nuXmv 2.0.0 User Manual

Marco Bozzano, Roberto Cavada,
Alessandro Cimatti, Michele Dorigatti,
Alberto Griggio, Alessandro Mariotti,
Andrea Micheli, Sergio Mover,
Marco Roveri, Stefano Tonetta

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

Email: nuxmv@list.fbk.eu

This document is part of the distribution package of the NUXMV model checker.
For any additional request for information please send an e-mail to:

* nuxmv-users@list.fbk.eu for technical questions about the usage of the tool
* nuxmv@list.fbk.eu for non-technical issues like licensing, cooperation requests, etc..

Please report bugs through the nuXmv Bug Tracker at https://nuxmv.fbk.eu/bugs, and then click "Login Anonymously" to access. As an alternative (less preferred), you can send an email to nuxmv-users@list.fbk.eu.

Contents

Chapter 1

Introduction

nuXmv inherits, and thus provides to the user, all the functionalities of NuSMV [CCG+02]. In this section we revise all the new features distinguishing them in those for the analysis of finite-state domains, those for the analysis of infinite-state domains, and other generic features.

1.1 Analysis of finite-state domains

nuXmv complements the NuSMV language with the aiger 1.9.4 [BHW11] format. aiger 1.9.4 is the language adopted in the hardware model checking competition. Once the aiger 1.9.4 file is read, the internal data structures of nuXmv are populated, and it is possible to verify the properties (if any) with any of the available verification algorithms, or specify new properties interactively "playing" with the design.

nuXmv implements a vast portfolio of algorithms for invariant checking. We extended MiniSat [ES03] to build a resolution proof. This enables for the extraction of interpolants [McM04], and opens for the implementation of interpolation based algorithms. We currently provide an implementation for the McMilllan approach [McM03] and for the interpolation sequence approach [VG09]. Interpolation based algorithms are complemented with k-induction algorithms [SSS00] and a family of algorithms based on IC3 [Bra11, HBS13, VGS12]. The IC3 algorithm using abstraction refinement [VGS12] comes in two variant depending on the approach to refinement: the original one based on IC3, and a new variant based on BMC. All these techniques, benefit from the use of temporal decomposition [CMBK09] and from the techniques to discover equivalences to simplify the problem.

Still related to the verification of invariants, we also improve the BDD based invariant checking algorithms by allowing the user to specify hints in the spirit of guided reachability [TCP08]. The hints are specified using a restricted fragment of the PSL SERE [EF06]. The hints can also be used to compute the full set of the reachable states.

For LTL SAT based model checking, we complement the BMC based algorithms of NuSMV [BCCZ99b, BHJ+06] with k-liveness [CS12] integrated within an IC3 framework. K-liveness is based on counting and bounding the number of times a fairness constraint can become true. This is used in conjunction with the construction of a monitor for LTL properties, for which we use the LTL2SMV [CGH97b] as provided by NuSMV.

1.2 Analysis of infinite-state domains

In order to allow the user to specify infinite-state systems, we extend the language of NuSMV with two new data types, namely Reals and unbounded Integers. This, for instance, enables to specify domains with infinite data types (e.g. the example in Fig. 1.1).

To analyze such kind of designs, we integrate in nuXmv several new verification algorithms based on Satisfiability Modulo Theory (SMT) [BSST09] and on abstraction, or combination of abstraction with other techniques.

We lift Simple Bounded Model Checking (SBMC) [BHJ+06] from the pure Boolean case to the SMT case. The encoding is the same as that of SBMC, but instead of using a SAT solver we use an SMT solver. The SBMC SMT based approach for LTL verification is complemented with k-liveness combined with IC3 extended

```
1  MODULE main
2  IVAR
3    d : Real;
4  VAR
5    state : {s0, s1};
6    res   : Real;
7  ASSIGN
8    init(state) := s0;
9    next(state) := case
10     state = s0 & res >= 0.10 : s1;
11     state = s1 & res >= 0.20 : s0;
12     TRUE                     : state;
13   esac;
14   next(t) := case
15     state = s0 & res < 0.10 : res + d;
16     state = s1 & res < 0.20 : res + d;
17     TRUE                    : 0.0;
18   esac;
19 INIT
20   res >= 0.0
21 TRANS
22   (state = s0 -> (d >= 0 & d <= 0.01)) &
23   (state = s1 -> (d >= 0 & d <= 0.02))
24 INVARSPEC res <= 0.3;
```

Figure 1.1: Example of the NUXMV language.

to the infinite-state case [CGMT14b]. This approach relies on recent results on applying an IC3-based approach to the verification of infinite-state systems [CG12]. We remark that, although these approaches are in general incomplete, if a lazo-shaped counterexample exists, it is guaranteed to be eventually found. Moreover, for certain designs, these approaches are able to conclude that the property hold.

As far as invariant checking is concerned, we lift the pure Boolean approaches like BMC, k-induction, interpolation, and IC3, to the case of verification of infinite-state systems. Intuitively, we use an SMT solver in place of a SAT solver. For the infinite case, similar to the finite case, we provide an SMT based implementation for McMilllan approach [McM03]; for the interpolation sequence approach [VG09]; for k-induction [SSS00]; and for a family of algorithms based on IC3 [CG12, CGMT14a].

NUXMV also implements several approaches based on abstraction refinement [CGJ+03]. We provide new algorithms combining abstraction with BMC and k-induction [Ton09]. The algorithms do not rely on quantifier elimination techniques to compute the abstraction, but encode the model checking problem over the abstract state space into an SMT problem. The advantage, is that they avoid the possible bottleneck of abstraction computation. The very same approach has been recently lifted and tightly integrated within the IC3 framework [CGMT14a], with very good results. All these techniques complement the "classical" counterexample guided (predicate) abstraction refinement (CEGAR) [CGJ+03], also implemented in NUXMV. The CEGAR approach requires the computation of a quantifier-free formula that is equivalent to the abstract transition relation w.r.t. a given set of predicates. This, in turn, requires the solving of an ALLSAT problem [LNO06]. For this step, NUXMV implements different techniques: the combination of BDD and SMT [CCF+07, CFG+10], where BDDs are used as compact Boolean model enumerator within an ALLSMT approach; a technique that exploits the structure of the system under verification, by partitioning the abstraction problem into the combination of several smaller abstraction problems [CDJR09]. For the refinement step to discard the spurious counterexample, NUXMV implements three approaches based on the analysis of the unsatisfiable core, on the analysis of the interpolants, and on the weakest preconditions.

1.3 Miscellaneous functionalities

NUXMV provides novel functionalities that aim at facilitating the modeling and the understanding of complex designs. For instance, it allows for the generation of an explicit state representation (subject to the projection over a set of user specified predicates) in XMI format of the design under verification. The generated XMI can be visualized in any UML based viewer supporting the import from XMI.

LTL and invariant properties have been extended to allow for the use of input signals and next values of state variables. These extensions do not add any expressive power to the language, but facilitates the writing of properties from the user's point of view. Internally, each state formula containing a reference to an input or next signal is replaced by a corresponding monitor allowing the reuse of off-the-shelf verification engines.

NUXMV also provides several model transformation techniques aiming to reduce the state space of the design. It uses static analysis techniques to extract possible values for variables, and then re-encode the design using such

information (e.g. using a bit-vector at 32 bit to store 2 values can be re-encoded with just one Boolean variable). These techniques are complemented with others aiming at simplifying the model through constants and free inputs propagation [AFF+07].

Finally, in NUXMV we remove the NUSMV limitation to have bit vectors with less than 64 bits only.

1.4 Differences with NUSMV

In this section we summarize the main differences at user level with NUSMV.

As far as the input language of NUXMV is concerned, we introduce two new types for state and input variables. Namely, `real` and `integer` (See sections 2.1.5, 2.1.4, and 2.3.1 for details). This enables the user to model the specification of infinite-state transition systems. We add new constructs for specifying the predicates to use in the predicate abstraction based techniques (See sections 2.3.18 and 5.7 for details). NUXMV does not support anymore the keyword `process`. Indeed, the NUXMV is targeting only finite- and infinite-state synchronous fair transition systems.

We provide the user with all the interactive commands provided by NUSMV (See chapter 4), and we complement them with a set of new commands to use the new features provided by NUXMV (See chapter 5 for a detailed list of the new functionalities provided). In particular, the new commands allows to use the new model checking algorithms for finite-state transition systems, and the new algorithms based on SMT for infinite-state transition systems.

Structure of this document

This document is structured as follows. First in chapter 2 we describe the syntax and the semantics of the input language of the NUXMV. In chapter 3 we describe how to execute the NUXMV in interactive mode. In chapter 4 we describe the interactive commands inherited from NUSMV, while in chapter 5 we describe the new commands of NUXMV. In chapter 7 we describe the command line switches to execute NUXMV in batch mode (only for finite-state domains).

Remark

This document is in continuous evolution to better document the features provided by NUXMV.

Chapter 2

Input Language of NUXMV

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

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

In the following, an `identifier` may be any sequence of characters starting with a character in the set {**`A-Za-z_`**} and followed by a possibly empty sequence of characters belonging to the set {**`A-Za-z0-9_$#-`**}. All characters and case in an identifier are significant. Whitespace characters are space (<SPACE>), tab (<TAB>) and newline (<RET>). Any string starting with two dashes ('`--`') and ending with a newline is a comment and ignored by the parser. The 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
        A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
        a b c d e f g h i j k l m n o p q r s t u v w x y z _

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

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

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

`@F~`, **`@O~`**, **`A`**, **`ABF`**, **`ABG`**, **`abs`**, **`AF`**, **`AG`**, **`array`**, **`ASSIGN`**, **`at next`**, **`at last`**, **`AX`**, **`bool`**, **`boolean`**, **`BU`**, **`case`**, **`Clock`**, **`clock`**, **`COMPASSION`**, **`COMPID`**, **`COMPUTE`**, **`COMPWFF`**, **`CONSTANTS`**, **`CONSTARRAY`**,CONSTRAINT, **`cos`**, **`count`**, **`CTLSPEC`**, **`CTLWFF`**, **`DEFINE`**, **`E`**, **`EBF`**, **`EBG`**, **`EF`**, **`EG`**, **`esac`**, **`EX`**, **`exp`**, **`extend`**, **`F`**, **`FAIRNESS`**, **`FALSE`**, **`floor`**, **`FROZENVAR`**, **`FUN`**, **`G`**, **`H`**, **`IN`**, **`in`**, **`INIT`**, **`init`**, **`Integer`**, **`integer`**, **`INVAR`**, **`INVARSPEC`**, **`ISA`**, **`ITYPE`**, **`IVAR`**, **`JUSTICE`**, **`ln`**, **`LTLSPEC`**, **`LTLWFF`**, **`MAX`**, **`max`**, **`MDEFINE`**, **`MIN`**, **`min`**, **`MIRROR`**, **`mod`**, **`MODULE`**, **`NAME`**, **`next`**, **`NEXTWFF`**, **`noncontinuous`**, **`O`**, **`of`**, **`PRED`**, **`PREDICATES`**, **`pi`**, **`pow`**, **`PSLSPEC`**, **`PARSYNTH`**, **`READ`**, **`Real`**, **`real`**, **`resize`**, **`S`**, **`SAT`**, **`self`**, **`signed`**, **`SIMPWFF`**, **`sin`**, **`sizeof`**, **`SPEC`**, **`swconst`**, **`T`**, **`tan`**, **`time`**, **`time_since`**, **`time_until`**, **`toint`**, **`TRANS`**, **`TRUE`**, **`typeof`**, **`U`**, **`union`**, **`unsigned`**, **`URGENT`**, **`uwconst`**, **`V`**, **`VALID`**, **`VAR`**, **`Word`**, **`word`**, **`word1`**, **`WRITE`**, **`X`**, **`xnor`**, **`xor`**, **`X~`** **`Y`**, **`Y~`**, **`Z`**

Note: NUXMV does no longer support the keyword **`process`** as only synchronous systems are supported.
Note: *Clock*, *clock* and *time* are reserved keywords only in TTS (2.4).

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

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

and `symbolic constants` which are `identifiers`

```
  symbolic_constant :: identifier
```

Examples of `integer numbers` and `symbolic constants` are `3, -14, 007, OK, FAIL, waiting, stop`. The values of `symbolic constants` and `integer numbers` do not intersect.

2.1 Types Overview

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

2.1.1 Boolean

The boolean type comprises symbolic values **`FALSE`** and **`TRUE`**.

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

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

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

Another enumeration type is integer enum. Example of enumerations of integers are {`2, 4, -2, 0`} and {`-1, 1`}. In the NUXMV type system an expression of the type integer enum is always converted to the type integer. For 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.

2.1.3 Word

The unsigned word[•] and signed word[•] 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 unsigned word[3] represents vector of three bits, which allows unsigned operations, and type signed word[7] represents vector of seven bits, which allows signed operations.

When values of 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 unsigned word[N] is able to represent values from 0 to $2^N - 1$.

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

2.1.4 Integer

The domain of the integer type is any Whole Number, positive or negative.

Although the integer type is used to represent integer enum type when explaining the NUXMV type system, there are important differences which are needed to keep in mind. First, using integer is not allowed in certain Model Checking engines and algorithms. Second, at the moment, there are implementation-dependent constraints on the integer enum type, as `integer numbers` can only be in the range $-2^{32}+1$ to $2^{32}-1$ (more accurately, these values are equivalent to the C/C++ macros `INT_MIN` +1 and `INT_MAX`).

2.1.5 Real

The domain of the real type is the Rational Numbers.

2.1.6 Clock

The clock type is available only in TTS (2.4). The domain of this type depends on the time domain of the module:

- `continuous` time domain: clock type domain is equivalent to the real type domain;
- `none` time domain: clock type can not be expressed.

2.1.7 Array

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

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

The type 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 array `1..8 of` array `-1..2 of` unsigned word[5] is array `-1..2 of` unsigned word[5] which has its own subtype 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 **`DEFINE`** (see 2.3.3) or variables of array type (see 2.3.1).

Internally, these arrays are treated as a set of variables. See next subsections for other kinds of arrays.

2.1.8 WordArray

The word-array types are used to model arrays whose size is bounded and is specified with unsigned word[•] type. Elements of the array can be of some type. For example,

```
array word[5] of unsigned word[3];
array word[4] of real;
```

The type `array word[4] of word[9]` means an array of 16 elements (from 0d4_0 to 0b4_15), each of which is unsigned word[9]. word-array types are distinguishable on their size and element type. Note also that the size has to be greater than zero.

word-array are very specific type and very few operators can be applied to expressions of these types. See also **READ**, **WRITE**, **CONSTARRAY**, **:=** , and **=** operators.

2.1.9 IntArray

The int-array types are used to model arrays whose size is unbounded. Similar to word-array, the elements of the int-array can be of any type. For example,

```
array integer of integer;
array integer of unsigned word[8];
```

The type `array integer of integer` means an unbounded array with integer type indices and integer type elements. int-array types are distinguishable on their element type.

int-array are very specific type and very few operators can be applied to expressions of these types. See also **READ**, **WRITE**, **CONSTARRAY**, **:=** , and **=** operators.

2.1.10 Set Types

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

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

the counterpart of a boolean set type is boolean,

the counterpart of a integer set type is integer,

the counterpart of a symbolic set type is symbolic enum,

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

Some types such as unsigned word[•], signed word[•], and real do not have a set type counterpart.

2.1.11 Type Order

Figure 2.1 depicts the order existing between types in NUXMV.
It means, for example, that integer is less than integers-and-symbolic enum and less than real; symbolic enum is less than integers-and-symbolic enum, etc. The unsigned word[•] and signed word[•] types are not comparable with any other type or between each other. Any type is equal to itself.

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

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

2.2 Expressions

In NUXMV all expressions are typed and there are constraints on the type of operands. An expression that violates the type system will be considered erroneous, and will raise a type error.

To maintain backward compatibility with old versions of NUSMV, there is a system variable called `backward_compatibility` (and a correponding `-old` command line option) that disables a few new features of NUSMV 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 the NUSMV user manual [CCCJ+10] for further information.

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

[1] For more information on these operators see the NUSMV user manual [CCCJ+10].

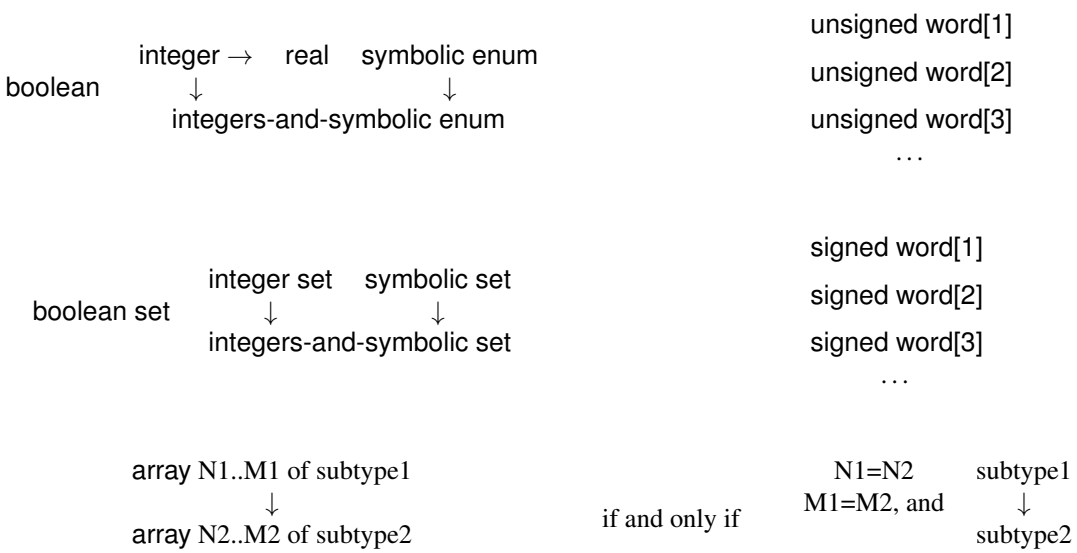

Figure 2.1: The ordering on the types in NUXMV

2.2.1 Implicit Type Conversion

In some expressions operands may be converted from one type to its set type counterpart (see 2.1.10). For example, integer can be converted to integer set type.

clock is implicitly converted to real.

Note: In old version of NUSMV, implicit type conversion from integer to boolean (and vice-versa) was performed. Since NUSMV version 2.5.1, and thus also in NUXMV, implicit integer <-> 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, real, symbolic, word or range constant.

```
constant ::
        boolean_constant
      | symbolic_constant
      | integer_constant
      | real_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
```

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 36 for more information about how `symbolic constants` are distinguished from other `identifiers`, i.e. variables, defines, etc.

Integer Constant

An `integer constant` is an `integer number`. The type of an `integer constant` is integer.

```
integer_constant :: integer_number
```

Real Constant

A `real constant` is an `real number`. The type of a `real constant` is real.

```
real_constant :: real_number
```

Definition of `real number` allows for different representations, namely floating point, fractional and exponential. Some examples:

float	`123.456`
fractional	`F'123/456`
fractional	`f'123/456`
exponential	`123e4`
exponential	`123.456e7`
exponential	`123.456E7`
exponential	`123.456E-7`

Word Constant

`Word constant` begins with digit `0`, followed by optional character `u` (unsigned) or `s` (signed) and one of the characters `b`/`B` (binary), `o`/`O` (octal), `d`/`D` (decimal) or `h`/`H` (hexadecimal) which gives the base that the actual constant is in. Next comes an optional decimal integer giving the number of bits, then the character `_`, and lastly the constant value itself. 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_b
```

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_010111111100`.
- 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 no longer the implementation limit of being 64 bits at most. In NUXMV it is possible to define words of arbitrary size.

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 NUXMV (as it is also the case in NUSMV).

```
basic_expr ::
      constant                        -- a constant
    | variable_identifier             -- a variable identifier
    | define_identifier               -- a define identifier
    | function_call                   -- a call to a function
    | ( basic_expr )
    | pi  -- the pi constant
    | abs ( basic_expr ) -- absolute value
    | max ( basic_expr , basic_expr ) -- max
    | min ( basic_expr , basic_expr ) -- min
    | sin ( basic_expr ) -- sin
    | cos ( basic_expr ) -- cos
    | exp ( basic_expr ) -- exp
    | tan ( basic_expr ) -- tan
    | ln ( basic_expr )  -- ln
    | ! basic_expr                    -- logical or bitwise NOT
    | basic_expr & basic_expr         -- logical or bitwise AND
```

```
| basic_expr | basic_expr        -- logical or bitwise OR
| basic_expr xor basic_expr      -- logical or bitwise exclusive OR
| basic_expr xnor basic_expr     -- logical or bitwise NOT exclusive OR
| basic_expr -> basic_expr       -- logical or bitwise implication
| basic_expr <-> basic_expr      -- logical or bitwise equivalence
| basic_expr = basic_expr        -- equality
| basic_expr != basic_expr       -- inequality
| basic_expr < basic_expr        -- less than
| basic_expr > basic_expr        -- greater than
| basic_expr <= basic_expr       -- less than or equal
| basic_expr >= basic_expr       -- greater than or equal
| - basic_expr                   -- integer or real or word unary minus
| basic_expr + basic_expr        -- integer or real or word addition
| basic_expr - basic_expr        -- integer or real or word subtraction
| basic_expr * basic_expr        -- integer or real or word multiplication
| basic_expr / basic_expr        -- integer or real or word division
| basic_expr mod basic_expr      -- integer or word 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_list )      -- count of true boolean expressions
| swconst ( basic_expr , 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
| floor ( basic_expr  )          -- from a real to an integer
| extend ( basic_expr , basic_expr)
                                 -- word width extension
| resize ( basic_expr , basic_expr)
                                 -- word width resize
| signed word[N] ( basic_expr )  -- integer to signed word conversion
| unsigned word[N] ( basic_expr ) -- integer to unsigned word conversion
| basic_expr union basic_expr    -- union of set expressions
| { set_body_expr }              -- set expression
| basic_expr in basic_expr       -- inclusion in a set expression
| basic_expr ? basic_expr : basic_expr
                                 -- if-then-else expression

| READ ( basic_expr , basic_expr ) -- read function with first argument
-- an array and second index
| WRITE ( basic_expr, basic_expr, basic_expr ) -- write function with first
-- argument an array, second index, and third value to be stored
| CONSTARRAY ( typeof ( variable_identifer ), basic_expr ) -- constant array
-- constructor function that takes the type of the array variable indentifier
| CONSTARRAY ( array word[n] of subtype, basic_expr ) -- constant array
-- constructor function for word-array that takes the array type explicitly
| CONSTARRAY ( array integer of subtype, basic_expr ) -- constant array
-- constructor function for int-array that takes the array type explicitly

| case_expr                      -- case expression
```

```
    | basic_next_expr                -- next expression

basic_expr_list ::
      basic_expr
    | basic_expr_list , basic_expr
```

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 11.

Variables and Defines

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

```
define_identifier :: complex_identifier

variable_identifier :: complex_identifier
```

The syntax and semantics of `complex_identifiers` are explained in Section 2.3.13 [References to Module Components], page 35. 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 29 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.

`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 25, Section 2.3.1 [State Variables], page 26, Section 2.3.1 [Input Variables], page 26, and Section 2.3.1 [Frozen Variables], page 27. Since a `symbolic constant` is syntactically indistinguishable from `variable_identifiers` and `define_identifiers`, a symbol table is used to distinguish them from each other.

Function calls

A `function_call` is a term which identify an uninterpreted function call. The syntax for function calls is:

```
function_call :: function_identifier ( fun_args_list )
function_identifier :: complex_identifier
fun_args_list :: next_expr | fun_args_list next_expr
```

`complex_identifiers` are explained in Section 2.3.13 [References to Module Components], page 35. The syntax for `next_expr` is explained in Section 2.2.4 [Simple and Next Expressions], page 23.

The type of a function is specified in its declaration. For more information about functions, see Section 2.3.5 [Function Declaration], page 30.

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 NUXMV 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
: integer * real → boolean
: real * integer → boolean
: clock * clock → boolean
: clock * integer → boolean
: integer * clock → boolean
: clock * real → boolean
: real * clock → 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
: array word[N] of subtype * array word[N] of subtype → boolean
: array integer of subtype * array integer of subtype → 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
: integer * real → boolean
: real * integer → boolean
: clock * clock → boolean
: clock * integer → boolean
: integer * clock → boolean
: clock * real → boolean
: real * clock → 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
: integer * real → real
: real * integer → real
: clock * clock → clock
: clock * integer → real
: integer * clock → real
: clock * real → real
: real * clock → real
: unsigned word[N] * unsigned word[N] → unsigned word[N]
: signed word[N] * signed word[N] → signed word[N]

− (unary) : integer → integer
: real → real
: clock → clock
: unsigned word[N] → unsigned word[N]
: 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. For operands of type integer, 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 NUXMV. The semantics is equivalent to the corresponding one of C/C++ languages. For operands of type real, the result of the **/** operator is the algebraic quotient where the fractional part (if any) is kept.

Similarly to NuSMV, we adopt this new semantics for the division / operator. We refer to the NuSMV user manual [CCCJ+10] for more details on this respect.

Remainder Operator `mod`

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

The signature of the remainder operator is:

`mod` : integer * integer → integer
: unsigned word[N] * unsigned word[N] → unsigned word[N]
: signed word[N] * signed word[N] → signed word[N]
: clock * clock → clock
: clock * integer → real
: integer * clock → real

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

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

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

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

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

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

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

whereas in the old semantics the equations were

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

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

Shift Operators `<<`, `>>`

The signature of the shift operators is:

`<<`, `>>` : unsigned word[N] * integer → unsigned word[N]
: signed word[N] * integer → signed word[N]
: unsigned word[N] * unsigned word[M] → unsigned word[N]
: signed word[N] * unsigned word[M] → 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 **`<<`** (right shift **`>>`**) 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 signed word[•], in which case a word is padded with its highest bit. For instance,

0ub4_0101 **<<** 2 is equal to	0sb4_1011 **>>** 2 is equal to
0ub4_0100 **<<** 1 is equal to	0sb4_1110 **>>** 1 is equal to
0ub4_1000 **<<** 0 is equal to	0sb4_1111 **>>** 0 is equal to
0ub4_1000 and	0sb4_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. NUXMV 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 must 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
: array N..M of subtype * integer → subtype

For example, for below declarations [2] :

```
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 raise an 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_h * integer_l → unsigned word[$\text{integer}_h - \text{integer}_l + 1$]
: signed word[N] * integer_h * integer_l → unsigned word[$\text{integer}_h - \text{integer}_l + 1$]

where $0 \leq \text{integer}_l \leq \text{integer}_h <$ 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]

where word[N] is unsigned word[N] or signed 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 word[•]. For example, given the two words `w1 := 0ub4_1101` and `w2 := 0sb2_00`, the result of `w1::w2` is `0ub6_110100`.

