $0 \ldots 2^{N-1}$.

• 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
   | abs ( basic_expr ) -- absolute value
   | max ( basic_expr , basic_expr ) -- max
   | min ( basic_expr , basic_expr ) -- min
   | 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 -- less than or equal
   | basic_expr > basic_expr -- greater than
   | 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 [ index ] -- index subscript
   | basic_expr [ basic_expr : basic_expr ] -- word bits selection
```
| basic_expr :: basic_expr -- word concatenation |
| word1 ( basic_expr ) -- boolean to unsigned word[1] conversion |
| bool ( basic_expr ) -- unsigned word[1] and int to boolean conversion |
| toint ( basic_expr ) -- word and boolean to integer constant conversion |
| count ( basic_expr, basic_expr ) -- count of true boolean expressions |
| swconst ( basic_expr ) -- integer to signed word constant conversion |
| uwconst ( basic_expr, basic_expr ) -- integer to unsigned word constant conversion |
| signed ( basic_expr ) -- unsigned word to signed word conversion |
| unsigned ( basic_expr ) -- signed word to unsigned word conversion |
| sizeof ( basic_expr ) -- word size as an integer |
| extend ( basic_expr, basic_expr ) -- word width extension |
| resize ( basic_expr, basic_expr ) -- word width resize |
| basic_expr union basic_expr -- union of set expressions |
| { set_body_expr } -- set expression |
| basic_expr .. basic_expr -- pure integer set expression |
| basic_expr in basic_expr -- inclusion in a set expression |
| basic_expr ? basic_expr : basic_expr -- if-then-else expression |
| case_expr -- case expression |
| basic_next_expr -- 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 10.

Variables and Defines

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

Variables

A variable_identifier is an expression that identifies a variable.

Define

A define_identifier is an expression that identifies a define.

Rules:

1. A variable_identifier must be a valid identifier.
2. A define_identifier must be a valid identifier.
3. The identifier must not be a reserved keyword.

Examples:

- variable_identifier: my_var
- define_identifier: my_define

Constants

A constant is an expression that represents a fixed value.

Types:

1. Integer Constants
2. Boolean Constants
3. Signed Constants
4. Unsigned Constants

Examples:

- Integer Constants: 0, 1, 2, ..., 100
- Boolean Constants: true, false
- Signed Constants: -1, -2, ..., -128, 128
- Unsigned Constants: 0, 1, 2, ..., 127
define_identifier :: complex_identifier

variable_identifier :: complex_identifier

The syntax and semantics of complex_identifiers are explained in Section 2.3.12 [References to Module Components], page 32. 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 27 and Section 2.3.1 [Variable Declarations], page 25.

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. Define expressions may contain next operators; Normal rules apply: No nested next operators.

variable_identifier represents state, input, and frozen 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 23, Section 2.3.1 [State Variables], page 24, Section 2.3.1 [Input Variables], page 24, and Section 2.3.1 [Frozen Variables], page 25. 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
    : unsigned word[N] → unsigned word[N]
    : signed word[N] → signed word[N]

This means that the operation can be applied to boolean, unsigned word[•] and signed word[•] operands. The type of the whole expression is the same as the type of the operand. If the operand is not boolean, unsigned word[•] or signed 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
    : unsigned word[N] * unsigned word[N] → unsigned word[N]
    : signed word[N] * signed word[N] → signed word[N]

the operands can be of boolean, unsigned word[•] or signed 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
- =, != : unsigned word[N] * unsigned word[N] → boolean
- =, != : signed word[N] * signed word[N] → boolean

No implicit type conversion is performed. 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 not correct, because no implicit type conversion will be performed. One can use the toint or the bool for explicit casts.

For example:

toint(TRUE) = 5

or

TRUE = bool(5)

This is also true if one of the operands is of type unsigned word[1] and the other one is of the type boolean. Explicit cast must be used (e.g. using word1 or bool)

Relational Operators >, <, >=, <=

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

- >, <, >=, <= : integer * integer → boolean
- >, <, >=, <= : unsigned word[N] * unsigned word[N] → boolean
- >, <, >=, <= : signed word[N] * signed word[N] → boolean

Arithmetic Operators +, -, *, /

The arithmetic operators + (addition), - (unary negation or binary subtraction), * (multiplication) and / (division) have the following signature:

- +, -, *, / : integer * integer → integer
- +, -, *, / : unsigned word[N] * unsigned word[N] → unsigned word[N]
- +, -, *, / : signed word[N] * signed word[N] → signed word[N]
- - (unary) : integer → integer
- - (unary) : unsigned word[N] → unsigned word[N]
- - (unary) : signed word[N] → signed word[N]

Before checking the expression for being correctly typed, the implicit type conversion can be applied to one of the operands. If the operators are applied to unsigned word[N] or signed 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. The result of the / operator is the algebraic quotient with any fractional part discarded (this is often called "truncation towards zero"). If the quotient $a/b$ 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 \texttt{mod}

The result of the \texttt{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:

\[
\texttt{mod} : \text{integer} \times \text{integer} \rightarrow \text{integer} \\
\text{unsigned word}[N] \times \text{unsigned word}[N] \rightarrow \text{unsigned word}[N] \\
\text{signed word}[N] \times \text{signed word}[N] \rightarrow \text{signed word}[N]
\]

The semantics of \texttt{mod} operator is equivalent to the corresponding operator \% of C/C++ languages. Thus if the quotient \(a/b\) is representable, the expression \((a/b) \times b + (a \mod b)\) shall equal \(a\).

\textbf{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 \texttt{mod} be of the current, i.e. C/C++’s, semantics the older semantics of division was given by the formula:

\[
\text{IF } (a \mod b < 0) \text{ THEN } (a / b - 1) \text{ ELSE } (a / b)
\]

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

\[
\text{IF } (a \mod b < 0) \text{ THEN } (a \mod b + b) \text{ ELSE } (a \mod b)
\]

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

\[
\begin{align*}
7/5 &= 1 & 7 \text{ mod } 5 &= 2 \\
-7/5 &= -1 & -7 \text{ mod } 5 &= -2 \\
7/-5 &= 1 & 7 \text{ mod } -5 &= 2 \\
-7/-5 &= 0 & -7 \text{ mod } -5 &= -7
\end{align*}
\]

whereas in the older versions on NuSMV the equations were

\[
\begin{align*}
7/5 &= 1 & 7 \text{ mod } 5 &= 2 \\
-7/5 &= -2 & -7 \text{ mod } 5 &= 3 \\
7/-5 &= 1 & 7 \text{ mod } -5 &= 2 \\
-7/-5 &= 0 & -7 \text{ mod } -5 &= -7
\end{align*}
\]

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

\textbf{Note:} semantics of modulo operator can be obtained from the remainder operator by exploiting the equivalence:

\[
(n \text{ modulo } M) \equiv ((n \text{ mod } M) + M) \text{ mod } M
\]

Where \texttt{mod} is the remainder operator.

Shift Operators \texttt{<<, >>}

The signature of the shift operators is:

\[
\texttt{<<, >>} : \text{unsigned word}[N] \times \text{integer} \rightarrow \text{unsigned word}[N] \\
\text{signed word}[N] \times \text{integer} \rightarrow \text{signed word}[N] \\
\text{unsigned word}[N] \times \text{unsigned word}[M] \rightarrow \text{unsigned word}[N] \\
\text{signed word}[N] \times \text{unsigned word}[M] \rightarrow \text{signed word}[N]
\]

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

Left shift \texttt{<<} (right shift \texttt{>>}) operation shifts to the left (right) the bits of the left operand by the number specified in the right operand. A shift by \(N\) bits is equivalent to \(N\) shifts by 1 bit. A bit shifted behind the word bound is lost. During shifting a word is padded with zeros with the exception of the right shift for \texttt{signed word}[\texttt{*}], in which case a word is padded with its highest bit. For instance,
0ub4_0101 << 2 is equal to 0sb3_1011 >> 2 is equal to
0ub4_0100 << 1 is equal to 0sb3_1110 >> 1 is equal to
0ub4_1000 << 0 is equal to 0sb3_1111 >> 0 is equal to
0ub4_1000 and 0sb3_1111

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

Index Subscript Operator [ ]

The index subscript operator extracts one element of an array in the typical fashion. On the left of [ ] there has to be an expression of array type. The index expression in the brackets has to be an expression of integer or word[] type with value greater or equal to lower bound and less or equal to the upper bound of the array. The signature of the index subscript operator is:

```
array N..M of subtype * word[N] → subtype
```

For example, for below declarations:

```
MODULE main

VAR a : array -1 .. 4 of array 1 .. 2 of boolean;
DEFINE d := [[12, 4], [-1,2]];
VAR r : 0..1;
```

expressions a[-1], a[0][r+1] and d[r][1] are valid whereas a[0], a[0][r] and d[0][r-1] will cause out of bound error.

Bit Selection Operator [ : ]

The bit selection operator extracts consecutive bits from a unsigned word[] or signed word[] expression, resulting in a new unsigned word[] expression. This operation always decreases the width of a word or leaves it intact. The expressions in the brackets have to be integer constants which specify the high and 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 unsigned 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, 0sb7_1011001[4:1] extracts bits 1 through 4 (including 1st and 4th bits) and is equal to 0ub4_1100. 0ub3_101[0:0] extracts bit number 0 and is equal to 0ub1_1.

The signature of the bit selection operator is:

```
[ : ]: unsigned word[N] * integer[ ] * integer[ ] → unsigned word[integer[ ] − integer[ ] + 1]
```

```
signed word[N] * integer[ ] * integer[ ] → unsigned word[integer[ ] − integer[ ] + 1]
```

where \( 0 \leq \text{integer}_1 \leq \text{integer}_2 < N \)

Word Concatenation Operator ::

The concatenation operator joins two words (unsigned word[] or signed word[] or both) together to create a larger unsigned word[] type. The operator itself is two colons (::), and its signature is as follows:

```
:: : word[M] * word[N] → unsigned word[M+N]
```

\(^2\)See 2.3.3 for array defines and \(^2\) for array variables.
where \( \text{word}[N] \) is unsigned \( \text{word}[N] \) or signed \( \text{word}[N] \). 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. The result is always unsigned \( \text{word}[*] \). For example, given the two words \( w_1 := \text{0ub}4_{1101} \) and \( w_2 := \text{0sb}2_{00} \), the result of \( w_1::w_2 \) is \( \text{0ub}6_{110100} \).

**Word sizeof Operator**

`sizeof` operator provides a very simple way for retrieving the width of a word. The behavior of this operator can be described as follows:

let \( w \) be a \( \text{word}[N] \):

\[
\text{sizeof}(w) \text{ returns } N.
\]

The signature of the operator is:

\[
\text{sizeof} : \text{unsigned word}[*] \rightarrow \text{integer} \\
\text{sizeof} : \text{signed word}[*] \rightarrow \text{integer}
\]

**Extend Word Conversions**

`extend` operator increases the width of a word by attaching additional bits on the left. If the provided word is unsigned then zeros are added, otherwise if the word is signed the highest (sing) bit is repeated corresponding number of times.

The signature of the operator is:

\[
\text{extend} : \text{unsigned word}[N] * \text{integer} \rightarrow \text{unsigned word}[N+\text{integer}] \\
\text{extend} : \text{signed word}[N] * \text{integer} \rightarrow \text{signed word}[N+\text{integer}]
\]

For example:

\[
\text{extend}(\text{0ub}3_{101}, 2) = \text{0ub}5_{00101} \\
\text{extend}(\text{0sb}3_{101}, 2) = \text{0sb}5_{11101} \\
\text{extend}(\text{0sb}3_{011}, 2) = \text{0sb}5_{00011}
\]

Note that the right operand of `extend` has to be an integer constant greater or equal to zero.

**Resize Word Conversions**

`resize` operator provides a more comfortable way of changing the width of a word. The behavior of this operator can be described as follows:

let \( w \) be a \( M \) bits unsigned \( \text{word}[*] \) and \( N \) be the required width: if \( M = N \), \( w \) is returned unmodified; if \( N \) is less than \( M \), bits in the range \([N-1:0]\) are extracted from \( w \); if \( N \) is greater than \( M \), \( w \) is extended of \((N - M)\) bits up to required width, padding with zeroes.

let \( w \) be a \( M \) bits signed \( \text{word}[*] \) and \( N \) be the required width: if \( M = N \), \( w \) is returned unmodified; if \( N \) is less than \( M \), bits in the range \([N-2:0]\) are extracted from \( w \), while \( N-1 \)-ith bit is forced to preserve the value of the original sign bit of \( w \) (\( M-1 \)-ith bit); if \( N \) is greater than \( M \), \( w \) is extended of \((N - M)\) bits up to required width, extending sign bit.

The signature of the operator is:

\[
\text{resize} : \text{unsigned word}[N] * \text{integer} \rightarrow \text{unsigned word}[\text{integer}] \\
\text{resize} : \text{signed word}[N] * \text{integer} \rightarrow \text{signed word}[\text{integer}]
\]

**Set Expressions**

The set expression is an expression defining a set of \text{boolean}, \text{integer} and \text{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:

```plaintext
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[^1] 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:

```plaintext
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.

Yet another way to write a set expression is to use the binary operator `..` (\texttt{<TWO DOTS>}). The two operands have both to be expressions that evaluate to constants integer numbers, and may contain names of defines and module formal parameters. For example, `-1 - P1 .. 5 + D1`, where `P1` refers to a module formal parameter, and `D1` refers to a define. Both `P1` and `D1` have to be statically evaluable to integer constants.

This is just a shorthand for a set expression containing the list of integer numbers included between the lower and the upper bound. For example, `-1..5` and `{-1,0,1,2,3,4,5}` are equivalent. Note that the evaluated number on the left from the two dots must be less than or equal to the evaluated number on the right.

Set expressions can be used only as operands of **union** and **in** operations, as the right operand of **case** and as the second and the third operand of (** ? • : •**) 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:

```plaintext
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 [^1]:See 2.1.6 for more information about the *set* types and their counterpart types

[^1]:See 2.1.6 for more information about the *set* types and their counterpart types
possible, both operands are converted to their counterpart set types. Then, if required, implicit type conversion is carried out on one of the operands.

Case Expressions

A case expression has the following syntax:

\[
\text{case_expr :: case case_body esac}
\]

\[
\text{case_body :: basic_expr : basic_expr ;}
| \text{case_body basic_expr : basic_expr ;}
\]

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 TRUE. For example, the result of the expression

\[
\text{case left_expression_1 : right_expression_1 ; left_expression_2 : right_expression_2 ; ... left_expression_N : right_expression_N ; esac}
\]

is right_expression_k such that for all \(i \text{ from } 0 \text{ to } k-1\), left_expression_i is FALSE, and left_expression_k is TRUE. It is an error if all expressions on the left hand side evaluate to FALSE.

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. The type of the whole expression is such a minimal type that all of the expressions on the right (after possible conversion to set types) can be implicitly converted to this type. If this is not possible, NuSMV throws an error.

Note: Prior to version 2.5.1, using 1 as left_expression_N was pretty common, e.g:

\[
\text{case cond1 : expr1; cond2 : expr2; ... 1 : exprN; -- otherwise esac}
\]

Since version 2.5.1 integer values are no longer implicitly casted to boolean, and 1 has to be written as TRUE instead. For backward compatibility options, please see page 50.

If-Then-Else expressions

In certain cases, the syntax described above may look a bit awkward. In simpler cases, it is possible to use the alternative, terser, (• ? • : •) expression. This construct is defined as follows:

\[
\text{cond_expr ? basic_expr1 : basic_expr2}
\]

This expression evaluates to basic_expr1 if the condition in cond_expr evaluates to true, and to basic_expr2 otherwise. Therefore, the expressions cond1 ? exp1 : exp2 and case cond1 : expr1; TRUE : expr2; esac are equivalent.

