
OpenVera Assertions v2.0 / ForSpec v2.0

Technical Language Specification

*Roy Armoni[†], Surrendra Dudani[‡], Alon Flaisher[†], Joseph Lis[‡],
Moshe Vardi[§], Yael Zbar[†]*

[†] - Intel Corporation

[‡] - Synopsys Inc.

[§] - Rice University

April 7, 2002

1 INTRODUCTION	4
2 OVA 2.0 BNF	4
2.1 NOTATIONS	4
2.2 SYNTAX	4
3 SEMANTICS	9
3.1 FORMULAE	9
3.1.A CLOCK TICKS	9
3.1.B BOOLEAN CONNECTIVES	9
3.1.C LTL OPERATORS	10
3.1.D LTL WITH TEMPORAL INTERVALS	11
3.1.E FOLLOWED_BY AND TRIGGERS	12
3.1.F CLOCKS	13
3.1.G RESETS	13
3.2 REGULAR EVENTS	14
3.2.A REGULAR EXPRESSIONS	14
3.2.B CLOCKED REGULAR EVENTS	14
3.3 REGULAR EXPRESSIONS	15
3.3.A BOOLEAN EXPRESSIONS	15
3.3.B INTERSECTION, && AND	15
3.3.C CONCATENATION AND TIME SHIFT	15
3.3.D INVERSE AND FIRST MATCH	16
3.3.E IF-THEN-ELSE	16
3.3.F REPETITION	16
3.3.G RESTRICTIONS	17
3.4 BOOLEAN EXPRESSIONS	17
3.4.A COUNT	17
3.4.B PAST, FUTURE, ANY	17
3.4.C EDGES	18

3 . 4 . _D ENDED	18
4 OPERATOR PRECEDENCE	19
5 TEMPLATES	20
5 . 1 MATCHED	20
5 . 1 . _A EXPLICIT SEMANTICS OF THE MATCHED OPERATOR	20
5 . 2 DATA CHECKING	20
5 . 3 IMPORTANT BOOLEAN OPERATORS	20
5 . 3 . _A MUTEX AND STRONG MUTEX	20
5 . 3 . _B SAME	20

1 Introduction

The enclosed is semantic support for the OVA donation language made to Accelera System Verilog.

2 OVA 2.0 BNF

2.1 Notations

The following notation is used to specify the syntax:

- **Bold** font for terminals
- plain text for non-terminals
- [] for empty or present
- {} for empty, or any number of times

2.2 Syntax

```
file ::= { declaration }
```

```
declaration ::=  
    module_spec  
| scope_spec  
| non_scoped_spec
```

```
non_scoped_spec ::=  
    library_use_clause  
| comment  
| clocked_template
```

```
module_spec ::=  
    module name { { entity } }
```

```
scope_spec ::=  
    scope name { { entity } }
```

```

clocked_template ::=
    template name [ ( parameter_list ) ] : { { entity } }

unclocked_template ::=
    template name [ ( parameter_list ) ] : { { unlocked_entity } }

entity ::=
    clocked_entity
| unlocked_entity

clocked_entity ::=
    clock boolean_expr { { unlocked_entity } }
| sclock boolean_expr { { unlocked_entity } }
| directive
| <name of clocked template> [name] [ ( argument_list ) ] ;

unclocked_entity ::=
    scope name { { unlocked_entity } }
| template
| free_var_spec
| free_var_assign
| non_scoped_spec
| unlocked_template
| <name of unlocked template> [name] [ ( argument_list ) ] ;

argument_list ::=
    formula_expr { , formula_expr }

library_use_clause ::=
    `lib_include name

comment ::= any Verilog style comment and any nested C language
style comments

free_var_spec ::=
    var [ [ int : int ] ] name [ [ int : int ] ] ;

```

```

free_var_assign ::=
    name = bit_vector_expr ;
| name <= bit_vector_expr ;
| init name = bit_vector_expr ;

directive ::=
    assume name[: formula_expr] ;
| assert name[: formula_expr] ;
| restrict name[: formula_expr] ;
| model name[: formula_expr] ;

template ::=
    unlocked_formula
| bool name [( parameter_list )]: boolean_expr ;

unlocked_formula ::=
    formula name [( parameter_list )]: formula_expr ;
| event name [( parameter_list )]: <clocked regular_expr> ;

parameter_list ::=
    name {, name}