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.

[2] See 2.3.3) for array defines and 2.3.1 for array variables.

The signature of the operator is:

extend : unsigned word[N] * integer → unsigned word[N+integer]
: signed word[N] * integer → signed word[N+integer]

For example:

extend(0ub3_101, 2) = 0ub5_00101
extend(0sb3_101, 2) = 0sb5_11101
extend(0sb3_011, 2) = 0sb5_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 word of a width. The behavior of this operator can be described as follows:

Let w be a M bits unsigned 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 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:

resize : unsigned word[•] * integer → unsigned word[integer]
: signed word[•] * integer → signed word[integer]

Set Expressions

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

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

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

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

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

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

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

[3] See 2.1.10 for more information about the set types and their counterpart types

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

Set expressions can be used only as operands of **union** and **in** operations, 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:

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

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

READ Expressions

The read operator '**READ**' extracts one element of an array at particular index. The first argument of the operator must be an expression of type either word-array or int-array, and the type of second argrument expression must be same as of the index type of the array expression in the first argument. The signature of the **READ** is:

READ : array word[N] of subtype * word[N] → subtype
: array integer of subtype * integer → subtype

WRITE Expressions

The write operator '**WRITE**' updates one element at a particular index of an array and returns the updated array as a new array. The first argument of the operator must be an expression of type either word-array or int-array. The type of the second and third argument expressions must be same as of the index type and element type of the array expression in the first argument. The signature of the **WRITE** is:

WRITE : array word[N] of subtype * word[N] * subtype → array word[N] of subtype
: array integer of subtype * integer * subtype → array integer of subtype

CONSTARRAY Expressions

The constant array '**CONSTARRAY**' is a special constructor to create an array of given type having elements set to a uniform given value. The signature of **CONSTARRAY** is the following:

[4] See 2.1.10 for more information about the set types and their counterpart types

CONSTARRAY : (array integer of subtype) * boolean → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * integer → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * real → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * symbolic enum → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * integers-and-symbolic enum → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * unsigned word[N] → (array integer of subtype)
CONSTARRAY : (array integer of subtype) * signed word[N] → (array integer of subtype)
CONSTARRAY : (array word[N] of subtype) * boolean → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * integer → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * real → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * symbolic enum → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * integers-and-symbolic enum → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * unsigned word[N] → (array word[N] of subtype)
CONSTARRAY : (array word[N] of subtype) * signed word[N] → (array word[N] of subtype)

For example, a constant array `CONSTARRAY(typeof(a), 0)` (suppose that `a` is of type `array integer of integer`), means an int-array of type `array integer of integer` with all elements value set to `0`.

Typeof Expressions

The typeof expression is specifically used as a first argument in the constant array expressions. Basically it is used to get the type of an array variable. The typeof expression has the following syntax:

```
typeof_expr ::  typeof (variable_identifier)
```

Case Expressions

A case expression has the following syntax:

```
case_expr :: case case_body esac

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

A `case_expr` returns the value of the first expression on the right hand side of '`:`', such that the corresponding condition on the left hand side evaluates to `TRUE`. For example, the result of the expression

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

is `right_expression_k` such that for all i from 0 to $k-1$, `left_expression_i` is `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 [5]. The type of the whole expression is such a minimal type[6] that all of the expressions on the right (after possible convertion to set types) can be implicitly converted to this type. If this is not possible, NUXMV throws an error.

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

[5] See 2.1.10 for information on set types and their counterpart types
[6] See Section 2.1.11 [Type Order], page 10 for the information on the order of types.

```
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.

If-Then-Else expressions

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

```
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 ? expr1 : expr2` and `case cond1 : expr1; TRUE : expr2; esac` are equivalent.

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

The syntactic rule is:

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

A `next expression` does not change the type.

Count Operator

The **`count`** operator counts the number of expressions which are true. The **`count`** operator is a syntactic sugar for

```
toint (bool_expr1) +

toint (bool_expr2) +

... +

toint (bool_exprN)
```

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 + ...  + bN) < 3 -- at most two bits are enabled
```

Since version 2.5.1, this expression can be written as:

```
count(b0, b1, ...  , 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 **`next`** operation inside and the syntax of `simple_expressions` is as follows:

```
simple_expr :: basic_expr
```

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

```
next_expr :: basic_expr
```

with the alternative `basic_next_expr` allowed.

2.2.5 Type conversion operators

Integer conversion operator

`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:

`toint` : integer → integer
`toint` : boolean → integer
`toint` : unsigned word[•] → integer
`toint` : signed word[•] → integer

`floor` convertion operator

The operator **`floor`** maps a real number to the largest previous integer. If applied to an integer it returns the integer itself. It has the following signature:

`floor` : integer → integer
: real → integer

Boolean conversion operator

`bool` converts unsigned word[1] and any expression of type integer (e.g. 1 + 2) to boolean. Also boolean expressions are allowed, but no action is perfomed. 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:

`bool`(0ub1_0) = FALSE
`bool`(0ub1_1) = TRUE

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:

`swconst` : integer * integer → signed word[•]
`uwconst` : integer * integer → unsigned word[•]

Word1 Explicit Conversions

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

`word1` : boolean → unsigned word[1]

The conversion obeys the following table:

`word1`(FALSE) = 0ub1_0
`word1`(TRUE) = 0ub1_1

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:

unsigned : signed word[N] → unsigned word[N]
signed : unsigned word[N] → signed word[N]

For example:

signed(0ub_101) = 0sb_101
signed(0ud3_5) = -0sd3_3
unsigned(0sb_101) = 0usb_101
unsigned(-0sd3_3) = 0ud3_5

General Integer to Word Conversions

unsigned word[N] converts an integer expression to an unsigned word[N], and **signed word[N]** converts an integer to signed word[N].

The signatures of these conversion operators are:

unsigned word[N] : integer → unsigned word[N]
signed word[N] : integer → signed word[N]

The semantics of **unsigned word[N]**(x) when x is *non-negative* is given by the following relation:

$$\exists y \geq 0.(x = \sum_{i=0}^{N}((\textbf{unsigned word[N]}(x)[i:i] = 0ub\_1)\ ?\ 2^i : 0) + 2^N * y).$$

When x is negative, **unsigned word[N]**$(x) = -(\textbf{unsigned word[N]}(-x))$. Finally, **signed word[N]**$(x) = \textbf{signed}(\textbf{unsigned word[N]}(x))$.

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 NUXMV 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:

```
type_specifier ::
        simple_type_specifier
      | module_type_specifier

simple_type_specifier ::
        boolean
```

```
      | word [ basic_expr ]
      | unsigned word [ basic_expr ]
      | signed word [ basic_expr ]
      | real
      | integer
      | { enumeration_type_body }
      | basic_expr .. basic_expr
      | array basic_expr .. basic_expr
          of simple_type_specifier

      | array word [ basic_expr ]
          of simple_type_specifier
      | array integer
          of simple_type_specifier


enumeration_type_body ::
        enumeration_type_value
      | enumeration_type_body , enumeration_type_value

enumeration_type_value ::
        symbolic_constant
      | integer_number
```

There are two kinds of `type specifier`: a `simple type specifier` and a `module type specifier`. The `module type specifier` is explained later in Section 2.3.12 [MODULE Instantiations], page 34. The `simple type specifier` comprises boolean type, integer type, enumeration types, unsigned word[•], signed word[•] and arrays types.

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:

```
var_declaration :: VAR var_list

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

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

Input Variables

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

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


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

- `INIT` statements. For example:

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

- Scope of `next` expressions. For example:

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

- Some specification kinds: `CTLSPEC`, `SPEC`, `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
INVARSPEC (i -> s) – this is allowed
```

Frozen Variables

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

```
ASSIGN next(fv) := fv;
```

As a consequence, frozen variables may not have their current and next value set in an `ASSIGN` statement, i.e. statements such as `ASSIGN next(fv) := expr;` and `ASSIGN fv := 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 typeunsigned word 3-bits-wide (i.e. unsigned word[3]), and `i` is an signed word 4-bits-wide (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 :: complex_identifier := next_expr ;
             | define_body identifier := next_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 `complex_identifier` occurs in an expression, the `complex_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 `complex_identifier` is declared (see Section 2.3.16 [Context], page 37 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 complex_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 `complex_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 `complex_identifier` occurs in an expression, the `complex_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 `complex_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 expecially useful in those conditions that require symbolic constants to occur only in **DEFINEs** body (e.g. in generated models). For an example of usage see also the command `write_boolean_model`. A constant is allowed to be declared multiple times, as after the first declaration any further declaration will be ignored. **CONSTANTS** declarations are an extension of the original SMV grammar. They have been integrated in NuSMV and inherited in nuXmv. The syntax for this kind of declaration is:

```
constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier
             | constants_body  , identifier
```

2.3.5 Function Declaration

In NUXMV it is also possible to define uninterpreted functions. These functions are rigid, i.e. their denotation does not change from two different time points. These functions can be seen as parameters: the denotation of the function is defined in the initial state and kept from that point on. The syntax for declaring functions is:

```
function_declaration :: FUN function_list

function_list :: function_declaration
                 | function_list function_declaration

function_declaration :: identifier : function_type_specifier ;
function_type_specifier :: function_args_type_specifier -> simple_type_specifier

function_args_type_specifier :: simple_type_specifier
                 | function_args_type_specifier * simple_type_specifier
```

below is reported a simple example of declaring a function `funct1` that takes two reals as arguments, and returns an integer, and a function `funct2` that takes two reals and returns an unsigned word of size 32.

```
FUN
    funct1 : real * real -> integer ;

    funct2 : real * real -> unsigned word[32] ;
```

Note: Currently in NUXMV we only support a limited number of data types both as return type and as type of each argument of a function. In particular, the supported types are: `boolean`, `real`, `integer`, and `word[N]`. Support for richer types (e.g. enumeratives, bouded integers and array) is ongoing.

2.3.6 `INIT` Constraint

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

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

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

2.3.7 `INVAR` Constraint

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

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

Since the expressions in the `INVAR` constraint are `simple_expression` s, they cannot contain the **`next()`** operator. If there is more than one `INVAR` constraint, the invariant set is the conjunction of all of the `INVAR` constraints.

2.3.8 TRANS Constraint

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

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

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

2.3.9 ASSIGN Constraint

An assignment has the form:

```
assign_constraint :: ASSIGN assign_list

assign_list :: assign ;
             | assign_list assign ;

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

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

Semantically assignments can be expressed using other kinds of constraints:

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

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

The allowed types of the assignment operator are:

`:=` : integer * integer
: real * integer
: real * real
: integer * integer set
: symbolic enum * symbolic enum
: symbolic enum * symbolic set
: integers-and-symbolic enum * integers-and-symbolic enum
: integers-and-symbolic enum * integers-and-symbolic set
: unsigned word[N] * unsigned word[N]
: signed word[N] * signed word[N]
: array word[N] of subtype * array word[N] of subtype
: array integer of subtype * array integer of subtype

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 (i.e. by a **next**, see examples below).

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 **:=** $expr_1$;
Example 2	**init(** x **) :=** $expr_1$;
Example 3	**next(** x **) :=** $expr_1$;
Example 4	**init(** x **) :=** $expr_1$; **next(** x **) :=** $expr_2$;

while the following are illegal assignments:

Example 1	x **:=** $expr_1$; x **:=** $expr_2$;
Example 2	**init(** x **) :=** $expr_1$; **init(** x **) :=** $expr_2$;
Example 3	x **:=** $expr_1$; **init(** x **) :=** $expr_2$;
Example 4	x **:=** $expr_1$; **next(** x **) :=** $expr_2$;

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

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

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

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

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

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

2.3.10 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.

NUXMV (as well as NUSMV) supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula `f`, which is assumed to be true infinitely often in all the fair paths. In NUXMV, justice constraints are identified by keywords **`JUSTICE`** and, for backward compatibility, **`FAIRNESS`**. A compassion constraint consists of a pair of formulas `(p,q)`; if property `p` is true infinitely often in a fair path, then also formula `q` has to be true infinitely often in the fair path. In NUXMV, compassion constraints are identified by keyword **`COMPASSION`**. [7]

Note: If compassion constraints are used, then the model must not contain any input variable. Currently, NUXMV does not enforce this so it is the responsibility of the user to make sure that this is the case.

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

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

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

2.3.11 MODULE Declarations

A module declaration is an encapsulated collection of declarations, constraints and specifications. A module declaration also opens a new identifier scope. Once defined, a module can be reused as many times as necessary. Modules are used in such a way that each instance of a module refers to different data structures. A module can contain instances of other modules, allowing a structural hierarchy to be built. The syntax of a module declaration is as follows:

```
module :: MODULE identifier [( module_parameters )] [module_body]

module_parameters ::
          identifier
        | module_parameters , identifier

module_body ::
          module_element
        | module_body module_element

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

[7] Similarly to NUSMV, in the current version of NUXMV, 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 forthcoming releases of NUXMV.

```
        | isa_declaration
        | pred_declaration
        | mirror_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.

2.3.12 `MODULE` Instantiations

An *instance* of a module is created using the **`VAR`** declaration (see Section 2.3.1 [State Variables], page 26) with a module type specifier (see Section 2.3.1 [Type Specifiers], page 25). The syntax of a `module type specifier` is:

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

parameter_list ::
        next_expr
      | parameter_list , next_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 23). It is an error if the number of actual parameters is different from the number of formal parameters. Whenever formal parameters occur in expressions within the module, they are replaced by the actual parameters. The semantic of module instantiation is similar to call-by-reference.[8]

Here are examples:

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

```
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.

Note: NUXMV does no longer support the keyword **`process`**.

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

2.3.13 References to Module Components (Variables and Defines)

As described in Section 2.2.3 [Variables and Defines], page 15, defines and variables can be referenced in expressions as `variable_identifiers` and `define_identifiers` respectively, both of which are `complex identifiers`. The syntax of a `complex identifier` is:

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

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

Notations with `.` (`<DOT>`) are used to access the components of modules. For example, if `m` is an instance of a module (see Section 2.3.12 [MODULE Instantiations], page 34 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`.

2.3.14 A Program and the `main` Module

The syntax of a NUXMV program is:

```
program :: module_list

module_list ::
          module
        | module_list module
```

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

2.3.15 Namespaces and Constraints on Declarations

Identifiers in the NUXMV 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 2.3.12 [MODULE Instantiations], page 34). 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 NUXMV 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 complex_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:

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

MODULE foo
  VAR c : moo;

MODULE moo
  VAR d : boolean;
```

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

2.3.17 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. Similarly to NUSMV, in NUXMV it is possible to declare the common parts as separate modules, and then use the **ISA** declaration to import the common parts inside a module declaration. The syntax of an isa_declaration is as follows:

```
isa_declaration :: ISA 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 NUXMV. Therefore, avoid the use of ISA_declarations. Use module instances instead.

2.3.18 PRED and MIRROR Declarations

When using abstraction-based techniques, such as Counterexample Guided Abstraction Refinement [CGJ+03] or k-induction with implicit abstraction [Ton09], one may want to declare an initial set of predicates to be used in the model. This is possible by using the keyword **PRED**.

The syntax of a pred_declaration is as follows:

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

```
pred_declaration :: PRED simple_expression [;]
                  | PRED < identifier > := simple_expression [;]
```

where `identifier` is an arbitrary name to be assigned to the predicate.

Another way to specify the predicates to be used in the abstraction-based techniques model consists in "mirroring", i.e. preserving in the abstract space, a variable. A mirrored variable with its type will be declared in the abstract model as it was declated in the concrete model. Mirrored variables are introduced with the keyword **MIRROR**. A `mirror_declaration` is as follows.

```
mirror_declaration :: MIRROR variable_identifier [;]
```

2.4 Definition of the Timed Transition System

The language used to describe FSM in NUXMV, showed in section 2.3, is extended to represent Timed Transition Systems (TTS). The commands to perform actions on these models are enabled by the command line option `-time`.

2.4.1 TIME_DOMAIN Annotation

The time domain annotation is optional and, if present, it must appear only once in the model and it must precede the MODULE declarations.

```
time_domain_annotation ::
       @TIME_DOMAIN [none | continuous ]
```

If the time domain annotation is absent the model is equivalent to one with time domain `none`. If the time domain is `none` all language extensions of TTS are not available.

2.4.2 Variable Declarations

This section shows the additional features available for variable declarations in TTS, with respect to the ones presented in 2.3.1.

Type Specifiers

In TTS an additional `simple type specifier` is available for input and state variables: clock. All symbols of this type increment of the same amount during time elapses and can not be used in specifications.

State Variables

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

In time elapse transitions all variables with type different from clock keep the same assignment.

In TTS a built-in clock state variable **time** is available. This variable represents the total amount of time elapsed. **time** can be used only in expression belonging to the following grammar:

```
op :: < | <= | = | != | >= | >
timed_expr ::
            time op untimed_expr
            time @F~ expr op untimed_expr
            time @O~ expr op untimed_expr
```

where `untimed_expr` is a **real** or **integer** type expression containing neither **time**, nor **time_until**, nor **time_since**.

2.4.3 `INVAR` Constraint

TTS supports the same invariant constraint described in 2.3.7 for variables of type different from clock. Variables of type clock are allowed to occur only in the rightmost `simple_expression` of the form below. Furthermore, this expression must be *convex* (i.e. a conjunction of atoms).

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

2.4.4 `URGENT` Constraint

The urgent constraints specify a set of states where time elapses are not allowed. The syntax of a **`URGENT`** constraint is:

```
urgent_constraint :: URGENT  next_expr [;]
```

The expression in the **`URGENT`** constraint can not contain variables of type clock. When the specified `next_expression` holds time does not elapse. If there is more than one **`URGENT`** constraint, the urgent set is the disjunction of all of the **`URGENT`** constraints.

2.4.5 `TRANS` Constraint

In TTS **`TRANS`** constraint have the same syntax described in 2.3.8. However, they constrain only the discrete transitions of the system.

2.4.6 `ASSIGN` Constraint

In TTS **`ASSIGN`** constraint have the same syntax described in 2.3.9. However, assignments that predicate over next states constrain only the discrete transitions of the system and not time elapses.

The following types of assignment operators are added:

`:=`	: clock * clock
	: clock * integer
	: clock * real [if time domain is continuous]
	: real * clock [if time domain is continuous]

2.4.7 `MODULE` Declarations

TTS add to the **`MODULE`** declarations showed in 2.3.11, the declaration of **`URGENT`** constraints.

2.5 Specifications

The specifications to be checked on the FSM can be expressed in temporal logics like Computation Tree Logic (CTL), Linear Temporal Logic (LTL)extended with Past Operators, and Property Specification Language (PSL) [psl03] that includes CTL and LTL with Sequential 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 NUXMV in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, NUXMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property.

2.5.1 CTL Specifications

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

```
ctl_specification :: CTLSPEC ctl_expr [;]
                   | SPEC ctl_expr [;]
                   | CTLSPEC NAME identifier := ctl_expr [;]
                   | SPEC NAME identifier := ctl_expr [;]
```

The syntax of CTL formulas recognized by NUXMV 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 $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$, $s_{n-1} \rightarrow s_n$ such that p is true in s_n.
- **`AF`** p is true in a state s_0 if *for all* series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$, $s_{n-1} \rightarrow s_n$ p is true in s_n.
- **`EG`** p is true in a state s_0 if *there exists* an infinite series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$ 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 $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$ 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 $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$, $s_{n-1} \rightarrow s_n$ such that p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.
- **`A[`**p **`U`** q**`]`** is true in a state s_0 if *for all* series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $\ldots$, $s_{n-1} \rightarrow s_n$ p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.

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

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

2.5.2 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 identifier := next_expr [;]
```

This statement is intuitively equivalent to

```
CTLSPEC AG next_expr ;
```

but can be checked by a specialised algorithm during reachability analysis. Invariant Specifications, differently from corresponding CTL, can contain **next** operators. Fairness constraints are not taken into account during invariant checking.

2.5.3 LTL Specifications

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

```
ltl_specification :: LTLSPEC ltl_expr [;]
                   | LTLSPEC NAME identifier := ltl_expr [;]
```

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

```
at_expr ::
    next_expr
    | ltl_expr
    | at_expr at next at_expr -- at next
    | at_expr @F~ at_expr     -- at next
    | at_expr at last at_expr -- at last
    | at_expr @O~ at_expr     -- at last
ltl_expr ::
    at_expr              -- a boolean expression with at and next
    | ( 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
    | G bound ltl_expr       -- bounded globally
    | F ltl_expr             -- finally
    | F bound ltl_expr       -- bounded finally
    | ltl_expr U ltl_expr    -- until
    | ltl_expr U bound ltl_expr    -- bounded until
    | ltl_expr V ltl_expr    -- releases
    | ltl_expr V bound ltl_expr    -- bounded 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 S bound ltl_expr    -- bounded since
    | ltl_expr T ltl_expr    -- triggered
    | ltl_expr T bound ltl_expr    -- bounded triggered

bound :: [ integer_number , integer_number ]
       | [ integer_number , +oo )
```

Intuitively the semantics of LTL operators is as follows:

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

Intuitively the semantics of AT operators is as follows:

- *expr* **@F~** p is the value *expr* will have the next time p will hold.
- *expr* **@O~** p is the value *expr* had the last time p held.