4See 2.1.6 for more information about the set types and their counterpart types
5See 2.1.6 for information on set types and their counterpart types
6See Section 2.1.7 [Type Order], page 9 for the information on the order of types.
Basic Next Expression

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

The syntactic rule is:

$$\text{basic\_next\_expr} ::= \text{next}\ (\ \text{basic\_expr}\ )$$

A next expression does not change the type.

Count Operator

The $\text{count}$ operator counts the number of expressions which are true. The $\text{count}$ operator is a syntactic sugar for

$$\begin{align*}
\text{toint} \ (\text{bool\_expr1}) & + \\
\text{toint} \ (\text{bool\_expr2}) & + \\
\ldots & + \\
\text{toint} \ (\text{bool\_exprN})
\end{align*}$$

This operator has been introduced in version 2.5.1, to simplify the porting of those models which exploited the implicit casting of integer to boolean to encoding e.g. predicates like:

$$(b0 + b1 + \ldots + bN) < 3 \quad \text{-- at most two bits are enabled}$$

Since version 2.5.1, this expression can be written as:

$$\text{count}(b0, b1\ldots, bN) < 3$$

2.2.4 Simple and Next Expressions

Simple expressions are expressions built only from the values of variables in the current state. Therefore, the simple_expression cannot have a $\text{next}$ operation inside and the syntax of simple_expression is as follows:

$$\text{simple\_expr} ::= \text{basic\_expr}$$

with the alternative $\text{basic\_next\_expr}$ not allowed. Simple expressions can be used to specify sets of states, for example, the initial set of states. The $\text{next\_expression}$ relates current and next state variables to express transitions in the FSM. The $\text{next\_expression}$ can have $\text{next}$ operation inside, i.e.

$$\text{next\_expr} ::= \text{basic\_expr}$$

with the alternative $\text{basic\_next\_expr}$ allowed.

2.2.5 Type conversion operators

Integer conversion operator

$\text{toint}$ converts an unsigned word[•] constant or a signed word[•] constant, or a boolean expression to an integer representing its value. Also integer expressions are allowed, but no action is performed. The signature of this conversion operator is:

21
toint : integer → integer
toint : boolean → integer
toint : unsigned word[*] → integer
toint : signed word[*] → integer

Warning: using the toint operator with word variables may cause bad performances of the system. Performances may degrade with the increase of the number of bits of the word expression.

Boolean conversion operator

bool converts unsigned word[1] and integer expressions to boolean. Also boolean expressions are allowed, but no action is performed. In case of integer expression, the result of the conversion is FALSE if the expression resolves to 0, TRUE otherwise. In case of unsigned word[1] expression, the conversion obeys the following table:

\[
\begin{align*}
\text{bool}(0\text{ub}1_0) &= \text{FALSE} \\
\text{bool}(0\text{ub}1_1) &= \text{TRUE}
\end{align*}
\]

Integer to Word Constants Conversion

swconst, uwconst convert an integer constant into a signed word[*] constant or unsigned word[*] constant of given size respectively. The signature of these conversion operator is:

\[
\begin{align*}
\text{swconst} &: \text{integer} \rightarrow \text{signed word[*]} \\
\text{uwconst} &: \text{integer} \rightarrow \text{unsigned word[*]}
\end{align*}
\]

Where the left integer parameter is the value and the right integer parameter is the size in bits of the generated unsigned word[*] or signed word[*] constant.

Word1 Explicit Conversions

word1 converts a boolean to a unsigned word[1]. The signature of this conversion operator is:

\[
\begin{align*}
\text{word1} &: \text{boolean} \rightarrow \text{unsigned word[1]}
\end{align*}
\]

The conversion obeys the following table:

\[
\begin{align*}
\text{word1}(\text{FALSE}) &= 0\text{ub}1_0 \\
\text{word1}(\text{TRUE}) &= 0\text{ub}1_1
\end{align*}
\]

Unsigned and Signed Explicit Conversions

unsigned converts a signed word[N] to an unsigned word[N], while signed performs the opposite operation and converts an unsigned word[N] to a signed word[N]. Both operations do not change the bit representation of a provided word. The signatures of these conversion operators are:

\[
\begin{align*}
\text{unsigned} &: \text{signed word[N]} \rightarrow \text{unsigned word[N]} \\
\text{signed} &: \text{unsigned word[N]} \rightarrow \text{signed word[N]}
\end{align*}
\]

For example:

\[
\begin{align*}
\text{signed}(0\text{ub}_101) &= 0\text{sb}_101 \\
\text{signed}(0\text{ud}_35) &= -0\text{sd}_33 \\
\text{unsigned}(0\text{sb}_101) &= 0\text{ub}_101 \\
\text{unsigned}(-0\text{sd}_33) &= 0\text{ud}_35
\end{align*}
\]
2.3 Definition of the FSM

We consider a Finite State Machine (FSM) described in terms of state variables, input variables, and frozen 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, a frozen, 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:

typeSpecifier ::

simpleTypeSpecifier
| moduleTypeSpecifier

simpleTypeSpecifier ::

boolean
| word [ basicExpr ]
| unsigned word [ basicExpr ]
| signed word [ basicExpr ]
| { enumerationTypeBody }
| basicExpr .. basicExpr
| array basicExpr .. basicExpr
  | of simpleTypeSpecifier

enumerationTypeBody ::

enumerationTypeValue
| enumerationTypeBody , enumerationTypeValue

enumerationTypeValue ::

symbolicConstant
| integerNumber

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 2.3.11 [MODULE Instantiations], page 31. The simple type specifier comprises boolean type, integer type, enumeration types, unsigned word[ ], signed 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\}. FALSE and TRUE values cannot be used as enumeration type specifiers. 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, 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 expressions separated by .. (<TWO DOTS>). The two expressions have both to evaluate to constants integer numbers, and may contain names of defines and module formal parameters. For example, \(-1 - P1 .. 5 + D1\), where \(P1\) refers to a module formal parameter, and \(D1\) refers to a define. Both \(P1\) and \(D1\) have to be statically evaluable to integer constants.

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 evaluated number on the left from the two dots must be less than or equal to the evaluated number on the right.

The unsigned word\[\] type is specified by the keywords unsigned word (where unsigned may be skipped) with a basic_expr supplied in square brackets. The expression must be statically evaluable to a constant integer number whose value must be greater than zero. The signed word\[\] type is specified in a similar way with the keywords signed word. 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, a basic_expr specifying the lower bound of the array index, two dots .. a basic_expr specifying the upper bound of the array index, the keyword of, and the type of array’s elements. The elements can themselves be arrays. The two bound expressions have to be statically evaluable to constant integer numbers, and may contain names of defines and module formal parameters.

State Variables

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

\[
\text{var\_declaration :: \text{VAR}\ var\_list}
\]

\[
\text{var\_list :: identifier : type\_specifier ;}
| \text{var\_list identifier : type\_specifier ;}
\]

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 an enumeration type may take only the values enumerated in the type specifier of the declaration.

Input Variables

IVARs (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:

\[
\text{ivar\_declaration :: \text{IVAR}\ simple\_var\_list}
\]

\[
\text{simple\_var\_list ::}
| \text{identifier : simple\_type\_specifier ;}
| \text{simple\_var\_list identifier : simple\_type\_specifier ;}
\]

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:

\[
\text{IVAR i : boolean;}
\]

\[
\text{ASSIGN}
\]

\[
\text{init(i) := TRUE;}
\]

\[
\text{next(i) := FALSE;}
\]
• 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 82 for further information.

Frozen Variables

FROZENVAR s (frozen variables) are variables that retain their initial value throughout the evolution of the state machine; this initial value can be constrained in the same ways as for normal state variables. Similar to input variables the difference between the syntax for the frozen and state variables declarations is the keyword indicating the beginning of a declaration:

frozenvar_declaration :: FROZENVAR simple_var_list

The semantics of some frozen variable \( a \) is that of a state variable accompanied by an assignment that keeps its value constant (it is handled more efficiently, though):

\[
\text{ASSIGN next}(a) := a;
\]

As a consequence, frozen variables may not have their current and next value set in an ASSIGN statement, i.e. statements such as \( \text{ASSIGN next}(a) := \text{expr} \); and \( \text{ASSIGN a} := \text{expr} \); are illegal. Apart from that frozen variables may occur in the definition of the FSM in any place in which a state variable may occur. Some examples are as follows:

• Left-side current and next state assignments are illegal, while init state assignments are allowed:
  FROZENVAR a : boolean;
  FROZENVAR b : boolean;
  FROZENVAR c : boolean;
  VAR d : boolean;
  FROZENVAR e : boolean;
  ASSIGN
  init(a) := d; -- legal
  next(b) := d; -- illegal
  c := d; -- illegal
  e := a; -- also illegal

• INIT, TRANS, INVAR, FAIRNESS, JUSTICE, and COMPASSION statements are all legal. So is the scope of a next expression. For example:

  -- the following has an empty state space
  FROZENVAR a : boolean;
INIT a
INVAR !a

-- alternatively, this has two initial states, deadlocking
FROZENVAR b : boolean;
TRANS next(b) <-> !b

-- and that’s just unfair
FROZENVAR c : boolean;
FAIRNESS c
FAIRNESS !c

• All kinds of specifications involving frozen variables are allowed, e.g.:
  FROZENVAR c : boolean;
  -- True by definition.
  SPEC AG ((c -> AG c) & (!(c) -> AG !c))
  -- Here, neither is true.
  INVARSPEC c
  INVARSPEC !c
  -- False (as above).
  LTLSPEC (G F c) & (G F !c)

Examples

Below are examples of state, frozen, and input variable declarations:

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

The variable a is a state variable, b is a frozen variable, and c is an input variable; 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 : unsigned word[3];
VAR h : word[3];
VAR i : signed word[4];

The variables g and h are of 3-bits-wide unsigned word type (i.e. unsigned word[3]), and i is of 4-bits-wide signed word type (i.e. signed word[4]).

VAR j : array -1..1 of boolean;

The variable j 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 statement where the `identifier` is declared (see Section 2.3.16 [Context], page 36 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 Array Define Declarations

It is possible to specify an array expressions. This feature is experimental and currently available only through `DEFINE` declaration. The syntax for this kind of declaration is:

```
array_define_declaration :: DEFINE identifier := array_expression ;
array_expression :: [ array_contents ]
| [ array_expression_list ]
array_expression_list :: array_expression
| array_expression , array_expression_list
array_contents :: next_expr , array_contents
| next_expr
```

Array `DEFINE` associates an `identifier` on the left hand side of the `':='` with an array expression. As a normal `DEFINE` statement an array define is considered as a macro. Whenever an array `identifier` occurs in an expression, the `identifier` is syntactically replaced by the array expression it is associated with. As with normal `DEFINE` an array `DEFINE` expression is always evaluated in the context of the statement where the `identifier` is declared and forward references to defined symbols are allowed but circular definitions are not.

The type of an array expression `[exp1, exp2, ..., expN]` is `array 0..N-1 of type` where `type` is the least type such that all `exp1, exp2, ..., expN` can be converted to it.

It is not possible to declare asymmetrical arrays. This means that it is forbidden to declare an array with a different number of elements in a dimension. For example, the following code will result in an error:

```
DEFINE x := [[1,2,3], [1,2]];
```

2.3.4 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 especially useful
in those conditions that require symbolic constants to occur only in \texttt{DEFINE}s body (e.g. in generated models). For an example of usage see also the command \texttt{write boolean model}. A constant is allowed to be declared multiple times, as after the first declaration any further declaration will be ignored. \texttt{CONSTANTS} declarations are an extension of the original SMV grammar, and they are supported since NuSMV 2.6. The syntax for this kind of declaration is:

\begin{verbatim}
constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier |
               constants_body , identifier
\end{verbatim}

2.3.5 INVAR Constraint

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

\begin{verbatim}
invar_constraint :: INVAR simple_expr [;]
\end{verbatim}

Since the expression in the \texttt{INVAR} constraint is a \texttt{simple_expression}, it cannot contain the \texttt{next()} operator. If there is more than one \texttt{INVAR} constraint, the invariant set is the conjunction of all of the \texttt{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 \texttt{boolean} expression, introduced by the \texttt{TRANS} keyword. The syntax of a \texttt{TRANS} constraint is:

\begin{verbatim}
trans_constraint :: TRANS next_expr [;]
\end{verbatim}

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

2.3.8 ASSIGN Constraint

An assignment has the form:

\begin{verbatim}
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
\end{verbatim}
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:

\[
\text{ASSIGN } a := \text{exp}; \quad \text{is equivalent to } \text{INVAR } a \text{ in exp;}
\]
\[
\text{ASSIGN init(a) := } \text{exp; is equivalent to } \text{INIT a in exp;}
\]
\[
\text{ASSIGN next(a) := } \text{exp; is equivalent to } \text{TRANS next(a) in exp;}
\]

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:

\[
\begin{align*}
\text{:=} & \quad : \text{integer } \times \text{integer} \\
& \quad : \text{integer } \times \text{integer set} \\
& \quad : \text{symbolic enum } \times \text{symbolic enum} \\
& \quad : \text{symbolic enum } \times \text{symbolic set} \\
& \quad : \text{integers-and-symbolic enum } \times \text{integers-and-symbolic enum} \\
& \quad : \text{integers-and-symbolic enum } \times \text{integers-and-symbolic set} \\
& \quad : \text{unsigned word[N]} \times \text{unsigned word[N]} \\
& \quad : \text{signed word[N]} \times \text{signed word[N]}
\end{align*}
\]

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:
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:

$\begin{align*}
  x &:= y; \\
  y &:= x;
\end{align*}$

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 \texttt{next()} operator.