```

```

formula_expr ::=
    regular_expr
    | <name of any template> [( formula_expr {, formula_expr } )]
    | ( formula_expr )
    | formula_expr || formula_expr
    | formula_expr && formula_expr
    | ! formula_expr
    | formula_expr implies formula_expr
    | formula_expr iff formula_expr
    | <clocked regular_expr> followed_by formula_expr
    | <clocked regular_expr> triggers formula_expr
    | formula_expr until [int_interval] formula_expr
    | formula_expr wuntil [int_interval] formula_expr
    | next [[ int ]] formula_expr
    | wnext [[ int ]] formula_expr
    | eventually [int_interval] formula_expr
    | globally [int_interval] formula_expr
    | reject boolean_expr in formula_expr
    | accept boolean_expr in formula_expr

regular_expr ::=
    boolean_expr
    | <name of unclocked event> [(regular_expr {, regular_expr } )]
    | ( regular_expr )
    | regular_expr || regular_expr
    | regular_expr && regular_expr
    | regular_expr intersect regular_expr
    | [regular_expr] time_shift regular_expr
    | inv regular_expr
    | first_match regular_expr
    | if boolean_expr then regular_expr [else regular_expr]
    | regular_expr *[ int ]
    | regular_expr *int_interval
    | restrict_reg_expr in regular_expr

```

```
restrict_reg_expr ::=
    istru boolean_expr
| length [ int ]
| length int_interval
```

boolean_expr ::= casting of bit_vector_expression to width 1.

```
bit_vector_expr ::=
    <name of bool> [(bit_vector_expr {, bit_vector_expr } )]
| ( bit_vector_expr )
| count ( bit_vector_expr )
| past ( bit_vector_expr [, int [, boolean_expr]] )
| future ( bit_vector_expr )
| any
| posedge boolean_expr
| negedge boolean_expr
| edge boolean_expr
| ended <clocked regular_expr>
| clock
| unary_vlog_ops bit_vector_expr
| bit_vector_expr binary_vlog_ops bit_vector_expr
| similarly add addintional vlog boolean ops
```

```
time_shift ::=
    #int
| #int_interval
| ->>
```

```
int_interval ::=
    [ int .. int ]
| [ int .. ]
```

name ::= any valid Verilog identifier

int ::= any Verilog non-negative integer expression which can be compile-time evaluated

3 Semantics

3.1 Formulae

The semantics of a OVA formula is defined when a context π, i, c, a, r (π is a trace, i is a point on the trace, c is the *clock-free* (does not contain the CLOCK keyword) clock and a and r are the accept and reject signals, respectively) satisfies a OVA formula f . This is called the *satisfaction relation*, and is denoted by $\pi, i, c, a, r \models f$. If both reset signals are false, we write $\pi, i, c \models f$, and if the clock is true and both reset signals are false we simply write $\pi, i \models f$. We say that a trace satisfies a OVA formula whenever the initial context satisfies the formula; that is,

$$\pi, 0, \text{true}, \text{false}, \text{false} \models f \quad \text{or for short} \quad \pi, 0 \models f$$

The definition of OVA semantics (the satisfaction relation) is by induction on the structure of the formula where the induction basis contains the semantics of Boolean expressions.

3.1.a Clock Ticks

TICK is an abbreviation for the regular event $((! \text{CLOCK})^*[0..] \#1 \text{CLOCK})$. Let π be a computation, i a point, c a clock expression, and a and r resets expressions. Then $\text{tick}(\pi, i, c, a, r)$ is the least j such that $j \geq i$ and $\pi, i, j, c, a, r \models \text{TICK}$ (note that there is at most one such j). Note also that $\text{tick}(\pi, i, c, a, r)$ need not be defined, since we may have that $\pi, i, j, c, a, r \not\models \text{TICK}$ for all $j \geq i$. See below the definition of tight satisfaction (\models) of regular events.

