Chapter 2

Input Language

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

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

In the following, an `identifier` may be any sequence of characters starting with a character in the set {**`A-Za-z_`**} and followed by a possibly empty sequence of characters belonging to the set {**`A-Za-z0-9_$#-`**}. All characters and case in an identifier are significant. Whitespace characters are space (<SPACE>), tab (<TAB>) and newline (<RET>). Any string starting with two dashes ('`--`') and ending with a newline is a comment and ignored by the parser. The 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 NuSMV language reserved keywords which are:

MODULE, **DEFINE**, **MDEFINE**, **CONSTANTS**, **VAR**, **IVAR**, **FROZENVAR**, **INIT**, **TRANS**, **INVAR**, **SPEC**, **CTLSPEC**, **LTLSPEC**, **PSLSPEC**, **COMPUTE**, **NAME**, **INVARSPEC**, **FAIRNESS**, **JUSTICE**, **COMPASSION**, **ISA**, **ASSIGN**, **CONSTRAINT**, **SIMPWFF**, **CTLWFF**, **LTLWFF**, **PSLWFF**, **COMPWFF**, **IN**, **MIN**, **MAX**, **MIRROR**, **PRED**, **PREDICATES**, **process**, **array**, **of**, **boolean**, **integer**, **real**, **word**, **word1**, **bool**, **signed**, **unsigned**, **extend**, **resize**, **sizeof**, **uwconst**, **swconst**, **EX**, **AX**, **EF**, **AF**, **EG**, **AG**, **E**, **F**, **O**, **G**, **H**, **X**, **Y**, **Z**, **A**, **U**, **S**, **V**, **T**, **BU**, **EBF**, **ABF**, **EBG**, **ABG**, **case**, **esac**, **mod**, **next**, **init**, **union**, **in**, **xor**, **xnor**, **self**, **TRUE**, **FALSE**, **count**, **abs**, **max**, **min**

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 NuSMV.

2.1.1 Boolean

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

2.1.2 Integer

The domain of the integer type is simply any whole number, positive or negative. At the moment, there are implementation-dependent constraints on the this type and `integer numbers` can only be in the range $-2^{31}+1$ to $2^{31}-1$ (more accurately, these values are equivalent to the C/C++ macros `INT_MIN` +1 and `INT_MAX`).

2.1.3 Enumeration Types

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

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

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

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

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

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

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

2.1.4 Word

The unsigned 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 $2^{N-1} - 1$.