$\begin{align*}
  x &:= y; \\
  \text{next}(y) &:= x;
\end{align*}$

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,

$\begin{align*}
  \text{next}(x) &:= \text{case } x = 1 : 1; \\
  &\quad \text{TRUE} : \{0,1\}; \\
  &\quad \text{esac};
\end{align*}$

### 2.3.9 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 $\phi$, which is assumed to be true infinitely often in all the fair paths. In NuSMV, justice constraints are identified by keywords \texttt{JUSTICE} and, for backward compatibility, \texttt{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 \texttt{COMPASSION}. 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 ::
  \texttt{FAIRNESS} \ simple_expr \ [;]
  | \texttt{JUSTICE} \ simple_expr \ [;]
  | \texttt{COMPASSION} \ (\ simple_expr , \ simple_expr \ ) \ [;]
```

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

\textsuperscript{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.

---

**Example 1**

| \texttt{x := expr\_1;} |
| \texttt{x := expr\_2;} |

| \texttt{Example 2} |
| \texttt{init(x) := expr\_1;} |
| \texttt{init(x) := expr\_2;} |

| \texttt{Example 3} |
| \texttt{x := expr\_1;} |
| \texttt{init(x) := expr\_2;} |

| \texttt{Example 4} |
| \texttt{x := expr\_1;} |
| \texttt{next(x) := expr\_2;} |
2.3.10 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:

\[
\text{module} ::= \text{MODULE} \text{identifier} \left[\left(\text{module}_\text{parameters}\right)\right] \left[\text{module}_\text{body}\right]
\]

\[
\text{module}_\text{parameters} ::= \text{identifier} | \text{module}_\text{parameters}, \text{identifier}
\]

\[
\text{module}_\text{body} ::= \text{module}_\text{element} | \text{module}_\text{body} \text{module}_\text{element}
\]

\[
\text{module}_\text{element} ::= \text{var}_\text{declaration} | \text{ivar}_\text{declaration} | \text{frozenvar}_\text{declaration} | \text{define}_\text{declaration} | \text{constants}_\text{declaration} | \text{assign}_\text{constraint} | \text{trans}_\text{constraint} | \text{init}_\text{constraint} | \text{invar}_\text{constraint} | \text{fairness}_\text{constraint} | \text{ctl}_\text{specification} | \text{invar}_\text{specification} | \text{ltl}_\text{specification} | \text{compute}_\text{specification} | \text{isa}_\text{declaration}
\]

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

2.3.11 MODULE Instantiations

An instance of a module is created using the VAR declaration (see Section 2.3.1 [State Variables], page 24) with a module type specifier (see Section 2.3.1 [Type Specifiers], page 23). The syntax of a module type specifier is:

\[
\text{module}_\text{type}_\text{specifier} ::= \text{identifier} \left[\left(\text{parameter}_\text{list}\right)\right] | \text{process} \text{identifier} \left[\left(\text{parameter}_\text{list}\right)\right]
\]

\[
\text{parameter}_\text{list} ::= \text{next}_\text{expr} | \text{parameter}_\text{list}, \text{next}_\text{expr}
\]

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 next expression (see Section 2.2.4 [Simple and Next Expressions], page 21). 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.

Here are examples:

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

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

Now consider the following program:

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

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

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

The keyword process is explained in Section 2.3.13 [Processes], page 34.

2.3.12 References to Module Components (Variables and Defines)

As described in Section 2.2.3 [Variables and Defines], page 13, 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:

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

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.
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 \( \cdot \) are used to access the components of modules. For example, if \( m \) is an instance of a module (see Section 2.3.11 [MODULE Instantiations], page 31 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 given in square brackets. See 2.2.3 for more information.

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`.
2.3.13 Processes

Important!
Since NuSMV version 2.5.0 processes are deprecated. In future versions of NuSMV processes may be no longer supported, and only synchronous FSM will be supported by the input language. Modeling of asynchronous processes will have to be resolved at higher level.

Processes are used to model interleaving concurrency. A process is a module which is instantiated using the keyword `process' (see Section [2.3.11] [MODULE Instantiations], page [11]). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Note that only assignments of the form

\[
\text{ASSIGN next(var\_name) := ... ;}
\]

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 TRUE if and only if the process instance is currently selected for execution. No two processes may be running at the same time.

Note that (only) 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 (when processes are used), 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:

\[
\text{MODULE my\_module}
\]
\[
\text{MODULE main}
\]

2.3.14 A Program and the main Module

The syntax of a NuSMV program is:

\[
\text{program ::= module\_list}
\]
\[
\text{module\_list ::=}
\]
\[
\text{module | module\_list module}
\]

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

2.3.15 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 2.3.11 [MODULE Instantiations], page 31). 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.
2.3.16 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:

```plaintext
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), 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).

2.3.17 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:

```plaintext
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) \[\text{PSL}\] 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:

```plaintext
CTLSPEC
```

The module `main` is instantiated with the so called empty identifier which cannot be referenced in a program.
ctl_specification ::= CTLSPEC ctl_expr [;]
| SPEC ctl_expr [;]
| CTLSPEC NAME name := ctl_expr [;]
| SPEC NAME name := 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 xnor ctl_expr -- logical NOT 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 \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \) such that \( p \) is true in \( s_n \).
- **AF** \( p \) is true in a state \( s_0 \) if for all series of transitions \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \), \( p \) is true in \( s_n \).
- **EG** \( p \) is true in a state \( s_0 \) if there exists an infinite series of transitions \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \) 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 \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \), \( p \) is true in every \( s_i \).
- **E[p U q]** is true in a state \( s_0 \) if there exists a series of transitions \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \) 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 \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \), \( p \) 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 next_expr ;
                   INVARSPEC NAME name := next_expr [;]
```

This statement is intuitively equivalent to

```
SPEC AG simple_expr ;
```

but can be checked by a specialised algorithm during reachability analysis and Invariant Specifications can contain next operators. 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 [;]
                   LTLSPEC NAME name := ltl_expr [;]
```

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

```
ltl_expr ::
    next_expr -- a next boolean expression
    | ( ltl_expr ) -- logical not
    | ! 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 xnor ltl_expr -- logical NOT exclusive or
    | ltl_expr -> ltl_expr -- logical implies
    | ltl_expr <-> ltl_expr -- logical equivalence
    -- FUTURE
    | X ltl_expr -- next state
    | G ltl_expr -- globally
    | G bound ltl_expr -- bounded globally
    | F ltl_expr -- finally
    | F bound ltl_expr -- bounded 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
    | H bound ltl_expr -- bounded historically
    | O ltl_expr -- once
    | O bound ltl_expr -- bounded once
    | ltl_expr S ltl_expr -- since
    | ltl_expr T ltl_expr -- triggered

bound :: [ integer_number , integer_number ]
```

Intuitively the semantics of LTL operators is as follows:
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 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 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 temporal operators.

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.

**Important Difference Between BDD and SAT Based LTL Model Checking**

If a FSM to be checked it not total (i.e. has deadlock state) the model checking may return different results for the same LTL specification depending on the verification engine used. For example, for below model:

```plaintext
MODULE main
VAR s : boolean;
TRANS s = TRUE
LTLSPEC G (s = TRUE)
```

the LTL specification is proved valid by BDD-based model checking but is violated by SAT-based bounded model checking. The counter-example found consists of one state \( s = \text{FALSE} \).
This difference between the results is caused by the fact that BDD model checking investigates only *infinite* paths whereas SAT-based model checking is able to deal also with *finite* paths. Apparently infinite paths cannot ever have \(s=\text{FALSE}\) as then the transition relation will not hold between the consecutive states in the path. A *finite* path consisting of just one state \(s=\text{FALSE}\) violates the specification \(G\ (s = \text{TRUE})\) and is still consistent with the FSM as the transition relation is not taken ever and there is not initial condition to violate. Note however that this state is a deadlock and cannot have consecutive states.

In order to make SAT-based bound model checking ignore finite paths it is enough to add a fairness condition to the main module:

```plaintext
JUSTICE TRUE;
```

Being limited to fair paths, SAT-based bounded model checking cannot find a finite counter-example and results of model checking become consistent with BDD-based model checking.

### 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:

```plaintext
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 ]
```

Given ranges must be non-negative.

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. (keyword ‘SPEC’ is deprecated)

```plaintext
rtctl_specification :: CTLSPEC rtctl_expr [;]
    | SPEC rtctl_expr [;]
    | CTLSPEC NAME name := rtctl_expr [;]
    | SPEC NAME name := 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 []
                    COMPUTE NAME name := 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 []
                    PSLSPEC NAME name := 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
    | word ;; a word constant
    | var_id ;; a variable identifier
    | { psl_expr , ... , psl_expr }
    | { psl_expr "{" psl_expr , ... , "psl_expr" }" }
    | ( psl_expr )
```

```
psl_unary_expr ::
    + psl_primary_expr
```

41
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:

```plaintext
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
```

42
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    ;; infinite 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 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 )

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 )

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

Please note that instances of FL_property cannot be combined with the “=”, “!=” and “==”. Sequences, i.e. instances 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 ]
Instances of \textit{OBE\_property} are CTL properties in the PSL style and are defined as follows:

\begin{verbatim}
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 ]
\end{verbatim}

The \textsc{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.

\section*{2.5 Variable Order Input}

It is possible to specify the order in which variables should appear in the BDD’s generated by \textsc{NuSmv}. The file which gives the desired order can be read in using the \texttt{-i} option in batch mode or by setting the \texttt{input\_order\_file} environment variable in interactive mode.\footnote{Note that if the ordering is not provided by a user then \textsc{NuSmv} decides by itself how to order the variables. Two shell variables \texttt{bdd\_static\_order\_heuristics} (see page 55) and \texttt{vars\_order\_type} (see page 53) allow to control the ordering creation.}

\subsection*{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:

\begin{verbatim}
vars\_list ::= EMPTY
| var\_list\_item vars\_list

var\_list\_item ::= complex\_identifier
| complex\_identifier . integer\_number
\end{verbatim}

Where \texttt{EMPTY} means parsing nothing.

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

- \texttt{Complete\_Var\_Name} \textit{--} to specify an ordinary variable
- \texttt{Complete\_Var\_Name[index]} \textit{--} to specify an array variable element
- \texttt{Complete\_Var\_Name.NUMBER} \textit{--} to specify a specific bit of a
  \textit{--} scalar variable

where \texttt{Complete\_Var\_Name} is just the name of the variable if it appears in the module \texttt{MAIN}, otherwise it has the module name(s) prepended to the start, for example:

\begin{verbatim}
  mod1.mod2...modN.varname
\end{verbatim}
where \textit{varname} is a variable in \texttt{modN}, and \texttt{modN.varname} is a variable in \texttt{modN-1}, and so on. Note that the module name \texttt{main} is implicitly 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 \texttt{NuSMV} displays a warning message stating these actions.

Comments can be included by using the same syntax as regular \texttt{NuSMV} files. That is, by starting the line with \texttt{--} or by entering text between limiters \texttt{/**} and \texttt{*/}.

\subsection*{2.5.2 Scalar Variables}

A variable, which has a finite range of values that it can take, is encoded as a set of \texttt{boolean} variables (i.e. bits). 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 \texttt{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 \texttt{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 the location of specific bit variables anywhere in the ordering. This is achieved by first specifying the scalar variable name in the desired location, then simply specifying \texttt{Complete.Var_Name.i} at the position where you want that bit variable to appear:

...  
\texttt{Complete.Var_Name}  
...  
\texttt{Complete.Var_Name.i}  
...  

The result of doing this is that the variable representing the \textit{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 \texttt{varname.0}, ..., \texttt{varname.i-1}, \texttt{varname.i+1}, ..., \texttt{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:

...  
\texttt{Complete.Var_Name.i}  
...  
\texttt{Complete.Var_Name}  
...  

The variable representing the \textit{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:
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 algorithm that the user previously 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 53 for further information).

Grammar of the clusters ordering file is the same of variable ordering file presented in section 2.5 at page 45.
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>
```

When running interactively, NuSMV first tries to read and execute commands from an initialization file if such file can be found and is readable unless -s is passed on the command line.

First, file master.nusmvrc is looked for in directory defined in environment variable NUSMV_LIBRARY_PATH or in default library path if no such variable is defined. If no such file exists, file .nusmvrc is looked for in user’s home directory and as a last attempt, .nusmvrc is looked for in current directory. Commands in the initialization file (if any) are executed consecutively. When initialization phase is completed the NuSMV shell is displayed and the system is now ready to execute user commands.

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 -source:

```
system_prompt> NuSMV -source cmd_file <RET>
```

- `source 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 -source implies -int.
In the following we present the possible commands followed by the related environment variables, classified in different categories. Every command answers to the option \texttt{-h} by printing out the command usage. When output is paged for some commands (option \texttt{-m}), it is piped through the program specified by the UNIX \texttt{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 3.11 depicted at page 113. 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 \texttt{history} command, on page 106 for more information.

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

<table>
<thead>
<tr>
<th>\texttt{verbose.level}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controls the verbosity of the system. Possible values are integers from 0 (no messages) to 4 (full messages). The default value is 0.</td>
<td></td>
</tr>
</tbody>
</table>

### 3.1 Model Reading and Building

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

<table>
<thead>
<tr>
<th>\texttt{read.model}</th>
<th>Reads a NuSMV file into NuSMV.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{read.model [\texttt{-h}] [\texttt{-i model-file}]}</td>
<td></td>
</tr>
</tbody>
</table>

Reads a NuSMV 2.6 file. If the \texttt{-i} option is not specified, it reads from the file specified in the environment variable \texttt{input.file}.

Command Options:

\texttt{-i model-file} \hspace{1cm} Sets the environment variable \texttt{input.file} to \texttt{model-file}, and reads the model from the specified file.

<table>
<thead>
<tr>
<th>\texttt{input.file}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stores the name of the input file containing the model. It can be set by the &quot;set&quot; command or by the command line option \texttt{-i}. There is no default value.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{pp.list}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 separated list of pre-processor names inside double quotes. Any invalid names are ignored. The default is none.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{flatten.hierarchy}</th>
<th>Flattens the hierarchy of modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{flatten.hierarchy [\texttt{-h}] [\texttt{-d}]}</td>
<td></td>
</tr>
</tbody>
</table>
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.

Command Options:
- \texttt{\textbf{-d}} Delays the construction of vars constraints until needed

<table>
<thead>
<tr>
<th>disable_syntactic_checks</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables or disables the syntactic checks that are performed by the \texttt{flatten_hierarchy} command. Warning: If the model is not well-formed, NuSMV may result in unpredictable results, use this option at your own risk.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>keep_single_value_vars</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables or disables the conversion of variables that can assume only one single possible value into constant \texttt{DEFINEs}.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>backward_compatibility</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is used to enable or disable type checking and other features provided by NuSMV 2.6. 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 and promotion of boolean constants to their integer counterpart. 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. Since NuSMV 2.5.1, backward compatibility mode introduces a porting feature from old models which use constant 1 as \texttt{case} conditions, instead of forcing the use of \texttt{TRUE}. The option by default it set to 0.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type_checking_warning_on</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>show_vars - Shows model’s symbolic variables and defines with their types</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prints a summary of the variables and defines declared in the input file. Moreover, it prints also the list of symbolic input, frozen and state variables of the model with their range of values (as defined in the input file) if the proper command option is specified. By default, if no type specifiers (-s, -f, -i) are used, all variable types will be printed. When using one or more type specifiers (e.g. -s), only variables belonging to selected types will be printed. Command Options: -s Prints only state variables.</td>
<td></td>
</tr>
</tbody>
</table>
-f Prints only frozen variables.
-i Prints only input variables.
-t Prints only the number of variables (among selected kinds), grouped by type. This option is incompatible with -V or -D
-V Prints only the list of variables with their types (among selected kinds), with no summary information. This option is incompatible with -t or -D
-D Prints only the list of defines with their types, with no summary information. This option is incompatible with -t or -V
-v Prints verbosely. Scalar variable's values are not truncated if too long for printing.
-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.

show_dependencies - Shows the dependencies for the given expression

show_dependencies [-h] [-k bound] -e expression
Prints the set of variables that are in the dependency set of the given expression. If the bound is specified using the -k argument, then the computation of the dependencies is done until the bound has been reached. If not specified, the computation is performed until no new dependencies are found.

Command Options:
-h Shows the command usage
-k bound Sets the bound limit for the dependencies computation
-e expr The expression on which the dependencies are computed

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

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. A value for this variable can also be provided with command line option -i. 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 1.

**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. A value for this variable can also be provided with command line option -o. 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 by a user and not computed statically by heuristics (see variables input_order_file on page 52 and bdd_static_order_heuristics on page 53).
The individual bits of variables may or may not be interleaved. When bits interleaving is not used then bits belonging to one variable are grouped together in the ordering. Otherwise, the bits interleaving is applied and all higher bits of all variables are ordered before all the lower bits, i.e. N-th bits of all variables go before (N-1)th bits. The exception is boolean variables which are ordered before variables of any other type though boolean variables consist of only 0-th bit.

The value of vars_order_type may be:

- **inputs_before**. Input variables are forced to be ordered before state and frozen variables (default). No bits interleaving is done.
- **inputs_after**. Input variables are forced to be ordered after state and frozen variables. No bits interleaving is done.
- **topological**. Input, state and frozen variables are ordered as they are declared in the input smv file. No bits interleaving is done.
- **inputs_before_bi**. Bits are interleaved and in every group of N-th bits input variables are forced to be ordered before state and frozen variables. This is the default value.
- **inputs_after_bi**. Bits are interleaved and in every group of N-th bits input variables are forced to be ordered after state and frozen variables.
- **topological_bi**. Bits are interleaved and in every group of N-th bits input, state and frozen variables are ordered as they are declared in the input smv file.
- **lexicographic**. This is deprecated value. topological has to be used instead.