An LTL formula is true if it is true at the initial time t_0.

In NUXMV, LTL specifications can be analyzed, depending on the fact that the model is finite-state or infinite state, by means of BDD-based reasoning, by means of SAT-based techniques or for SMT based techniques. In the case of BDD-based reasoning, NUXMV proceeds according to [CGH97a]. For each LTL specification, a tableau of the behaviors falsifying the property is constructed, and then synchronously composed with the model. Similarly to NUSMV, the [CGH97a] approach is fully integrated within NUXMV, 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/SMT-based reasoning, a similar tableau construction is carried out to encode the paths of limited length, violating the property. NUXMV, similarly to NUSMV, generates a propositional satisfiability problem, that is then tackled by means of efficient SAT or SMT solvers.

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

LTL Specifications in TTS

Some additional LTL operators are available for TTS and the semantic of some of the existing ones is updated. In this context also the following LTL formulas are recognized by NUXMV:

```
op :: < | <= | = | != | >= | >
at_expr ::
         at_expr
         | time_until ltl_expr op at_expr  -- time until
         | time_since ltl_expr op at_expr  -- time since
ltl_expr ::
         ltl_expr
         -- FUTURE
         | X~ ltl_expr  -- timed next state
         -- PAST
         | Y~ ltl_expr  -- timed previous state
```

Where in these cases the `at_expr` used as operand of the comparison operator can contain neither **`time`**, nor **`time_until`**, nor **`time_since`**.
In the following the semantic of these new operators and the changes applied to existing ones is provided. In particular bounds on LTL operators predicate over time and not on the number of discrete transitions. The possible intervals are:

```
bound :: [ number , +oo ) | [ 0 , number ]
```

where number can be an expression containing constant numbers and frozen variables. This expression must be of type real.

- **`X`** p is true at configuration s if from s there is a discrete transition to a state in which p is true.
- **`X~`** p is true in configuration s at time t if from s there is a time elapse and p holds in the left open interval $(t, t+\epsilon]$.
- **`F`** `[l,+oo)` p is true at time t if p is true at *some* time $t + l \leq t'$.
- **`G`** `[l,+oo)` p is true at time t if p is true at *all* times $t + l \leq t'$.
- **`Y`** p is true at configuration $s\neg = s_0$ if p holds in s' and there is a discrete transition from s' to s. **`Y`** p is *false* in s_0.
- **`Y~`** p is true in configuration s at time $t > 0$ if p holds in the right open interval $[t-\epsilon, t)$. **`Y~`** p is *false* at time 0.
- **`H`** `[l,+oo)` p is true at time t if p holds in *all* previous time steps $t' \leq t - l$.
- **`O`** `[l,+oo)` p is true at time t if p held in *at least one* of the previous time steps $t' \leq t - l$.
- *`expr`* **`@F~`** p is the value *expr* will have the next time `p` **`| X~`** `p` will hold.
- *`expr`* **`@O~`** p is the value *expr* had the last time `p` **`| Y~`** `p` held.
- **`time_until`** p is the time elapse required to reach the next state in which p holds.
- **`time_since`** p is the time elapsed from the last state in which p held.

Important Difference Between BDD and SAT/SMT Based LTL Model Checking

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

```
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/SMT-based bounded model checking. The counter-example found consists of one state `s=FALSE`.

This difference between the results is caused by the fact that BDD model checking investigates only *infinite* paths whereas SAT/SMT-based model checking is able to deal also with *finite* paths. Apparently infinite paths cannot ever have `s=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=FALSE` violates the specification `G (s = 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/SMT-based bound model checking ignore finite paths it is enough to add a fairness condition to the `main` module:

```
JUSTICE TRUE;
```

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

2.5.4 Real Time CTL Specifications and Computations

NUXMV (as well as NUSMV) allows for Real Time CTL specifications [EMSS91]. Similarly to NUSMV, in NUXMV we assume that each transition takes unit time for execution. RTCTL extends the syntax of CTL path expressions with the following bounded modalities:

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

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)

```
rtctl_specification :: CTLSPEC rtctl_expr [;]
                     | SPEC rtctl_expr [;]
                     | CTLSPEC NAME identifier := rtctl_expr [;]
                     | SPEC NAME identifier := 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 identifier := 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`.

2.5.5 Parameter Synthesis Specifications

In many application domains it is necessary to model and reason about parameterized systems, where parameters are variables whose value is invariant over time, but is only partially constrained (a.k.a. in NUXMV as frozen variable Section 2.3.1 [Frozen Variables], page 27). Choosing an appropriate value of the parameters is a widely spread engineering problem, a form of design space exploration where the parameters can represent different design or deployment decisions.

NUXMV provides the possibility to specify and then construct the space of parameter valuations that satisfy a parameterized model checking problem. We focus on universal parameter valuations, that guarantee the satisfaction of a property for all associated execution traces.

The syntax for specifying parameter synthesis propblems is the following:

```
parameter_synth_problem :: PARSYNTH par_synth_problem [;]
```

where

```
par_synth_problem ::  identifier := { id_list | ltl_expr synt_opt_funct }
                    | identifier := { id_list | VALID ltl_expr synth_opts }
                    | identifier := { id_list | SAT ltl_expr synth_opts }

id_list ::   identifier
           | id_list , identifier

synth_opts ::  /* empty */
             | , synth_opts_list

synth_opts_list ::  /* empty */
```

```
                    | synth_opt
                    | , synth_opts_list , synth_opt

synth_opt ::  /* empty */
              | MAX ( simple_expr )
              | MIN ( simple_expr )
              | MONOPOS
              | MONONEG
```

The `identifier` is a unique identifier within the model. The `id_list` is the list of parameters to be synthesized: they must correspond to frozen variable names (Section 2.3.1 [Frozen Variables], page 27). The `ltl_expr` is the LTL expression for which the parameters have to be computed to guarantee the validity (**`VALID`**) or the satisfaction (**`SAT`**). If not specified, the default meaning is `VALID`. The optional `synth_opts` production specifies the problem characteristics such as the optimization (given a function (`simple_expr`)), or the monotonicity. In particular, `MONOPOS` assumes that $if\{p1\&\neg p2\} \in Badthen\{p1\} \in Bad$, and $if\{p1\&p2\} \in Goodthen\{p1\} \in Goodand\{p2\} \in Good$, and `MONONEG` assumes that $if\{p1\&\neg p2\} \in Badthen\{\neg p2\} \in Bad$, and $if\{\neg p1\&\neg p2\} \in Goodthen\{\neg p1\} \in Goodand\{\neg p2\} \in Good$.

2.5.6 PSL Specifications

NUXMV, similarly to NUSMV, allows to specify PSL properties that comply with 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 identifier := 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
 | - psl_primary_expr
 | ! psl_primary_expr
 | bool ( psl_expr )
 | word1 ( psl_expr )
 | uwconst ( psl_expr, psl_expr )
 | swconst ( psl_expr, psl_expr )
 | sizeof ( psl_expr )
 | toint ( psl_expr )
 | signed ( psl_expr )
```

```
  | unsigned ( psl_expr )
  | extend ( psl_expr, psl_primary_expr )
  | resize ( psl_expr, psl_primary_expr )
  | select ( psl_expr, psl_expr, psl_expr )

psl_binary_expr ::
    psl_expr + psl_expr
  | psl_expr union psl_expr
  | psl_expr in psl_expr
  | psl_expr - psl_expr
  | psl_expr * psl_expr
  | psl_expr / psl_expr
  | psl_expr % psl_expr
  | psl_expr == psl_expr
  | psl_expr != psl_expr
  | psl_expr < psl_expr
  | psl_expr <= psl_expr
  | psl_expr > psl_expr
  | psl_expr >= psl_expr
  | psl_expr & psl_expr
  | psl_expr | psl_expr
  | psl_expr xor psl_expr
  | psl_expr xnor psl_expr
  | psl_expr << psl_expr
  | psl_expr >> psl_expr
  | psl_expr :: psl_expr
psl_conditional_expr ::
 psl_expr ? psl_expr : psl_expr
psl_case_expr ::
 case
     psl_expr : psl_expr ;
     ...
     psl_expr : psl_expr ;
 endcase
```

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

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

```
  | boolean
value_range ::
    psl_expr
  | range
```

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

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

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

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

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

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

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

The NUXMV 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.

Note: Full support for PSL will be integrated in forthcoming releases of NUXMV.

2.6 Variable Order Input

As it is the case in NUSMV, NUXMV allows to specify the order in which variables should appear in the generated BDDs. The file which gives the desired order can be read in using the `-i` option in batch mode or by setting the

input_order_file environment variable in interactive mode. [10]

2.6.1 Input File Syntax

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

```
vars_list :: EMPTY
          | var_list_item vars_list

var_list_item :: complex_identifier
              | complex_identifier . integer_number
```

Where *EMPTY* means parsing nothing.

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

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

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

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

where varname is a variable in modN, and modN.varname is a variable in modN-1, and so on. Note that the module name main is implicitely prepended to every variable name and therefore must not be included in their declarations.

Any variable which appears in the model file, but not the ordering file is placed after all the others in the ordering. Variables which appear in the ordering file but not the model file are ignored. Similarly to NuSMV, in both cases nuXmv displays a warning message stating these actions.

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

2.6.2 Scalar Variables

A variable, which has a finite range of values that it can take, is encoded as a set of 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 boolean variables to represent it.

It is possible not only to declare the position of a scalar variable in the ordering file, but each of the boolean variables which represent it.

If only the scalar variable itself is named then all the boolean variables which are actually used to encode it are grouped together in the BDD package.

Variables which are grouped together will always remain next to each other in the BDD package and in the same order. When dynamic variable re-ordering is carried out, the group of variables are treated as one entity and moved as such.

If a scalar variable is omitted from the ordering file then it will be added at the end of the variable order and the specific-bit variables that represent it will be grouped together. However, if any specific-bit variables have been declared in the ordering file (see below) then these will not be grouped with the remaining ones.

It is also possible to specify 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 Complete_Var_Name.i at the position where you want that bit variable to appear:

[10] Note that if the ordering is not provided by a user then nuXmv decides by itself how to order the variables. Two shell variables bdd_static_order_heuristics (see the NuSMV user manual [CCCJ+10] for further information) and vars_order_type allow to control the ordering creation.

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

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

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

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

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

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

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

2.6.3 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.

2.7 Clusters Ordering

When NUXMV builds a clusterized BDD-based FSM during model construction (as it is the case for NUSMV), an initial simple clusters list is roughly constructed by iterating through a *list of variables*, and by constructing the clusters by picking the transition relation associated to each variable in the list. Later, the clusters list will be refined and improved by applying the clustering alghorithm that the user previoulsy selected (for further information, see partitioning methods in the NUSMV user manual [CCCJ+10]).

NUXMV, similar to NUSMV, allows to specify an ordering for the initial list of variables that is used to build the clusters. The option `trans_order_file` can be used to specify a file containing a variable ordering. This feature is inherited directly from NUSMV (for further information see the NUSMV user manual [CCCJ+10]).

Grammar of the clusters ordering file in NUXMV is the same of the one in the NUSMV user manual [CCCJ+10].

Chapter 3

Running NUXMV interactively

NUXMV inherits from NUSMV the interactive shell. In this mode NUXMV, like NUSMV, enters a read-eval-print loop. The user can activate the various NUXMV computation steps as system commands with different options. These steps can therefore be invoked separately, possibly undone or repeated under different modalities. As for NUSMV, the interactive shell of NUXMV is activated from the system prompt as follows ('`nuXmv >`' is the default NUXMV shell prompt):

```
system_prompt> nuXmv -int <RET>
nuXmv >
```

When running interactively, NUXMV 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.
Search is done in this order:

1. File `master.nuxmvrc` is looked for in directory defined in environment variable `NUXMV_LIBRARY_PATH` or in default library path (e.g. `/usr/local/share/nusmv` under GNU/Linux) if no such variable is defined.
2. If no such file exists, file `.nuxmvrc` is looked for in user's home directory.
3. If no such file exists, `.nuxmvrc` is looked for in current directory.

Tip: To see which file are searched for and the search paths we recommend to use the command line option **`-h`** and to look at the description for option **`-s`**.

Commands in the initialization file (if any) are executed consecutively. When initialization phase is completed the NUXMV shell is displayed and the system is now ready to execute user commands.

Similar to NUSMV, a NUXMV 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 NUXMV 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 NUXMV read and execute a sequence of commands from a file, through the command line option **`-source`**:

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

`-source` *`cmd-file`* Starts the interactive shell and then executes NUXMV 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`**.

Command sequences in NUXMV (similar to NUSMV) must obey the (partial) order specified in Figure 3.1 (at page 54) for untimed systems. While for timed systems (**-time** command line flag specified), the command sequences must obey the (partial) order depicted in Figure 3.2 (at page 55). From the picture, it is clear that, for instance, it is not possible to evaluate CTL/LTL or invariant properties before the model is built. In Figure 3.1 and Figure 3.2 a star (*) is used as prefix to denote new commands provided by NUXMV, and not available in NUSMV. Furthermore, loop backs are not represented to ease the reading. Nevertheless, the flow should be still well comprehensible. For a description of the commands to analyze untimed models see chapter 5 (page 116). While the description of the commands to analyze timed models can found in chapter 6 (page 150).

A number of commands and environment variables, like those dealing with file names, accept arbitrary strings. There are a few reserved characters which must be escaped if they are to be used literally in such situations. See the section describing the `history` command, on page 109, for more information.

The verbosity of NUXMV is controlled by the `verbose_level` environment variable.

verbose_level Environment Variable

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

NUXMV provides to the user all the commands that NUSMV provides. Moreover, it extends the set of commands with new ones. In Chapter 4 we describe all the NUSMV commands, while in Chapter 5 we describe all the new NUXMV commands. All the commands are organizez for functionality.

Command Annotations

All the NUSMV commands operates over finite domains. In the following we annotated each new commands with a flag stating whether the command is applicable only to finite-state domain, or if it can be applied to infinite-state domains (which include finite domains). We use the folloiwng annotations:

- **[F]**: the command can be applied only to finite-state domains.
- **[I]**: the command can be applied only to infinite-state domains, which include finite-state ones.
- **[F,I]**: the command provides the user with command line options to invoke the finite-state or the infinite-state version (that uses SMT) for the underlying algorithm.

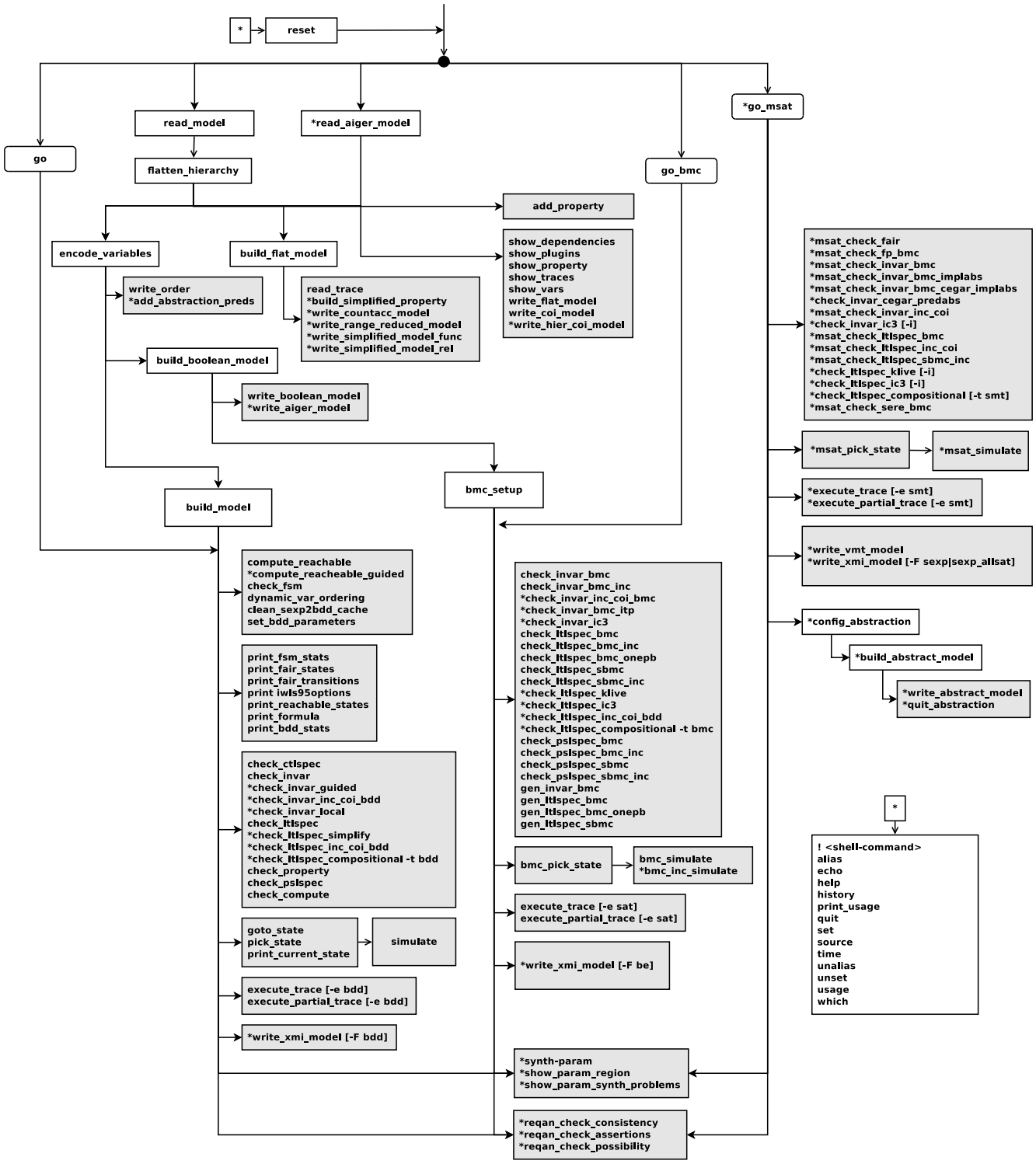

Figure 3.1: The dependency among NUXMV commands.

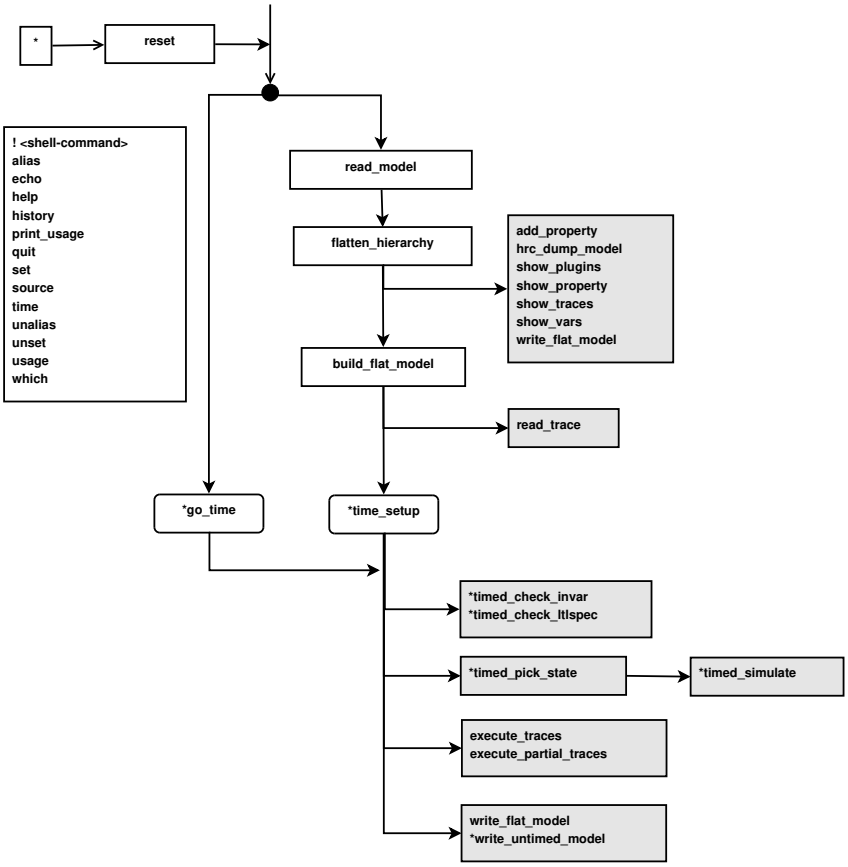

Figure 3.2: The dependency among NUXMV commands for timed models.

Chapter 4

Commands from NuSMV

In the following we present the commands inherited from NuSMV. We also describe the environment variables that may affect the behavior of the commands. All the commands have been classified in different categories.
Tip: Each command has the command line option **-h** that provides the short description for the command itself.

4.1 Model Reading and Building

The following commands allow for the parsing and compilation of the model in order to enable all the verification algorithms.

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

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

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

Command Options:

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

input_file Environment Variable

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

pp_list Environment Variable

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

flatten_hierarchy - *Flattens the hierarchy of modules* Command

`flatten_hierarchy [-h] [-d] [-e]`

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:

`-d`	Delays the construction of vars constraints until needed
`-e`	Expands the word-array expressions into individual elements expressions

disable_syntactic_checks Environment Variable

Enables or disables the syntactic checks that are performed by the "flatten_hierarchy" command. Warning: If the model is not well-formed, NUXMV may result in unpredictable results, use this option at your own risk.

keep_single_value_vars Environment Variable

Enables or disables the conversion of variables that can assume only one single possible value into constant DEFINEs.

backward_compatibility Environment Variable

As in NUSMV, this variable enables/disables type checking and other features provided by NUXMV. If set to `1` then the type checking is turned off, and NUXMV behaves as the old versions of NUSMV (for the pure boolean case) 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 NUXMV program the user is informed and the execution is stopped.

Since NUSMV 2.5.1, backward compatibility mode introduces a porting feature from old models which use constant `1` as `case` conditions, instead of forcing the use of `TRUE`.

The option by default it set to `0`.

type_checking_warning_on Environment Variable

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

show_vars - *Shows model's symbolic variables and defines with their types* Command

```
show_vars [-h] [-s] [-f] [-i] [-t | -V | -D] [-v] [-m | -o output-file]
```

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.
`-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* Command

```
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.* Command

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

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

Command Options:

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

input_order_file Environment Variable

Indicates the file name containing the variable ordering to be used in building the model by the '`encode_variables`' command. 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 58 and `bdd_static_order_heuristics` on page 60).

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 Environment Variable

When a variable ordering is not specified (see variable `input_order_file` on page 58) NUXMV 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 `TRANS next(a)=b+1 -> (next(c)=d/e & next(f)!=a)` will results in 2 groups of variables {`a,b,f`} and {`c,d,e`}.

  Shell variable `vars_order_type` (page 59) 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:

- **Monolithic**. No partitioning at all.