2.1.5 Array

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

```
array 0..3 of boolean
array 10..20 of {OK, y, z}
array 1..8 of array -1..2 of 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).

2.1.6 Set Types

set types are used to identify expressions representing a set of values. There are four set types: boolean set, integer set, symbolic set, integers-and-symbolic set. The set types can be

used in a very limited number of ways. In particular, a variable cannot be of a set type. Only `range constant` and **`union`** operator can be used to create an expression of a set type, and only **`in`**, **`case`**, (• **`?`** • **`:`** •) and assignment[1] expressions can have imediate operands of a set type.

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

the counterpart of a boolean set type is boolean,

the counterpart of a integer set type is integer,

the counterpart of a symbolic set type is symbolic enum,

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

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

2.1.7 Type Order

Figure 2.1 depicts the order existing between types in NuSMV.

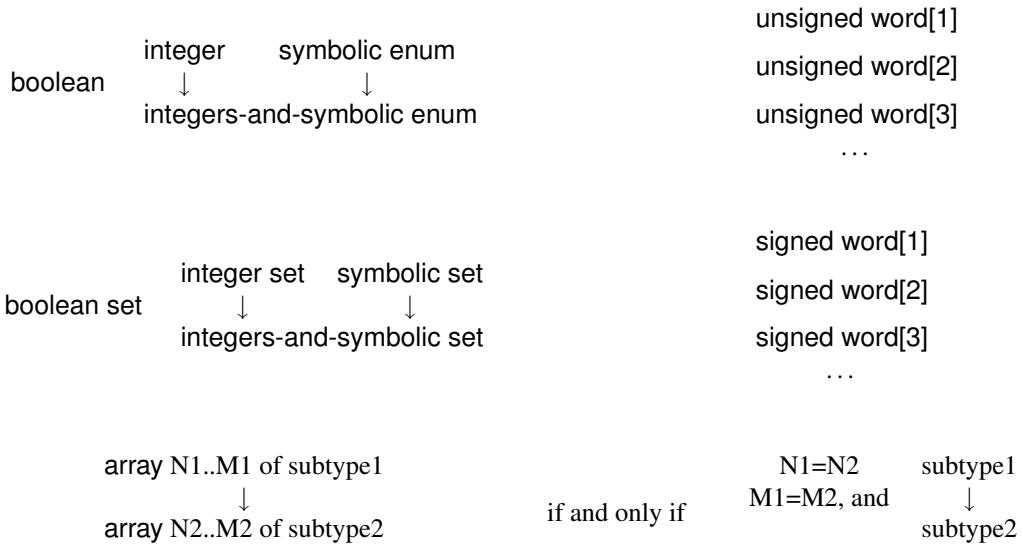

Figure 2.1: The ordering on the types in NuSMV

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

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

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

2.2 Expressions

The previous versions of NuSMV (prior to 2.4.0) did not have the type system and as such expressions were untyped. In the current version all expressions are typed and there are constraints

[1] For more information on these operators see pages 12, 18, 19, 20 and 28, respectively.

on the type of operands. Therefore, an expression may now potentially violate the type system, i.e. be erroneous.

To maintain backward compatibility, there is a new system variable called `backward_compatibility` (and a correponding `-old` command line option) that disables a few new features of version 2.4 to keep backward compatibility with old version of NuSMV. In particular, if this system variable is set then type violations caused by expressions of old types (i.e. enumeration type, boolean and integer) will be ignored by the type checker, instead, warnings will be printed out. See description at page 50 for further information.

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

2.2.1 Implicit Type Conversion

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

Note: Prior to version 2.5.1, implicit type conversion from integer to boolean (and vicevversa) was performed. Since version 2.5.1, 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, symbolic, word or range constant.

```
constant ::
        boolean_constant
      | integer_constant
      | symbolic_constant
      | word_constant
      | range_constant
```

Boolean Constant

A `boolean constant` is one of the symbolic values **`FALSE`** and **`TRUE`**. The type of a `boolean constant` is boolean.

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

Integer Constant

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

```
integer_constant :: integer_number
```

Symbolic Constant

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

```
symbolic_constant :: identifier
```

The type of a `symbolic constant` is symbolic enum. See Section 2.3.15 [Namespaces], page 34 for more information about how `symbolic constants` are distinguished from other `identifiers`, i.e. variables, defines, etc.

Word Constant

`Word constant` begins with digit `0`, followed by optional character `u` (unsigned) or `s` (signed) and one of the characters `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 an implementation limit which for most systems is 64 bits.

Range Constant

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

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

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

2.2.3 Basic Expressions

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

```
basic_expr ::
      constant                        -- a constant
    | variable_identifier             -- a variable identifier
    | define_identifier               -- a define identifier
    | ( basic_expr )
    | ! basic_expr                    -- logical or bitwise NOT
    | abs ( basic_expr )                        -- absolute value
    | max ( basic_expr , basic_expr ) -- max
    | min ( basic_expr , basic_expr ) -- min
    | basic_expr & basic_expr         -- logical or bitwise AND
    | basic_expr | basic_expr         -- logical or bitwise OR
    | basic_expr xor basic_expr       -- logical or bitwise exclusive OR
    | basic_expr xnor basic_expr      -- logical or bitwise NOT exclusive OR
    | basic_expr -> basic_expr        -- logical or bitwise implication
    | basic_expr <-> basic_expr       -- logical or bitwise equivalence
    | basic_expr = basic_expr         -- equality
    | basic_expr != basic_expr        -- inequality
    | basic_expr < basic_expr         -- less than
    | basic_expr > basic_expr         -- greater than
    | basic_expr <= basic_expr        -- less than or equal
    | basic_expr >= basic_expr        -- greater than or equal
    | - basic_expr                    -- integer unary minus
    | basic_expr + basic_expr         -- integer addition
    | basic_expr - basic_expr         -- integer subtraction
    | basic_expr * basic_expr         -- integer multiplication
    | basic_expr / basic_expr         -- integer division
    | basic_expr mod basic_expr       -- integer remainder
    | basic_expr >> basic_expr        -- bit shift right
    | basic_expr << basic_expr        -- bit shift left
    | basic_expr [ 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
    | extend ( basic_expr , basic_expr)
                                    -- word width extension
    | resize ( basic_expr , basic_expr)
                                    -- word width resize
    | basic_expr union basic_expr   -- union of set expressions
    | { set_body_expr }             -- set expression
    | basic_expr .. basic_expr      -- pure integer set expression
    | basic_expr in basic_expr      -- inclusion in a set expression
    | basic_expr ? basic_expr : basic_expr
                                    -- if-then-else expression
    | case_expr                     -- case expression
    | basic_next_expr               -- next expression