### bdd_static_order_heuristics

<table>
<thead>
<tr>
<th>Environment Variable</th>
</tr>
</thead>
</table>
| When a variable ordering is not specified (see variable input_order_file on page 52)
NuSMV can try to guess a good ordering by analyzing the input model.
Possible values are:

- **none** No heuristics are applied.
- **basic** This heuristics creates some initial ordering and then moves scalar and word variables in this ordering to form groups. Groups go one after another and every group contains variables which interact with each other in the model. For example, having variables $a,b,c,d,e,f$ and a single model constraint

$$\text{TRANS next}(a) = b + 1 \rightarrow (\text{next}(c) = d \lor e \land \text{next}(f) = a)$$

will results in 2 groups of variables $\{a,b,f\}$ and $\{c,d,e\}$.

Shell variable vars_order_type (page 52) provides additional control over the heuristics. In particular, it allows to put input/state variables in the initial ordering at the begin, the end or in topological order. Moreover, if the value of this variable is ending in _bi then in very individual group the bits of variables are additionally interleaved.

Note that variable groups created by the heuristics has nothing to do with BDD package groups which disallow dynamic reordering of variables in one group. After the heuristics is applied the dynamic reordering may move any bit of any variable at any position.

### 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:
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td></td>
<td>No partitioning at all.</td>
</tr>
<tr>
<td>Threshold</td>
<td></td>
<td>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 <code>conj_part_threshold</code>. It is possible (default) to use affinity clustering to improve model checking performance. See <code>affinity</code> variable.</td>
</tr>
<tr>
<td>Iwls95CP</td>
<td></td>
<td>Conjunctive partitioning, with clusters generated and ordered according to the heuristic described in [RAP+95]. Works in conjunction with the variables <code>image_cluster_size</code>, <code>image_W1</code>, <code>image_W2</code>, <code>image_W3</code>, <code>image_W4</code>. It is possible (default) to use affinity clustering to improve model checking performance. See <code>affinity</code> variable. It is also possible to avoid (default) preordering of clusters (see [RAP+95]) by setting the <code>iwls95preorder</code> variable appropriately.</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>conj_part_threshold</th>
<th>Environment Variable</th>
<th>The limit of the size of clusters in conjunctive partitioning. The default value is 0 BDD nodes.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>affinity</th>
<th>Environment Variable</th>
<th>Enables affinity clustering heuristic described in [MHS00], possible values are 0 or 1. The default value is 1.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>trans_order_file</th>
<th>Environment Variable</th>
<th>Reads the a variables list from file <code>tv_file</code>, 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].</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>image_cluster_size</th>
<th>Environment Variable</th>
<th>One of the parameters to configure the behaviour of the Iwls95CP partitioning algorithm. <code>image_cluster_size</code> is used as threshold value for the clusters. The default value is 1000 BDD nodes.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>image_W[1,2,3,4]</th>
<th>Environment Variable</th>
<th>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].)</th>
</tr>
</thead>
</table>
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.*

`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.

`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.

get_internal_status - Prints out the internal status of the system.

`get_internal_status [-h]`

Prints out the internal status of the system. i.e.
- `-1: read_model` has not yet been executed or an error occurred during its execution.
- `0: flatten_hierarchy` has not yet been executed or an error occurred during its execution.
- `1: encode_variables` has not yet been executed or an error occurred during its execution.
- `2: build_model` has not yet been executed or an error occurred during its execution.

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

`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 2.6 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 option 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.

---

**build flat model** - Compiles the flattened hierarchy into a Scalar FSM

Command:

```
build flat model [-h]
```

Compiles the flattened hierarchy into SEXP (initial states, invariants, and transition relation).

---

**build boolean model** - Compiles the flattened hierarchy into boolean Scalar FSM

Command:

```
build boolean model [-h] [-f]
```

Compiles the flattened hierarchy into boolean SEXP (initial states, invariants, and transition relation).

Command Options:
- **-f** Forces the boolean model construction.

---

**write flat model** - Writes a flat model to a file

Command:

```
write flat model [-h] [-A] [-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`
- **-A** Writes the flat SMV model using a renaming map to “anonimize” the model. All the symbols except numerical constant will be renamed.

---

**output_flatten_model_file** Environment Variable

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

---

**daggifier_enabled** Environment Variable

56
Determines whether the expression daggifier in the model dumping features is enabled or not. The default is enabled.

<table>
<thead>
<tr>
<th><strong>daggifier_depth_threshold</strong></th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the minimum threshold for expressions depth to be daggified.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>daggifier_counter_threshold</strong></th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the minimum threshold for expressions count to be daggified. (i.e. expression must show at least Number time to be daggified</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>daggifier_statistics</strong></th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prints daggifier statistics after model dumping.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>write_boolean_model</strong></th>
<th>- Writes a flat and boolean model to a file</th>
</tr>
</thead>
<tbody>
<tr>
<td>-write_boolean_model [-h] [-o filename]</td>
<td></td>
</tr>
<tr>
<td>Writes the currently loaded NuSMV 2.6 model in the specified file, after having flattened and booleanized it. Processes are eliminated and a corresponding equivalent model is printed out.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Command Options:</td>
<td></td>
</tr>
<tr>
<td>-o filename</td>
<td>Attempts to write the flat and boolean NuSMV 2.6 model in filename</td>
</tr>
</tbody>
</table>

In NuSMV 2.6 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 2.6 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.

<table>
<thead>
<tr>
<th><strong>output_boolean_model_file</strong></th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>The file where the flattened and booleanized model has to be written. The default value is ‘stdout’.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>output_word_format</strong></th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>This variable sets in which base <strong>unsigned word[●]</strong> and <strong>signed word[●]</strong> 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 <strong>unsigned word[●]</strong> and <strong>signed word[●]</strong> constants remain in same format as they were written in the input file.</td>
<td></td>
</tr>
</tbody>
</table>
dump fsm - Dumps (in DOT format) selected parts of the bdd fsm, with optional expression

```bash
```

Dumps selected parts of the bdd fsm, with optional expression, in DOT format. At least one among options [iIte] must be specified.

**Command Options:**

- `-o fname` Dumps to the specified file name.
- `-i` Dumps the initial states of the FSM, among with other selected outputs.
- `-I` Dumps the invariant states of the FSM, among with other selected outputs.
- `-t` Dumps the (monolithic) transition relation of the FSM, among with other selected outputs.
- `-f` Dumps the (monolithic) fair states of the FSM, among with other selected outputs.
- `-r` Dumps the (monolithic) reachable states of the FSM, among with other selected outputs.
- `-e expr` Dumps the specified expression, among with other selected outputs (see also command dump expr).

### 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**

```bash
compute reachable [-h] [-k number] [-t seconds]
```

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 the execution of this command can take much time because the computation of reachable states may be very expensive. Use the `-k` option to limit the number of forward step to perform. If the reachable states has been already computed the command returns immediately since there is nothing more to compute.

**Command Options:**

- `-k number` If specified, limits the computation of reachable states to perform number steps forward starting from the last computed frontier. This means that you can expand the computed reachable states incrementally using this option.
- `-t seconds` If specified, forces the computation of reachable states to end after "seconds" seconds. This limit could not be precise since the if the computation of a step is running when the limit occurs, the computation is not interrupted until the end of the step.
**print_reachable_states** - Prints out the number of reachable states

```
print_reachable_states [-h] [-v] [-d] [-f] [-o filename]
```

Prints the number of reachable states of the given model. In verbose mode, prints also the list of all reachable states, if they are less than \(2^{16}\). The reachable states are computed if needed.

Command Options:

- `-v` Prints the list of reachable states
- `-d` Prints the list of reachable states with defines (Requires `-v`)
- `-f` Prints the formula representing the reachable states
- `-o filename` Prints the result on the specified `filename` instead of on standard output

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

```
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.

```
print_fsm_stats [-h] | [-m] | [-p] | [-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.

Also the command can print all the normalized predicates the FMS consists of. A normalized predicate is a boolean expression which does not have other boolean sub-expressions. For example, expression \((b<0 ? a/b : 0) = c\) is normalized into \((b<0 ? a/b=c : 0=c)\) which has 3 normalized predicates inside: \(b<0, a/b=c, 0=c\).
Command Options:

- **-h**
  Prints the command usage.

- **-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”.

- **-p**
  Prints out the normalized predicates the FSM consists of. Expressions in properties are ignored.

- **-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, if they are less than $2^{16}$.

---

**print_fair_transitions** - *Prints out the number of fair transitions, and optionally list them*

**Command**

`print_fair_transitions [-h] [-v [-f format] [-o out_fname]]`

Prints the number of fair transitions of the given model. In verbose mode, prints also the list of all fair transitions, with a limit of $2^{16}$. The transitions are displayed as state-input-next triples, in three possible formats: smv (default), dot and csv. Also, each transition is tagged with a current state ID and next state ID.

---

**check_ctlspec** - *Performs fair CTL model checking.*

**Command**

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

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 is used, all the SPEC formulas in the database are checked. See variable `use_coil_size_sorting` for changing properties verification order.

**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.
-P name Checks the CTL property named name 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.

<table>
<thead>
<tr>
<th>ag-only_search</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>forward_search</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ltl-tableau-forward_search</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forces the computation of the set of reachable states for the tableau resulting from BDD-based LTL model checking, performed by command check-ltl-spec. 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-ltl-spec, 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).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oreg-justice-emptiness-bdd-algorithm</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>The algorithm used to determine language emptiness of a Büchi fair transition system. The algorithm may be used from the following commands: check-ltl-spec, check-psl-spec. Possible values are:</td>
<td></td>
</tr>
<tr>
<td>- EL_bwd The default value. The Emerson-Lei algorithm [EL86] in its usual backwards direction, i.e., using backward image computations.</td>
<td></td>
</tr>
<tr>
<td>- EL_fwd A variant of the Emerson-Lei algorithm that uses only forward image computations (see, e.g., [HKQ03]). This variant requires the variables forward_search, ltl-tableau-forward_search, use reachable states to be set. Furthermore, counterexample computation is not yet implemented, i.e., counter_examples should not be set. When invoking one of the commands mentioned above, all required settings are performed automatically if not already found as needed, and are restored after execution of the command.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check_invar - Performs model checking of invariants</th>
<th>Command</th>
</tr>
</thead>
</table>
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.

A search strategy can be specified with \(-s\) option. This is useful to speed up the check in some situations. If “forward-backward” or “bdd-bmc” strategy is specified then it is possible to choose a search heuristic with \(-e\) option; “bdd-bmc” strategy has some other options explained below.

See variable use\_col\_size\_sorting for changing properties verification order.
Command Options:

- **-m**
  Pipes the output generated by the program in processing INVARSPECs 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 INVARSPECs to the file output-file.

- **-n number**
  Checks the INVAR property with index number in the property database.

- **-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.

- **-P name**
  Checks the INVAR property named name in the property database.

- **-s strategy**
  Chooses the strategy to use while performing reachability analysis. Possible strategies are:

  - “forward” Explore the search space from initial states and try to reach bad states.
  - “backward” Explore the search space from bad states and try to reach initial states.
  - “forward-backward” Explore the search space using a heuristic to decide at each step whether to move from bad states or from reachable states.
  - “bdd-bmc” Explore the search space using BDD with “forward-backward” strategy and use a heuristic (specified with -j option) to decide if to switch from BDD technology to BMC. The idea is to expand the sets of states reachable from both bad and initial states, eventually stop and search for a path between frontiers using BMC technology. Options -j, -t and -k are enabled only when using this strategy. Note that the algorithm used for the BMC approach is the one specified in the variable “bmc_invar_alg”.

  If this option is not specified, the value of the environment variable “check_invar_strategy” is considered.

- **-e f-b-heuristic**
  Specify the heuristic that decides at each step if we must expand reachable states or bad states. This option is enabled only when using “forward-backward” or “bdd-bmc” strategies. Possible values are “zigzag” and “smallest”. “zigzag” forces to perform a step forward and the next step backward and so on, while “smallest” performs a step from the frontier with the BDD representing the state is smaller. If this option is not specified, the value of the environment variable “check_invar_forward_backward_heuristic” is considered.
When using “bdd-bmc” strategy specify the heuristic that decides at which step we must switch from BDD to BMC technology. You should use the option `-t` to specify the threshold for the chosen heuristic. Possible heuristics are “steps” and “size”. “steps” forces to switch after a number of steps equal to the threshold, while “size” switch when BDDs are bigger (in the number of nodes) than the threshold. If this option is not specified, the value of the environment variable “check_invar_bdbmc_heuristic” is considered.

When using “bdd-bmc” strategy specify the threshold for the chosen heuristic. If this option is not specified, the value of the environment variable “check_invar_bdbmc_threshold” is considered.

When using “bdd-bmc” strategy specify the maximum length of the path to search for during BMC search. If this option is not specified, the value of the environment variable “bmc_length” is considered.

---

**check_invar_strategy**

Determines default search strategy to be used when using command “check_invar”. See the documentation of “check_invar” for a detailed description of possible values and intended semantics.

**check_invar_forward_backward_heuristic**

Determines default forward-backward heuristic to be used when using command “check_invar”. See the documentation of “check_invar” for a detailed description of possible values and intended semantics.

**check_invar_bdd_bmc_heuristic**

Determines default bdd-bmc heuristic to be used when using command “check_invar”. See the documentation of “check_invar” for a detailed description of possible values and intended semantics.

**check_invar_bdd_bmc_threshold**

Determines default bdd-bmc threshold to be used when using command “check_invar”. See the documentation of “check_invar” for a detailed description of possible values and intended semantics.

**check_ltlspec**

Performs LTL model checking

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

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.

See variable `use_col_size_sorting` for changing properties verification order.
Command Options:

- \texttt{-m} \hfill Pipes the output generated by the command in processing \texttt{LTLSPECs} to the program specified by the \texttt{PAGER} shell variable if defined, else through the UNIX command “more”.

- \texttt{-o output-file} \hfill Writes the output generated by the command in processing \texttt{LTLSPECs} to the file \texttt{output-file}.

- \texttt{-p "ltl-expr [IN context]"} \hfill An LTL formula to be checked. \texttt{context} is the module instance name which the variables in \texttt{ltl-expr} must be evaluated in.

- \texttt{-P "name"} \hfill Checks the LTL property named \texttt{name}

- \texttt{-n number} \hfill Checks the LTL property with index \texttt{number} in the property database.

\begin{tabular}{|c|c|}
\hline
\textbf{ltl2smv\_single\_justice} & Environment Variable \\
\hline
\end{tabular}

![ltl2smv\_single\_justice](image.png)

Informs the \texttt{ltl2smv} tableau constructor to generate a symbolic fair transition system for the given LTL formula with one single Justice constraint instead of possibly more than one. (This is achieved by replacing the multiple Justice with a single Justice plus an additional monitor.) By default multiple Justice are built.

\begin{tabular}{|c|c|}
\hline
\textbf{check\_compute} & - Performs computation of quantitative characteristics \hfill Command \texttt{check\_compute [-h] [-m | -o output-file] [-n number | -p "compute-expr [IN context]" | -P "name"]} \\
\hline
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.

\[
\begin{align*}
\text{MAX} & : \{\alpha, \beta\} \\
\text{MIN} & : \{\alpha, \beta\}
\end{align*}
\]

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

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

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

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

See variable \texttt{use\_coi\_size\_sorting} for changing properties verification order.

Command Options:

- \texttt{-m} \hfill Pipes the output generated by the command in processing \texttt{COMPUTE}s to the program specified by the \texttt{PAGER} shell variable if defined, else through the UNIX command “more”.  

65
-o output-file
  Writes the output generated by the command in processing COMPUTEs 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.

-P name
  Checks the COMPUTE property named name in the property database.

---

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

```bash
check_property [-h] [-n number | -P "name"] | [(-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.