- **Threshold**. Conjunctive partitioning, with a simple threshold heuristic. Assignments are collected in a single cluster until its size grows over the value specified in the variable `conj_part_threshold`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable.
- **Iwls95CP**. Conjunctive partitioning, with clusters generated and ordered according to the heuristic described in [RAP+95]. Works in conjunction with the variables `image_cluster_size`, `image_W1`, `image_W2`, `image_W3`, `image_W4`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable. It is also possible to avoid (default) preordering of clusters (see [RAP+95]) by setting the `iwls95preorder` variable appropriately.

conj_part_threshold Environment Variable

The limit of the size of clusters (expressed as number of BDD nodes) in conjunctive partitioning. The default value is `1000` BDD nodes.

affinity Environment Variable

This variable controls whether to enables the affinity clustering heuristic as described in [MHS00]. Possible values are `0` or `1`: the default value is `1`.

trans_order_file Environment Variable

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

image_cluster_size Environment Variable

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

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

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

iwls95preorder Environment Variable

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

image_verbosity Environment Variable

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

print_iwls95options - *Prints the Iwls95 Options.* Command

```
print_iwls95options [-h]
```

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

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

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

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

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

Command Options:

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

get_internal_status - *Prints out the internal status of the system.* Command

```
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.* Command

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

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

Command Options:

`-f`	Forces the model construction even when Cone Of Influence is enabled.
`-r`	Forces a variable reordering at the end of the computation, and dumps the new variables ordering to the default ordering file. This options acts like the command line option `-reorder`.
`-i model-file`	Sets the environment variable `input_file` to file `model-file`, and reads the model from file `model-file`.
`-m Method`	Sets the environment variable `partition_method` to `Method` and uses it as partitioning method.

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 constanst 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

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

daggifier_depth_threshold — Environment Variable

Sets the minimum threshold for expressions depth to be daggified.

daggifier_counter_threshold — Environment Variable

Sets the minimum threshold for expressions count to be daggified. (i.e. expression must show at least `Number` time to be daggified

daggifier_statistics — Environment Variable

Prints daggifier statistics after model dumping.

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

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

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

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

Command Options:

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

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

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

output_boolean_model_file Environment Variable

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

dump_fsm - *Dumps (in DOT format) selected parts of the bdd fsm, with optional expression* Command

`dump_fsm [-h] -o <fname> [-i] [-I] [-t] [-f] [-r] [-e <expr>]`

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).

output_word_format Environment Variable

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

4.2 Commands for Checking Specifications

The following commands allow for the BDD-based model checking of a NUXMV model. These commands can be used only for NUXMV models that do not contain Real or Integers.

compute_reachable - *Computes the set of reachable states* Command

```
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* Command

```
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.* Command

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

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

Command Options:

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

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

check_fsm Environment Variable

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

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

```
print_fsm_stats [-h] | [-m] | [-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` nor `-P` are used, all the SPEC formulas in the database are checked.
See variable `use_coi_size_sorting` for changing properties verification order.

Command Options:

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

ag_only_search Environment Variable

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

forward_search Environment Variable

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

ltl_tableau_forward_search Environment Variable

Forces the computation of the set of reachable states for the tableau resulting from BDD-based LTL model checking, performed by command `check_ltlspec`. If the variable `ltl_tableau_forward_search` is not set (default), the resulting tableau will inherit the computation of the reachable states from the model, if enabled (see environment variable `use_reachable_states`). If the variable is set to true, the set of reachable states will be calculated for the model *and* for the tableau resulting from LTL model checking.

Remark. This might improve performances of the command `check_ltlspec`, but may also lead to a dramatic slow down for some modeld. This variable has effect only when the calculation of reachable states for the model is enabled (see `forward_search`).

oreg_justice_emptiness_bdd_algorithm Environment Variable

The algorithm used to determine language emptiness of a Büchi fair transition system. The algorithm may be used from the following commands: `check_ltlspec`, `check_pslspec`. Possible values are:

- **EL_bwd** The default value. The Emerson-Lei algorithm [EL86] in its usual backwards direction, i.e., using backward image computations.
- **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.

check_invar - *Performs model checking of invariants* Command

```
check_invar [-h] [-m | -o output-file] [-n number | -p
"invar-expr  [IN context]" | -P "name"] [-s strategy] [-e
f-b-heuristic] [-j b-b-heuristic] [-t threshold] [-k length]
```

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_coi_size_sorting` for changing properties verification order.

Command Options:

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

`-j b-b-heuristic`	When using “bdd-bmc” strategy specify the heuristic that decides at which step we must switch from BDD to BMC technolgy. 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_bddbmc_heuristic” is considered.
`-t threshold`	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_bddbmc_threshold” is considered.
`-k length`	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 Environment Variable

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 Environment Variable

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 Environment Variable

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 Environment Variable

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* Command

```
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 [CGH97a].

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_coi_size_sorting` for changing properties verification order.

Command Options:

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

`-o output-file`	Writes the output generated by the command in processing `LTLSPEC` formulas to the file `output-file`.
`-p "ltl-expr [IN context]"`	An LTL formula to be checked. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-P "name"`	Checks the LTL property named `name`
`-n number`	Checks the LTL property with index `number` in the property database.

ltl2smv_single_justice Environment Variable

Informs the 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 a an additional monitor.) By default multiple Justice are built.

check_compute - *Performs computation of quantitative characteristics* Command

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

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

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

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

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

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

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

See variable `use_coi_size_sorting` for changing properties verification order.

Command Options:

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

```
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_coi_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* Command

```
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 INVAR property.
-s	Adds a PSL property.
-q	Adds a quantitative (COMPUTE) property.
-p "formula [IN context]"	Adds the formula specified on the command-line. context is the module instance name which the variables in formula must be evaluated in.
-n "name"	Sets the name of the property to “name”

show_property - *Shows the currently stored properties* Command

```
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, INVAR, 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, INVAR, 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 INVAR 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. *tabular* 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* Command

```
convert_property_to_invar[-n number | -P "name" | -l -p G next-expr | -c
-p 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 or -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, see -l and -c.
`-l`	use with -p to specify a LTL formula.
`-c`	use with -p to specify a CTL formula.

write_coi_model - *Writes a restricted flat model to a file* Command

`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
`-s`	Dumps COI model for all PSL properties
`-q`	Dumps COI model for all COMPUTE properties
`-C`	Only prints the list of variables that are in the COI of properties
`-g`	Dumps the COI model that represents the union of all COI properties

cone_of_influence Environment Variable

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.

use_coi_size_sorting Environment Variable

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

prop_print_method Environment Variable

Determines how properties are printed. The following methods are available:

name. Prints the property name. If not available, defaults to method "index".
index. Prints the property index. If not available, defaults to method "truncated".
truncated. Prints the formula of the property. If the formula is longer than 40 characters, it is truncated.
formula. The default method, simply prints the formula.

4.3 Commands for Bounded Model Checking

In this section we describe in detail the commands for doing and controlling Bounded Model Checking in NUXMV. Bounded Model Checking is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. After the problem is generated, NUXMV internally calls a propositional SAT solver in order to find an assignment which satisfies the problem. Currently NUXMV supplies two SAT solvers: Zchaff and MiniSat. If none of the two is enabled, all Bounded Model Checking part in NUXMV 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 NUXMV.

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 NUXMV, 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 NUXMV with a separate external SAT solver. At the moment, the DIMACS files can be generated only by commands which do not use incremental algorithms.

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

```
bmc_setup [-h]
```

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

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

```
go_bmc [-h] [-f]
```

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

Command Options:

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

sexp_inlining Environment Variable

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

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

rbc_inlining Environment Variable

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

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

rbc_rbc2cnf_algorithm Environment Variable

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 '`sheridan`' refers to [She04] algorithm which allows to obtain a more compact CNF formulas. The other value '`tseitin`' refers to a standard Tseiting transformation algorithm.

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

```
check_ltlspec_bmc [-h ] | [-n idx | -p "formula [IN context]" | -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* — Command

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

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

Command Options:

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

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

`-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 '$\star$', which means "all possible loopback from zero to *length-1*".

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

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

- @@: the '@' character.

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

```
gen_ltlspec_bmc [-h] | [ -n idx | -p "formula" [IN context] | -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]"` — 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 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 '$\star$', which means "all possible loopback from zero to *length-1*".

`-o` *filename* — *filename* is the name of dumped dimacs files. If this options is not specified, variable *bmc_dimacs_filename* will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:

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

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

```
gen_ltlspec_bmc_onepb [-h ] | [ -n idx | -p "formula" [IN context] | -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, -*length*). Any invalid combination of length and loopback will be skipped during the generation process.
- the symbol 'X', which means "no loopback".
- the symbol '$\star$', which means "all possible loopback from zero to *length-1*".

`-o` *filename* — *filename* is the name of the dumped dimacs file. If this options is not specified, variable `bmc_dimacs_filename` will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:

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

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

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

For each problem this command incrementally generates many satisfiability subproblems and calls the SAT solver on each one of them. The incremental algorithm exploits the fact that subproblems have common subparts, so information obtained during a previous call to the SAT solver can be used in the consecutive ones. Logically, this command does the same thing as `check_ltlspec_bmc` (see the description on page 76) 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 `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 must be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc_length* is considered instead.
`-l` *loopback*	The *loopback* value may be: • a natural number in (0, *max_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process. • a negative number in (-1, *-bmc_length*). In this case *loopback* is considered a value relative to *max_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process. • the symbol 'X', which means "no loopback". • the symbol '$\star$', which means "all possible loopback from zero to *length-1*" .

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

```
check_ltlspec_sbmc [-h] | [-n idx | -p "formula [IN context]" | -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 `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"`.

See variable `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.
`-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 '$\star$', which means "all possible loopbacks from zero to *length-1*" .
`-o` *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @F: model name with path part. • @f: model name without path part. • @k: current problem bound. • @l: current loopback value. • @n: index of the currently processed formula in the property database. • @@: the '@' character.

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

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

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

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

See variable `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.* Command

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

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

bmc_length Environment Variable

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

bmc_loopback Environment Variable

Sets the generated problem loop. Possible values are:

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

The default value is `*`.

bmc_optimized_tableau Environment Variable

Uses depth1 optimization for LTL Tableau construction in BMC.

bmc_force_pltl_tableau Environment Variable

Forces to use PLTL instead of LTL for BMC tableau construction.

bmc_dimacs_filename Environment Variable

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

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

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

bmc_sbmc_gf_fg_opt Environment Variable

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

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

```
check_invar_bmc [-h | -n idx | -p "formula" [IN context] | -P "name"]
[-a alg] [-e] [-k bmc_bound] [-o filename]
```

In Bounded Model Checking, invariants are proved using induction. For this, satisfiability problems for the base and induction step are generated and a SAT solver is invoked on each of them. At the moment, two algorithms can be used to prove invariants. In one algorithm, which we call "classic" (i.e. k-induction with k=1), 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_coi_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` (i.e. k-induction with k=1) or `een-sorensson`. If no value is given the environment variable *bmc_invar_alg* is considered instead.
`-o` *filename*	*filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: • @**F**: model name with path part • @**f**: model name without path part • @**n**: index of the currently processed formula in the properties database • @@: the '@' character

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

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

At the moment, the invariants are generated using "classic" (k-induction with k=1) algorithm only (see the description of `check_invar_bmc` on page 84).

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.
`-o` *filename*	*filename* is the name of the dumped dimacs file. If you do not use this option the dimacs file name is taken from the environment variable `bmc_invar_dimacs_filename`. File name may contain special symbols which will be macro-expanded to form the real dimacs file name. Possible symbols are: • @**F**: model name with path part • @**f**: model name without path part • @**n**: index of the currently processed formula in the properties database • @@: the '@' character

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

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

This command does the same thing as `check_invar_bmc` (see the description on page 84) 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_coi_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 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` (for k-induction with k=1) and `een-sorensson`. The default value is `classic`.

bmc_inc_invar_alg Environment Variable

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

bmc_invar_dimacs_filename Environment Variable

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

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

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

sat_solver Environment Variable

The SAT solver's name actually to be used. Default SAT solver is MiniSat. Depending on the NUXMV 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* Command

```
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 first 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`	Verbosely 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* Command

```
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 `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

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

```
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 `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

bmc_simulate_check_feasible_constraints - *Checks feasability for the given constraints* Command

```
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)

4.4 Commands for checking PSL specifications

The following commands allow for model checking of PSL specifications.

check_pslspec - *Performs BDD-based PSL model checking* Command

```
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`. 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.

See variable `use_coi_size_sorting` for changing properties verification order.

Command Options:

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

check_pslspec_bmc - *Performs SAT-based PSL model checking* Command

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

Check psl properties using 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_lenght* 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_coi_size_sorting` for changing properties verification order.

Command Options:

`-m`	Pipes the output generated by the command in processing `PSLSPEC` formulas to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o` *output-file*	Writes the output generated by the command in processing `PSLSPEC` formulas 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`.
`-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_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"] [-1] [-k
bmc_lenght] [-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_lenght* 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_coi_size_sorting` for changing properties verification order.

Command Options:

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

`-o` *output-file*	Writes the output generated by the command in processing `PSLSPEC` formulas 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* Command

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

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_lenght* 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_coi_size_sorting` for changing properties verification order.

Command Options:

`-m`	Pipes the output generated by the command in processing `PSLSPEC` formulas to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o` *output-file*	Writes the output generated by the command in processing `PSLSPEC` formulas 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`.
`-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 '$\star$', 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_inc - *Performs incremental SAT-based PSL model checking* — Command

```
check_pslspec_sbmc_inc [-h] [-m | -o output-file] [-n number | -p
"psl-expr  [IN context]" | -P "name"] [-1] [-k
bmc_lenght] [-l loopback] [-c] [-N]
```

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_lenght* 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`. 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_coi_size_sorting` for changing properties verification order.

Command Options:

`-m`	Pipes the output generated by the command in processing `PSLSPEC` formulas to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o` *output-file*	Writes the output generated by the command in processing `PSLSPEC` formulas 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.
`-c`	Performs completeness check.
`-N`	Does not perform virtual unrolling.

4.5 Simulation Commands

In this section we describe the commands that allow to simulate a NUXMV specification. See also the section Section 4.7 [Traces], page 97 that describes the commands available for manipulating traces.

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

```
pick_state [-h] [-v] [-r | -i [-a]] [-c "constraints" | -s trace.state]
[-S seed]
```

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.
`-S seed`	Sets the seed to be used for random simulation.

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

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

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
`-S seed`	Sets the seed to be used for random simulation.

default_simulation_steps Environment Variable

Controls the default number of steps performed by all simulation commands. The default is 10.

shown_states Environment Variable

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`.

traces_hiding_prefix Environment Variable

see section 4.7.2 for a detailed description.

traces_regexp Environment Variable

see section 4.7.2 for a detailed description.

4.6 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 4.7 [Traces], page 97 that describes the commands available for manipulating traces.

execute_traces - *Executes complete traces on the model FSM* Command

`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* Command

`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 succesful 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.

4.7 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 NUXMV when a formula is found to be false; they are also generated as a result of a simulation (Section 4.5 [Simulation Commands], page 94) or partial trace re-execution (Section 4.6 [Execution Commands], page 96). Each trace has a number, and the states-inputs pairs are numbered within the trace. Trace *n* has states/inputs *n.1, n.2, n.3, "..."* where *n.1* represents the initial state.

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.

4.7.1 Inspecting Traces

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

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

```
goto_state [-h] state_label
```

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

`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.

print_current_state - *Prints out the current state* Command

```
print_current_state [-h] [-v]
```

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

Command Options:

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

4.7.2 Displaying Traces

NUXMV comes with three trace plugins (see Section 4.8 [Trace Plugins], page 101) which can be used to display traces in the system. Once a trace has been generated by NUXMV it is printed to `stdout` using the trace explanation plugin which has been set as the current default. The command `show_traces` (see Section 4.5 [Simulation

Commands], page 94) 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`.

counter_examples Environment Variable

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 NUXMV may introduce hidden placeholding 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 `m1.m2.name` only `name` is checked for filtering.

Notice that depending on the underlaying 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.

4.7.3 Trace Plugin Commands

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

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 NUXMV are to be shown. The values that this variable can take depend on which trace plugins are installed. Use the command `show_plugins` to see which ones are available. The default value is `0`.

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

```
show_traces [-h] [-v] [-t] [-A] [-m | -o output-file]
[-p plugin-no] [-a | trace_number[.from_state[:[to_state]]]]
```

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`. If -a is also specified, then each trace is stored in a separate file named "num_output-file" where num is the trace number.
`-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).

read_trace - *Loads a previously saved trace* Command

```
read_trace [-h | [-i filename] [-u] [-s] filename]
```

Command Options:

`-i filename`	Reads in a trace from the specified file. Note that the file must only contain one trace. *This option has been deprecated.* Use the explicit filename argument instead.

`-u` Turns "undefined symbol" error into a warning. The loader will ignore assignments to undefined symbols.

`-s` Turns "wrong section" error into a warning. The loader will accept symbol assignment even if they are in a different section than expected. Assignments will be silently moved to appropriate section, i.e. misplaced assignments to state symbols will be moved back to previous state section and assignments to input/combinatorial symbols will be moved forward to successive input/-combinatorial section. Such a way if a variable in a model was input and became state or vice versa the existing traces still can be read and executed.

Loads a trace which has been previously output to a file with the XML Format Output plugin. The model from which the trace was originally generated must be loaded and built using the command "`go`" first. Please note that this command is only available on systems that have the libxml2 XML parser library installed.

4.8 Trace Plugins

NUXMV 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 previuosly generated, or loaded, trace to a specific file.

4.8.1 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:

`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:

```
-> Input: TRACE_NO.STATE_NO <-
    /* for each input var (being printed), i: */
    INPUT_VARi = VALUE
-> State: TRACE_NO.STATE_NO <-
    /* for each state and frozen var (being printed), j: */
```

```
STATE_VARj = VALUE
/* for each combinatorial constant (being printed), k: */
CONSTANTk = VALUE
```

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

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

4.8.2 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:

```
S1 C2 I2 S2 ... Cn In Sn
```

where S1 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:

```
Step1 Step2 ... Stepn
```

4.8.3 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_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>
    <combinatorial id=i+1>

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

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

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

```
    </input>
  </node>

</counter-example>
```

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

4.8.4 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[1] has been loaded, and built using the command `go`.

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

4.8.5 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.

4.9 Interface to the DD Package

NUXMV uses the state of the art BDD package CUDD [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 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 — Environment Variable

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 `-dynamic` command line option when invoking NUXMV.

reorder_method — Environment Variable

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

[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.

`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.
`window2_converge`: `window3_converge`: `window4_converge`:	The `window{2,3,4}` method is iterated until no further improvement is obtained.
`group_sift`:	This method is similar to `symmetry_sift`, but uses more general criteria to create groups.
`group_sift_converge`:	The `group_sift` method is iterated until no further improvement is obtained.
`annealing`:	This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.

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

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

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

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

Command Options:

`-d` Disable dynamic ordering from triggering automatically.

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

- **sift**: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.
- **random**: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.
- **random_pivot**: Same as **random**, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.
- **sift_converge**: The **sift** method is iterated until no further improvement is obtained.
- **symmetry_sift**: This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.

- **symmetry_sift_converge**: The **symmetry_sift** method is iterated until no further improvement is obtained.
- **window{2,3,4}**: Permutes the variables within windows of "n" adjacent variables, where "n" can be either 2, 3 or 4, so as to minimize the overall BDD size.
- **window{2,3,4}_converge**: The **window{2,3,4}** method is iterated until no further improvement is obtained.
- **group_sift**: This method is similar to **symmetry_sift**, but uses more general criteria to create groups.
- **group_sift_converge**: The **group_sift** method is iterated until no further improvement is obtained.
- **annealing**: This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.
- **genetic**: This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.
- **exact**: This method implements a dynamic programming approach to exact reordering. It only stores a BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.
- **linear**: This method is a combination of sifting and linear transformations.
- **linear_converge**: The **linear** method is iterated until no further improvement is obtained.

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

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

```
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.* Command

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

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

Command Options:

`-v` Prints explicit models of the given simple_expression.

`-f` Prints the simplified and canonical simple_expression.

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 unmodified 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* Command

```
print_bdd_stats [-h]
```

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

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

```
set_bdd_parameters [-h] [-s]
```

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

Command Options:

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

4.10 Administration Commands

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

! - *shell_command* Command

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

alias - *Provides an alias for a command* Command

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

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

Command Options:

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

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

For example:

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

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

```
nuXmv > alias echo2 "echo Hi ; echo %* !"
nuXmv > 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.

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

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

echo - *Merely echoes the arguments* Command

```
echo [-h] [-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 overwritting it. Use only with the option `-o`.

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

```
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* Command

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

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

Command Options:

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

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

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

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

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

Command Options:

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

print_usage - *Prints processor and BDD statistics.* Command

`print_usage [-h]`

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

quit - *exits NuSMV* Command

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

reset - *Resets the whole system.* Command

`reset [-h]`

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

set - *Sets an environment variable* Command

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

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

Command Options:

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

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

```
nuXmv > 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:

```
nuXmv > set foo bar
nuXmv > 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:

```
nuXmv > set foo bar
nuXmv > set foo $foo:foobar
nuXmv > 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

```
nuXmv > set "foo bar" this
nuXmv > 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: `nuXmv > set "foo/bar" this`

```
nuXmv > 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 `NUXMV_LIBRARY_PATH`) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.
`nusmv_stderr`	Standard error (normally `(` stderr)) can be re-directed to a file by setting the variable `nusmv_stderr`.
`nusmv_stdout`	Standard output (normally `(` stdout)) can be re-directed to a file by setting the variable `nusmv_stdout`.

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

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

Reads and executes commands from a file.

Command Options:

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

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


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

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

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

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

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

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

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

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

See the variable `on_failure_script_quits` for further information.

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

```
time [-h]
```

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

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

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

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

Command Options:

`<alias-names>`	Aliases to be removed

unset - *Unsets an environment variable* Command

`unset [-h] <variables>`

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

Command Options:

`<variables>`	Variables to be unset.

usage - *Provides a dump of process statistics* Command

`usage [-h]`

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

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

`which [-h] <file_name>`

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

Command Options:

`<file_name>`	File to be searched

4.11 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 NuXmv. This behaviour might be useful when the command `source` is controlled by either a system pipe or a shell script. Under these conditions a mistake within the script interpreted by `source` or any unexpected error might hang the controlling script or pipe, as by default the interpreter would simply give up the current execution, and wait for further commands. The default value of this environment variable is `0`.

filec Environment Variable

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

shell_char Environment Variable

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

history_char Environment Variable

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

open_path Environment Variable

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

nusmv_stderr Environment Variable

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

nusmv_stdout Environment Variable

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

nusmv_stdin Environment Variable

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

Chapter 5

Commands of NUXMV

In the following we present the new commands provided by NUXMV. Similarly to the case of the commands inherited from NUSMV, we also describe the environment variables that may affect the behavior of the commands. All the commands have been classified in different categories.

5.1 Commands for Initialization

go_msat - *Initializes the system for the infinite state verification via SMT.* Command

```
go_msat [-h] [-f]
```

This command initializes the system for verification of finite and infinite state systems. It is equivalent to a series of internal commands. 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.

5.2 Commands for Model Simulation

In this section we describe the new commands that allow to simulate a NUXMV specification that may contains Integers and Reals. See also the section Section 4.7 [Traces], page 97 that describes the commands available for manipulating traces.

msat_pick_state - *Picks a state from the set of initial states* Command **[I]**

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

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).
`-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 "constraint"`	Uses `constraint` 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.

msat_simulate - *Generates a trace of the model from 0 (zero) to k* Command [I]

```
msat_simulate [-h] [-v] [-i [-a]] [-e] [-l] [-k length] [[-c
"simple_expr"] | [-t "next_expr"] | [-p "formula"]]
```

`msat_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:

`-v`	Prints the generated trace (all variables).
`-e`	Extends the previous simulation if any. Option `-p` below cannot be specified in conjunction with this option.
`-l`	Performs look-ahead while doing the simulation to see whether the trace can be extended, thus trying to bump in possible deadlocks.
`-i`	Enables the user to interactively pick up a next state.
`-c simple_expr`	Performs a simulation in which computation is restricted to states satisfying those `simple_expr`. 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: `simple_expr` 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` *next_expr*	Performs a simulation in which computation is restricted to states satisfying those `next_expr`. The desired sequence of states could not exist if such next_expr was too strong or it may happen that at some point of the simulation a future state satisfying that next_expr doesn't exist: in that case a trace with a number of states less than `steps` trace is obtained. Note: `next_expr` 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`
`-p` *"formula"*	Performs a simulation in which computation is restricted to states satisfying the given LTL formula. Option `-e` cannot be used in conjunction with this option.
`-k` *length*	Maximum length of the path according to the constraints. The length of a trace could contain less than `length` 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

5.3 Commands for Invariant Checking

In this section we list the new commands for checking invariants. Some of these commands can be applied only to finite-state models (e.g. "check_invar_guided"), some others can be applied both to finite and infinite-state models (e.g. "check_invar_ic3").

check_invar_guided - *Guided reachability invariant checking* Command **[F]**