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 10.

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 2.3.12 [References to Module Components], page 32. All defines and variables referenced in expressions should be declared. All identifiers (variables, defines, symbolic constants, etc) can be used prior to their definition, i.e. there is no constraint on order such as in C where a declaration of a variable should always be placed in text above the variable use. See more information about define and variable declarations in Section 2.3.2 [DEFINE Declarations], page 27 and Section 2.3.1 [Variable Declarations], page 23.

A define is a kind of macro. Every time a define is met in expressions, it is substituted by the expression associated with this define. Therefore, the type of a define is the type of the associated expression in the current context. Define expressions may contain **`next`** operators; Normal rules apply: No nested **`next`** operators.

`variable_identifier` represents state, input, and frozen variables. The type of a variable is specified in its declaration. For more information about variables, see Section 2.3 [Definition of the FSM], page 23, Section 2.3.1 [State Variables], page 24, Section 2.3.1 [Input Variables], page 24, and Section 2.3.1 [Frozen Variables], page 25. Since a `symbolic constant` is syntactically indistinguishable from `variable_identifiers` and `define_identifiers`, a symbol table is used to distinguish them from each other.

Parentheses

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

Logical and Bitwise !

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

`!` : boolean → boolean
: unsigned word[N] → unsigned word[N]
: signed word[N] → signed word[N]

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

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

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

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

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

Equality (=) and Inequality (!=)

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

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

No implicit type conversion is performed. For example, in the expression
`TRUE = 5`
the left operand is of type boolean and the right one is of type integer. Though the signature of the operation does not have a boolean * integer rule, the expression is not correct, because no implicit type conversion will be performed. One can use the **`toint`** or the **`bool`** for explicit casts.
For example:

```
toint(TRUE) = 5
```

or

```
TRUE = bool(5)
```

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

Relational Operators >, <, >=, <=

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

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

Arithmetic Operators +, −, *, /

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

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

− (unary) : integer → integer
: 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. The result of the **/** operator is the algebraic quotient with any fractional part discarded (this is often called "truncation towards zero"). If the quotient `a/b` is representable, the expression `(a/b)*b + (a mod b)` shall equal `a`. If the value of the second operand is zero, the behavior is undefined and an error is thrown by NuSMV. The semantics is equivalent to the corresponding one of C/C++ languages.

In the versions of NuSMV prior to 2.4.0 the semantics of division was different. See page 16 for more detail.

Remainder Operator `mod`

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

The signature of the remainder operator is:

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

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

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

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

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

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

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

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

whereas in the older versions on NuSMV the equations were

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

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

Note: semantics of *modulo* operator can be obtained from the remainder operator by exploiting the equivalence:

(n modulo M) $\equiv$ ((n mod M) $+$ M) mod M

Where mod is the remainder operator.

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	0sb3_1011 **>>** 2 is equal to
0ub4_0100 **<<** 1 is equal to	0sb3_1110 **>>** 1 is equal to
0ub4_1000 **<<** 0 is equal to	0sb3_1111 **>>** 0 is equal to
0ub4_1000 and	0sb3_1111

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

Index Subscript Operator `[ ]`

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

`[ ]` : array N..M of subtype * word[N] → subtype
: 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 cause out of bound error.