See variable use_coil_size_sorting for changing properties verification order.

**Command Options:**

- **-n number**
  Checks the property stored at the given index

- **-P name**
  Checks the property named name in the property database.

- **-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

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

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 INV 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.

-n "name" Sets the name of the property to “name”

---

**show_property** - Shows the currently stored properties

```
show_property [-h] [-n idx | -P "name"] [-c | -l | -i | -s | -q] [-f | -v | -u] [-m | -o] [-F format]
```

Shows the properties currently stored in the list of properties. This list is initialized with the properties (CTL, LTL, INV, COMPUTE) present in the input file, if any; then all of the properties added by the user with the relative **check_property** or **add_property** commands are appended to this list. For every property, the following informations are displayed:

- the identifier of the property (a progressive number);
- the property name if available;
- the property formula;
- the type (CTL, LTL, INV, PSL, COMPUTE)
- the status of the formula (Unchecked, True, False) or the result of the quantitative expression, if any (it can be infinite);
- if the formula has been found to be false, the index number of the corresponding counterexample trace.

By default, all the properties currently stored in the list of properties are shown. Specifying the suitable options, properties with a certain status (Unchecked, True, False) and/or of a certain type (e.g. CTL, LTL), or with a given identifier, it is possible to let the system show a restricted set of properties. It is allowed to insert only one option per status and one option per type.

**Command Options:**

- `-P name` Prints out the property named "name"
- `-n idx` Prints out the property numbered "idx"
- `-c` Prints only CTL properties
- `-l` Prints only LTL properties
- `-i` Prints only INV properties
- `-q` Prints only COMPUTE properties
- `-u` Prints only unchecked properties
- `-t` Prints only those properties found to be true
- `-f` Prints only those properties found to be false
-s          Prints the number of stored properties
-o filename  Writes the output generated by the command to filename
-F format    Prints with the specified format. tablear and xml are common formats, however use -F help to see all available formats.
-m          Pipes the output through the program specified by the PAGER shell variable if defined, else through the UNIX "more" command

**convert.property.to.invar** - Convert, when possible, properties to invariant properties

```
convert.property.to.invar[-n number | -P "name" | -p (G next-expr | AG next-expr)]
```

Convert CTL and LTL properties to invariant ones. Only properties of the form “AG next-expr” and “G next-expr” are processed. The conversion is performed over the specification selected with one between -n, -P or -p, if given, or all the CTL and LTL properties in the model. The generated properties are added to the database (they can be listed with the command show.property).

Command Options:
- n number      Convert CTL or LTL property with index “number”.
- P "name"     Convert CTL or LTL property named “name”.
- p G next-expr | AG next-expr     Convert the given CTL or LTL formula.

**write.coi.model** - Writes a restricted flat model to a file

```
write.coi.model [-h] [-n idx | -p "expr" | -P "name"] [-c | -l | -i | -s | -q] [-C] [-g]
```

Writes the currently loaded SMV model in the specified file, after having flattened it. If a property is specified, the dumped model is the result of applying the Cone Of Influence over that property. otherwise, a restricted SMV model is dumped for each property in the property database.

Processes are eliminated and a corresponding equivalent model is printed out.

If no file is specified, stderr is used for output.

Command Options:
- o filename     Attempts to write the flat SMV model in filename
- p expr        Applies COI for the given expression expression. Notice that also the property type has to be specified
- P name        Applies COI for property named "name"
- n idx         Applies COI for property stored with index "idx"
- c             Dumps COI model for all CTL properties
- l             Dumps COI model for all LTL properties
- i             Dumps COI model for all INVAR properties
Dumps COI model for all PSL properties

-\texttt{q} Dumps COI model for all COMPUTE properties

-\texttt{c} Only prints the list of variables that are in the COI of properties

-\texttt{g} Dumps the COI model that represents the union of all COI properties

<table>
<thead>
<tr>
<th>\texttt{cone_of_influence}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses the cone of influence reduction when checking properties. When cone of influence reduction is active, the problem encoded in the solving engine consists only of the relevant parts of the model for the property being checked. This can greatly help in reducing solving time and memory usage. Note however, that a COI counter-example trace may or may not be a valid counter-example trace for the original model.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{use_coi_size_sorting}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses the cone of influence variables set size for properties sorting, before the verification step. If set to 1, properties are verified starting with the one that has the smallest COI set, ending with the property with the biggest COI set. If set to 0, properties are verified according to the declaration order in the input file</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{prop_print_method}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determines how properties are printed. The following methods are available:</td>
<td></td>
</tr>
<tr>
<td>\texttt{name} Prints the property name. If not available, defaults to method “index”</td>
<td></td>
</tr>
<tr>
<td>\texttt{index} Prints the property index. If not available, defaults to method “truncated”</td>
<td></td>
</tr>
<tr>
<td>\texttt{truncated} Prints the formula of the property. If the formula is longer than 40 characters, it is truncated</td>
<td></td>
</tr>
<tr>
<td>\texttt{formula} The default method, simply prints the formula</td>
<td></td>
</tr>
</tbody>
</table>

### 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 two SAT solvers: Zchaff and MiniSat. If none of the two is enabled, all Bounded Model Checking part in NuSMV will not be available. 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.

<table>
<thead>
<tr>
<th>Command</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bmc_setup</strong> - Builds the model in a Boolean Expression format.</td>
<td></td>
</tr>
<tr>
<td>bmc_setup [-h]</td>
<td>You must call this command before use any other bmc-related command. Only one call per session is required.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>go_bmc</strong> - Initializes the system for the BMC verification.</td>
<td></td>
</tr>
<tr>
<td>go_bmc [-h] [-f]</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command Options:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f Forces model construction even when Cone Of Influence is enabled.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sexp_inlining</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rbc_inlining</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rbc_rbc2cnf_algorithm</strong></td>
<td>This variable defines the algorithm used for conversion from RBC to CNF format in which a problem is supplied to a SAT solver. The default value &quot;sheridan&quot; refers to [She04] algorithm which allows to obtain a more compact CNF formulas. The other value &quot;tseitin&quot; refers to a standard Tseitin transformation algorithm.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>check_ltlspec_bmc</strong> - Checks the given LTL specification, or all LTL specifications if no formula is given. Checking parameters are the maximum length and the loopback value</td>
<td></td>
</tr>
</tbody>
</table>
check_ltlspec_bmc [-h] | [-n idx | -p "formula [IN context]" | -P "name"] [-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.
- **-P name**
  - Checks the LTL property named name in the property database.
- **-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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_ltlspec_bmc_onepb [-h]</td>
</tr>
</tbody>
</table>

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.

- **-P name**
  Checks the LTL property named name in the property database.

- **-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.

**Command**

```bash
gen_ltlspec_bmc [-h] | [ -n idx | -p "formula" [IN context] | -P "name"] [-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]"
  `formula` specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.

- `-P name`
  Checks the LTL property named `name` in the property database.

- `-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

```
gen_ltlspec_bmc_onepb [-h | [ -n idx | -p "formula" [IN context] | -P "name"] [-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.

- **-P name**
  
  Checks the LTL property named name in the property database.

- **-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 \((-1, -\text{length})\). 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”.

\[-o \text{filename}\]

\(\text{filename}\) is the name of the dumped dimacs file. If this option is not specified, variable \(\text{bmc} \_\text{dimacs}\_\text{filename}\) 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

\text{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.

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 \text{check\_ltlspec\_bmc} (see the description on page 70) 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.

See variable \text{use\_coi\_size\_sorting} for changing properties verification order.

Command Options:
- \(-n \text{index}\)  \(\text{index}\) is the numeric index of a valid LTL specification formula actually located in the properties database.
- \(-p \text{"formula [IN context]"}\)  Checks the formula specified on the command-line. context is the module instance name which the variables in formula must be evaluated in.
- \(-P \text{name}\)  Checks the LTL property named \text{name} in the property database.
- \(-k \text{max\_length}\)  \text{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 \text{bmc\_length} is considered instead.
The loopback value may be:

- a natural number in \((0, \text{max}\_\text{length}-1)\). 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 \((-1, -\text{bmc}\_\text{length})\). In this case loopback is considered a value relative to \text{max}\_\text{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.

```bash
check_ltlspec_sbmc [-h] | [-n idx | -p "formula [IN context]" | -P "name"] [-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 \text{max}\_\text{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 \text{bmc}\_\text{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 \text{bmc}\_\text{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 \(-o\ "filename\"\).

See variable use\_col\_size\_sorting for changing properties verification order.

**Command Options:**

- \(-n\ index\)
  - \text{index}\ is the numeric index of a valid LTL specification formula actually located in the properties database.

- \(-p\ "formula [IN context]\"
  - Checks the \text{formula} specified on the command-line. \text{context}\ is the module instance name which the variables in \text{formula} must be evaluated in.

- \(-P\ name\)
  - Checks the LTL property named \text{name} in the property database.

- \(-k\ max\_length\)
  - \text{max}\_\text{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 \text{bmc}\_\text{length} is considered instead.
The loopback value may be:

- a natural number in \((0, \text{max\_length}-1)\). 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 \((-1, -\text{bmc\_length})\). In this case loopback is considered a value relative to \text{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 \text{length}-1".

check_ltlspec sbmc inc

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 \text{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 \text{bmc\_length}.

The property to be checked may be specified using the \(-n\) idx, the \(-p\) "formula\(\)\) [IN context]" or the \(-P\) "name" options.

See variable \text{use\_coi\_size\_sorting} for changing properties verification order.

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.

\(-P\) name
Checks the LTL property named name in the property database.
-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.

-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.

gen_ltlspec sbmc [\(-h\) | \[-n idx | -p "formula [IN context]\] | -P "name\] [-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 the 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.

- P "name"

Checks the LTL property named name

- 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 \((-1, -\text{bmc}\_\text{length})\). In this case \text{loopback} is considered a value relative to \text{max}\_\text{length}. Any invalid combination of \text{length} and \text{loopback} will be skipped during the generation/solving process.
- the symbol ‘\(\times\)’, which means “no loopback”.
- the symbol ‘\(*\)’, which means “all possible loopbacks from zero to \text{length}-1”.

\[-o\text{ }\text{filename}\]

\text{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.

<table>
<thead>
<tr>
<th>\text{bmc}_\text{length}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the generated problem bound. Possible values are any natural number, but must be compatible with the current value held by the variable \text{bmc}_\text{loopback}. The default value is 10.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{bmc}_\text{loopback}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the generated problem loop. Possible values are:</td>
<td></td>
</tr>
<tr>
<td>- Any natural number, but less than the current value of the variable \text{bmc}_\text{length}. In this case the loop point is absolute.</td>
<td></td>
</tr>
<tr>
<td>- Any negative number, but greater than or equal to (-\text{bmc}_\text{length}). In this case specified loop is the loop length.</td>
<td></td>
</tr>
<tr>
<td>- The symbol ‘(\times)’, which means “no loopback”.</td>
<td></td>
</tr>
<tr>
<td>- The symbol ‘(*)’, which means “any possible loopbacks”.</td>
<td></td>
</tr>
<tr>
<td>The default value is ‘(*)’.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{bmc}_\text{optimized}_\text{tableau}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses depth1 optimization for LTL Tableau construction in BMC.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{bmc}_\text{force}_\text{pltl}_\text{tableau}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forces to use PLTL instead of LTL for BMC tableau construction.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{bmc}_\text{dimacs}_\text{filename}</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 \text{gen}_\text{ltl}_\text{spec}_\text{bmc} family. DIMACS file name can contain special symbols which will be expanded to represent the actual file name. Possible symbols are:</td>
<td></td>
</tr>
<tr>
<td>- @F The currently loaded model name with full path.</td>
<td></td>
</tr>
<tr>
<td>- @f The currently loaded model name without path part.</td>
<td></td>
</tr>
<tr>
<td>- @n The numerical index of the currently processed formula in the property database.</td>
<td></td>
</tr>
</tbody>
</table>

79
• @k The currently generated problem length.
• @l The currently generated problem loopback value.
• @@ The ‘@’ character.

The default value is “@f @k @l @n”.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Command Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmc_sbmc_gf_fg.opt</td>
<td>check_invar_bmc [-h</td>
</tr>
</tbody>
</table>

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.

See variable use, col, size, sorting for changing properties verification order.

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.

- **-P name**
  - Checks the INVAR property named name in the property database.

- **-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.

- **-e**
  - Performs an extra induction step for finding a proof. Can be used only with the “een-sorensson” algorithm

- **-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.
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

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

At the moment, the invariants are generated using “classic” algorithm only (see the description of **check_invar_bmc** on page 80).

**Command Options:**

- \-n index
- \-p "formula [IN context]"

index is the numeric index of a valid INV AR specification formula actually located in the property database. The validity of index value is checked out by the system.

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

Checks the INV AR property named name in the property database.

- \-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

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

This command does the same thing as **check_invar_bmc** (see the description on page 80) 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.

See variable use_col_size_sorting for changing properties verification order.

Command Options:
- `-n index`  
  *index* is the numeric index of a valid INV AR 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.

- `-P "name"`  
  Checks the INVARSPEC property named *name*

- `-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.

- `-K step_size`  
  Only for falsification: increment the search of *step_size* at a time. Must be greater than zero (1 by default).

- `-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”.

---

82
The SAT solver’s name actually to be used. Default SAT solver is MiniSat. Depending on the NuSMV configuration, also the Zchaff SAT solver can be available or not. Notice that Zchaff and MiniSat are for non-commercial purposes only. If no SAT solver has been configured, BMC commands and environment variables will not be available.

**bmc_pick_state - Picks a state from the set of initial states**

```
bmc_pick_state [-h] [-v] [-c "constraint" | -s trace.state] [-r | -i [-a]]
```

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 rst state of a new trace ready to be lengthened by steps states by the `bmc_simulate` command or the `bmc_inc_simulate` command.

Command Options:
- `-v` Verbose prints the generated trace
- `-c constraint` Set a constraint to narrow initial states.
- `-s state` Picks state from trace.state label.
- `-r` Randomly picks a state from the set of initial states.
- `-i` Enters simulation’s interactive mode.
- `-a` Displays all the state variables (changed and unchanged) in the interactive session

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

```
bmc_simulate [-h] [-p | -v] [-r] [[-c "constraints"] | [-t "constraints"]] [-k steps]
```

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:
- `-p` Prints the generated trace (only changed variables).
- `-v` Prints the generated trace (all variables).
- `-r` Picks a state from a set of possible future states in a random way.
- `-c constraint` 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 " ". The expression cannot contain next operators, and is automatically shifted by one state in order to constraint only the next steps.
-t "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 \( k \) trace is obtained. Note: constraints must be enclosed between double quotes " ". The expression can contain next operators, and is NOT automatically shifted by one state as done with option -c.

-k steps  Maximum length of the path according to the constraints.
The length of a trace could contain less than \( k \) states: this is the case in which simulation stops in an intermediate step because it may not exist any future state satisfying those constraints. The default value is determined by the default_simulation_steps environment variable.

---

### `bmc_inc_simulate` - Generates a trace of the model from 0 (zero) to \( k \)