```
check_invar_guided [-n <index> | -p <prop> | -P <name>] [-s] [-S] [-R]
[-d] [-a] [-u] [-f] (-h | -e "sere_expr" | -i sere_file)
```

Performs invariant checking on the given model using Guided Reachability algorithm over the given strategy. Checking the invariant is the process of determining that all states reachable from the initial states of the model lie in the invariant. For each falsified property, a counterexample is built and stored in the TraceMgr according to the global option about counterexample generation.

By default in GR the computation of reachable states is done in two steps. At first, states satisfying the strategy are computed until fixpoint is reached. Then, starting from the previous fixpoint states, the image computation is applied regardless of the strategy until the global fixpoint, i.e. until all the reachable states are detected.

Invariants to be verified have to be provided as simple formulas (the only temporal operator allowed is "next") in the input model file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

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

If option `-d` is used, it is not possible to mark as verified the properties not falsified during the strategy application because the set of reached states might be not complete.

If generalized invariant (invariant containing IVAR and NEXT variables) are checked, the BDD version of the input model is needed to perform property rewriting. Using the option `-d`, the command avoid to build it, so we need to force the construction using the option `-f`.

When option `-a` is used, the verification process (and the underlying reachability analysis) stops as soon as the first checked property is found false. Since the exploration resuming is not allowed, any successive call to the command will start the reachability analysis from scratch.

The strategy must be a valid PSL formula. Allowed PSL operators are: "; $[*]$ $[*N](withN > 0)$ |"

Command Options:

`-h`	Prints the command usage.
`-s`	Uses model simplification over the given model
`-S`	Enables the use of a further simplified FSM for each atomic part of the given SERE
`-R`	Enables the use of Implicit Frame Condition for each atomic part of the given SERE
`-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. The property is added to the Property Database.
`-n <index>`	Verifies the invariant with index "index" within the Property Database
`-P <name>`	Verifies the invariant named "name" within the Property Database
`-d`	Disables the reachability analysis completion. This means that only the strategy provided with the command is executed. The resulting set of reachable states is **not** guaranteed to be complete. This option makes invariant checking algorithm incomplete, therefore no "invariant is true" response can be given
`-f`	Forces the building of the BDD FSM. Use this option if using option -d and verifying generalized invariants
`-a`	Stop verification at the first property found false.
`-u`	Change the semantics of the ";" operator from SEQUENCE to UNION.
`-e "sere_expr"`	Provide the strategy from command line. This is an alternative to provide the strategy with an external file. In this case, the SERE formula must not begin with keyword 'grsequence'
`-i sere_file`	Provide the strategy from file. This is an alternative to provide the strategy with the -e option. The SERE expression must start with the 'grsequence' keyword and must end with a ";"

msat_check_invar_bmc - *Invariant property check with BMC* Command [I]

```
msat_check_invar_bmc [-h | -n idx | -p "formula" | -P "name"] [-d
"mathsat" | "smtlib"] [-o filename] [-a alg] [-i] [-k max_len] [-K
step_size] [-e]
```

Performs invariant checking with BMC.

Command Options:

`-h`	Shows a brief description of the available options
`-u`	Disables SMT solver invocation
`-n idx`	Checks the invariant (INVARSPEC) property specified with `idx`
`-p "formula"`	Checks the specified invariant property
`-P name`	Checks the invariant property with given name
`-a alg`	Uses the specified algorithm. Valid values are: • classic (it is k-induction with k=1) • een-sorensson • falsification • dual • zigzag • interp_seq • interpolants Default value is taken from variable `bmc_invar_alg`
`-k max_len`	Maximum bound for BMC instead of using the variable `bmc_length` value. Use only when een-sorensson, falsification, dual or zigzag algorithm is selected
`-K step_size`	Only for falsification: increment the search of `step_size` at a time. Must be greater than zero (1 by default).
`-i`	Use incremental version of falsification algorithm. Requires -a falsification
`-e`	Performs an extra step for finding a proof. Can be used only with the een-sorensson algorithm
`-o filename`	Instead of checking the property the SMT problems are dumped into file `filename` with an additional suffix. For example: **`'_bmc_classic'`** for classic algorithm, **`'_bmc_base_n'`** for een-sorensson base problem **`'_bmc_step_n'`** for een-sorensson step problem where 'n' is the length of a path taken into account
`-d`	Enables dump of the problems on the selected format.
`-f format`	Selects the dumping format. Valid values are: "smtlib1" and "smtlib2"

Notice that if no property is specified, checks all LTL properties.

check_invar_bmc_itp - *Interpolation based invariant verification algorithms* Command **[F]**

```
check_invar_bmc_itp [-h] | [-n idx | -p ``expr'' | -P ``name''] [-k
``bound''] [-a ``alg'']
```

Performs invariant checking using interpolants based BMC algorithms. If no property is specified, checks all INVAR properties.

IMPORTANT: This command does not accept mixed integer and real types in the model's FSM constraints.

Command Options:

`-h`	Shows a brief description of the available options.
`-n number`	Checks the property stored at the given index
`-P name`	Checks the property named `name` in the property database.
`-p "formula [IN context]"`	Checks the formula specified on the command-line. `context` is the module instance name which the variables in `formula` must be evaluated in.
`-k idx`	Sets the BMC bound limit to be used.
`-a alg`	Use the given algorithm for verification. Possible values are "mcmillan". "itp_seq", "mcmillan2", "itp_seq2", "avy", "falsification" (Default) and "itp_seq"

check_invar_ic3 - *Verifies invariant properties using IC3 engines* Command **[F,I]**

```
check_invar_ic3 [-h] [-d] [-i] [-O 0|1|2] [-a 0|1] [-u num] [-g] [-Y]
[-m num] [-v num] [-n number | -p "invar-expr" | -P "name"] [-k number]
```

Checks invariant properties using the ic3 engines (simplic3 or smt)

When the domain is infinite (or when forced explicitly with option -i), msatic3 library is used to check the property.

IMPORTANT: When the domain is infinite, the verification problem is in general undecidable and this command may fail in proving the property. In particular, it may not terminate or it may terminate with an unknown result when it cannot refine the abstraction (this may be due to the presence of mixed integer/real predicates).

Command Options:

`-h`	Shows a brief description of the available options.
`-d`	Disables the counterexample construction when proving false a property.
`-i`	Forces the use of the engine for infinite domains (msatic3)
`-O 0|1|2`	Only for finite: sets the preprocessing level (default: 2) **0** : No preprocessing. **1** : Enables sequencial preprocessing which searches equivalent latches and or constants **2** : Adds 2-step temporal decomposition to the preprocessor (default).
`-a 0|1`	If true, enable abstraction/refinement
`-u num`	Only for finite: perform property unrolling (i.e. target enlargement) for the given number of steps. Default 4.
`-g`	Only for finite: enables clause generalization, according to Zyad Hassan, Aaron R. Bradley, Fabio Somenzi, "Better Generalization in IC3", FMCAD'13
`-Y`	Invokes a portfolio of different algorithms in parallel. Takes precedence over all the other options. The variable `ic3.portfolio_exe` must be set to the name of the executable implementing the portfolio.

`-v num`	Enables verbosity. Default 0. Must be greater or equal to 0.
`-m num`	Only for finite: Max number of solvers to use for frames in IC3. Default 1. Must be greater or equal to 1.
`-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.
`-k number`	Sets the bound of IC3 to the given `number`.

check_invar_local - *Localized invariant checking* Command [F]

```
check_invar_local [-h] [-k] [-n idx | -p "expr" | -P "name"]
```

Performs invariant checking on the localized model. Model localization is performed before starting the checking process. This can greatly help in reducing resources (and time) required to check the property. Localization takes place using the property context as the input variable set.

Command Options:

`-h`	Shows a brief description of the available options.
`-n idx`	Checks the property stored at the given index
`-P ''name''`	Checks the property named `name` in the property database.
`-p ''expr''`	Checks the formula specified on the command-line.
`-k`	This flag enables recursive dependencies resolving when performing simplification, thus resulting in a stricter, behavior-preserving, approximation.

5.3.1 Incremental Cone Of Influence for Invariant Checking

In this section we list the commands for checking invariant properties that exploits abstraction based on incremental cone of influence reduction. The analysis starts considering the finite state transition corresponding to the cone of the property at distance 0. If it succeeds in proving the property holds, it terminates. Otherwise, the counter-example is analyzed to see if it corresponds to a concrete counter-example. If the counter-example can be concretized, then we are done: the property is violated. Otherwise, the cone is refined adding new variables until either the property has been proved or disproved.

check_invar_inc_coi_bdd - *BDD-based Incremental COI invariant checking* Command [F]

```
check_invar_inc_coi_bdd [-h] | [-n number | -p "invar-expr  [IN context]"
| -P "name"] [-I]
```

Performs invariant checking using the BDD-based Incremental COI algorithm. Invariants to be verified can be provided as simple formulas (Only temporal operator allowed is “next”) in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

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

Command Options:

`-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.
`-n "idx"`	Verifies the invariant with index "idx" within the Property Database
`-P "name"`	Verifies the invariant named "name" within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI

check_invar_inc_coi_bmc - *SAT-based Incremental COI invariant checking* Command **[F]**

```
check_invar_inc_coi_bmc [-h] | [-n number | -p "invar-expr [IN context]"
| -P "name"] [-I] [-k bound]
```

Performs invariant checking using the SAT-based Incremental COI algorithm. Invariants to be verified can be provided as simple formulas (Only temporal operator allowed is "next") in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

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

Command Options:

`-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.
`-n "idx"`	Verifies the invariant with index "idx" within the Property Database
`-P "name"`	Verifies the invariant named "name" within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-k ``bound''`	The bound to be used for SAT algorithms

msat_check_invar_inc_coi - *SMT-based Incremental COI invariant checking* Command **[I]**

```
msat_check_invar_inc_coi [-h] | [-n number | -p "invar-expr [IN
context]" | -P "name"] [-I] [-u] [-i] [-k bound]
```

Performs invariant checking using the SMT-based Incremental COI algorithm. Invariants to be verified can be provided as simple formulas (Only temporal operator allowed is "next") in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

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

IMPORTANT: In current implementation options `-i` and `-u` are disabled, and an error is reported to the user when used.

IMPORTANT: When using interpolation (option `-i`), integer and real types in the model's FSM constraints cannot be mixed.

Command Options:

`-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.

`-n idx`	Verifies the invariant with index “idx” within the Property Database
`-P "name"`	Verifies the invariant named “name” within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-u`	Use unsat-cores variables for refinement (unsupported yet)
`-i`	Use interpolants variables for refinement (unsupported yet)
`-k bound`	The bound to be used for SMT algorithms

check_invar_inc_coi - *Incremental COI invariant checking* Command **[F,I]**

```
check_invar_inc_coi [-h] | -v eng -e eng [-n number | -p "invar-expr  [IN
context]" | -P "name"] [-I] [-u] [-i] [-k bound]
```

Performs invariant checking using the Incremental COI algorithm. Invariants to be verified can be provided as simple formulas (Only temporal operator allowed is “next”) in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

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

IMPORTANT: In current implementation options `-i` and `-u` are disabled, and an error is reported to the user when used.

IMPORTANT: For SMT, when using interpolation (option `-i`), integer and real types in the model’s FSM constraints cannot be mixed.

Command Options:

`-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.
`-n "idx"`	Verifies the invariant with index “idx” within the Property Database
`-P "name"`	Verifies the invariant named “name” within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-u`	Available only with SMT: Use unsat-cores variables for refinement
`-i`	Available only with SMT: Use interpolants variables for refinement
`-k bound`	The bound to be used for SAT / SMT algorithms
`-v "eng"`	Specifies the engine to be used for verification. Can be ”bdd”, ”sat” or ”smt”
`-e "eng"`	Specifies the engine to be used for traces execution. Can be ”bdd”, ”sat” or ”smt”

5.4 Commands for LTL Model Checking

In this section we list the new commands for checking LTL properties. All these commands can be applied both to finite and infinite-state models (e.g. “ic3_check_ltlspec”).

msat_check_ltlspec_bmc - *LTL property check with BMC* Command **[I]**

```
msat_check_ltlspec_bmc [-h | -n idx | -p "formula" | -P "name"] [-k
max_length] [-l loopback] [-d mathsat|smtlib] [-o filename]
```

Performs LTL checking with BMC. Currently, past operators and option bmc_force_pltl_tableau are not supported.

Command Options:

`-h`	Shows a brief description of the available options
`-u`	Disables SMT solver invocation
`-t max_timespan`	Shows theory lemmas up to max_timespan
`-n idx`	Checks the LTL property specified with `idx`
`-p "formula"`	Checks the specified LTL property
`-P name`	Checks the LTL property with given name
`-k max_length`	Maximum bound for BMC instead of using the variable `bmc_length` value
`-l loopback`	Checks the property using `loopback` value instead of using the variable `bmc_loopback value`
`-o filename`	Uses `filename` as file to dump the generated problem. `filename` may contain patterns
`-d`	Enables dump of the problems on the selected format.
`-f format`	Selects the dumping format. Valid values are: “smtlib1” and “smtlib2”

Notice that if no property is specified, checks all LTL properties.

msat_check_ltlspec_sbmc_inc - *LTL property check with Incremental SBMC* Command **[I]**

```
msat_check_ltlspec_sbmc_inc [-h | -n idx | -p "formula" | -P "name"] [-k
max_length] [-N] [-c]
```

Performs LTL incremental checking with SBMC.

Currently, past operators and option bmc_force_pltl_tableau are not supported.

Command Options:

`-h`	Shows a brief description of the available options
`-n idx`	Checks the LTL property specified with `idx`
`-p "formula"`	Checks the specified LTL property
`-P name`	Checks the LTL property with given name
`-k max_length`	Maximum bound for BMC instead of using the variable `bmc_length` value
`-N`	Does not perform virtual unrolling
`-c`	Performs completeness check

Notice that if no property is specified, checks all LTL properties.

check_ltlspec_ic3 - *Verifies LTL properties using ic3 engines, either with K-Liveness or with (abstract) liveness-to-safety transformation* Command **[F,I]**

```
check_ltlspec_ic3 [-h] [-d] [-m num] [-i] [-Y] [-u num] [-g] [-O [0|1]]
[-e] [-E num] [-L] [-v num] [-a 0|1] [-n number | -p "ltl-expr" | -P
"name"] [-k number] [-l number] [-K 0|1]
```

Checks LTL properties using the ic3 engines (simplic3 or smt), either with the K-Liveness algorithm or with (abstract) liveness-to-safety transformation.

When the domain is infinite (or when forced explicitly with option -i), msatic3 library is used to check the property.

IMPORTANT: When the domain is infinite, the verification problem is in general undecidable and this command may fail in proving the property. In particular, it may not terminate or it may terminate with an unknown result when it cannot refine the abstraction (this may be due to the presence of mixed integer/real predicates).

Command Options:

`-h`	Shows a brief description of the available options.
`-d`	Disables the counterexample construction when proving false a property.
`-i`	Forces the use of the engine for infinite domains (msatic3)
`-O 0|1`	Only for finite: sets the preprocessing level (default: 1) **0** : No preprocessing. **1** : Enables sequencial preprocessing which searches equivalent latches and or constants (default).
`-u num`	Only for finite: perform property unrolling (i.e. target enlargement) for the given number of steps. Default 4.
`-g`	Only for finite: enables clause generalization, according to Zyad Hassan, Aaron R. Bradley, Fabio Somenzi, "Better Generalization in IC3", FMCAD'13
`-Y`	Invokes a portfolio of different algorithms in parallel. Takes precedence over all the other options. The variable `ic3.portfolio_exe` must be set to the name of the executable implementing the portfolio.
`-e`	Only for finite: enables extraction additional liveness stabilizing constraints in preprocessing generalization.
`-E num`	Only for finite: number of candidates to consider for liveness constraint extraction.(0: disabled, 3000 default).
`-L`	Disables complementation of k-liveness with BMC (if disabled no counterexample can be computed).
`-m`	Only for finite: max number of solvers to use for frames in IC3. Default 1. Must be greater or equal to 1.
`-a 0|1`	If true, enable abstraction/refinement (only for infinite-state models).
`-v <num>`	Enables verbosity. Default 0. Must be greater or equal to 0.
`-n number`	Checks the LTL property with index `number` in the property database.
`-p "invar-expr [IN context]"`	The command line specified LTL formula to be verified. `context` is the module instance name which the variables in `invar-expr` must be evaluated in.
`-P name`	Checks the LTL property named `name` in the property database.
`-k number`	Sets the bound of IC3 to the given `number`.
`-l number`	Sets the bound of K-Liveness to the given `number`.
`-K 0|1`	Select between K-Liveness (-K 1, default) or liveness-to-safety (-K 0).

check_ltlspec_simplify - *LTL model checking using simplifications* Command **[F]**

```
check_ltlspec_simplify [-h] [-n index | -p ``prop'' | -P ``name''] [-s]*
[-r]*
```

Performs model checking of LTL formulas. LTL model checking is reduced to CTL model checking as described in the paper by [CGH97a].

The model on which the model checking is performed is simplified using the Model Simplifier and the Range Reduction systems.

By default, Model Simplification and Range Reduction are used, but a chain of simplifications to be performed over the model can be specified using the `-s` and the `-r` command options.

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

Command Options:

`-p ''prop''`	An LTL formula to be checked.
`-P "name"`	Checks the LTL property named "name"
`-n index`	Checks the LTL property with index `index` in the property database.
`-s`	Adds Model Simplification to the chain of simplifications. This option can be used multiple times
`-r`	Adds Range Reduction to the chain of simplifications. This option can be used multiple times

5.4.1 Incremental Cone Of Influence for LTL Model Checking

In this section we list the commands for checking LTL properties that exploits abstraction based on incremental cone of influence reduction. Similarly to the case of verification of invariants, the analysis starts considering the finite state transition corresponding to the cone of the property at distance 0. If it succeeds in proving the property holds, it terminates. Otherwise, the counter-example is analyzed to see if it corresponds to a concrete counter-example. If the counter-example can be concretized, then we are done: the property is violated. Otherwise, the cone is refined adding new variables until either the property has been proved or disproved.

check_ltlspec_inc_coi_bdd - *BDD-based Incremental COI LTL properties checking* Command **[F]**

```
check_ltlspec_inc_coi_bdd [-h] | [-n number | -p "ltl-expr  [IN context]"
| -P "name"] [-I]
```

Performs LTL checking using the BDD-based Incremental COI algorithm. LTL properties to be verified can be provided as LTL formulas in the input file via the `LTLSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` or `-P` can be used for checking a particular LTL property of the model. If neither `-n` nor `-p` nor `-P` are used, all the LTL properties are checked.

Command Options:

`-p "ltl-expr [IN context]"`	The command line specified LTL formula to be verified. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-n number`	Verifies the LTL property with index number within the Property Database
`-P "name"`	Verifies the LTL property named "name" within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI

check_ltlspec_inc_coi_bmc - *SAT-based Incremental COI LTL properties checking* Command **[F]**

```
check_ltlspec_inc_coi_bmc [-h] | [-n number | -p "ltl-expr  [IN context]"
| -P "name"] [-I] [-k bound]
```

Performs LTL checking using the SAT-based Incremental COI algorithm. LTL properties to be verified can be provided as LTL formulas in the input file via the `LTLSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` or `-P` can be used for checking a particular LTL property of the model. If neither `-n` nor `-p` nor `-P` are used, all the LTL properties are checked.

Command Options:

`-p "ltl-expr [IN context]"`	The command line specified LTL formula to be verified. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-n number`	Verifies the LTL property with index number within the Property Database
`-P "name"`	Verifies the LTL property named “name” within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-k bound`	The bound to be used for SAT algorithms

msat_check_ltlspec_inc_coi - *SMT-based Incremental COI LTL properties checking* Command **[I]**

```
msat_check_ltlspec_inc_coi [-h] | [-n number | -p "ltl-expr  [IN
context]" | -P "name"] [-I] [-u] [-i] [-k bound]
```

Performs LTL checking using the SMT-based Incremental COI algorithm. LTL properties to be verified can be provided as LTL formulas in the input file via the `LTLSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` or `-P` can be used for checking a particular LTL property of the model. If neither `-n` nor `-p` nor `-P` are used, all the LTL properties are checked.

IMPORTANT: In current implementation options `-i` and `-u` are disabled, and an error is reported to the user when used.

IMPORTANT: When using interpolation (option `-i`), integer and real types in the model’s FSM constraints cannot be mixed.

Command Options:

`-p "ltl-expr [IN context]"`	The command line specified LTL formula to be verified. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-n idx`	Verifies the LTL property with index “idx” within the Property Database
`-P "name"`	Verifies the LTL property named “name” within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-u`	Use unsat-cores variables for refinement
`-i`	Use interpolants variables for refinement
`-k bound`	The bound to be used for SMT algorithms

check_ltlspec_inc_coi - *Incremental COI LTL properties checking* Command **[F,I]**

```
check_ltlspec_inc_coi [-h] | -v eng -e eng [-n number | -p "ltl-expr  [IN
context]" | -P "name"] [-I] [-u] [-i] [-k bound]
```

Performs LTL checking using the Incremental COI algorithm. LTL properties to be verified can be provided as LTL formulas in the input file via the `LTLSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` or `-P` can be used for checking a particular LTL property of the model. If neither `-n` nor `-p` nor `-P` are used, all the LTL properties are checked.

IMPORTANT: In current implementation options `-i` and `-u` are disabled, and an error is reported to the user when used.

IMPORTANT: For SMT, when using interpolation (option `-i`), integer and real types in the model's FSM constraints cannot be mixed.

Command Options:

`-p "ltl-expr [IN context]"`	The command line specified LTL formula to be verified. `context` is the module instance name which the variables in `ltl-expr` must be evaluated in.
`-n number`	Verifies the LTL property with index number within the Property Database
`-P "name"`	Verifies the LTL property named "name" within the Property Database
`-I`	Execute traces over increasing size FSMs, based on Incremental COI
`-u`	Available only with SMT: Use unsat-cores variables for refinement
`-i`	Available only with SMT: Use interpolants variables for refinement
`-k bound`	The bound to be used for SAT / SMT algorithms
`-v "eng"`	Specifies the engine to be used for verification. Can be "bdd", "sat" or "smt"
`-e "eng"`	Specifies the engine to be used for traces execution. Can be "bdd", "sat" or "smt"

5.4.2 Compositional Reasoning for LTL Model Checking

Compositional model checking is a verification method that aims at reducing the verification problem of large systems to smaller, possibly localized verification problems. This is done in order to try to avoid the "state explosion problem". When reasoning compositionally about two systems A and B, it is often necessary to assume the correctness of A to verify B, and vice-versa. In the literature this "apparent" circularity has been resolved by induction over time. The induction over time is made explicit by assuming that a property P *only* up to time $t-1$ when proving Q at time t, and vice-versa. The proof obligations incurred using this method can be discharge with model checking. This approach has been described in [McM99]. In this section we describe the commands for performing the above outlined compositional reasoning.

check_ltlspec_compositional - *Circular compositional reasoning model checking* Command **[F,I]**

```
check_ltlspec_compositional [-h] | -f ``proof-file'' [-n ``node''] [-t
``check-technique'']
```

Performs circular compositional reasoning model checking as described in [McM99] and [PK99].

An assertion is a condition that must hold true in every possible execution of the program. Assertions refer to properties in LTL.

In order to apply the circular compositional rule, one has to supply the set of assertions to be proved and the proof graph. From these, a sufficient set of proof obligations on the form of LTL formulas are built.

A property is specified using the following syntax:

$name : assert\langle formula \rangle;$

When specifying the proof graph, properties are refererred to by their names. An arc (p1, p2) in the proof graph is specified as follows:

using n1 prove n2;

where n1 and n2 are the respective names of properties p1 and p2. A list of assumption can also be used when verifying a property, specifying a comma-separated list of assumptions.

using n1, n2, n3 prove n4;

Such a "proof" may not contain circular chains of reasoning. The system verifies that every cycle in the proof graph is cut by a unit delay arc. A unit delay arc is specified by putting the assumption in parentheses, as follows:

using (n1) prove n2;

Command Options:

`-t technique`	Use the specified technique to perform model checking. Valid techniques are {bdd, bmc, smt}
`-n node`	Perform the check only for the specified node instead of checking all the nodes
`-f proof-file`	Reads the proof graph from the specified file

5.5 Commands for Requirements Analysis

NUXMV provides commands for supportin requirements analysis. In particular, it provides commands for checking the consistency of a set of requirements, checking whether a set of requirements is consistent with another requirement, and finally to check whether a set of requirement entails another requirement.

reqan_check_consistency - *Checks consistency of a set of requirements.* Command **[F,I]**