Bit Selection Operator `[ : ]`

The bit selection operator extracts consecutive bits from a unsigned word[•] or signed word[•] expression, resulting in a new unsigned word[•] expression. This operation always decreases the width of a word or leaves it intact. The expressions in the brackets have to be integer constants which specify the high and low bound. The high bound must be greater than or equal to the low bound. The bits count from 0. The result of the operations is unsigned word[•] value consisting of the consecutive bits beginning from the high bound of the operand down to, and including, the low bound bit. For example, 0sb7_1011001[4:1] extracts bits 1 through 4 (including 1st and 4th bits) and is equal to 0ub4_1100. 0ub3_101[0:0] extracts bit number 0 and is equal to 0ub1_1.

The signature of the bit selection operator is:

`[ : ]` : unsigned word[N] * integer_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 $0 \leq \text{integer}_l \leq \text{integer}_h < \text{N}$

Word Concatenation Operator `::`

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

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

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

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`.

Word sizeof Operator

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

let w be a word[N] : **`sizeof`** (w) returns N.

The signature of the operator is:

`sizeof` : unsigned word[•] → integer
: signed word[•] → integer

Extend Word Conversions

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

The signature of the operator is:

`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 width of a word. 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 NuSMV. Sets can contain only singleton values, but not other sets.

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

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

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

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

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

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

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

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

Set expressions can be used only as operands of **union** and **in** operations, as the right operand of **case** and as the second and the third operand of (• **?** • **:** •) expressions and assignments. In all other places the use of set expressions is prohibited.

Inclusion Operator `in`

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

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

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

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

possible, both operands are converted to their counterpart set types [4]. Then, if required, implicit type conversion is carried out on *one* of the operands.

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, NuSMV throws an error.

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

```
case
  cond1 : expr1;
  cond2 : expr2;
  ...
  1     : exprN; -- otherwise
esac
```

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

If-Then-Else expressions

In certain cases, the syntax described above may look a bit 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 ? exp1 : exp2` and `case cond1 : exp1; TRUE : expr2; esac` are equivalent.

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

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

[6] See Section 2.1.7 [Type Order], page 9 for the information on the order of types.

Basic Next Expression

`Next expressions` refer to the values of variables in the next state. For example, if a variable **`v`** is a state variable, then **`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

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

Boolean conversion operator

bool converts unsigned word[1] and integer expressions to boolean. Also boolean expressions are allowed, but no action is 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[•]

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

Word1 Explicit Conversions

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

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

2.3 Definition of the FSM

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

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

2.3.1 Variable Declarations

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

Type Specifiers

A `type specifier` has the following syntax:

```
type_specifier ::
        simple_type_specifier
      | module_type_specifier

simple_type_specifier ::
        boolean
      | word [ basic_expr ]
      | unsigned word [ basic_expr ]
      | signed word [ basic_expr ]
      | { enumeration_type_body }
      | basic_expr .. basic_expr
      | array basic_expr .. basic_expr
          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 2.3.11 [MODULE Instantiations], page 31. The `simple type specifier` comprises boolean type, integer type, enumeration types, unsigned word[•], signed word[•] and arrays types.

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

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

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

A `type specifier` can be given by two expressions separated by `..` (`<TWO DOTS>`). The two expressions have both to evaluate to constants integer numbers, and may contain names of defines and module formal parameters. For example, `-1 - P1 .. 5 + D1`, where `P1` refers to a module formal parameter, and `D1` refers to a define. Both `P1` and `D1` have to be statically evaluable to integer constants.

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

The unsigned word[•] type is specified by the keywords **`unsigned word`** (where **`unsigned`** may be skipped) with a `basic_expr` supplied in square brackets. The expression must be statically evaluable to a constant integer number whose value must be greater than zero. The signed word[•] type is specified in a similar way with the keywords **`signed word`**. The purpose of the word types is to offer integer and bitwise arithmetic.

An array type is denoted by a sequence of the keyword **`array`**, a `basic_expr` specifying the lower bound of the array index, two dots `..`, a `basic_expr` specifying the upper bound of the array index, the keyword **`of`**, and the type of array's elements. The elements can themselves be arrays. The two bound expressions have to be statically evaluable to constant integer numbers, and may contain names of defines and module formal parameters.

State Variables

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

```
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`, `INVARSPEC`, `COMPUTE`, `PSLSPEC`. For example:

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

- Anywhere in the FSM when checking invariants with BMC and the "DUAL" algorithm. See at page 82 for further information.