```
bmc_inc_simulate [-h] [-p | -v] [-r | -i [-a]] [\ [-c "constraints"] | [-t "constraints"] ] [-k steps]
```

Performs incremental simulation of the model. `bmc_inc_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:**

- `-p`  Prints the generated trace (only changed variables).
- `-v`  Prints the generated trace (all variables).
- `-r`  Picks a state from a set of possible future states in a random way.
- `-i`  Enters simulation’s interactive mode.
- `-a`  Displays all the state variables (changed and unchanged) in the interactive session.
- `-c constraint`  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 \( k \) trace is obtained. Note: constraints must be enclosed between double quotes " ". The expression cannot contain next operators, and is automatically shifted by one state in order to constraint only the next steps.
-t "constraints"        Performs a simulation in which computation is restricted
to states satisfying those constraints. The desired se-
quence 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 " ". The expression can contain
next operators, and is NOT automatically shifted by one
state as done with option -c

-k 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 intermedi-
ate step because it may not exist any future state satisfying
those constraints. The default value is determined by the
default_simulation_steps environment variable

---

### bmc_simulate_check_feasible_constraints - Checks feasibility for the given constraints

```bash
bmc_simulate_check_feasible_constraints [-h] [-q] [-c "constr"]
```

Checks if the given constraints are feasible for BMC simulation.

**Command Options:**

- `-q` Prints the output in compact form.
- `-c constr` Specify one constraint whose feasability has to be checked
  (can be used multiple times, order is important to read the result)

---

### 3.4 Commands for checking PSL specifications

The following commands allow for model checking of PSL specifications.

#### check_pslspec - Performs bdd-based PSL model checking

```bash
check_pslspec [-h] [-m | -o output-file] [-n number] [-p "psl-expr [IN context]" | -P "name"]
```

Check psl properties using bdd-based model checking.

- A `psl-expr` to be checked can be specified at command line using option `-p`. Alternati-
  vely, 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.
- See variable `use_coil.size_sorting` for changing properties verification order.

**Command Options:**

- `-m` Pipes the output generated by the command in processing PSLSPECs 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 PSLSPECs 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.

-P name  Checks the PSL property named name in the property database.

<table>
<thead>
<tr>
<th>check_pslspec_bmc</th>
<th>Performs SAT-based PSL model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Options:</td>
<td></td>
</tr>
</tbody>
</table>
| -m                | Pipes the output generated by the command in processing PSLSPECs 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 PSLSPECs 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.
| -P name           | Checks the PSL property named name in the property database.
| -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 -l.

See variable use.col.size_sorting for changing properties verification order.
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.

---

**check_pslspec_bmc_inc** - Performs incremental SAT-based PSL model checking

**Command**

```
check_pslspec_bmc_inc [-h] [-m | -o output-file] [-n number | -p "psl-expr [IN context]" | -P "name"] [-l loopback]
```

Check psl properties using incremental SAT-based 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. 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_length* and *bmc_loopback*. Single problems can be generated by using option `-l`. Bounded model checking problems can be generated and dumped in a file by using option `-g`.

See variable *use_col_size_sorting* for changing properties verification order.

**Command Options:**

- `-m`
  Pipes the output generated by the command in processing PSLSPECs 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 PSLSPECs 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.

-P name  Checks the PSL property named name in the property database.

-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.

---

**check_pslspec_sbmc - Performs SAT-based PSL model checking**

Check psl properties using SAT-based model checking. Use the SBMC algorithms.
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. 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_length and bmc_loopback. Single problems can be generated by using option -1. Bounded model checking problems can be generated and dumped in a file by using option -g.

See variable use_col_size_sorting for changing properties verification order.

**Command Options:**

-m  Pipes the output generated by the command in processing PSLSPECs to the program specified by the PAGER shell variable if defined, else through the UNIX command “more”.

---

88
-o output-file  Writes the output generated by the command in processing PSLSPECs 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.
-P name  Checks the PSL property named name in the property database.
-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.
-l  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.

check_pslspec_sbmcc_inc - Performs incremental SAT-based PSL model checking


Check psl properties using incremental SAT-based model checking. Use the SBMC algorithms. 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. 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_length and bmc_loopback. Single problems can be generated by using option -l. Bounded model checking problems can be generated and dumped in a file by using option -g. With the option -c is possible to perform a completeness check, while with the option -N is possible to disable the virtual unrolling.
See variable use.col.size.sorting for changing properties verification order.

Command Options:

- `m` Pipes the output generated by the command in processing PSLSPECs 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 PSLSPECs 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.

- `P name` Checks the PSL property named name in the property database.

- `l` 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.

- `c` Performs completeness check.

- `N` Does not perform virtual unrolling.

### 3.5 Simulation Commands

In this section we describe the commands that allow to simulate a NuSMV specification. See also the Section 3.7 [Traces], page 94, that describes the commands available for manipulating traces.

<table>
<thead>
<tr>
<th>pick_state</th>
<th>-p</th>
<th>Picks a state from the set of initial states</th>
</tr>
</thead>
<tbody>
<tr>
<td>pick_state</td>
<td>[-h] [-v] [-r</td>
<td>-i [-a]] [-c &quot;constraints&quot;</td>
</tr>
</tbody>
</table>
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 and frozen 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 frozen and 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 and frozen 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 " ".

- **-s trace.state** Picks state from trace.state label. A new simulation trace will be created by copying prefix of the source trace up to specified state.

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

```
simulate [-h] [-p | -v] [-r | -i [-a]] [-c "constraints" | -t "constraints"] [-k 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 and frozen 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 and frozen 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 " ". The expression cannot contain next operators, and is automatically shifted by one state in order to constraint only the next steps.
-t "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 " ". The expression can contain next operators, and is NOT automatically shifted by one state as done with option -c.
-k 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. The default value is determined by the default_simulation_steps environment variable.

<table>
<thead>
<tr>
<th>default_simulation_steps</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Controls the default number of steps performed by all simulation commands. The default is 10.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shown_states</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Controls the maximum number of states tail will be shown during an interactive simulation session. Possible values are integers from 1 to 100. The default value is 25.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>traces_hiding_prefix</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>see section 3.7.2 for a detailed description.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>traces_regexp</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>see section 3.7.2 for a detailed description.</td>
</tr>
</tbody>
</table>

## 3.6 Execution Commands

In this section we describe the commands that allow to perform trace re-execution on a given model. See also the section Section 3.7 [Traces], page 94 that describes the commands available for manipulating traces.

<table>
<thead>
<tr>
<th>execute_traces</th>
<th>- Executes complete traces on the model FSM</th>
</tr>
</thead>
</table>

execute_traces [-h] [-v] [-m | -o output-file] -e engine [-a | trace_number]

Executes traces stored in the Trace Manager. If no trace is specified, last registered trace is executed. Traces must be complete in order to perform execution.  
Command Options:

- **-v**  Verbosely prints traces execution steps.
- **-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.
- **-e engine**  Selects an engine for trace re-execution. It must be one of ‘bdd’, ‘sat’ or ‘smt’.
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 executed.

**execute_partial_traces** - Executes partial traces on the model FSM

```
execute_partial_traces [-h] [-v] [-r] [-m | -o output-file]
-e engine [-a | trace_number]
```

Executes traces stored in the Trace Manager. If no trace is specified, last registered trace is executed. Traces are not required to be complete. Upon successful termination, a new complete trace is registered in the Trace Manager.

**Command Options:**

- **-v**
  Verbosely prints traces execution steps.

- **-a**
  Prints all the currently stored traces.

- **-r**
  Performs restart on complete states. When a complete state (i.e. a state which is non-ambiguosly determined by a complete assignment to state variables) is encountered, the re-execution algorithm is re-initialized, thus reducing computation time.

- **-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.

- **-e engine**
  Selects an engine for trace re-execution. It must be one of ‘bdd’, ‘sat’ or ‘smt’.

- **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 executed.

### 3.7 Traces

A trace consists of an initial state, optionally followed by a sequence of states-inputs pairs corresponding to a possible execution of the model. Apart, from the initial state, 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 and frozen 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/frozen variables and/or `NEXT`ed state variables, then it gets displayed in a separate “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 90) or partial trace re-execution (Section 3.6 [Execution Commands], page 93). Each trace has a number, and the
states-inputs pairs are numbered within the trace. Trace \( n \) has states/inputs \( n.1, n.2, n.3, \ldots \) where \( n.1 \) represents the initial state.

When Cone of Influence (COI) is enabled when generating a trace (e.g. when performing model checking), the generated trace will contain only the relevant symbols (variables and DEFINEs) which are in the COI projected by the variables occurring in the property which is being checked. The symbols which are left out of the COI, will be not visible in the generated trace, as they do not occur in the problem encoded in the solving engine. Notice that when COI is enabled, the generated trace may or may not be a valid counter-example trace for the original model.

### 3.7.1 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto_state ([-h]) state_label</td>
<td>Goes to a given state of a trace. ( state_label ) is in the form ( trace.state ) where ( trace ) is the index of the trace which the state has to be taken from, ( state ) is the index of the state within the given trace. If ( state ) is a negative number, then the state index is intended to be relative to the length of the given trace. For example ( 2.-1 ) means the last state of the trace ( 2 ). ( 2.-2 ) is the state before the last state, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_current_state ([-h]) ([-v])</td>
<td>Prints the name of the current state if defined.</td>
</tr>
<tr>
<td>(-v)</td>
<td>Prints the value of all the state and frozen variables of the current state.</td>
</tr>
</tbody>
</table>

### 3.7.2 Displaying Traces

NuSMV comes with three trace plugins (see Section 3.8 Trace Plugins, page 98) 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 90) 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.

Generation and displaying of traces can be enabled/disabled by setting variable counter_examples. Some filtering of symbols that are presented when showing traces can be controlled by variables traces_hiding_prefix and traces_regexp.
This determines whether traces are generated when needed. See also command line option 
-dcx.

**traces_hiding_prefix** Environment Variable

Symbols names that match this string prefix will be not printed out when showing a trace. 
This variable may be used to avoid displaying symbols that are expected to be not visible to the user. For example, this variable is exploited when dumping booleanized models, as NuSMV may introduce hidden placement symbols as DEFINES that do not carry any useful information for the user, and that would make traces hardly readable if printed. Default is __.

**traces_regexp** Environment Variable

Only symbols whose names match this regular expression will be printed out when showing a trace. This option might be used by users that are interested in showing only some symbol names. Names are first filtered out by applying matching of the dual variable traces_hiding_prefix, and then filtered names are checked against content of traces_regexp. Given regular expression can be a Posix Basic Regular Expression. Matching is carried out on symbol names without any contextual information, like module hierarchy. For example in ml.m2.name only name is checked for filtering. Notice that depending on the underlying platform and operating system this variable might be not available.

**show_defines_in_traces** Environment Variable

Controls whether defines should be printed as part of a trace or be skipped. Skipping printing of the defines can help in reducing time and memory usage required to build very big traces.

**traces_show_defines_with_next** Environment Variable

Controls whether defines containing next operators should be printed as part of a trace or be skipped.

### 3.7.3 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:

```
```

Command Options:

- **-v**
  Verbosely prints traces content (all state and frozen 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. Omitting the trace number causes the most recently generated trace to be printed.

- **from_step**
  The number of the first step of the trace to be printed. Negative numbers can be used to denote right-to-left indexes from the last step.

- **to_step**
  The number of the trace to be printed. Negative numbers can be used to denote right-to-left indexes from the last step. Omitting this parameter causes the entire suffix of the trace to be printed.

- **-A**
  Prints the trace(s) using a rewriting mapping for all symbols. The rewriting is the same used in `write_flat_model` with option `-A`.

Shows the traces currently stored in system memory, if any. By default it shows the last generated trace, if any. Optional trace number can be followed by two indexes (from_state, to_state), denoting a trace “slice”. Thus, it is possible to require printout only of an arbitrary fragment of the trace (this can be helpful when inspecting very big traces).

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 filename] [-u] [-s] filename]
```

97
3.8 Trace Plugins

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

- Basic Trace Explainer
- States/Variables Table
- XML Format Printer
- Empty Trace

There is also an xml loader which can read in any trace which has been output to a file by the XML Format Printer. Note however that this loader is only available on systems that have the libxml2 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 previously generated, or loaded, trace to a specific file.

3.8.1 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 and frozen variables and their initial values. States are numbered in the following fashion:

\[
\text{trace\_number.state\_number}
\]

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 or frozen 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:

\[
\text{-> Input: TRACE\_NO.STATE\_NO <-} \\
\text{/* for each input var (being printed), i: */} \\
\text{INPUT\_VARi = VALUE} \\
\text{-> State: TRACE\_NO.STATE\_NO <-} \\
\text{/* for each state and frozen var (being printed), j: */} \\
\text{STATE\_VARj = VALUE} \\
\text{/* for each combinatorial constant (being printed), k: */} \\
\text{CONSTANTk = VALUE}
\]

where INPUT\_VAR, STATE\_VAR and CONSTANT have the relevant module names prepended to them (separated 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.

### 3.8.2 States/Variables Table

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

\[
S_1 \ C_2 \ I_2 \ S_2 \ ... \ C_n \ In \ S_n
\]

where \( S_1 \) 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 or frozen 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 and frozen 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.

The compact version has the states on the rows and no distinction is made between variables:

\[
\text{Step1} \ \text{Step2} \ ... \ \text{Stepn}
\]

### 3.8.3 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
<?XML_VERSION_STRING?>
<counter-example type=TRACE_TYPE desc=TRACE_DESC>
  /* for each state, i: */
  <node>
    <state id=i>
      /* for each state and frozen var, j: */
      <value variable=j>VALUE</value>
    </state>
  </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.

### 3.8.4 XML Format Reader

This plugin makes use of the libxml2 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 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.

### 3.8.5 Empty Trace

This plugin simply disables trace printing. Traces are still computed and stored: unset system option `counter_examples` for performance gain if traces are of no interest.

### 3.9 Interface to the DD Package

NuSMV uses the state of the art BDD package CUDD \[\text{Som98}\]. Control over the BDD package can be 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

---

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 $M_2$. 

100
reordering is enabled with the shell command `dynamic_var_ordering` with the option `-e`, and disabled with the `-d` option. Variable `dynamic_reorder` can also be used to determine whether dynamic reordering is active. If dynamic reordering is enabled it may be beneficial also to disable BDD caching by unsetting variable `enable_sexp2bdd_caching`.

### dynamic_reorder

<table>
<thead>
<tr>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determines whether dynamic reordering is active. If this variable is set, dynamic reordering will take place as described above. If not set (default), no dynamic reordering will occur. This variable can also be set by passing <code>-dynamic</code> command line option when invoking NuSMV.</td>
</tr>
</tbody>
</table>

### reorder_method

<table>
<thead>
<tr>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 <code>sift</code>.</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

- **window2:**

- **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.
The window\{2,3,4\} method is iterated until no further improvement is obtained.

This method is similar to symmetry.sift, but uses more general criteria to create groups.

The group.sift method is iterated until no further improvement is obtained.

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

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

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.

This method is a combination of sifting and linear transformations.

The linear method is iterated until no further improvement is obtained.

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

```
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.
  - **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>
```

**clean_sexp2bdd_cache** - Cleans the cached results of evaluations of symbolic expressions to ADD and BDD representations.

```
clean_sexp2bdd_cache [-h]
```
During conversion of symbolic expressions to ADD and BDD representations the results of evaluations are normally cached (see additionally the environment variable `enable_sexp2bdd_caching`). This allows to save time by avoid the construction of BDD for the same symbolic expression several time.

In some situations it may be preferable to clean the cache and free collected ADD and BDD. This operation can be done, for example, to free some memory. Another possible reason is that dynamic reordering may modify all existing BDDs, and cleaning the cache thereby freeing the BDD may speed up the reordering.

This command is designed specifically to free the internal cache of evaluated expressions and their ADDs and BDDs. Note that only the cache of symbolic-expression-to-bdd evaluator is freed. BDDs of variables, constants and expressions collected in BDD FSM or anywhere else are not freed.

**print_formula** - Prints a formula in canonical format.