If it is defined, we write $\text{tick}(\pi, i, c, a, r) \downarrow$. Note that $\pi, i, j, c, a, r \models \text{TICK}$ can hold in two (nonexclusive) ways:

- $\pi, j \models c$, in which case we write $\text{tick}(\pi, i, c, a, r) \downarrow_c$, or
- $\pi, j, c \models a$, in which case we write $\text{tick}(\pi, i, c, a, r) \downarrow_a$.

We write " $j = \text{tick}(\pi, i, c, a, r)$ " as an abbreviation for " $\text{tick}(\pi, i, c, a, r) \downarrow$ and $j = \text{tick}(\pi, i, c, a, r)$ ".

3.1.b Boolean connectives

- $\pi, i, c, a, r \models f \mid\mid g$ if $\pi, i, c, a, r \models f$ or $\pi, i, c, a, r \models g$.
- $\pi, i, c, a, r \models f \ \&\& \ g$ if $\pi, i, c, a, r \models f$ and $\pi, i, c, a, r \models g$.
- $\pi, i, c, a, r \models !f$ if $\pi, i, c, r, a \not\models f$. (Note how negation reverses the roles of the accept and reject signals).

3.1.c LTL operators

- $\pi, i, c, a, r \models \text{next } f$ if either
 - $\pi, i, c \models a$, or
 - $\pi, i, c \models !r$, and
 - either $\text{tick}(\pi, i+1, c, a, r) \downarrow_a$ or
 - $j = \text{tick}(\pi, i+1, c, a, r)$ and $\pi, j, c, a, r \models f$
- $\pi, i, c, a, r \models \text{wnext } f$ if either
 - $\pi, i, c \models a$, or
 - $\pi, i, c \models !r$, and
 - if $j = \text{tick}(\pi, i+1, c, r, a)$, then
 - $\text{tick}(\pi, i+1, c, r, a) \downarrow_r$ and $\pi, j, c, a, r \models f$
- $\pi, i, c, a, r \models \text{eventually } f$ if for some $j \geq i$ we have that the following holds:
 - for all k , $i \leq k < j$, we have $\pi, k, c \models !r$, and
 - either $\pi, j, c \models a$ or
 - $\pi, j \models c$ and $\pi, j, c, a, r \models f$.

Intuitively, `eventually f` says that `f` holds at some future tick point or that the accept signal `a` is eventually asserted. The reject signal `r` is allowed to be on only after the accept signal is on.

- $\pi, i, c, a, r \models \text{globally } f$ if for all $j \geq i$ we have
 - for some k , $i \leq k < j$, we have $\pi, k, c \models a$, or
 - both $\pi, j, c \models !r$, and
 - if $\pi, j \models c$ then $\pi, j, c, a, r \models f$.

Intuitively, `globally f` says that `f` holds at every future tick point until the accept signal `a` is on. The reject signal `r` is allowed to be on only after the accept signal is on.

- $\pi, i, c, a, r \models f \text{ until } g$ if for some $j \geq i$ we have that the following holds:
 - for all k , $i \leq k < j$, we have $\pi, k, c \models !r$, and
 - if $\pi, k \models c$, then $\pi, k, c, a, r \models f$, and
 - either $\pi, j, c \models a$, or
 - $\pi, j \models c$ and $\pi, j, c, a, r \models g$.

The semantics of the until involves the current time point `i`, if it is a tick point, and future tick points. In addition it involves all time points, as the reject signal is not allowed to hold at any point until the operator is satisfied. Also, the accept signal is

not required to occur at the tick point. This again is a feature of our asynchronous style of resets.

- $\pi, i, c, a, r \models f \text{ wuntil } g$ if either
 - $\pi, i, c, a, r \models f \text{ until } g$, or
 - $\pi, i, c, a, r \models \text{globally } f$.