Frozen Variables

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

`frozenvar_declaration :: **FROZENVAR** simple_var_list`

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

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

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

- Left-side current and next state assignments are illegal, while init state assignments are allowed:

  ```
  FROZENVAR a : boolean;
  FROZENVAR b : boolean;
  FROZENVAR c : boolean;
  VAR d : boolean;
  FROZENVAR e : boolean;
  ASSIGN
  init(a) := d; -- legal
  next(b) := d; -- illegal
  c := d; -- illegal
  e := a; -- also illegal
  ```

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

  ```
  -- the following has an empty state space
  FROZENVAR a : boolean;
  ```

```
INIT a
INVAR !a

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

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

- All kinds of specifications involving frozen variables are allowed, e.g.:

```
FROZENVAR c : boolean;
-- True by definition.
SPEC AG ((c -> AG c) & ((!c) -> AG !c))
-- Here, neither is true.
INVARSPEC c
INVARSPEC !c
-- False (as above).
LTLSPEC (G F c) & (G F !c)
```

Examples

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

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

The variable `a` is a state variable, `b` is a frozen variable, and `c` is an input variable; In the following examples:

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

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

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

VAR h : word[3];

VAR i : signed word[4];
```

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

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

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

2.3.2 DEFINE Declarations

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

```
define_declaration :: DEFINE define_body

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

DEFINE associates an `identifier` on the left hand side of the `':='` with an expression on the right side. A define statement can be considered as a macro. Whenever a define `identifier` occurs in an expression, the `identifier` is syntactically replaced by the expression it is associated with. The associated expression is always evaluated in the context of the statement where the `identifier` is declared (see Section 2.3.16 [Context], page 36 for an explanation of contexts). Forward references to defined symbols are allowed but circular definitions are not, and result in an error. The difference between defined symbols and variables is that while variables are statically typed, definitions are not.

2.3.3 Array Define Declarations

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

```
array_define_declaration ::
                  DEFINE identifier := array_expression ;

array_expression :: [ array_contents ]
                  | [ array_expression_list ]

array_expression_list :: array_expression
                       | array_expression , array_expression_list

array_contents :: next_expr , array_contents
                | next_expr
```

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

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

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

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

2.3.4 CONSTANTS Declarations

CONSTANTS declarations allow the user to explicitly declare symbolic constants that might occur or not within the FSM that is being defined. **CONSTANTS** declarations are expecially useful

in those conditions that require symbolic constants to occur only in **DEFINEs** body (e.g. in generated models). For an example of usage see also the command `write_boolean_model`. A constant is allowed to be declared multiple times, as after the first declaration any further declaration will be ignored. **CONSTANTS** declarations are an extension of the original SMV grammar, and they are supported since NuSMV 2.6. The syntax for this kind of declaration is:

```
constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier
             | constants_body  , identifier
```

2.3.5 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.

2.3.6 INVAR Constraint

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

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

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

2.3.7 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.

2.3.8 ASSIGN Constraint

An assignment has the form:

```
assign_constraint :: ASSIGN assign_list

assign_list :: assign ;
             | assign_list assign ;

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

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

Semantically assignments can be expressed using other kinds of constraints:

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

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

The allowed types of the assignment operator are:

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

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

Rules for assignments

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

The restriction rules for assignments are:

- **The single assignment rule** – each variable may be assigned only once.
- **The circular dependency rule** – a set of equations must not have "cycles" in its dependency graph not broken by delays.

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

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

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

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

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

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

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

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

2.3.9 `FAIRNESS` Constraints

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

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

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

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

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

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

2.3.10 MODULE Declarations

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

```
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
        | compute_specification
        | isa_declaration
```

The `identifier` immediately following the keyword **MODULE** is the name associated with the module. Module names have a separate name space in the program, and hence may clash with names of variables and definitions. The optional list of identifiers in parentheses are the formal parameters of the module.