```
print_formula [-h] [-v] [-f] "expression"
```

Prints the number of satisfying assignments for the given formula. In verbose mode, prints also the list of such assigments. In formula mode, a canonical representation of the formula is printed.

Command Options:

- **-v**
  - Prints explicit models of the formula.

- **-f**
  - Prints the simplified and canonical formula.

**enable_sexp2bdd_caching** - Environment Variable

This variable determines if during evaluation of symbolic expression to ADD and BDD representations the obtained results are cached or not. Note that if the variable is set down consequently computed results are not cached but the previously cached data remain un-modified and will be used during later evaluations.

The default value of this variable is 1 which can be changed by a command line option `-disable_sexp2bdd_caching`.

For more information about the reasons of why BDD cache should be disabled in some situations see command `clean_sexp2bdd_cache`.

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

```
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.

```
set_bdd_parameters [-h] [-s]
```
Applies the variables table of the NuSMV environment 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.

### 3.10 Administration Commands

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>! - shell_command</td>
<td>! &quot;shell_command&quot; executes a shell command. The &quot;shell_command&quot; is executed by calling &quot;bin/sh -c shell_command&quot;. If the command does not exist or you have not the right to execute it, then an error message is printed.</td>
</tr>
</tbody>
</table>
| alias [-h] [<name> [<string>]] | 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

```
echo [-h] [-2] [-n] [-o filename [-a]] <string>
```

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

Command Options:

- `-2` Redirects output to the standard error instead of the standard output. This cannot be used in combination with the option -o.
- `-n` Does not output the trailing newline.
- `-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 overwriting it. Use only with the option -o.

### help - Provides on-line information on commands

```
help [-h] [-a] [-p] [<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.
- `-p` Disables the use of a pager like "more" or any set in environment variable PAGER.

### history - list previous commands and their event numbers

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

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

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

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

The default history substitution character is the ‘%’ (‘!’ is default for shell escapes, and ‘#’ marks the beginning of a comment). This can be changed using the set command. In this description ‘%’ is used as the history_char. The ‘%’ can appear anywhere in a line.

A line containing a history substitution is echoed to the screen after the substitution takes place. ‘%’ can be preceded by a ‘\’ in order to escape the substitution, for example, to enter a ‘%’ into an alias or to set the prompt.

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

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

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.

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

quit [-h] [-s] [-x]
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.
Leaves immediately. Skip all the cleanup code, leaving it to the OS. This can save quite a long time.

**reset - Resets the whole system.**

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

**set - Sets an environment variable**

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;name&gt;</td>
<td>Variable name</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>Value to be assigned to the variable</td>
</tr>
</tbody>
</table>

Using the set command to set a variable, without giving any explicit value is allowed, and sets the variable to 1:

```NuSMV> set foo
```
will set the variable foo to 1.

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**

```
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 NUWSMV 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

### 3.11 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.
<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>filec</strong></td>
<td>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 &lt;TAB&gt; 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 <code>set filec</code> 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.</td>
</tr>
<tr>
<td><strong>shell_char</strong></td>
<td><code>shell_char</code> specifies a character to be used as shell escape. The default value of this environment variable is ‘!’ .</td>
</tr>
<tr>
<td><strong>history_char</strong></td>
<td><code>history_char</code> specifies a character to be used in history substitutions. The default value of this environment variable is ‘%’.</td>
</tr>
<tr>
<td><strong>open_path</strong></td>
<td><code>open_path</code> (in analogy to the shell-variable <code>PATH</code>) 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 (<code>NuSMV_LIBRARY_PATH</code>) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.</td>
</tr>
<tr>
<td><strong>nusmv_stderr</strong></td>
<td>Standard error (normally stderr) can be re-directed to a file by setting the variable <code>nusmv_stderr</code>.</td>
</tr>
<tr>
<td><strong>nusmv_stdout</strong></td>
<td>Standard output (normally stdout) can be re-directed to a file by setting the internal variable <code>nusmv_stdout</code>.</td>
</tr>
<tr>
<td><strong>nusmv_stdin</strong></td>
<td>Standard input (normally stdin) can be re-directed to a file by setting the internal variable <code>nusmv_stdin</code>.</td>
</tr>
<tr>
<td><strong>pp_cpl_path</strong></td>
<td><code>pp_cpl_path</code> is absoult or relative path to the preprocessor. If this value is set, preprocessor from the path will be invoked. If this value is not set, the system will invoke default cpp preprocessor.</td>
</tr>
<tr>
<td><strong>pp_m4_path</strong></td>
<td><code>pp_m4_path</code> is absoult or relative path to the preprocessor. If this value is set, preprocessor from the path will be invoked. If this value is not set, the system will invoke default m4 preprocessor.</td>
</tr>
</tbody>
</table>
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:

[-s] [-old] [-old_div_op] [-smv_old]
[-disable_syntactic_checks] [-keep_single_value_vars]

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

-v verbose-level
-verbose-level

Prints the command line help.

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

-int

Enables interactive mode

-source sc_file
-load sc_file

Executes NuSMV commands from file sc_file

Avoids to load the NuSMV commands contained in ~/.nusmvrc or in .nusmvrc or inNuSMV_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. In addition, if enabled, case conditions also accepts "1" which is semantically equivalent to the truth value "TRUE". This backward compatibility feature has been added in NuSMV 2.5.1 in order to help porting of old SMV models. In fact, in versions older than 2.5.1, it was pretty common to use 1 in case conditions expressions. For an example please see section 2.2.3

-old_div_op

Disables the old semantics of "/" and "mod" operations (from NuSMV 2.3.0) instead of ANSI C semantics.

-disable_syntactic

Checks

Disables all syntactic checks that will be performed when flattening the input model. Warning: If the model is not well-formed, NuSMV may result in unpredictable results, use this option at your own risk.

-disable_daggifier

Disables the daggification feature of model dumping

-keep_single

_value_vars

Disables the generation of counter-examples for properties that are proved to be false. See also variable counter_examples

-dcx

Runs pre-processor on SMV files before any of those specified with the -pre option. The user can set his own preprocessor by setting system variables pp_cpp_path and pp_m4_path.
-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-separated list of pre-processor names contained within double quotes.

-ofm fm_file prints flattened model to file fm_file

-obm bm_file Prints boolean model to file bm_file

-lp Lists all properties in SMV model

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

-is Does not check SPEC properties. Sets to “1” the ignore_spec environment variable.

-ic Does not check COMPUTE properties. Sets to “1” the ignore_compute environment variable.

-ils Does not check LTLSPEC properties. Sets to “1” the ignore_ltlspec environment variable.

-ips Does not check PSLSPEC properties. Sets to “1” the ignore_pslspec environment variable.

-ii Does not check INVARSPEC properties. Sets to “1” the ignore_invariant environment variable.

-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_ltlspcmd, but may also lead to a dramatic slowing down. This options has effect only when the calculation of reachable states is enabled (see -f).
Verifies only AG formulas using an ad hoc algorithm (see documentation for the ag_only_search environment variable).

Enables cone of influence reduction. Sets to “1” the cone_of_influence environment variable. We remark that, when cone of influence reduction is enabled, a counter-example trace for a property that does not hold may not be a valid counter-example trace for the original model. We refer the reader to the Frequently Asked Questions (FAQ) [FAQ].

Reads the variable ordering from file iv_file.

Writes the variable ordering to file ov_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].

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

Enables dynamic reordering of variables

Uses method when variable ordering is enabled. Possible values for method are those allowed for the reorder_method environment variable (see Section 3.9 [Interface to DD package], page 100).

Sets the default value of environment variable enable_bdd_cache to 0, i.e. the evaluation of symbolic expression to ADD and BDD representations are not cached. See command clean_sexp2bdd_cache for reasons of why BDD cache should be disabled sometimes.

Sets the default value of environment variable bdd_static_order_heuristics to heuristics, i.e. the option sets up the heuristics to be used to compute BDD ordering statically by analyzing the input model. See the documentation about variable bdd_static_order_heuristics on page 53 for more details.

Enables monolithic transition relation

conjunctive partitioning with threshold of each partition set to cp_f (DEFAULT, with cp_f=1000)

DEPRECATED: use thresh instead.

Enables iwls95 conjunctive partitioning and sets the threshold of each partition to cp_f

Disables affinity clustering

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.

-ojeba algorithm Sets the algorithm used for BDD-based language emptiness of Büchi fair transition systems by setting system variable oreg_justice emptiness_bdd_algorithm (default is EL_bwd). The available algorithms are: EL_bwd EL_fwd
Bibliography


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

\begin{itemize}
  \item \texttt{F, G, X, U, V,}
  \item \texttt{W, H, O, Y, Z,}
  \item \texttt{S, T, B}
  \item \texttt{CTLSPEC}
  \item \texttt{LTLSPEC}
  \item \texttt{INVARSPEC}
  \item \texttt{PSLSPEC}
  \item \texttt{IVAR}
  \item \texttt{FROZENVAR}
  \item \texttt{JUSTICE}
  \item \texttt{COMPASSION}
  \item \texttt{CONSTANTS}
  \item \texttt{word}
  \item \texttt{word1}
  \item \texttt{bool}
  \item \texttt{unsigned}
  \item \texttt{signed}
  \item \texttt{extend}
\end{itemize}

These names are reserved for the LTL temporal operators.

\begin{itemize}
  \item \texttt{TRANS}
  \item \texttt{next\{alpha & next\{beta | next\{gamma\}\}\} -> delta}
\end{itemize}

The \texttt{IMPLEMENTS}, \texttt{INPUT}, \texttt{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 \texttt{\texttt{next}}.

• 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’.

\[
\begin{align*}
\text{INVAR} \\
\quad \text{next}(\alpha) & \land \beta \\
\text{INIT} \\
\quad \text{next}(\beta) & \rightarrow \alpha \\
\text{ASSIGN} \\
\quad \delta := \alpha & \land \text{next}(\gamma); \quad -- \text{normal assignments} \\
\quad \text{init}(\gamma) := \alpha & \land \text{next}(\delta); \quad -- \text{init assignments}
\end{align*}
\]

• It is no longer possible to write ‘SPEC’, ‘FAIRNESS’ statements containing ‘next’.

\[
\begin{align*}
\text{FAIRNESS} \\
\quad \text{next}(\text{running}) \\
\text{SPEC} \\
\quad \text{next}(x) & \land y
\end{align*}
\]

• 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**

\( y: \text{boolean} \);

\( x: \text{boolean} \);

**ASSIGN**

\( \text{next}(x) := x & \text{next}(y); \)

\( \text{next}(y) := y & \text{next}(x); \)

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
- unsigned word\([N]\) (where \(N\) is any whole number \( \geq 1 \))
- signed word\([N]\) (where \(N\) is any whole number \( \geq 1 \))

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

B.2 Implicit Conversion

There is only one kind of implicit conversion. For more information on type ordering see Section 2.2.1 [Implicit Type Conversion], page 10.

Implicit type conversions changes the type of an expression to its counterpart set type. The Figure B.2 shows the direction of such conversions. For more information on set types and their counterpart types see Section 2.1.6 [Set Types], page 8.
### 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 : unsigned word$[N]$ or signed 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

- : integer \( \rightarrow \) integer
  - : unsigned word\([N]\] \( \rightarrow \) unsigned word\([N]\]
  - : signed word\([N]\] \( \rightarrow \) signed word\([N]\]
- \(\text{+, -, /, }\star\) : integer \(\star\) integer \(\rightarrow\) integer
  - : unsigned word\([N]\] \(\star\) unsigned word\([N]\] \(\rightarrow\) unsigned word\([N]\]
  - : signed word\([N]\] \(\star\) signed word\([N]\] \(\rightarrow\) signed word\([N]\]
- \(\text{mod}\) : integer \(\star\) integer \(\rightarrow\) integer
  - : unsigned word\([N]\] \(\star\) unsigned word\([N]\] \(\rightarrow\) unsigned word\([N]\]
  - : signed word\([N]\] \(\star\) signed word\([N]\] \(\rightarrow\) signed word\([N]\]

For operations on words, the result is taken modulo \(2^N\).

- \(\text{\:'+, -\', =\', !=\'}\) : integer \(\star\) integer \(\rightarrow\) boolean
  - : unsigned word\([N]\] \(\star\) unsigned word\([N]\] \(\rightarrow\) boolean
  - : signed word\([N]\] \(\star\) signed word\([N]\] \(\rightarrow\) boolean

### Logic Operators

- ! (negation) : boolean \(\rightarrow\) boolean
  - : unsigned word\([N]\] \(\rightarrow\) unsigned word\([N]\]
  - : signed word\([N]\] \(\rightarrow\) signed word\([N]\]
- 
  \&, |, \neg, <\neg>, \neg\neg, \neg\neg\neg : boolean \(\star\) boolean \(\rightarrow\) boolean
  - : unsigned word\([N]\] \(\star\) unsigned word\([N]\] \(\rightarrow\) unsigned word\([N]\]
  - : signed word\([N]\] \(\star\) signed word\([N]\] \(\rightarrow\) signed word\([N]\]
- \(\text{\:'+, -\', =\', !=\'}\) : integer \(\star\) boolean \(\rightarrow\) boolean
  - : integer \(\star\) boolean \(\rightarrow\) boolean
  - : symbolic enum \(\star\) symbolic enum \(\rightarrow\) boolean
  - : integers-and-symbolic enum \(\star\) integers-and-symbolic enum \(\rightarrow\) boolean
  - : unsigned word\([N]\] \(\star\) unsigned word\([N]\] \(\rightarrow\) boolean
  - : unsigned word\([N]\] \(\star\) signed word\([N]\] \(\rightarrow\) boolean

### Index Subscript Operator

\(\text{\textit{exp}}\_1\ [\text{\textit{exp}}\_2] : \text{array } N..M \text{ of subtype } \text{word}[N] \rightarrow \text{subtype}\)

\(\text{\textit{exp}}\_1\ [\text{\textit{exp}}\_2\ , \text{\textit{exp}}\_3] : \text{array } N..M \text{ of subtype } \text{integer} \rightarrow \text{subtype}\)

The value of \(\text{\textit{exp}}\_2\) has to be in range \([N, M]\).

### Bit-Wise Operators

- \(\text{\textit{\texttt{\textbackslash :}}}(\text{concatenation})\) : word\([N]\] \(\star\) word\([N]\] \(\rightarrow\) unsigned word\([N+M]\]
  
  Where word\([\star]\) can be any of unsigned word\([\star]\] or signed word\([\star]\]

\(\text{\textit{exp}}\_1\ [\text{\textit{exp}}\_2\ , \text{\textit{exp}}\_3] : \text{unsigned word}[N] \star \text{integer} \star \text{integer} \rightarrow \text{unsigned word}[\text{\textit{exp}}\_3 - \text{\textit{exp}}\_2 + 1]\)

\(\text{\textit{exp}}\_1\ [\text{\textit{exp}}\_2\ , \text{\textit{exp}}\_3] : \text{signed word}[N] \star \text{integer} \star \text{integer} \rightarrow \text{unsigned word}[\text{\textit{exp}}\_3 - \text{\textit{exp}}\_2 + 1]\)

Expressions \(\text{\textit{exp}}\_2\) and \(\text{\textit{exp}}\_3\) must be integers such that \(0 \leq \text{\textit{exp}}\_2 \leq \text{\textit{exp}}\_3 < n\).