3.1.d LTL with Temporal Intervals

- For $m > 0$, $\pi, i, c, a, r \models \text{next}[m] f$ if $\pi, i, c, a, r \models \text{next} \dots \text{next } f$, where the next is iterated m times.
- For $m > 0$, $\pi, i, c, a, r \models \text{wnext}[m] f$ if $\pi, i, c, a, r \models \text{wnext} \dots \text{wnext } f$, where the wnext is iterated m times.
- For $0 \leq m \leq n \leq \infty$ ($m = \infty$ is not allowed), $\pi, i, c, a, r \models \text{globally}[m, n] f$ is defined in two steps:
 - $\pi, i, c, a, r \models \text{globally}[0, n] f$ if for all $j \geq i$ such that there are at most n tick points between $i+1$ and j (inclusive), we have that:
 - For some k , $i \leq k \leq j$, we have $\pi, k, c \models a$, or
 - both
 - $\pi, j, c \models !r$, and
 - if $\pi, j \models c$ then $\pi, j, c, a, r \models f$
 - For $m > 0$, $\pi, i, c, a, r \models \text{globally}[m, n] f$ if either
 - $\pi, i, c \models a$, or
 - both
 - $\pi, i, c \models !r$, and
 - $\pi, i+1, c, a, r \models \text{TICK TRIGGERS globally}[m-1, n-1] f$
- For $0 \leq m \leq n \leq \infty$ ($m = \infty$ is not allowed), $\pi, i, c, a, r \models \text{eventually}[m, n] f$ is defined in two steps:
 - $\pi, i, c, a, r \models \text{eventually}[0, n] f$ if for some $j \geq i$ we have:
 - There are at most n tick points between $i+1$ and j (inclusive), and
 - For all k , $i \leq k \leq j$, we have $\pi, k, c \models !r$, and
 - either
 - $\pi, j, c \models a$, or
 - $\pi, j \models c$ and $\pi, j, c, a, r \models f$
 - For $m > 0$, $\pi, i, c, a, r \models \text{eventually}[m, n] f$ if either
 - $\pi, i, c \models a$, or

-
- both
 - $\pi, i, c \models !r$, and
 - $\pi, i+1, c, a, r \models \text{TICK FOLLOWED_BY}$

$$\text{eventually}[m-1, n-1] f$$
 - For $0 \leq m \leq n \leq \infty$ ($m = \infty$ is not allowed), $\pi, i, c, a, r \models f \text{ until}[m, n] g$ is defined in two steps:
 - $\pi, i, c, a, r \models f \text{ until}[0, n] g$ if for some $j \geq i$ we have:
 - There are at most n tick points between $i+1$ and j (inclusive), and
 - For all k , $i \leq k \leq j$, we have $\pi, k, c \models !r$, and if $\pi, k \models c$ then $\pi, k, c, a, r \models f$, and
 - either
 - $\pi, j, c \models a$, or
 - $\pi, j \models c$ and $\pi, j, c, a, r \models g$
 - For $m > 0$, $\pi, i, c, a, r \models f \text{ until}[m, n] g$ if either
 - $\pi, i, c \models a$, or
 - both
 - $\pi, i, c \models !r$, and
 - $\pi, i+1, c, a, r \models \text{TICK FOLLOWED_BY}$

$$(f \text{ until } [m-1, n-1] g)$$
 - For $0 \leq m \leq n \leq \infty$ ($m = \infty$ is not allowed), $\pi, i, c, a, r \models f \text{ wuntil}[m, n] g$ if either
 - $\pi, i, c, a, r \models f \text{ until}[m, n] g$, or
 - $\pi, i, c, a, r \models \text{globally}[m, n] f$

Note that the semantics of $\text{globally}[m, n]$, $\text{eventually}[m, n]$, $\text{until}[m, n]$ and $\text{wuntil}[m, n]$ is sensitive to resets (accept and reject signals) occurring before the time window starts.

3.1.e FOLLOWED_BY and TRIGGERS

- $\pi, i, c, a, r \models_e \text{followed_by } f$ if
 - either $\text{tick}(\pi, i, c, a, r) \downarrow_a$ or
 - $j = \text{tick}(\pi, i, c, a, r)$ and there exists $k \geq j$ such that $\pi, j, k, c, a, r \models_e e$ and
 - $\pi, j, k, c, a, r \models_a e$ or
 - $\text{tick}(\pi, k, c, a, r) \downarrow_a$ or
 - $l = \text{tick}(\pi, k, c, a, r)$ and $\pi, l, c, a, r \models f$.

