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

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

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

2.4.2 Invariant Specifications

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

```
invar_specification :: INVARSPEC 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 counter-example and results of model checking become consistent with BDD-based model checking.

2.4.4 Real Time CTL Specifications and Computations

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

```
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
```