\(\ll, \gg\) (shift) : unsigned word\([N]\] \(\star\) integer \(\rightarrow\) unsigned word\([N]\]

\(\ll, \gg\) (shift) : unsigned word\([N]\] \(\star\) integer \(\rightarrow\) signed word\([N]\]

\(\ll, \gg\) (shift) : signed word\([N]\] \(\star\) integer \(\rightarrow\) signed word\([N]\]

\(\ll, \gg\) (shift) : signed word\([N]\] \(\star\) unsigned word\([\star]\] \(\rightarrow\) signed 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 conversion is performed on one of the operands

Case and If-Then-Else Expression

\[\text{case } cond_1 : \ result_1; \]
\[cond_2 : \ result_2; \]
\[
\ldots
\]
\[cond_n : \ result_n; \] 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
- 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
  : unsigned word[1] → boolean
  : integer → boolean
toint
  : boolean → integer
  : unsigned word[N] constant → integer
  : signed word[N] constant → integer
word1
  : boolean → unsigned word[1]
signed
  : unsigned word[N] → signed word[N]
unsigned
  : signed word[N] → unsigned word[N]
extend
  : unsigned word[•]* integer → unsigned word[N+integer ]
  : signed word[•]* integer → signed word[N+integer ]
next, init
  : any type → the same type
() :
  : any type → the same type
:=
  : boolean * boolean → 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
  : unsigned word[N] * unsigned word[N] → no type
  : signed word[N] * signed word[N] → 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
  FALSE TRUE

integer_constant :: integer_number

symbolic_constant :: identifier

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

word_sign_specifier :: one of
  u s

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 )
  | abs basic_expr -- absolute value
  | max ( basic_expr , basic_expr ) -- max
  | min ( basic_expr , basic_expr ) -- min
  | ! 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  -- 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[ index ]  -- index subscript
| basic_expr[ integer_number : integer_number ]  -- word bits selection
| basic_expr  :: basic_expr  -- word concatenation
| wordl ( basic_expr )  -- boolean to word[1] conversion
| bool ( basic_expr )  -- word[1] and integer to boolean conversion
| toint ( basic_expr )  -- word[N] and boolean to integer conversion
| signed ( basic_expr )  -- unsigned to signed word conversion
| unsigned ( basic_expr )  -- signed to unsigned word conversion
| extend ( basic_expr , basic_expr )  -- word width increase
| resize ( basic_expr , basic_expr )  -- word width resizing
| basic_expr union basic_expr  -- union of set expressions
| { set_body_expr }  -- set expression
| basic_expr in basic_expr  -- inclusion expression
| basic_expr ? basic_expr : basic_expr  -- if-then-else expression
| count ( basic_expr_list )  -- count of TRUE boolean expressions
| case_expr  -- case expression
| next ( basic_expr )  -- next expression

basic_expr_list ::
| basic_expr
| basic_expr_list , basic_expr

set_body_expr ::
| basic_expr
| set_body_expr , basic_expr

**Case Expression and If-Then-Else Expression**

case_expr :: case case_body esac

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

basic_expr ? 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_typeSpecifier

simple_type_specifier ::
    boolean
| word [ integer_number ]
| unsigned word [ integer_number ]
| signed 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

Module Type Specifier

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

parameter_list ::
    simple_expr
| parameter_list , simple_expr

State, Input and Frozen Variables

var_declaration :: VAR var_list

ivar_declaration :: IVAR simple_var_list

132
frozenvar_declaration :: FROZENVAR simple_var_list
var_list :: identifier : type_specifier ;
      | var_list identifier : type_specifier ;

simple_var_list :: identifier : simple_type_specifier ;
      | simple_var_list identifier : simple_type_specifier ;

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 ::
    var_declaration |
    ivar_declaration |
    frozenvar_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

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 ) -- logical not
    | ! 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 xnor ctl_expr -- logical NOT 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 xnor ltl_expr -- logical NOT 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 quantitative 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 )

psi_unary_expr ::
  + psl_primary_expr
  | - psl_primary_expr
  | ! psl_primary_expr

psi_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

psi_conditional_expr ::
  psl_expr ? psl_expr : psl_expr

psi_case_expr ::
  case
    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

136
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 ;; infinite 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
ƯŚ 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

138
| 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

!, see bang 104

convert property to invar 67
add property 65
alias 104
bmc_simulate 83
bmc_pick_state 82
bmc_setup 69
bmc_simulate_check_feasible_constraints 82
bmc_simulate 82
build boolean model 55
build flat model 55
build model 52
check compute 64
check_ctlspec 59
check fsm 58
check_invar_bmc_inc 80
check_invar_bmc 70
check_invar 60
check_ltlspec_bmc_inc 74
check_ltlspec_bmc_onepb 71
check_ltlspec_bmc 69
check_ltlspec_abmc_inc 76
check_ltlspec_abmc 75
check_ltlspec 63
check property 65
check_pslspec_bmc_inc 86
check_pslspec_bmc 85
check_pslspec_abmc_inc 88
check_pslspec_abmc 87
check_pslspec 84
clean_sexp2bdd_cache 102
compute reachable 57
dump fsm 57
dynamic_var_ordering 101
echo 105
encode_variables 59
execute_partial_traces 93
execute_traces 92
flatten hierarchy 48
gen_invar_bmc 80
gen_ltlspec_bmc_onepb 73
gen_ltlspec_bmc 72
gen_ltlspec_abmc 77
get_internal_status 54
go_bmc_69
goto_state 94
goto 54
help 105
history 105
pick_state 89
print_bdd_stats 103
print_current_state 94
print_fair_states 59
print_fair_transitions 59
print_formula 103
print_fsm_stats 58
print_iwls95options 54
print_reachable_states 58
print_usage 106
process_model 54
quit 106
read_model 48
read_trace 96
reset 107
set_bdd_parameters 103
set 107
show_dependencies 59
show_plugins 95
show_property 66
show_traces 96
show_vars 49
simulate 90
source 108
time 109
unalias 109
unset 110
usage 110
which 110
write_boolean_model 56
write_coi_model 67
write_flat_model 55
write_order 31
Variable Index

NuSMV_LIBRARY_PATH, 111, 114
affinity, 53
ag_only_search, 60
autoexec, 110
backward_compatibility, 49
bdd_static_order_heuristics, 52
bmc_dimacs_filename, 78
bmc_force_ltl_tableau, 78
bmc_inc_invar_alg, 81
bmc_invar_alg, 81
bmc_invar_dimacs_filename, 81
bmc_length, 78
bmc_loopback, 78
bmc_optimized_tableau, 78
bmc_smbc_gf_fg_opt, 79
check_fsm, 58
check_invar_bdd_bmc_heuristic, 63
check_invar_bdd_bmc_threshold, 63
check_invar_forward_backward_heuristic, 63
check_invar_strategy, 63
cone_of_influence, 68
counter_examples, 95
dagifier_counter_threshold, 58
dagifier_depth_threshold, 56
dagifier_enabled, 56
dagifier_statistics, 55
default_simulation_steps, 92
default_trace_plugin, 96
disable_syntactic_checks, 49
dynamic_reorder, 100
enable_sexp2bdd_caching, 103
file, 111
forward_search, 60
history_char, 111
image(?{1,2,3,4}), 53
image_cluster_size, 53
image verbosity, 54
input_file, 48
input_order_file, 51
iwls95preorder, 54
keep_single_value_vars, 49
ltl2smv_single_justice, 62
ltl2tableau_forward_search, 60
nusmv_stderr, 111
nusmv_stdin, 111
nusmv_stdout, 111
on_failure_script quits, 110
open_path, 111
oreg_justice_emptiness_bdd_algorithm, 60
output_boolean_model_file, 56
output_flatten_model_file, 55
output_order_file, 51
output_word_format, 56
partition_method, 53
pp_cpp_path, 111
pp_list, 48
pp_m4_path, 111
prop_print_method, 68
rbc_inlining, 69
rbc_rbc2cnf_algorithm, 69
reorder_method, 100
sat_solver, 82
sexp_inlining, 69
shell_char, 111
show defines in traces, 95
shown_states, 92
traces_hiding_prefix, 92
traces_regexp, 92
traces_show defines with next, 95
trans_order_file, 53
type checking warning on, 49
use_coi_size_sorting, 68
vars_order_type, 57
verbose level, 48
write_order_dumps bits, 51
## Index

### Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nusmvrc</td>
<td>114</td>
</tr>
<tr>
<td>-AG</td>
<td>116</td>
</tr>
<tr>
<td>-bdd</td>
<td>116</td>
</tr>
<tr>
<td>-bmc_length_k</td>
<td>116</td>
</tr>
<tr>
<td>-coi</td>
<td>116</td>
</tr>
<tr>
<td>-cp</td>
<td>116</td>
</tr>
<tr>
<td>-ctt</td>
<td>115</td>
</tr>
<tr>
<td>-dct</td>
<td>114</td>
</tr>
<tr>
<td>-disable_dagifier</td>
<td>114</td>
</tr>
<tr>
<td>-disable_sexp2bdd_caching</td>
<td>116</td>
</tr>
<tr>
<td>-disable_syntactic_checks</td>
<td>114</td>
</tr>
<tr>
<td>-dynamic</td>
<td>116</td>
</tr>
<tr>
<td>-fit</td>
<td>115</td>
</tr>
<tr>
<td>-f</td>
<td>115</td>
</tr>
<tr>
<td>-help</td>
<td>114</td>
</tr>
<tr>
<td>-h</td>
<td>114</td>
</tr>
<tr>
<td>-ic</td>
<td>115</td>
</tr>
<tr>
<td>-iis</td>
<td>115</td>
</tr>
<tr>
<td>-is</td>
<td>115</td>
</tr>
<tr>
<td>-iwls95preorder</td>
<td>116</td>
</tr>
<tr>
<td>-iwls95_cp</td>
<td>116</td>
</tr>
<tr>
<td>-iwls95_cp_file</td>
<td>116</td>
</tr>
<tr>
<td>-keep_single_value_vars</td>
<td>114</td>
</tr>
<tr>
<td>-lp</td>
<td>115</td>
</tr>
<tr>
<td>-mono</td>
<td>116</td>
</tr>
<tr>
<td>-m_method</td>
<td>116</td>
</tr>
<tr>
<td>-noaffinity</td>
<td>110</td>
</tr>
<tr>
<td>-n_idx</td>
<td>115</td>
</tr>
<tr>
<td>-obm bm_file</td>
<td>115</td>
</tr>
<tr>
<td>-ofm fn_file</td>
<td>115</td>
</tr>
<tr>
<td>-ojeba algorithm</td>
<td>116</td>
</tr>
<tr>
<td>-old_div_op</td>
<td>116</td>
</tr>
<tr>
<td>-old</td>
<td>113</td>
</tr>
<tr>
<td>-o ov_file</td>
<td>116</td>
</tr>
<tr>
<td>-prepps</td>
<td>115</td>
</tr>
<tr>
<td>-reorder</td>
<td>116</td>
</tr>
<tr>
<td>-rin on.off</td>
<td>116</td>
</tr>
<tr>
<td>-r</td>
<td>115</td>
</tr>
<tr>
<td>-sat_solver name</td>
<td>116</td>
</tr>
<tr>
<td>.nusmvrc</td>
<td>114</td>
</tr>
<tr>
<td>-AG</td>
<td>116</td>
</tr>
<tr>
<td>-bdd</td>
<td>116</td>
</tr>
<tr>
<td>-bmc_length_k</td>
<td>116</td>
</tr>
<tr>
<td>-coi</td>
<td>116</td>
</tr>
<tr>
<td>-cp</td>
<td>116</td>
</tr>
<tr>
<td>-ctt</td>
<td>115</td>
</tr>
<tr>
<td>-dct</td>
<td>114</td>
</tr>
<tr>
<td>-disable_dagifier</td>
<td>114</td>
</tr>
<tr>
<td>-disable_sexp2bdd_caching</td>
<td>116</td>
</tr>
<tr>
<td>-disable_syntactic_checks</td>
<td>114</td>
</tr>
<tr>
<td>-dynamic</td>
<td>116</td>
</tr>
<tr>
<td>-fit</td>
<td>115</td>
</tr>
<tr>
<td>-f</td>
<td>115</td>
</tr>
<tr>
<td>-help</td>
<td>114</td>
</tr>
<tr>
<td>-h</td>
<td>114</td>
</tr>
<tr>
<td>-ic</td>
<td>115</td>
</tr>
<tr>
<td>-iis</td>
<td>115</td>
</tr>
<tr>
<td>-is</td>
<td>115</td>
</tr>
<tr>
<td>-iwls95preorder</td>
<td>116</td>
</tr>
<tr>
<td>-iwls95_cp</td>
<td>116</td>
</tr>
<tr>
<td>-iwls95_cp_file</td>
<td>116</td>
</tr>
<tr>
<td>-keep_single_value_vars</td>
<td>114</td>
</tr>
<tr>
<td>-lp</td>
<td>115</td>
</tr>
<tr>
<td>-mono</td>
<td>116</td>
</tr>
<tr>
<td>-m_method</td>
<td>116</td>
</tr>
<tr>
<td>-noaffinity</td>
<td>110</td>
</tr>
<tr>
<td>-n_idx</td>
<td>115</td>
</tr>
<tr>
<td>-obm bm_file</td>
<td>115</td>
</tr>
<tr>
<td>-ofm fn_file</td>
<td>115</td>
</tr>
<tr>
<td>-ojeba algorithm</td>
<td>116</td>
</tr>
<tr>
<td>-old_div_op</td>
<td>116</td>
</tr>
<tr>
<td>-old</td>
<td>113</td>
</tr>
<tr>
<td>-o ov_file</td>
<td>116</td>
</tr>
<tr>
<td>-prepps</td>
<td>115</td>
</tr>
<tr>
<td>-reorder</td>
<td>116</td>
</tr>
<tr>
<td>-rin on.off</td>
<td>116</td>
</tr>
<tr>
<td>-r</td>
<td>115</td>
</tr>
<tr>
<td>-sat_solver name</td>
<td>116</td>
</tr>
</tbody>
</table>

### A

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>administration commands</td>
<td>104</td>
</tr>
</tbody>
</table>

### AND

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical and bitwise</td>
<td>13</td>
</tr>
</tbody>
</table>

### Array Define Declarations

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>array define declarations</td>
<td>26</td>
</tr>
</tbody>
</table>

### Array Variables

<table>
<thead>
<tr>
<th>Array Variables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>array type</td>
<td>7</td>
</tr>
</tbody>
</table>

### B

<table>
<thead>
<tr>
<th>Basic Next Expression</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic next expression</td>
<td>20</td>
</tr>
</tbody>
</table>

### Basic Trace Explainer

<table>
<thead>
<tr>
<th>Basic Trace Explainer</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Trace Explainer</td>
<td>97</td>
</tr>
</tbody>
</table>

### Batch

<table>
<thead>
<tr>
<th>Batch</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch, running NuSMV</td>
<td>113</td>
</tr>
</tbody>
</table>

### Boolean Type

<table>
<thead>
<tr>
<th>Boolean Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean type</td>
<td>6</td>
</tr>
</tbody>
</table>

### Bool Operator

<table>
<thead>
<tr>
<th>Bool Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool operator</td>
<td>21</td>
</tr>
</tbody>
</table>

### C

<table>
<thead>
<tr>
<th>Case Expressions</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>case expressions</td>
<td>19</td>
</tr>
</tbody>
</table>

### Commands for Bounded Model Checking

<table>
<thead>
<tr>
<th>Commands for Bounded Model Checking</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commands for Bounded Model Checking</td>
<td>68</td>
</tr>
</tbody>
</table>

### Commands for Checking PSL Specifications

<table>
<thead>
<tr>
<th>Commands for Checking PSL Specifications</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commands for checking PSL specifications</td>
<td>54</td>
</tr>
</tbody>
</table>

### Comments in NuSMV Language

<table>
<thead>
<tr>
<th>Comments in NuSMV Language</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>comments in NuSMV language</td>
<td>5</td>
</tr>
</tbody>
</table>