Intuitively, we wait for a tick, then check e , then wait for another tick, and then check f . An accept signal along the way terminates the check successfully.

- $\pi, i, c, a, r \models e$ triggers f if
 - $\text{tick}(\pi, i, c, r, a) \downarrow_r$ does not hold and
 - if $j = \text{tick}(\pi, i, c, r, a)$, then, for all $k \geq j$ such that $\pi, j, k, c, r, a \models e$ we have that
 - $\pi, j, k, c, r, a \models_r e$ does not hold and
 - $\text{tick}(\pi, k, c, r, a) \downarrow_r$ does not hold and
 - if $l = \text{tick}(\pi, k, c, r, a)$, then, $\pi, l, c, a, r \models f$.

Note that waiting for a tick, the regular event e , and again waiting for a tick function as antecedents of an implication, and thus have negative polarity, which explains why the roles of a and r are reversed.

3.1.f Clocks

- $\pi, i, c, a, r \models \text{sclock } d \ f$ if either
 - $\text{tick}(\pi, i, [c \mapsto d], a, r) \downarrow_a$, or
 - $j = \text{tick}(\pi, i, [c \mapsto d], a, r)$ and $\pi, j, [c \mapsto d], a, r \models f$.

Note the substitution $[c \mapsto d]$ that substitutes c in every occurrence of **CLOCK** in d , which guarantee that the keyword **CLOCK** does not occur on the left-hand side of the relation \models . The **sclock** operator forces the evaluation of the formula at the nearest tick point.

- $\pi, i, c, a, r \models \text{clock } d \ f$ if $j = \text{tick}(\pi, i, [c \mapsto d], r, a)$ entails that either $\text{tick}(\pi, i, [c \mapsto d], r, a) \downarrow_r$, or $\pi, j, [c \mapsto d], a, r \models f$.

(Note again the reversal of a and r in the antecedent.)

3.1.g Resets

- $\pi, i, c, a, r \models \text{accept } b \text{ in } f$ if $\pi, i, c \models a \mid (b \& \& !r)$ or $\pi, i, c, a \mid (b \& \& !r), r \models f$

The important feature of reset signals in OVA is that they are **asynchronous**, that is, they are not required to occur at tick points; rather, they are allowed to occur at any point. To specify a **synchronous** reset the keyword **CLOCK** needs to be included in the accept expression. For example, the formula “accept $b \& \& \text{CLOCK}$ in f ” says that the accept signal b is going to be sampled only at tick points. It is also possible to define *semi-synchronous* resets, by using constructions such as “accept $b \& \& \text{past}(\text{clock}, 2)$ in f ”, which says that the accept signal b will be sampled two phases after clock ticks.

- $\pi, i, c, a, r \models \text{reject } b \text{ in } f$ if $\pi, i, c \not\models r \mid (b \& \& !a)$ and $\pi, i, c, a, r \mid (b \& \& !a) \models f$

3.2 Regular Events

We start by defining the satisfaction of regular events in terms of tight satisfaction:

$\pi, i, c, a, r \models e$ if for some $j \geq i$ we have $\pi, i, j, c, a, r \models e$.

The intuition of tight satisfaction is that the event e holds **between** the points i and j . We continue by defining the tight satisfaction for regular expressions whose language does not contain the empty word, and for clocked regular events.

3.2.a Regular expressions

For a regular expression e whose language $L(e)$ does not contain the empty word, we define tight satisfaction as follows:

$\pi, i, j, c, a, r \models e$ if

- $\pi, k, c \models !a \ \&\& \ !r$ for $i \leq k < j$
- there are precisely $l \geq 0$ tick points $i_1 < \dots < i_l$ of c such that $i = i_0 < i_1$ and $i_l \leq j$
- there is a word $B = b_0 b_1 \dots b_n \in L(e)$, $n \geq 1$ such that $\pi, i_m, c \models b_m$ for $0 \leq m < l$, and either
 - $\pi, j, c \models a$ and $i_l = j$, or
 - $\pi, j, c \models a$ and $\pi, i_1, c \models b_1$, or
 - $l = n$ and $i_n = j$ and $\pi, j, c \models b_n$ and $\pi, j, c \models !r$.