2.3.11 MODULE Instantiations

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

```
module_type_specifier ::
      | identifier [ ( [ parameter_list ] ) ]
      | process 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 21). It is an error if the number of actual parameters is different from the number of formal parameters. Whenever formal parameters occur in expressions within the module, they are replaced by the actual parameters. The semantic of module instantiation is similar to call-by-reference.[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.

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

2.3.12 References to Module Components (Variables and Defines)

As described in Section 2.2.3 [Variables and Defines], page 13, defines and variables can be referenced in expressions as `variable_identifiers` and `define_identifiers` respectively, both of which are `complex identifiers`. The syntax of a `complex identifier` is:

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

[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.

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 2.3.11 [MODULE Instantiations], page 31 for information about instances of modules) then the expression `m.c` identifies the component `c` of the module instance `m`. This is precisely analogous to accessing a component of a structured data type.

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

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

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

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

Individual elements of an array are accessed in the typical fashion with the index given in square brackets. See 2.2.3 for more information.

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

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

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

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

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

2.3.13 Processes

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

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword '**process**' (see Section 2.3.11 [MODULE Instantiations], page 31). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Note that only assignments of the form

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

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

Each instance of a process has a special boolean variable associated with it, called `running`. The value of this variable is `TRUE` if and only if the process instance is currently selected for execution. No two processes may be running at the same time.

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

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

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

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

2.3.14 A Program and the `main` Module

The syntax of a NuSMV program is:

```
program :: module_list

module_list ::
          module
        | module_list module
```

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

2.3.15 Namespaces and Constraints on Declarations

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

Module identifiers have their own separate namespace. Module identifiers can be used in `module type specifiers` only, and no other kind of identifiers can be used there (see Section 2.3.11 [MODULE Instantiations], page 31). Thus, module identifiers may be equal to other kinds of identifiers without making the program ambiguous. However, no two modules should be declared with the same identifier. Modules cannot be declared in other modules, therefore they are always referenced by simple `identifiers`.

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

Let us consider the following:

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

MODULE foo
  VAR q : boolean;
      e : moo;

MODULE moo
  DEFINE f := 0 < 1;

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

MODULE used
  VAR k : boolean;
```

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

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

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

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

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

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

2.3.16 Context

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

```
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. In NuSMV it is possible to declare the common parts as separate modules, and then use the **`ISA`** declaration to import the common parts inside a module declaration. The syntax of an `isa_declaration` is as follows:

```
isa_declaration :: ISA identifier
```

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

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

2.4 Specifications

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

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

2.4.1 CTL Specifications

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

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

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

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

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

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

Intuitively the semantics of CTL operators is as follows:

- **EX** p is true in a state s if *there exists* a state s' such that a transition goes from s to s' and p is true in s'.
- **AX** p is true in a state s if *for all* states s' where there is a transition from s to s', p is true in s'.
- **EF** p is true in a state s_0 if *there exists* a series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ..., $s_{n-1} \rightarrow s_n$ 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 $s_0 \rightarrow s_1, s_1 \rightarrow s_2$, ... 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 $s_0 \rightarrow s_1, s_1 \rightarrow s_2$, ..., $s_{n-1} \rightarrow s_n$ 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].

2.4.2 Invariant Specifications

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

```
invar_specification :: INVARSPEC next_expr ;
                       INVARSPEC NAME name := next_expr [;]
```

This statement is intuitively equivalent to

```
SPEC  AG simple_expr ;
```

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

2.4.3 LTL Specifications

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

```
ltl_specification :: LTLSPEC ltl_expr [;]
                     LTLSPEC NAME name := ltl_expr [;]
```

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

```
ltl_expr ::
    next_expr              -- a next boolean expression
    | ( ltl_expr )
    | ! ltl_expr             -- logical not
    | ltl_expr & ltl_expr    -- logical and
    | ltl_expr | ltl_expr    -- logical or
    | ltl_expr xor ltl_expr  -- logical exclusive or
    | ltl_expr xnor ltl_expr -- logical NOT exclusive or
    | ltl_expr -> ltl_expr   -- logical implies
    | ltl_expr <-> ltl_expr  -- logical equivalence
    -- FUTURE
    | X ltl_expr             -- next state
    | G ltl_expr             -- globally
    | G bound ltl_expr       -- bounded globally
    | F ltl_expr             -- finally
    | F bound ltl_expr       -- bounded finally
    | ltl_expr U ltl_expr    -- until
    | ltl_expr V ltl_expr    -- releases
    -- PAST
    | Y ltl_expr             -- previous state
    | Z ltl_expr             -- not previous state not
    | H ltl_expr             -- historically
    | H bound ltl_expr       -- bounded historically
    | O ltl_expr             -- once
    | O bound ltl_expr       -- bounded once
    | ltl_expr S ltl_expr    -- since
    | ltl_expr T ltl_expr    -- triggered