```
reqan_check_consistency[-h] [-i] [-e bdd|sat|msat|ic3] [-t id] -r
"props"
```

Checks the consistency of the provided set of requirements specified by their property index.

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Ignores the user's FSM (takes only the language).
`-e engine`	Uses the given engine, i.e. `bdd` `sat`, `msat`, `ic3` (default: `sat`).
`-t id`	Specify the time port for kzeno algorithm (default=none).
`-r "props"`	A subset of properties given as indices. Indices are separated by comma ',' or colon ':'. Ranges are allowed where lower and upper bounds are separated by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8.

reqan_check_possibility - *Checks consistency of a possibility with a set of requirements.* Command **[F,I]**

```
reqan_check_possibility[-h] [-i] [-e bdd|sat|msat|ic3] [-t id] [-r
"props"] -p idx
```

Checks the consistency of the provided set of requirements specified by their property index and a given possibility (also specified with an index).

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Ignores the user's FSM (takes only the language).
`-e engine`	Uses the given engine, i.e. `bdd` `sat`, `msat`, `ic3` (default: `sat`).
`-t id`	Specify the time port for kzeno algorithm (default=none).
`-r "props"`	A subset of properties given as indices. Indices are separated by comma ',' or colon ':'. Ranges are allowed where lower and upper bounds are separated by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8.
`-p idx`	Index of a possibility.

reqan_check_assertion - *Checks the assertion w.r.t. of a set of requirements.* Command **[F,I]**

```
reqan_check_assertion[-h] [-i] [-e bdd|sat|msat|ic3] [-t id] [-r
"props"] -p idx
```

Checks whether the set of requirements entails the assertion.

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Ignores the user's FSM (takes only the language).
`-e engine`	Uses the given engine, i.e. `bdd` `sat`, `msat`, `ic3` (default: `sat`).
`-t id`	Specify the time port for kzeno algorithm (default=none).
`-r "props"`	A subset of properties given as indices. Indices are separated by comma ',' or colon ':'. Ranges are allowed where lower and upper bounds are separated by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8.
`-p idx`	Index of an assertion.

5.6 Commands for Computing Reachable States

compute_reachable_guided - *Guided reachability set of reachable states* Command **[F]**

```
compute_reachable_guided [-s] [-S] [-R] [-P] [-d] [-u] (-h | -e
"sere_expr" | -i sere_file)
```

Computes the set of reachable states of the given model using Guided Reachability algorithm over the given strategy.

If the set of reachable states has already been computed, the command returns immediately since there is nothing more to compute.

The resulting reachable states are globally stored and used to simplify the execution of model checking commands (e.g. check_invar). This can result in improved performance on models with sparse state spaces. The exploration DAG is not stored, so exploration resuming is not allowed.

By default GR performs the computation of reachable states in two steps. At first, states satisfying the strategy are computed until fixpoint is reached. Then, starting from the previous fixpoint states, the image computation is applied regardless of the strategy until the global fixpoint, i.e. until all the reachable states are detected.

If option `-d` is used, it is not possible to mark the reached states as complete and to store them globally, since it is not implemented any completeness checking on them.

The strategy must be a valid PSL formula. Allowed PSL operators are: "; $[*]$ $[*N](withN > 0)$ |"

Command Options:

`-h`	Prints the command usage.
`-s`	Performs syntactic simplification on the given model.
`-S`	Performs syntactic simplification on the FMSs related to each disjunct specified in the given SERE.
`-R`	Enables the use of Implicit Frame Conditions during the reachability analysis.
`-d`	Disables the reachability analysis completion. This means that only the strategy provided with the command is executed and the original FSM is not used to discover the possible unreached states. The resulting set of reachable states is **not** guaranteed to be complete.
`-u`	Changes the semantics of the ";" operator from SEQUENCE to UNION.
`-P`	Enables command profiling. The resulting time values are valid only if a verbose level lower or equal than 2 is specified.
`-e "sere_expr"`	Provides the strategy from command line. In this case, the SERE formula must not begin with keyword 'grsequence'. This is an alternative to provide the strategy with an external file (-i option).
`-i sere_file`	Provides the strategy from file. The SERE expression must start with the 'grsequence' keyword and must end with a ";". This is an alternative to provide the strategy from command line (-e option).

5.7 Commands for Reasoning via Abstraction

Predicate abstraction is a technique that is used to prove properties of finite- and infinite-state systems. It is a combination of theorem proving and model checking techniques. Given a concrete finite- or infinite-state system and a set of predicates, a conservative finite state abstraction is generated. (For every execution in the concrete system there is a corresponding execution in the abstract system.) The abstract version of the verification condition is model checked in this abstract system. If the property is verified then it holds in the concrete system. Otherwise an abstract counter-example trace is generated. There could be a concrete counter-example corresponding to it, in which case there is a bug in the design, or the abstract counter-example is an artifact of the abstraction. The counter-example can be analyzed to find a real bug or to suggest extra predicates to refine the abstraction thereby avoiding that particular spurious trace. Then the process starts anew. The abstraction refinement process is guaranteed to terminate for finite-state systems (if resources permits). However, the process is not guaranteed to terminate for infinite-state systems: proving arbitrary (safety) properties of an infinite state system is not decidable, but for a large number of problems this method can successfully prove properties.

In NUXMV we provide two complementary approaches, both based on predicate abstraction. The first relies on the explicit computation of the abstract transition system. The second, uses an implicit abstraction to avoid the expensive computation of the abstract transition system.

5.7.1 Explicit Predicate Abstraction

These commands implements the functionalities for performed Counterexample Guided predicate Abstraction Refinement (CEGAR) [CGJ+03]. The CEGAR approach requires the computation of a quantifier-free formula that is equivalent to the abstract transition relation w.r.t. a given set of predicates. This, in turn, requires the solving of an ALLSAT problem [LNO06]. For this step, NUXMV implements different techniques: the combination of BDD and SMT [CCF+07, CFG+10], where BDDs are used as compact Boolean model enumerator within an ALLSMT approach; the technique that exploits the structure of the system under verification, to partition the abstraction problem into the combination of several smaller abstraction problems [CDJR09]. For the refinement step to discard the spurious counterexample, NUXMV implements three approaches based on the analysis of the unsatisfiable core, on the analysis of the interpolants, and on the weakest preconditions..

The command "config_abstraction" sets the options that control how the abstraction is performed. The command "add_abstraction_preds" allows to specify the predicates to be used for CEGAR. The command "build_abstract_model" computes the abstract transition system w.r.t. the specified set of predicates. Then, the model can be dumped into a file with the command "write_abstract_model". The command "check_invar_cegar_predabs" performs the CEGAR loop checking the given property.

config_abstraction - *Configures the options for abstraction computation* Command

```
config_abstraction [-h] [-e <output>] [-d <output>] [-a <engine>] [-c
(0|1)] [-t (0|<number>)] [-b (0|1)] [-s]
```

This command sets the options for abstraction computation using the `build_abstract_model` command.

Command Options:

`-h`	Shows the online help message.
`-a engine`	The abstraction engine to be used. The parameter `engine` can be `msat` (ALLSMT) or `bdd` [CFG+10] or `bddarray` [CFG+10] or `bool` or `structural` [CDJR09].
`-e output`	Enable the given output. The output can be either `bdd` or `sexp` or `boolsexp`.
`-d output`	Disable the given output. The output can be either `bdd` or `sexp` or `boolsexp`.
`-c 0 | 1`	Disable/Enable D'Agostino optimization (bddarray only).
`-t 0 | <number>`	Disable/Set the threshold limit (bddarray only).
`-b 0 | 1`	Disable/Enable backjumping (bddarray only).
`-s`	Shows the updated configuration.

add_abstraction_preds - *Extracts the abstraction precision from a model or from an external file* Command **[I]**

```
add_abstraction_preds [-h] [-a | -p number | -m number | -i file] [-o
file] [-s]
```

Extracts and/or shows the abstraction precision. The precision is the set of mirror variables and predicates to be used in the abstraction. The precision can be specified in two ways. First, mirrors and predicates can be declared in the model using the `MIRROR` and `PRED` keywords. A second possibility is to specify the list of predicates and mirrors in a separate file and use the `-i` option. The file format is the following. The file starts with the string `PREDICATES` followed by an arbitrary list of either `PRED [predicate]` or `MIRROR [mirror variable]`. The extracted precision is then used in the commands `build_abstract_model` and `check_invar_cegar_predabs`.

Command Options:

`-h`	Shows the online help message.
`-a`	Adds all predicates and mirrors from model.
`-p number`	Adds the predicate with the specified number from model.
`-m number`	Adds the mirror with the specified number from model.
`-i file`	Reads the precision specified in the given file.
`-o file`	Writes the precision extracted so far in the specified file.

`-s`	Shows the precision extracted so far.

abstraction_use_expression_as_predicate_name Environment Variable

If set to true generates a name for "unamed predicates" that corresponds to the string of the predicate. For instance for `PRED x = y + 1` it will be internally generated `_____a_p_1` if this variable is `0`, otherwise if the value is `1`, it will be generated the name `"x = y + 1"`. Default value is `0`.

build_abstract_model - *Computes the abstraction* Command [I]

```
build_abstract_model [-h]
```

This command computes and internally stores the abstraction of the model given the precision extracted using `add_abstraction_preds` with the options set by `config_abstraction`. The generated FSM can be dumped using `write_abstract_model` and disposed using `quit_abstraction`.

Command Options:

`-h`	Shows the online help message.

quit_abstraction - *Computes the abstraction* Command [I]

```
quit_abstraction [-h]
```

This command disposes the FSM computed by `build_abstract_model` in order to allow for another one to be generated (with different options or with a different precision).

Command Options:

`-h`	Shows the online help message.

write_abstract_model - *Dumps the abstracted FSM to a file* Command [I]

```
write_abstract_model [-h] <filename>
```

This command dumps the FSM computed by `build_abstract_model` to the given filename.

Command Options:

`-h`	Shows the online help message.

check_invar_cegar_predabs - *Checks a property using CEGAR* Command [I]

```
check_invar_cegar_predabs [-h] [-n <num> | -P <name> | -p <formula>] [-l
<num>]
```

This command sets performs the model checking of a property using the CEGAR loop. The options for the abstraction phase can be set using `config_abstraction` command. The property to be checked can be specified by its number (`-n` parameter), by its name (`-P` parameter) or by entering an invariant formula (`-p` option). The number of abstraction-refinement steps is bounded by the `-l` option and by default it is unlimited It is possible to specify predicates and mirrors to be used in all the abstractions cycles by using the `add_abstraction_preds` command.

Command Options:

`-h`	Shows the online help message.
`-n num`	The number of the property to be checked.
`-P name`	The name of the property to be checked.
`-p formula`	Adds an invariant property with the given formula and checks it.
`-l num`	Bounds the number of cycles to num.

5.7.2 Implicit Predicate Abstraction

We also complement the CEGAR based predicate abstraction algorithms with new algorithms that combine abstraction with BMC and k-induction [Ton09]. The algorithms do not rely on quantifier elimination techniques to compute the abstraction, but encode the model checking problem over the abstract state space into SMT problems. The advantage, is that they avoid the possible bottleneck of abstraction computation.

msat_check_invar_bmc_implabs - *Verifies invariant properties using BMC in combination with Abstraction* Command **[I]**

```
msat_check_invar_bmc_implabs [-h] [-k bound] [-a] (-n prop_index | -P
prop_name | -p formula)
```

Checks invariant properties using bounded model checking with k-induction in combination with abstraction w.r.t. a given set of predicates specified in the input file. The technique that combines abstraction and k-induction has been described in [Ton09].

Command Options:

`-h`	Shows a brief description of the available options.
`-k bound`	Specifies the maximum bound for bounded model checking and k-induction. Default value is the one specified by variable `bmc_length`. If value 0 is used, the algorithms continues to increment k until either the property has been proved, or a counterexample has been found, or the resources are exhausted.
`-n num`	Specifies the id of an invariant property. If the id does not corresponds to an invariant property, then an error is issued.
`-P prop_name`	Specifies the name of an invariant property. If the name does not corresponds to an invariant property, then an error is issued.
`-p formula`	Specifies an invariant property. It will be added to the property database.
`-a`	Disables the use of abstraction, and simply performs the verification using the standard bounded model checking algorithms. The specification of one invariant property is mandatory.

msat_check_invar_bmc_cegar_implabs - *Implicit abstract model checking* Command **[I]**

```
msat_check_invar_bmc_cegar_implabs [-h] | [-k bound] [-r meth] [-i meth]
[-s] [-m] [-d] [-c] [[-n prop] | [-p "invar"] | [-P name]]
```

Performs invariant checking with implicit abstract model checking as described in [Ton09]. This technique does not compute the abstraction explicitly like it is the case in classical Counterexample Guided Abstraction Refinement (CEGAR). This would save resources (memory and time) to check the property. If neither of -o, -p, or -P is specified, the command tries to

Command Options:

`-h`	Shows a brief description of the available options.
`-k bound`	Specifies the maximum bound for bounded model checking and k-induction. Default value is the one specified by variable `bmc_length`. If value 0 is used, the algorithms continues to increment k until either the property has been proved, or a counterexample has been found, or the resources are exhausted.
`-r meth`	Refinement method for generating new predicates. (a:automatic (default), m:manual, h: hybrid). automatic : Ignore predicates in smv file and generate new predicates manual : Take predicates in smv file and do not generate any new predicates hybrid : Take predicates in smv file and generate new predicates
`-i meth`	K-induction method for termination condition. (f:full(BW+FW) (default), b:backward only, n:none). full : Check backward induction first and then forward induction; backward : Only check backward condition; none : None. This is fully Abstract BMC. Program will only terminate if it reaches the bound k or find a bug.
`-s`	Incrementally add simulation condition.
`-m`	Tries to minimize the number of predicates added during the search.
`-d`	Enables for the discovery of invariants during the search.
`-c`	Enables for the fresh restart after each refinement step.
`-n prop`	Checks the property stored at the given index, assuming it is an invariant. If it is not an invariant, an error is issued.
`-p ``invar'' [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.

5.8 Commands for Format Conversions

This subsection contains commands for converting the NUXMV format into other external formats.

5.8.1 Commands for aiger 1.9.4 format support

aiger 1.9.4 is a format, library and set of utilities for And-Inverter Graphs (AIGs) [BHW11]. The aiger 1.9.4 format has an ASCII and a binary version. As described in the documentation of aiger 1.9.4 [BHW11], the ASCII version is the format of choice if an AIG is to be saved by an application which does not want to use the aiger 1.9.4 library. We refer the reader to the aiger 1.9.4 [BHW11] documentation for details about the format.

In this section we describe the commands for reading the aiger 1.9.4 format (both in the ASCII and in the binary formats). Moreover, we also describe the command to dump a NUXMV model without Reals and Integers into a set of aiger 1.9.4 files (one for each property).

read_aiger_model - *Reads and loads an aiger model* Command **[F]**

`read_aiger_model [-h] | [-i filename] [-r] [-m]`

The command imports a model in aiger 1.9.4 format into NUXMV. The loaded model can then be used as any other input file in the extended language accepted by the tool.

If the aiger file is in the aiger 1.9.4 format with both bad, justice, and fairness, the the outputs (if any) are not considered as properties. Each fairness conditions f_i is transformed in `FAIRNESS` f_i. Each bad conditions b_i is transformed in `INVARSPEC` $!b_i$. Each justice constraint $J_i = j_0^i, \ldots, j_n^i$ is transformed in `LTLSPEC` $!\bigwedge_{j_k^i \in J_i} GF_k^i$.

On the other hand, if the input file does not contain neither fairness, nor justice and nor bad, than if there is a single output o_0 it is interpreted as an `INVARSPEC` $!o_0$. Otherwise, if more than one output is specified, i.e. $O = o_0, \ldots, o_n$, each output o_i is interpreted as a GFo_i and the corresponding property is `LTLSPEC` $!\bigwedge_{o_i \in O} GFo_i$.

Command Options:

`-h`	Shows a brief description of the available options.
`-i filename`	Reads and loads the model from "input-file".
`-r`	Builds a relational hierarchy instead of building a functional one, i.e. uses INIT / TRANS instead of ASSIGN
`-m`	Uses monitor variables recognition, which tries to detect which variables in the aiger model are monitor/support variables, and on success, slightly reduces the number of variables and simplifies the Transition Relation. This should work almost every time when loading "write_aiger_model"-generated files.

write_aiger_model - *Dump of the current model in aiger format* Command **[F]**

```
write_aiger_model [-h] | -p "prefix" | -f "output" [-i | -l] [-n index]
[-b] [-d path]
```

Dumps the currently loaded model in aiger format using the aiger 1.9.4 format. Input format before aiger 1.9.4 is only supported for reading (see `read_aiger_model` for details).

If *-p "prefix"* is specified, a various number of aiger 1.9.4 files is generated, one for each property of kind LTL or INVARSPEC in the property database. (Other types of properties are not supported.) Each file will be named "*prefix_proptype_propidx*.[aig|aag]". Each generated file represents a model checking instance for the corresponding property.

If option *-f "output"* is specified instead, one file named "output" will be generated, which represents the Finite State Machine of the input model only without properties.

Command Options:

`-h`	Shows a brief description of the available options.
`-b`	Dumps the output files in binary format instead of ASCII format
`-i`	Dumps models only for invariant properties
`-l`	Dumps models only for LTL properties.
`-n index`	Dumps models only for the property at index n.
`-d path`	The directory where to save files. Default is "."
`-p prefix`	Dumps one model foreach property. Each generated file will be named "*prefix_proptype_propidx*.[aig|aag]"

5.8.2 Commands for VMT format support

The VMT format is an extension of the SMT-LIBv2 [BST12] (SMT2 for short) format to represent symbolic transition systems.

VMT exploits the capability offered by the SMT2 language of attaching *annotations* to terms and formulas in order to specify the components of the transition system and the properties to verify. More specifically, the following annotations are used:

`:next` *name* is used to represent state variables. For each variable x in the model, the VMT file contains a pair of variables, x^c and x^n, representing respectively the current and next version of x. The two variables are linked by annotating x^c with the attribute `:next` x^n. All the variables that are not in relation with another by means of a `:next` attribute are considered inputs.

`:init true` is used to specify the formula for the initial states of the model. This formula should contain neither next-state variables nor input variables. (The "dummy" value `true` in the annotation is needed because the current SMT2 standard requires annotations to always have an associated value.)

`:trans true` is used to specify the formula for the transition relation.

`:invar-property` *idx* is used to specify invariant properties, i.e. formulas of the form Gp, where p is the formula annotated with `:invar-property`. The non-negative integer *idx* is a unique identifier for the property.

`:live-property` *idx* is used to specify an LTL property of the form FGp, where p is the formula annotated with `:live-property`. The non-negative integer *idx* is a unique identifier for the property.

In a VMT file, only annotated terms and their sub-terms are meaningful. Any other term is ignored. Moreover, only the following commands are allowed to occur in VMT files: `set-logic`, `set-option`, `declare-sort`, `define-sort`, `declare-fun`, `define-fun`.(For convenience, an additional `(assert true)` command is allowed to appear at the end of the file.)
The following example shows a simple NUXMV model (left) and its corresponding VMT translation (right).

NUXMV	VMT

`-- this is a comment`
`MODULE main`
`VAR x : integer;`
`INIT x = 1;`
`TRANS next(x) = x + 1;`
`INVARSPEC x > 0;`
	`; this is a comment`
	`(declare-fun x () Int)`
	`(declare-fun xn () Int)`
	`(define-fun .sv0 () Int (! x :next xn))`
	`(define-fun .init () Bool (! (= x 1) :init true))`
	`(define-fun .trans () Bool (! (= xn (+ x 1)) :trans true))`
	`(define-fun .p0 () Bool (! (> x 0) :invar-property 0))`

Since the SMT2 format (and thus also the VMT one that inherits from SMT2) does not allow to annnotate the declaration of variables, it is a good practice to insert immediately after the declaration of the variables a set of defines to specify the relations among variables. See for instance the define `.sv0` in the example above that introduces the relation between `x` and `xn`.
In the distribution of the NUXMV (within directory `contrib`), we also provide conversion scripts from other formats (e.g. the from the BTOR language of Boolector [Boo] to the language of the NUXMV and vice-versa.

write_vmt_model - *Dumps the model with a single property in VMT format* Command **[I]**

```
write_vmt_model [-h] [-o filename]
[-n prop_number | -i "invar_expr" | -l "ltl_expr"]
```

Dumps the model with the specified property in VMT format. VMT is an extension of the SMT-LIBv2 format for specifying fair symbolic transition systems, and for specifying properties over the transition system.

Command Options:

`-h`	Shows a brief description of the available options.
`-o filename`	The filename where to dump the model and the possible LTL or invariant property.

`-n prop_number`	Index in the property database of the property to dump. Only invariants and LTL properties are supported.
`-i "invar_expr"`	An expression specifying the invariant property to dump. The property is also added to the property database.
`-l "ltl_expr"`	An expression specifying the LTL property to dump. The property is also added to the property database.

5.9 Commands for Model Transformation

In this section we report a set of commands that could be used to generate simplified models, and to explore the model e.g. generating XMI format.

5.9.1 Commands for Model Simplification

write_simplified_model_rel - *Model simplification* Command **[I]**

```
write_simplified_model_rel [-h] [-o filename] [-e ``expr'']* [-l
``prop'']* [-i ``prop'']* [-c ``prop'']* [-I | -D] [-r]
```

Writes the currently loaded SMV model in the specified file, after having flattened and simplified it. INVARs and ASSIGNs are taken as assumptions for simplification. Those expressions are processed in order to find strong dependencies between variables, so that some of them can be simply removed, reducing the space search. For example, having "INVAR a = b" means that all occurrences in the input model of one of the two variables involved in the expression can be replaced with the other one. Also inlining is applied as much as possible in order to reduce the expressions size.

During simplification, processes are eliminated and equivalent structures are set up.

If no file is specified, resulting simplified flat model is dumped to standard output.

Command Options:

`-h`	Shows a brief description of the available options.
`-o filename`	Attempts to write the simplified SMV model in `filename`
`-e expr`	Adds the given assumption to the simplifier
`-l prop`	Adds the given LTL property to the simplifier
`-i prop`	Adds the given INVAR property to the simplifier
`-c prop`	Adds the given CTL property to the simplifier
`-D`	Does not add any define declaration to the output model
`-I`	Disable defines and array defines inlining
`-r`	Disable properties rewriting

write_simplified_model_func - *Model simplification* Command **[I]**

```
write_simplified_model_func [-h] | [-o filename] [-d] [-D]
```

Writes the currently loaded SMV model in the specified file, after having flattened and simplified it. Assignments of the form A := B, in which A is a variable and B is a constant or a variable are transformed into defines, thus reducing the state space of the model. Additionally, when 2 variables are assigned the same expression then one of variables is converted to a define equal to the second variable. Assignments to init, current and next variables are taken into account. Daggification can be used in order to share common subformulas.

If no file is specified, resulting reduced simplified model is dumped to standard output.

Command Options:

`-o filename`	Attempts to write the simplified SMV model in `filename`
`-D`	Enables daggification and simplification of expressions to make the detection of equivalent variables more effective
`-d`	Disables optimization that creates one fresh variable to distinguish the first state for assignments

write_range_reduced_model - *Writes a reduced flat model to a file* Command **[I]**