Note that in the first case, only the prefix $b_0 b_1 \dots b_{l-1}$ is checked, in the second case, only the prefix $b_0 b_1 \dots b_l$ is checked, and in the third case the word B is checked in full. In the first two cases, the checking of B is terminated due to an accept signal. We denote this by $\pi, i, j, c, a, r \models_a e$.

3.2.b Clocked regular events

Clocked regular events are obtained from regular events by closing them under the operator **sclock**. For example

```
sclock c {  
  event e: p #1 q;  
}
```

defines the event e as a clocked regular event. In contrast, below

```
sclock c1 {  
  event e1: p #1 q;  
}  
  
sclock c2 {  
  event e2: e1 #1 q;  
}
```

e_2 is not a clocked regular event, since clocked regular events are not closed under regular operators. Thus, the definition of e_2 is syntactically illegal. The tight satisfaction for clocked regular events is defined as follows:

$$\begin{aligned} \pi, i, j, c, a, r &\models \text{sclock } d \ e \\ \text{if } k = \text{tick}(\pi, i, [c \mapsto d], a, r), \quad i \leq k \leq j, \quad \text{and either} \\ \text{tick}(\pi, i, [c \mapsto d], a, r) \downarrow_a \text{ or } \pi, k, j, [c \mapsto d], a, r &\models e. \end{aligned}$$

(Thus, SCLOCK always causes the evaluation of the formula at the nearest tick point, which could be the present).

3.3 Regular Expressions

For every regular expression in OVA we define a regular language in terms of the alphabet in which every letter is a subset of the states of the model, or a predicate over the states. For the sake of simplicity, we will denote these predicate as Boolean expressions. Although syntactically this notation allows for infinite number of expressions, one has to keep in mind that the number of states in the model is finite, thus, also the number of predicates over the states is finite, which results in a finite alphabet.

3.3.a Boolean expressions

For a Boolean expression b , we define $L(b) = \{ b \}$. That is, the language contains a single word, which in turns consists of a single letter, which is the predicate b . Note that in terms of subsets of states, the predicate b corresponds to the subset $\{ s : s \models b \}$, that is, the subset of all states that satisfy the Boolean expression b . In other words, b is the characteristic function of the set $\{ s : s \models b \}$.

3.3.b Intersection, && and ||

$$\begin{aligned} L(e_1 \text{ intersect } e_2) &= \\ \{ (w_1 \& u_1 \dots w_n \& u_n) : (w_1 \dots w_n) \in L(e_1), (u_1 \dots u_n) \in L(e_2) \} \\ \\ L(e_1 \&\& e_2) &= \\ \{ (w_1 \& u_1 \dots w_m \& u_m w_{m+1} \dots w_n) : m \leq n, (w_1 \dots w_n) \in L(e_1), (u_1 \dots u_m) \in L(e_2) \} \\ \cup \\ \{ (w_1 \& u_1 \dots w_n \& u_n u_{n+1} \dots u_m) : n \leq m, (w_1 \dots w_n) \in L(e_1), (u_1 \dots u_m) \in L(e_2) \} \\ \\ L(e_1 || e_2) &= \{ w : w \in L(e_1) \text{ or } w \in L(e_2) \} \end{aligned}$$

3.3.c Concatenation and Time Shift

$$L(\#[n..m] e) = L(\text{any } \#[n..m] e)$$

$$L(\#k\ e) = L(\text{any}\ \#k\ e)$$

$$L(e_1\ \#[n..m]\ e_2) = \bigcup_{(k \in [n,m])} L(e_1\ \#k\ e_2)$$

$$L(e_1\ \#0\ e_2) = \{ (w_1 \dots w_n \&\& u_1 \dots u_m) : (w_1 \dots w_n) \in L(e_1), (u_1 \dots u_m) \in L(e_2) \}$$

Note: the languages $L(e_1)$ and $L(e_2)$ must **not** contain the empty word.