bound :: [ integer_number , integer_number ]
```

Intuitively the semantics of LTL operators is as follows:

- **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 **V** q is true at time t if q holds at *all* time steps $t' \geq t$ up to and including the time step t'' where p also holds. Alternatively, it may be the case that p *never* holds in which case q must hold in *all* time steps $t' \geq t$.
- **Y** p is true at time $t > t_0$ 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 **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 $t_0 \leq t'' \leq t$

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

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

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

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

Important Difference Between BDD and SAT Based LTL Model Checking

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

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

the LTL specification is proved valid by BDD-based model checking but is violated by SAT-based bounded model checking. The counter-example found consists of one state `s=FALSE`.

This difference between the results is caused by the fact that BDD model checking investigates only *infinite* paths whereas SAT-based model checking is able to deal also with *finite* paths. Apparently infinite paths cannot ever have `s=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-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-based bounded model checking cannot find a finite counterexample and results of model checking become consistent with BDD-based model checking.

2.4.4 Real Time CTL Specifications and Computations

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

```
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 name := rtctl_expr [;]
                     | SPEC NAME name := rtctl_expr [;]
```

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

```
compute_specification :: COMPUTE compute_expr [;]
                         COMPUTE NAME name := compute_expr [;]
```

where

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

MIN [*start , final*] returns the length of the shortest path from a state in *start* to a state in *final*. For this, the set of states reachable from *start* is computed. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach the state. If a fixed point is reached and no computed states intersect *final* then *infinity* is returned.
MAX [*start , final*] returns the length of the longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, then *infinity* is returned. If any of the initial or final states is empty, then *undefined* is returned.

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

2.4.5 PSL Specifications

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

```
pslspec_declaration :: PSLSPEC psl_expr [;]
                       PSLSPEC NAME name := psl_expr [;]
```

where

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

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

```
psl_primary_expr ::
   number                                ;; a numeric constant
 | boolean                               ;; a boolean constant
 | word                                  ;; a word constant
 | var_id                                ;; a variable identifier
 | { psl_expr , ... , psl_expr }
 | { psl_expr "{" psl_expr , ... , "psl_expr" }}
 | ( psl_expr )

psl_unary_expr ::
   + psl_primary_expr
```

```
  | - 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
```

Please note that instances of `FL_property` cannot be combined with the "**=**", "**!=**" and "**==**". Sequences, i.e. istances of class `sequence`, are defined as follows:

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

```
count ::
   number
 | range
```

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

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

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

2.5 Variable Order Input

It is possible to specify the order in which variables should appear in the BDD's generated by NuSMV. The file which gives the desired order can be read in using the `-i` option in batch mode or by setting the `input_order_file` environment variable in interactive mode. [10]

2.5.1 Input File Syntax

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

```
vars_list :: EMPTY
          | var_list_item vars_list

var_list_item :: complex_identifier
              | complex_identifier . integer_number
```

Where *EMPTY* means parsing nothing.

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

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

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

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

[10] Note that if the ordering is not provided by a user then NuSMV decides by itself how to order the variables. Two shell variables `bdd_static_order_heuristics` (see page 53) and `vars_order_type` (see page 52) allow to control the ordering creation.

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

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

2.5.2 Scalar Variables

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

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

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

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

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

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

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

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

2.5.3 Array Variables

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

2.6 Clusters Ordering

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

In [WJKWLvdBR06], Wendy Johnston and others from University of Queensland, showed that choosing a good ordering for the initial list of variables that is used to build the clusters list may lead to a dramatic improvement of performances. They did experiments in a modified version of NuSMV, by allowing the user to specify a variable ordering to be used when constructing the initial clusters list. The prototype code has been included in version 2.4.1, that offers the new option `trans_order_file` to specify a file containing a variable ordering (see at page 53 for further information).

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