```
write_range_reduced_model [-h] | [-c] [-d] [-g] [-o file] [-f fixp]
```

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

If no file is specified, resulting reduced flat model is dumped to standard output.

Command Options:

`-h`	Shows a brief description of the available options.
`-c`	Enables detection of counters from data clusters.
`-d`	Disables normal range detection (counters still may be detected).
`-g`	Adds invariant about guessed ranges to the property database.
`-o file`	Attempts to write the flat SMV model in `file`
`-f fixp`	Sets the fixpoint to be used for range extraction. Default is 20. Must be a non-negative integer

build_simplified_property - *Property simplification* Command **[I]**

```
build_simplified_property [-h] | [-a] | [ [-n <index>] | [-N <name>] [-c
<ctlspec>] | [-l <ltlspec>] | [-i <invarspec] ]* [-e]
```

Performs property simplifications on a set of properties. INVARs and ASSIGN are taken as assumptions for simplification, as done for command `write_simplified_model_rel`. During simplification, processes are eliminated and equivalent structures are set up. Each simplified property is associated to a simplified model and registered in the property database, so that future calls to usual Model Checking commands (e.g. `check_invar`) will verify freshly created properties using the simplified model.

Using `build_simplified_property` with input model M and then performing model checkin, is actually equivalent to reading model M, dumping the simplified version of M, M', using command `write_simplified_model_rel`, reading M' and finally perform model checking.

Command Options:

`-h`	Shows a brief description of the available options.
`-a`	Selects all registered properties for simplification.
`-l prop`	Specifies a LTLSPEC property to simplify.
`-i prop`	Specifies an INVARSPEC property to simplify.
`-c prop`	Specifies a CTLSPEC property to simplify.

`-e expr`	Assume given expression.
`-n number`	Simplifies property with index `number` in the property database.
`-N name`	Simplifies property with name `name` in the property database.

write_hier_coi_model - *Localize a model* Command [I]

```
write_hier_coi_model [-h] | -i iv-file [-k] [-o filename]
```

Given an input variables description file, this commands creates a localized version of the current model, that is restricted to the parts of the model which depend on the variables given as input. The optional parameter keep behavior forces adding the dependencies of the constraints in which a variable in input set occurs.

Command Options:

`-h`	Shows a brief description of the available options.
`-i iv_file`	Reads the variable names from the specified "iv_file" instead of searching in the model matching the counter structure. Input file may contain variable names and/or instances with the intended meaning to include all variables within the instance
`-k`	This flag enables recursive dependencies resolving when performing simplification, thus resulting in a stricter, behavior-preserving, approximation.
`-o filename`	Writes the output generated by the command in to the file `filename`.

write_countacc_model - *Create "accelerated" models.* Command [I]

```
write_countacc_model [-h] | [-c] [-o filename] [-i iv_file] [-l
list_file] [-v] [-V] [-s] [-p]
```

This commands creates an "accelerated" version of the current model changing the behavior of its counter variables. A counter is a word variable that is initialized to 0, can be enabled by an arbitrary boolean expression and that when enabled, increases by 1 at each step until a limit value is reached or the enabling condition is false. When a counter reaches the limit or is disabled it is reset to 0. The accelerated model is such that in a single step counters possibly increase their value by a number greater than 1.

Note that the accelerated model does not preserve all properties of the model except invariants.

Command Options:

`-h`	Shows a brief description of the available options.
`-c`	Disables the check on the constraints on counter variables.
`-o filename`	Writes output to "filename" instead of stdout.
`-i iv_file`	Read the counter names from the specified "iv_file" instead of searching in the model matching the counter structure. This option is incompatible with the option `-l`.
`-l list_file`	Read the counter names and limit values from the specified "list_file" instead of searching in the model matching the counter structure. The file must be in the following format: `SIMPWFF` $\bigwedge c <= L$ where c is the counter name and L is the limit value. This option is incompatible with the option `-l`.

`-v`	Removes the properties of the model and adds three properties for every counter. These properties must hold for a valid counter. This option is incompatible with option `-V`.
`-V`	Adds an invariant property in the accelerated model to check whether the counter acceleration is really useful or not. If the property does not hold, then the counter acceleration may be useful, otherwise the counter acceleration is totally useless. This option is incompatible with option `-v`.
`-s`	This option has to be used in conjunction with `-i`. If specified this option enables the synthetization of limits for the counters specified in the "iv_file".
`-p`	This option has to be used in conjunction with `-s`. If enabled, instead of writing the accelerated model with the synthetized limits, outputs a list of pairs (counter, limit) in the format of the "counter_limit_file".

5.9.2 Commands for Model Exploration

write_xmi_model - *Conversion of a symbolic FSM to an explicit FSM, and printing to XMI format* Command **[F,I]**

```
write_xmi_model [-F method] [-o filename] [-a explist] [-f format]
```

Converts the symbolic FSM representing the model to an explicit finite state machine (EFSM). Then prints it in XMI format to the file specified with the `-o` option.

If the option `-a` is not used, all finite variables of the model are taken into consideration.

Otherwise, the EFSM is abtstracted to the specified expressions: the states will be made of the specified expressions only. The expressions can be boolean, or single variables with boolean, integer, enumeration or word type. Expressions made by a single variable are expanded to all the valid assignments of the variable. This behavior may be critical with words variables, that could easily have a wide range.

Command Options:

`-h`	Shows a brief description of the available options.
`-F method`	Allows the user to choose which engine to use for the computation. Valid values are: bdd, sexp, sexp_allsat, be.
`-a expr_list`	Abstract the EFSM to the specified expressions. Explist is "exp_1, ..., exp_n" where every exp_i is a variable or a boolean expression.
`-o file`	Redirects the output to the specified file; default: standard output.
`-f format`	Format the XMI in a specific way. Currently, the only valid value is "ea", for making the xmi readable by Enterprise Architect.

5.10 Other Commands

check_ltlspec_on_trace - *Checks whether an LTL property is satisfied on a trace* Command

```
check_ltlspec_on_trace [-h] [-i] (-n number | -p "ltl_expr" | -P "name")
[-l loopback] trace_number
```

Checks whether an LTL property is satisfied on a given trace. The problem generated can be checked using SAT/SMT backend.

Option `-i` forces the use of the engine for infinite domains. In case the user does not provide this option, a SAT solver is called by default for checking the problem generated if only the formula and trace does not contain infinite precision variables. Otherwise, a SMT solver will be called for solving the problem generated.

We take into account that each NUXMV trace may correspond to an infinite number of traces due to the possible presence of more than one loopbacks. So, it is not possible (or at least straightforward) to check all of them. Therefore, we consider just one loopback and provide the user with the possibility to select it.

A `ltl-expr` to be checked can be specified at command line using option `-p`. Alternatively, options `-n` and `-P` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` nor `-P` are used, then an error message is printed.

The loopback value can be specified at command line using option `-l`. This must a valid `loopback` value on the given trace. If it is not valid, then an error message is printed and also the available loopbacks are provided. In case that option `-l` is not used, then a warning is printed and the check is performed using the first loopback found on the given trace.

Finally, the last argument of the command is the trace number which has to correspond to a trace stored in the system memory. If the trace number is omitted, then an error message is printed. In case that the trace has not loopbacks, then an error message is printed informing the user that the selected trace is finite and cannot satisfy any LTL formula.

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Forces the use of the engine for infinite domains.
`-n number`	Checks the LTL property with index `number` in the property database.
`-p "ltl_expr"`	An LTL formula to be checked.
`-P "name"`	Checks the LTL property named “name”
`-loopback`	Checks the property on the trace using `loopback` value. This must a valid `loopback` value on the given trace.
`trace_number`	The (ordinal) identifier number of the trace to be used to check the property. This must be the last argument of the command.

check_traces_properties - *Checks the traces in the trace database against the invar properties in the property database, if a trace falsifies a property* — Command

```
check_traces_properties [-h] [-i] [-p] [-t] [-o <fname>]
```

Checks which traces in the trace database falsify invar properties in the property database. The problem generated for checking a trace and a property can be solved using SAT or SMT solver.

Option `-i` forces the use of the an SMT backend solver. In case the user does not provide this option, a SAT solver is called by default for checking the problem generated if only the formula and trace does not contain infinite precision variables. Otherwise, a SMT solver will be called for solving the problem generated.

Option `-p` prints result on the terminal ordered by property number.

Option `-t` prints result on the terminal ordered by trace number.

Option `-o <fname>` writes result in the specified file, which is in XML format.

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Forces the use an SMT backend solver.

`-p`	Prints the result ordered by property number.
`-t`	Prints the result ordered by trace number.
`-o <fname>`	Writes the result in the specified file, which is in XML format

5.11 NUXMV environment variables

In this section we describe all the environment variables that may affect the behavior of the new features of NUXMV.

abstraction.engine Environment Variable

Specifies the engine to be used while computing the abstraction during the CEGAR loop. Possible values for this variable are:

- `msat`: (default) This approach uses ALLSMT [LNO06] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.
- `structural`: This approach uses the structural abstraction approach [CDJR09] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.
- `hybrid`: This approach uses the hybrid BDD+SMT abstraction approach [CCF+07, CFG+10] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.
- `bdd`: This approach uses the BDD existential quantification to compute the abstraction. It assumes all the predicates and mirror variables have finite range. Moreover, it assumes the input model to be finite-state (i.e. it must not contain neither `real` nor `integer` variables).

cegar.refinement Environment Variable

Specifies the refinement strategy to be used within the CEGAR loop to refine the abstraction when the abstract counter-example is spurious. Possible values for this variable are:

- `itp`: (default) It preforms the refinement analyzing the interpolants [McM03] induced by the spurious counter-example. The counter-example is split in two parts by considering each state of the counter-example. The first part of the formula consists of the prefix of the counter-example from initial state to the considered state (included). The second part is the suffix of the counter-example from the considered state (included) till the last state of the counter-example. The interpolants will be on the variables corresponding to the considered states (i.e. on the language in the intersection among the two formulas).
- `uc`: It preforms the refinement analyzing the unsatisfiable core induced by the unsatisfiability in the concrete model of the abstract counter-example. This approach has a limitation that considers only the predicates in the extracted unsatisfiability core that do not refer to variables at different states in the counter-example. (The predicate `@(a,0)=@(b,2)` where variable `a` is at state 0 and variable `b` at state 2 will be discarded. While `@(a,0)=@(b,0)` will be considered. Both variable refer to the same state.)
- `wp`: It preforms the refinement computing weakest preconditions induced by the spurious abstract counter-example.

msat_dump_format Environment Variable

This variable controls the format used by commands like e.g. `msat_check_invar_bmc` when option `-d` is given to the command. The valid values for this variable are:

- `mathsat`: (default) This format is the language specific of the MATHSAT [CGSS13] SMT solver.
- `smtlib`: This format is standard format [BST12] adopted in the SMT competition and accepted by (almost) all the SMT solvers at the state-of-the-art.

msat_dump_frac_as_float Environment Variable

This is a Boolean variable that specifies the format used while printing counter-examples or while writing infinite precision Real constants. Its default value is `0`, meaning that for instance `f'2/3` is used to represent the rational number 2/3. If set to `1`, then the output would be `0.6666666667`.

msat_native_word_support Environment Variable

This is a Boolean variable that specifies whether when interacting with the SMT solver to perform upfront bit-blasting or to pass the words directly to the SMT solver. By default this variable is set to `1`, meaning that the words are passed natively to the SMT solver.

qe.engine Environment Variable

Specifies the high level quantifier elimination engine to be used while computing the abstraction during e.g. the CEGAR loop. Possible values for this variable are:

- `msat`: This approach uses ALLSMT [LNO06] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.
- `structural`: This approach uses the structural abstraction approach [CDJR09] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.
- `hybrid`: (default) This approach uses the hybrid BDD+SMT abstraction approach [CCF+07, CFG+10] to compute the abstraction. It assumes all the predicates and mirror variables have finite range.

qe.hybrid.backjumping_enabled Environment Variable

This is a Boolean variable that enables the back-jumping optimization within the TCC encoder [CCF+07, CFG+10] when the `qe.engine` is set to `hybrid`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.hybrid.dagostino_enabled Environment Variable

This is a Boolean variable that enables the D'Agostino optimization when the `qe.engine` is set to `hybrid`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.hybrid.partitioning_enabled Environment Variable

This is a Boolean variable that enables the conjunctive partitioning of the formula to abstract when the `qe.engine` is set to `hybrid`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.hybrid.threshold_enabled Environment Variable

This is a Boolean variable that enables the use of a threshold for computing the conjunctive partitioning of the formula to abstract when the `qe.engine` is set to `hybrid`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.hybrid.threshold_value Environment Variable

This is an positive integer variable that specifies the maximm size in terms of BDD nodes for each conjuct of the formula to abstract when the conjunctive partitioning is enabled and the `qe.engine` is set to `hybrid`. By default this variable is set to `300` BDD nodes.

qe.msat.engine Environment Variable

This variable controls the low level engine used when the `qe.engine` is set to `msat` or `structural`. Possible values are:

- `allsmt`: (default) This value specifies to use AllSMT approach. It assumes that all the infinite domain variables (i.e. `real` and `integer`) are quantified out.
- `fm`: This value specifies to use Fourier-Motzking [Sch98] quantifier elimination technique. It assumes the model contains `real` variables and no `integer`.
- `lw`: This value specifies to use Loos-Weispfenning [LW93, Mon08] quantifier elimination technique. It assumes the model contains `real` variables and no `integer`.

qe.msat.remove_redundant_constraints_enabled Environment Variable

This is a Boolean variable that enables the optimization that pre-process the formula to remove redundant part of the formula within the SMT solver. This option takes effect when the `qe.engine` is set to `msat` and the `qe.msat.engine` is set to `lw` or `fm`. By default this variable is set to `0`, i.e. the optimization is enabled.

qe.msat.boolean_simplifications_enabled Environment Variable

This is a Boolean variable that enables the optimization that enables the Boolean simplification of the formula within the SMT solver. This option takes effect when the `qe.engine` is set to `msat` and the `qe.msat.engine` is set to `lw` or `fm`. By default this variable is set to `0`, i.e. the optimization is enabled.

qe.msat.top_level_propagation_enabled Environment Variable

This is a Boolean variable that enables the optimization within the SMT solver to push quantifiers inside the formula. This option takes effect when the `qe.engine` is set to `msat` and the `qe.msat.engine` is set to `lw` or `fm`. By default this variable is set to `0`, i.e. the optimization is enabled.

qe.structural.analyze_conjuncts_enabled Environment Variable

This is a Boolean variable that enables the analysis of the possible conjuncts of the formula to abstract to see whether pushing of quantifiers could be performed. This option takes effect when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.assert_conjuncts_enabled Environment Variable

This is a Boolean variable that enables the assertion in the SMT solver of all the partial results of quantification of the conjuncts of the formula to abstract while performing the quantification. This option takes effect when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.core_engine Environment Variable

This variable specifies the core engine used to perform quantifier elimination when the `qe.engine` is set to `structural`. By default this variable is set to `hybrid`, i.e. the engine used is the BDD+SMT approach of [CCF+07, CFG+10].

qe.structural.dagostino_enabled Environment Variable

This is a Boolean variable that enables the D'Agostino optimization when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.dnf_enabled Environment Variable

This is a Boolean variable that enables the conversion of the formula to quantify in Disjunctive Normal Form (DBF) when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.genbdds_enabled Environment Variable

This is a Boolean variable that enables the generation of intermediate BDDs for each leaf (quantifier free formula resulting from the quantifier elimination) element of the formula to quantify when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.incrementality_enabled Environment Variable

This is a Boolean variable that enables the exploitation of the incrementality of the SMT solver while performing the quantification of the formula to abstract when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.inlining_enabled Environment Variable

This is a Boolean variable that enables the inlining of equalities while performing the quantification of the formula to abstract when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.inlining_value Environment Variable

This variable affects the behavior of the code that performs the inlining of expressions. In particular it specifies the number of iterations to perform to discover possible equivalences to perform the inlining of conjuncts. Default value is `0`, i.e. perform a fix-point.

qe.structural.low_level_enabled Environment Variable

This is a Boolean variable that enables the low level quantifier optimizations [CDJR09] while performing the quantification of the formula to abstract when the `qe.engine` is set to `structural`. By default this variable is set to `1`, i.e. the optimization is enabled.

qe.structural.preassert_conjuncts_enabled Environment Variable

This is a Boolean variable that enables the pre-assertion in the SMT solver of all the conjuncts, if any, of the formula to abstract when the `qe.engine` is set to `structural`. By default this variable is set to `0`, i.e. the optimization is disabled.

qe.structural.varsampling_enabled Environment Variable

This is a Boolean variable that enables the variable sampling optimization [CDJR09] while performing the quantification of the formula to abstract when the `qe.engine` is set to `structural`. By default this variable is set to `1`, i.e. the optimization is enabled.

write_xmi_max_word_width Environment Variable

This variable controls the maximum number of bits allowed for the bit vectors when dumping the model in XMI format with the command `write_xmi_model`. The default value is `6`.

Remark: Large values may lead to huge times in dumping the XMI format. Indeed, for each value of the word an XMI state may be created.

expand_wordarrays Environment Variable

This variable controls if word-array variables are expanded into individual word variables or not. By doing the expansion, the expressions containing word-array variables are also modified, i.e. `=` and `:=` between array expressions is pushed to each index subscript, **`READ`** is converted to a index subscript operator **`[ ]`**, and **`WRITE`** is treated as with if-then-else expression.

- `0`: (default) No expansion is performed.
- `1`: Expansion is performed.

There is another way to expanding word-array variables: by using `-e` option during flattening of the design. See `flatten_hierarchy` command in Section 4

5.12 Commands for Parameter Synthesis

show_param_synth_problems - *Shows the parameter synthesis problems* Command **[F,I]**

```
show_param_synth_problems [-h] [[-s | -u] | [[-n prob_no] | [-P prob_name
]]]
```

It prints the set of parameter synthesis problems that have been specified so far. Few filters are provided to only print solved or unsolved problems, or to print only a specified problem (either via its unique id number in the parameter synthesis problem data base, or with its unique name).

Command Options:

`-h`	Shows a brief description of the available options.
`-s`	Prints only solved problems
`-u`	Prints only not solved problems
`-n prob_no`	Prints only problem `prob_no`
`-P prob_name`	Prints only problem named `"prob_name"`

show_param_synth_problems - *Shows the parameter synthesis problems* Command **[F,I]**

```
show_param_region [-h] -n prob_no | -P prob_name
```

It printes the region of parameter for the specified problem. If the region has not yet been computed, the user is informed.

Command Options:

`-h`	Shows a brief description of the available options.
`-n prob_no`	Prints the region for problem `prob_no`
`-P prob_name`	Prints the region for problem named `"prob_name"`

synth_params - *Synthetize a region of parameters* Command **[F,I]**

`synth_param [-h] [-h] [-i] [-c] [-s] [-v] [-a alg] [-n prob_no | -P prob-name | -p prob_str]`

Computes the region of parameters for the specified parameter synthesis problems

When the domain is infinite (or when forced explicitly with option `-i`, the `msatic3` library is used to solve the problem.

IMPORTANT: When the domain is infinite, the parameter synthesis problem is in general undecidable (as well as the verification problem) and this command may fail in synthesising the parameters. In particular, it may not terminate or it may terminate with an unknown result when it cannot refine the abstraction (this may be due to the presence of mixed integer/real predicates).

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Forces the use of the engine for infinite domains (msatic3).
`-c`	Enables region validation.
`-s`	Enables region simplification via BDD (it applies only to finite states parameters).
`-V`	Disables the print of the computed regions for each solved problem.
`-a alg`	Solves the problme using the specified algorithm. • `ic3` uses Pure IC3 (only invariants, i.e., G(expr)) • `bmc` used Pure BMC algorithm • `bmc_ic3` uses a combination of BMC and K-Live with IC3 (default)
`-n number`	Checks the INVAR property with index `number` in the property database.
`-P name`	Checks the property syntheis problem named `name` in the parameter synthesis problem database.
`-p "prob_str"`	The command line specified parameter synthesis problem. Where `"prob_str"` is a string (surrounded by `"`) of the form `"name := { id_list | ltl_expr [, MAX|MIN (simple_expr)] } [IN context]"` and `context` is the module instance name in which all the variables in ltl_expr must be evaluated in.

Chapter 6

Commands of timed NUXMV

In the following we present the commands provided by NUXMV when the system operates on timed domain. These commands are only available if the `-time` command line option has been specified. Similarly to the case of the commands inherited from NUSMV, we also describe the environment variables that may affect the behavior of the commands. These commands are time aware and allow to process timed models.

6.1 Commands for Initialization

time_setup - *Initializes the system for the verification of timed models via SMT.* Command

```
time_setup [-h]
```

This command initializes the system for verification of timed finite and infinite state systems.

go_time - *Initializes the system for the infinite state verification of timed models via SMT.* Command

```
go_time [-h]
```

This command initializes the system for verification of timed finite and infinite state systems. It is equivalent to the command sequence `read_model`, `flatten_hierarchy`, `encode_variables`, `build_flat_model`.

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

6.2 Commands for Invariant Checking

In this section we describe the command for checking invariants in timed models.

timed_check_invar - *Invariant property check.* Command

```
timed_check_invar [-h | -n idx | -p "formula" | -P "name"] [-b] [-k N]
[-a 0|1]
```

This command performs invariant checking of a timed model.

Command Options:

`-h`	Shows a brief description of the available options
`-n idx`	Checks the invariant (INVARSPEC) property specified with `idx`
`-p "formula"`	Checks the specified invariant property
`-P name`	Checks the invariant property with given name
`-b`	Use BMC, default : ic3
`-k max_len`	Maximum bound for bmc or ic3, instead of using the variable `bmc_length` value.
`-a 0|1`	If true, enable abstraction/refinement; it can not be used with -b.

6.3 Commands for LTL Model Checking

In this section we describe the command for checking LTL properties in timed models.

timed_check_ltlspec - *LTL property check.* Command

```
timed_check_ltlspec [-h | -n idx | -p "formula" | -P "name"] [-b] [-k N]
```

This command performs LTL property checking on a timed model.

Command Options:

`-h`	Shows a brief description of the available options
`-n idx`	Checks the LTL (LTLSPEC) property specified with `idx`
`-p "formula"`	Checks the specified LTL property
`-P name`	Checks the LTL property with given name
`-b`	Use BMC, default : ic3
`-k max_len`	Maximum bound for bmc or ic3, instead of using the variable `bmc_length` value.

6.4 Command for dumping discrete model

In this section we describe a command that could be used to dump the discrete model corresponding to the timed one.

write_untimed_model - *Dump corresponding discrete model* Command

```
write_untimed_model [-h] [-s] [-o "file_name"]
```

This command dumps on the specified file, or stdout if not specified, the discrete model corresponding to the timed one.

Command Options:

`-h`	Shows a brief description of the available options
`-s`	Discard specifications when dumping the discrete model
`-o "file_name"`	Allows to dump the discrete model on the specified file

6.5 Timed Simulation Commands

In this section we describe the commands that allow to simulate a NUXMV timed specification. See also the section Section 6.7 [Time aware traces], page 154 that describes the commands available for manipulating time aware traces.

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

```
timed_pick_state [-h] [-v] [-i [-a]] [-c "constr" | -s trace.state]
```

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 `timed_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).
`-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.

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

```
timed_simulate [-h] [-v] [-l] [-i [-a]] [ [-c "simple_expr" ] | [-t
"next_expr" ] ] [-k length]
```

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 `timed_pick_state` command.

It is possible to run the simulation in two ways (according to different command line policies): deterministic (the default mode), 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:

`-v`	Verbosely prints current generated trace (changed and unchanged state and frozen variables).
`-l`	Performs look-ahead while doing the simulation to see whether the trace can be extended, thus trying to avoid deadlocks.
`-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 does not exist: in that case a trace with a number of states less than `steps` 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` 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

6.6 Timed Execution Commands

In this section we describe the commands that allow to perform time aware trace re-execution on a given model. See also the section Section 6.7 [Time aware traces], page 154 that describes the commands available for manipulating traces.

execute_traces - *Executes complete time aware traces on the model FSM* Command

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

Executes time aware 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.
`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 time aware traces on the model FSM* Command

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

Executes time aware traces stored in the Trace Manager. If no trace is specified, last registered trace is executed. Traces are not required to be complete. Upon succesful 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.
`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.

6.7 Time aware traces

The following commands are available for time aware traces: `show_traces`, `read_trace`, `execute_traces` and `execute_partial_traces`.
They have the same syntax of the ones defined in 4.7.3 but they have the additional capability to handle traces with 2 kinds of transitions: discrete or delta.
The same plugins described in 4.8 are available, traces of timed models contain additional infomation that allows to distinguish discrete transitions from delta transitions.

6.7.1 Basic Trace Explainer

The trace explainer described in 4.8.1 is extended by adding before every state the information about the transition type. The syntax is either :