For $k > 0$,

$$L(e_1\ \#k\ e_2) =$$

$$\{ (w_1 \dots w_n\ \text{any}^*[k-1]\ u_1 \dots u_m) : (w_1 \dots w_n) \in L(e_1), (u_1 \dots u_m) \in L(e_2) \}$$

Where $\text{any}^*[k-1]$ is $k-1$ consecutive appearances of any .

$$L(e_1\ ->>\ e_2) = L(e_1\ \#[1..]\ e_2)$$

3.3.d Inverse and first match

$$L(\text{inv}\ e) = \{ (b_1, \dots, b_{k-1}, !b_k) : k \leq n, (b_1, \dots, b_n) \in L(e) \}$$

Note: the languages $L(e)$ must **not** contain the empty word.

$$L(\text{first_match}\ e) = \{ (b_1) : (b_1) \in L(e) \} \cup$$

$$\{ (b_1 \dots b_n) \ \&\&\ K : n > 1, (b_1 \dots b_n) \in L(e),$$

$$K = \&\&_{[w \in L(\text{length}[1..n-1]\ \text{in}\ e)]} \{ (u_1 \dots u_k) : k < n, (u_1 \dots u_k) \in L(\text{inv}\ w) \} \}$$

Note: the languages $L(e)$ must **not** contain the empty word.

3.3.e if-then-else

$$L(\text{if}\ b\ \text{then}\ e) = \{ (b\&\&w_1\ w_2 \dots w_n) : (w_1 \dots w_n) \in L(e) \} \cup \{ (!b) \}$$

Note: the languages $L(e)$ must **not** contain the empty word.

$$L(\text{if}\ b\ \text{then}\ e_1\ \text{else}\ e_2) = \{ (b\&\&w_1\ w_2 \dots w_n) : (w_1 \dots w_n) \in L(e_1) \} \cup$$

$$\{ (!b\&\&u_1\ u_2 \dots u_n) : (u_1 \dots u_n) \in L(e_2) \}$$

Note: the languages $L(e_1)$ and $L(e_2)$ must **not** contain the empty word.

3.3.f Repetition

$$L(e^*[n..m]) = \bigcup_{(k \in [n,m])} L(e^*[k])$$

$$L(e^*[n..]) = \bigcup_{(k > n)} L(e^*[k])$$

$$L(e^*[0]) = \{\varepsilon\}$$

For $k > 0$,

$$L(e^*[k]) = \{ w_1 w_2 \dots w_k : \forall i \leq k, w_i \in L(e) \}$$

3.3.g Restrictions

$$L(\text{istrue } b \text{ in } e) = \{ (b \& w_1 \ b \& w_2 \dots b \& w_n) : (w_1 \dots w_n) \in L(e) \}$$

$$L(\text{length } [n..m] \text{ in } e) = \{ (w_1 w_2 \dots w_k) : n \leq k \leq m, (w_1 \dots w_k) \in L(e) \}$$

$$L(\text{length } [n..] \text{ in } e) = \{ (w_1 w_2 \dots w_k) : n \leq k, (w_1 \dots w_k) \in L(e) \}$$

$$L(\text{length } [n] \text{ in } e) = \{ (w_1 w_2 \dots w_n) : (w_1 \dots w_n) \in L(e) \}$$

3.4 Boolean expressions

For Boolean expressions, we start by defining the “clock and reset free” semantics of the Boolean atoms.

$$\pi, i \models b \text{ if } \pi_i(b) == \text{true}$$

That is, if b is true at the i th state of the trace π . We continue with the clock semantics of general expressions, which is defined as follows:

$$\pi, i, c \models d \text{ if } \pi, i \models [c \mapsto d]$$

Note that in particular, the above definition implies that

$$\pi, i, c \models \text{clock} \text{ if } \pi, i \models c$$

In this case we say that c *ticks at the i th point of π* , or equivalently, that i *is a tick point of c* . It is now possible to define the 6-place relation for a Boolean expression as follows:

$$\pi, i, c, a, r \models b \text{ if } \pi, i, c \models a \text{ or } \pi, i, c \models b \text{ and } \pi, i, c \models !r$$

We are now left to define the semantics of Boolean expressions in terms of 3-place relations. That will be done in the subsections of 3.4.

3.4.a Count

For a constant cnst , $\pi, i \models (\text{count}(v) == \text{cnst})$ if the number of 1's in the bit vector v is exactly cnst . The width of the return value of $\text{count}(v)$ is $\lceil \log(\text{sizeof}(v) + 1) \rceil$.

3.4.b Past, future, any

- We define the semantics of $\text{past}(v, n, c)$ for any bit vector v , $n > 1$ and a Boolean expression c . To do that, we need first to define $\text{past}(v, 0, c)$ as equal to v (although not allowed in the language) for the base of the inductive definition. Also, our semantics here will be a 4-place relation:

For a constant cnst , $\pi, i, d \models (\text{past}(v, n, c) == \text{cnst})$ if either

-
- o There is a point $j < i$ such that $\pi, j \models [d \mapsto c]$, and for the largest such j , $\pi, j, d \models (\text{past}(v, n-1, c) == \text{cnst})$, or
 - o There is no point $j < i$ such that $\pi, j \models [d \mapsto c]$, and cnst is 0.

We also define the default values of `past` as:

- o `past(v, n) : past(v, n, clock)`
- o `past(v) : past(v, 1)`

- For a constant `cnst`, $\pi, i \models (\text{future}(v) == \text{cnst})$ if $\pi, i+1 \models (v == \text{cnst})$
- The following is a tautology: $\pi, i \models \text{any}$

3.4.c Edges

- `posedge b: b && !past(b, 1, any)`
- `negedge b: !b && past(b, 1, any)`
- `edge b: b != past(b, 1, any)`

3.4.d Ended

$\pi, j, c, a, r \models \text{ended}(e)$ if either

- $\pi, j, c \models a$, or
- there exists $i \leq j$ s.t. $\pi, i, j, c \models e$ and $\pi, j, c \models !r$.

4 Operator Precedence

Priority	Operators	Associativity
22	t(params)	
21	! ~ [] Unary + -	
20	**	
19	* / %	Left to right
18	Binary + -	Left to right
17	<< >> <<< >>>	Left to right
16	< > <= >=	Left to right
15	== != === !==	Left to right
14	& ~&	Left to right
13	^ ^~ ~^	
12	~	
11	*[..]	Left to right
10	intersect	
9	&&	
8	ended inv first_match	Right to left
7		
6	posedge negedge edge	Right to left
5	If-then-else ()?:	
4	#k #[..]	Left to right
3	#0	
2	implies iff	Left to right
1	globally, eventually, followed_by, triggers, until, wuntil, next, wnext, accept, reject	Right to left

5 Templates

5.1 matched

```
matched(e) : ended(e) &&CLOCK || ended(ended(e) &&!CLOCK #1 any);
```

5.1.a Explicit Semantics of the matched Operator

$\pi, j, c, a, r \models \text{matched}(e)$ if either

- $\pi, j, c \models a$, or
- the following 3 hold:
 - $\pi, j, c \models !r$, and
 - $\pi, j \models c$, and
 - there exists $i \leq k \leq j$ s.t. $\pi, i, k, c \models e$ and for all l , $k \leq l < j$, $\pi, l \models !c$

5.2 Data checking

```
template data_check (pre1, data_in, pre2, post, data_out, clk)
{
    bit [sizeof(data_in)] data_store;
    data_store <= data_in;

    clock clk {
        formula data_consistent:
            (true* #1 pre1 #0 (data_store == data_in) #0 pre2)
            triggers (post #0 (data_out == data_store));
    }
    assert data_consistent;
}
```

5.3 Important Boolean operators

5.3.a Mutex and strong mutex

```
bool mutex(v) : count(v) <= 1;
bool strong_mutex(v) : count(v) == 1;
```

5.3.b Same

```
Bool same(v) : (& v) || (~| v);
```