```
-- [discrete transition ] --
```

for discrete transitions,

```
-- [time elapse: time = time_val; delta = N] --
```

for delta transitions not in a loop, where `N` is a number that represents the amount of time elapsed and `time_val` is the value of time in the previous step or diverging if the transition is in a loop in which time diverges.

6.7.2 States/Variables Table

The state/varialbes table described in 4.8.2 is extended by adding on the states axis either :

```
[D]
```

for discrete transitions or

```
[T N]
```

for delta transitions, where N is a number that represents the amount of time elapsed.

6.7.3 XML Format Printer

The XML format printer described in 4.8.3 is extended with the additional tag:

```
<transition type="discrete"></transition>
```

to label the current transition as discrete and

```
<transition type="timed" from="init_time" to="next_time" delta="d_val"></transition>
```

if the transition is a delta transition, `init_time` is the value of time when the transition begins, `next_time` is the value of time when the transition ends and `d_val` is the amount of time elapsed. N is a number that represents the amount of time elapsed. The new tag is placed between each pair of `node` tags and gives information about the transition from the previous node to the following one. where `var_name` is the name of the variable and `value` is the constant value that such variable assumes in the delta transition.

6.7.4 XML Format Reader

Time aware traces can be loaded in the same way of other traces, as described in 4.8.4, using the command `read_trace`.

Chapter 7

Running NUXMV batch

nuXmv so far provides an batch interaction inherited from the original NUSMV. We report here the different command line options provided both by NUSMV and by nuXmv.

When the `-int` option is not specified, nuXmv 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>` **`nuXmv [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:

```
nuXmv [-h | -help] [-v vl] [-int]
       [[-source script_file | -load script_file]]
       [-s] [-old] [-old_div_op] [-smv_old]
       [-disable_syntactic_checks]
       [-keep_single_value_vars]
       [-disable_daggifier] [-dcx] [-cpp] [-pre pps]
       [-ofm fm_file] [-obm bm_file] [-lp]
       [-n idx] [-is] [-ic] [-ils] [-ips] [-ii] [-ctt]
       [[-f] [-r]]|[-df] [-flt] [-AG]
       [-coi] [-i iv_file] [-o ov_file]
       [-t tv_file] [-reorder] [-dynamic] [-m method]
       [-disable_sexp2bdd_caching] [-bdd_soh heuristics]
       [[-mono]|[-thresh cp_t]|[-cp cp_t]|[-iwls95 cp_t]]
       [-noaffinity] [-iwls95preorder]
       [-bmc] [-bmc_length k]
       [-sat_solver name] [-sin on|off] [-rin on|off] [-time]
       [-ojeba algorithm] [-ewa] [ input-file]
```

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

`-help`	
`-h`	Prints the command line help.
`-v` *verbose-level*	Enables printing of additional information on the internal operations of nuXmv. 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*	Executes nuXmv commands from file `sc_file`
`-load` *sc_file*	same as -source (deprecated)
`-s`	Avoids to load the nuXmv commands contained in `~/.nusmvrc` or in `.nusmvrc` or in `${NUXMV_LIBRARY_PATH }/master.nusmvrc`.
`-old`	Keeps backward compatibility with older versions of nuXmv. 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. Infact, in versions older than 2.5.1, it was pretty common to use `1` in `case` conditions expressions. For an example please see the NuSMV user manual [CCCJ+10].
`-old_div_op`	Enables 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, nuXmv may result in unpredictable results, use this option at your own risk.
`-disable_daggified`	Disables the daggification feature of model dumping
`-keep_single_value_vars`	Does not convert variables that have only one single possible value into constant DEFINEs
`-dcx`	Disables the generation of counter-examples for properties that are proved to be false. See also variable `counter_examples`
`-cpp`	Runs pre-processor on SMV files before any of those specified with the -pre option.
`-pre` *pps*	Specifies a list of pre-processors to run (in the order given) on the input file before it is parsed by nuXmv. 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 *fn_file*
`-obm` *bm_file*	Prints boolean model to file *bn_file*
`-lp`	Lists all properties in SMV model
`-n` *idx*	Specifies which property of SMV model should be checked
`-is`	Does not check `SPEC` 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_ltlspec`, but may also lead to a dramatic slowing down. This options has effect only when the calculation of reachable states is enabled (see `-f`).
`-AG`	Verifies only AG formulas using an ad hoc algorithm (see documentation for the `ag_only_search` environment variable).
`-coi`	Enables cone of influence reduction. 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].

`-i` *iv_file*	Reads the variable ordering from file *iv_file*.
`-o` *ov_file*	Writes the variable ordering to file *ov_file*.
`-t` *tv_file*	Reads a variable list from file *tv_file*. This list defines the order for clustering the transition relation. This feature has been provided by Wendy Johnston, University of Queensland. The results of Johnston's et al. research have been presented at FM 2006 in Hamilton, Canada. See [WJKWLvdBR06].
`-reorder`	Enables variable reordering after having checked all the specification if any.
`-dynamic`	Enables dynamic reordering of variables
`-m` *method*	Uses *method* when variable ordering is enabled. Possible values for method are those allowed for the `reorder_method` environment variable (see the NuSMV user manual [CCCJ+10]).
`-disable_sexp2bdd_caching`	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.
`-bdd_soh` *heuristics*	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` in the NuSMV user manual [CCCJ+10] for more details.
`-mono`	Enables monolithic transition relation
`-thresh` *cp_t*	conjunctive partitioning with threshold of each partition set to *cp_t* (DEFAULT, with *cp_t*=1000)
`-cp` *cp_t*	DEPRECATED: use `thresh` instead.
`-iwls95` *cp_t*	Enables Iwls95 conjunctive partitioning and sets the threshold of each partition to *cp_t*
`-noaffinity`	Disables affinity clustering
`-iwls95preoder`	Enables *Iwls95CP* preordering
`-bmc`	Enables BMC instead of BDD model checking (works only for LTL properties and PSL properties that can be translated into LTL)
`-bmc_length` *k*	Sets `bmc_length` variable, used by BMC
`-sat_solver` *name*	Sets `sat_solver` variable, used by BMC so select the sat solver to be used.
`-sin` *on,off*	Enables (on) or disables (off) Sexp inlining, by setting system variable `sexp_inlining`. Default value is `off`.
`-rin` *on,off*	Enables (on) or disables (off) RBC inlining, by setting system variable `rbc_inlining`. Default value is `on`. The idea about inlining was taken from [ABE00] by Parosh Aziz Abdulla, Per Bjesse and Niklas Eén.
`-time`	Specifies the input file is a timed model and as such it shall be interpreted. If `-time` is specified, then the input file shall start with `@TIME_DOMAIN continuous`.

`-ojeba` *algorithm*	Sets the algorthim 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`
`-ewa`	Enables the expansion of wordarray variables, by setting the system variable `expand_wordarrays`.

Bibliography

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

[AFF+07] Roy Armoni, Limor Fix, Ranan Fraer, Tamir Heyman, Moshe Y. Vardi, Yakir Vizel, and Yael Zbar. Deeper Bound in BMC by Combining Constant Propagation and Abstraction. In *ASP-DAC*, pages 304–309. IEEE, 2007.

[BCCZ99a] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, In TACAS'99*, March 1999.

[BCCZ99b] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded ltl model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER*, 2011. http://fmv.jku.at/aiger/.

[Boo] The Boolector Boolector SMT solver. http://fmv.jku.at/boolector/.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[BST12] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*, 2012. http://smtlib.cs.uiowa.edu/docs.html.

[CCCJ+10] R. Cavada, A. Cimatti, E. Olivetti C.A. Jochim, G. Keighren, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.5 User Manual*, 2010.

[CCF+07] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD*, pages 69–76. IEEE Computer Society, 2007.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CDJR09] Alessandro Cimatti, Jori Dubrovin, Tommi A. Junttila, and Marco Roveri. Structure-aware computation of predicate abstraction. In *FMCAD*, pages 9–16. IEEE, 2009.

[CFG+10] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Krishnamani Kalyanasundaram, and Marco Roveri. Tighter integration of BDDs and SMT for Predicate Abstraction. In *DATE*, pages 1707–1712. IEEE, 2010.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[CGH97a] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In *Formal Methods in System Design, 10(1):57–71*, February 1997.

[CGH97b] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMT14a] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *TACAS*, 2014.

[CGMT14b] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Verifying ltl properties of hybrid systems with k-liveness. Technical report, Fondazione Bruno Kessler, 2014. Under review.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[CMBK09] Michael L. Case, Hari Mony, Jason Baumgartner, and Robert Kanzelman. Enhanced verification by temporal decomposition. In *FMCAD*, pages 17–24. IEEE, 2009.

[CS12] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 52–59. IEEE, 2012.

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EL86] E. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278. IEEE Computer Society, 1986.

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

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

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

[FAQ] Frequently Asked Questions (FAQ). Available at `http://nusmv.fbk.eu/faq.html` or within the NuSMV distribution package.

[HBS13] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in ic3. In *FMCAD*, pages 157–164. IEEE, 2013.

[HKQ03] T. A. Henzinger, O. Kupferman, and S. Qadeer. From *Pre*-historic to *Post*-modern symbolic model checking. *Formal Methods in System Design*, 23(3):303–327, 2003.

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

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

[LNO06] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 36(5):450–462, 1993.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In Pierre and Kropf [PK99], pages 342–345.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM04] Kenneth L. McMillan. An interpolating theorem prover. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 16–30. Springer, 2004.

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

[Mon08] David Monniaux. A Quantifier Elimination Algorithm for Linear Real Arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - LPAR*, volume 5330 of *LNCS*, pages 243–257. Springer, 2008.

[PK99] Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.

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

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

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

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. J. Wiley & Sons, 1998.

[She04] Daniel Sheridan. The optimality of a fast cnf conversion and its use with sat. In *SAT*, 2004.

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

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[TCP08] Dina Thomas, Supratik Chakraborty, and Paritosh K. Pandya. Efficient guided symbolic reachability using reachability expressions. *STTT*, 10(2):113–129, 2008.

[Ton09] Stefano Tonetta. Abstract model checking without computing the abstraction. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *LNCS*, pages 89–105. Springer, 2009.

[VG09] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *FMCAD*, pages 1–8. IEEE, 2009.

[VGS12] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 173–181. IEEE, 2012.

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

Appendix A

Typing and Production Rules

Appendix B

Typing Rules

This appendix gives the explicit formal typing rules for NUXMV'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 NUXMV are as follows:

boolean

integer

real

clock

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 ≥ 1)

signed word[N] (where N is any whole number ≥ 1)

For more detalied description of existing types see Section 2.1 [Types], page 8.

B.2 Implicit Conversion

There is only one kind of implicit convertion. For more information on type ordering see Section 2.2.1 [Implicit Type Conversion], page 11.

Implicit type convertions changes the type of an expression to its counterpart set type. The Figure B.2 shows the direction of such convertions. For more information on set types and their counterpart types see Section 2.1.10 [Set Types], page 10.

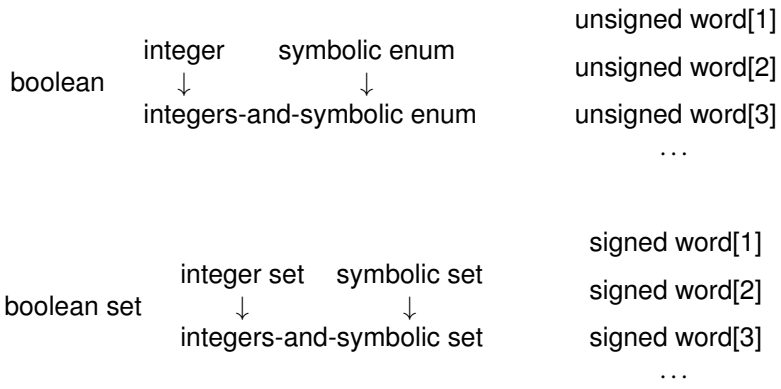

Figure B.1: The ordering on the types in NuSMV

boolean → boolean set
integer → integer set
symbolic enum → symbolic set
integers-and-symbolic enum → integers-and-symbolic set

Figure B.2: Implicit convertion to counterpart set types

B.3 Type Rules

The type rules are presented below with the operators on the left and the signatures of the rules on the right. To save space, more than one operator may be on the left-hand side, and it is also the case that an individual operator may have more than one signature. For more information on these expressions and their type rules see Section 2.2 [Expressions], page 10.

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 → integer
	: unsigned word[N] → unsigned word[N]
	: signed word[N] → signed word[N]
`+`, `-`, `/`, `*`	: integer * integer → integer
	: unsigned word[N] * unsigned word[N] → unsigned word[N]
	: signed word[N] * signed word[N] → signed word[N]
	: clock * integer → real
	: clock * real → real
`mod`	: integer * integer → integer
	: unsigned word[N] * unsigned word[N] → unsigned word[N]
	: signed word[N] * signed word[N] → signed word[N]
	For operations on words, the result is taken modulo 2^N
`>`, `<`, `>=`, `<=`	: integer * integer → boolean
	: clock * clock → boolean
	: clock * integer → boolean
	: clock * real → boolean
	: unsigned word[N] * unsigned word[N] → boolean
	: signed word[N] * signed word[N] → boolean

Logic Operators

`!` (negation)	: boolean → boolean
	: unsigned word[N] → unsigned word[N]
	: signed word[N] → signed word[N]
`&`, `|`, `->`, `<->`, `xor`, `xnor`	: boolean * boolean → boolean
	: unsigned word[N] * unsigned word[N] → unsigned word[N]
	: signed word[N] * signed word[N] → signed word[N]
`=`, `!=`	: boolean * boolean → boolean
	: integer * integer → boolean
	: clock * integer → boolean
	: clock * real → 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

Index Subscript Operator

exp_1 `[` exp_2 `]`	: array N..M of subtype * word[N] → subtype
	: array N..M of subtype * integer → subtype

the value of exp_2 has to be in range [N, M]

Bit-Wise Operators

`::` (concatenation) : word[N] * word[M] → unsigned word[N+M]
where word[•] can be any of unsigned word[•] or signed word[•]
exp_1`[`exp_2`,` exp_3`]` : unsigned word[N] * integer * integer → unsigned word[$exp_3 - exp_2 + 1$]
: signed word[N] * integer * integer → unsigned word[$exp_3 - exp_2 + 1$]
exressions exp_2 and exp_3 must be integers such that $0 \leq exp_2 \leq exp_3 <$ N
`<<`, `>>` (shift) : unsigned word[N] * integer → unsigned word[N]
: unsigned word[N] * unsigned word[•] → unsigned word[N]
: signed word[N] * integer → signed word[N]
: signed word[N] * unsigned word[•] → 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 convertion is performed on one of the operands

Case and If-Then-Else Expression

`case`
$cond_1$ `:` $result_1$`;`
$cond_2$ `:` $result_2$`;`
...
$cond_n$ `:` $result_n$`;`
`esac`

$cond$? $result_1$: $result_2$

$cond_i$ must be of type boolean. If one of $result_i$ is of a set type then all other $result_k$ are converted to their counterpart set types. The overall type of the expression is such a minimal type that each $result_i$ can be implicitly converted to.

Formula Operators

`EX`, `AX`, `EF`, `AF`, `EG`, `AG`,
`X`, `Y`, `Z`, `G`, `H`, `F`, `O` : boolean → boolean
`A-U`, `E-U`, `U`, `S` : boolean * boolean → boolean
`A-BU`, `E-BU` : boolean * integer * integer * boolean → boolean
`EBF`, `ABF`, `EBG`, `ABG` : integer * integer * boolean → boolean

Miscellaneous Operators

Integer . . Integer	: `integer_number` * `integer_number` → integer
`bool`	: 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

This appendix contains the syntactic production rules for writing a nuXmv 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 7).

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 ::
        pos_integer_number
      | - pos_integer_number

pos_integer_number ::
      digit
      | pos_integer_number digit
```

Real Numbers

```
real_number ::
        float_number
      | fractional_number
      | exponential_number

float_number ::
        pos_integer_number . pos_integer_number

fractional_number ::
        fraction_prefix ' pos_integer_number / pos_integer_number

fraction_prefix ::
        one of f F

exponential_number ::
        pos_integer_number exponential_prefix integer_number
      | float_number exponential_prefix integer_number

exponential_prefix ::
        one of e E
```

Constants

```
constant ::
        boolean_constant
      | integer_constant
      | real_constant
      | symbolic_constant
      | word_constant
      | range_constant

boolean_constant :: one of
                    FALSE TRUE

integer_constant :: integer_number

real_constant :: real_number

clock_constant :: time
```

Note that time is interpreted as a Clock constant only in timed nuXmv models.

```
symbolic_constant :: complex_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 13).

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

Basic Expressions

```
basic_expr ::
       constant                    -- a constant
      | variable_identifier        -- a variable identifier
      | define_identifier          -- a define identifier
      | function_call              -- a call to a function
      | ( basic_expr )
      | pi  -- the pi constant
      | abs ( basic_expr ) -- absolute value
      | max ( basic_expr , basic_expr ) -- max
      | min ( basic_expr , basic_expr ) -- min
      | sin ( basic_expr ) -- sin
      | cos ( basic_expr ) -- cos
      | exp ( basic_expr ) -- exp
      | tan ( basic_expr ) -- tan
      | ln ( basic_expr )  -- ln
      | pow ( basic_expr , simple_expr ) -- pow
      | pi -- pi
      | ! 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
      | word1 ( basic_expr )
                                -- boolean to word[1] convertion
      | bool ( basic_expr )
                                -- word[1] and integer to boolean convertion
      | toint ( basic_expr )
                                -- word[N] and boolean to integer convertion
      | signed ( basic_expr )
                                -- unsigned to signed word convertion
      | unsigned ( basic_expr )
                                -- signed to unsigned word convertion
      | 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
      | floor ( basic_expr )
      | 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_type_spicifier

simple_type_specifier ::
```

```
        boolean
      | word [ integer_number ]
      | unsigned word [ integer_number ]
      | signed word [ integer_number ]
      | integer
      | real
      | clock
      | { 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
```

Note that clock and Clock are interpreted as type specifiers only in timed nuXmv models.

Module Type Specifier

```
module_type_specifier ::
      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

frozenvar_declaration :: FROZENVAR simple_var_list

var_list :: complex_identifier : type_specifier ;
          | var_list complex_identifier : type_specifier ;

simple_var_list :: complex_identifier : simple_type_specifier ;
      | simple_var_list complex_identifier : simple_type_specifier ;
```

Functions

```
function_declaration :: FUN function_list

function_list :: function_declaration
                | function_list function_declaration

function_declaration :: complex_identifier : function_type_specifier ;
function_type_specifier :: function_args_type_specifier -> simple_type_specifier

function_args_type_specifier :: simple_type_specifier
                | function_args_type_specifier * simple_type_specifier
```

DEFINE Declaration

```
define_declaration :: DEFINE define_body

define_body :: complex_identifier := next_expr ;
             | define_body complex_identifier := next_expr ;
```

CONSTANTS Declaration

```
constants_declaration :: CONSTANTS constants_body ;

constants_body :: complex_identifier
             | constants_body  , complex_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 [;]

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

Clock variables are allowed to occur only in the second form above, and only in the rightmost simple expression, which is furthermore required to be *convex* (i.e. a conjunction of atoms).

FAIRNESS Constraints

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

Time domain annotation

```
  time_domain_annotation :: @TIME_DOMAIN time_domain
  time_domain :: none | continuous
```

Note that the time domain annotation must precede every mudule declarations. If it is missing the model is assumed to have time domain *none*.

Module Declarations

```
module :: MODULE identifier [(module_parameters)] [module_body]

module_parameters ::
          identifier
        | module_parameters , identifier

module_body ::
          module_element
        | module_body module_element

module_element ::
          var_declaration
        | ivar_declaration
        | frozenvar_declaration%
        | function_declaration
        | define_declaration
        | constants_declaration
        | assign_constraint
        | trans_constraint
        | init_constraint
        | invar_constraint
        | fairness_constraint
        | ctl_specification
        | invar_specification
        | ltl_specification
        | compute_specification
        | parameter_synth_problem

        | isa_declaration
        | pred_declaration
        | mirror_declaration
```

PRED and MIRROR Declarations

```
pred_declaration :: PRED simple_expression [;]
                  | PRED < identifier > := simple_expression [;]

mirror_declaration :: MIRROR variable_identifier [;]
```

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 :: CTLSPEC ctl_expr ;
                   | SPEC ctl_expr [;]
                   | CTLSPEC NAME identifier := ctl_expr [;]
                   | SPEC NAME identifier := ctl_expr [;]

ctl_expr ::
    simple_expr                -- a simple boolean expression
    | ( ctl_expr )
    | ! ctl_expr               -- logical not
    | ctl_expr & ctl_expr      -- logical and
```

```
    | ctl_expr | ctl_expr       -- logical or
    | ctl_expr xor ctl_expr     -- logical exclusive or
    | ctl_expr 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 next_expr ;
                     | INVARSPEC NAME identifier := next_expr [;]
```

This is equivalent to

```
SPEC  AG next_expr ;
```

but is checked by a specialised algorithm during reachability analysis.

LTL Specification

```
ltl_specification :: LTLSPEC ltl_expr [;]
                   | LTLSPEC NAME identifier := ltl_expr [;]

ltl_expr ::
    next_expr              -- a boolean expression with possibly next operator
    | ( 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 :: CTLSPEC rtctl_expr [;]
                     | SPEC rtctl_expr [;]
                     | CTLSPEC NAME identifier := rtctl_expr [;]
                     | SPEC NAME identifier := rtctl_expr [;]
```

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

It is also possible to compute quantative information for the FSM:

```
compute_specification :: COMPUTE compute_expr [;]
                       | COMPUTE NAME identifier := compute_expr [;]

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

PSL Specification

```
pslspec_declaration :: PSLSPEC psl_expr ;
                     | PSLSPEC NAME identifier := psl_expr ;
```

Notice that here the **;** is mandatory.

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

number :: integer_number
identifier::
   variable_identifier
 | define_identifier
psl_primary_expr ::
   constant
 | identifier                          ;; an identifier
 | { psl_expr , ... , psl_expr }
 | { psl_expr "{" psl_expr , ... , "psl_expr" }}
 | ( psl_expr )
psl_unary_expr ::
   + psl_primary_expr
 | - psl_primary_expr
 | ! psl_primary_expr
psl_binary_expr ::
   psl_expr + psl_expr
 | psl_expr union psl_expr
 | psl_expr in psl_expr
 | psl_expr - psl_expr
 | psl_expr * psl_expr
 | psl_expr / psl_expr
 | psl_expr % psl_expr
 | psl_expr == psl_expr
 | psl_expr != psl_expr
 | psl_expr < psl_expr
 | psl_expr <= psl_expr
 | psl_expr > psl_expr
```

```
 | psl_expr >= psl_expr
 | psl_expr & psl_expr
 | psl_expr | psl_expr
 | psl_expr xor psl_expr
psl_conditional_expr ::
 psl_expr ? psl_expr : psl_expr
psl_case_expr ::
 case
     psl_expr : psl_expr ;
     ...
     psl_expr : psl_expr ;
 endcase
```

Among the subclasses of `psl_expr` we depict the class `psl_bexpr` that will be used in the following to identify purely boolean, i.e. not temporal, expressions.

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

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

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

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

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

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

Command Index

Variable Index

Index