SystemVerilog Concurrent Assertions Semantics

1 Introduction

This appendix presents a formal semantics for SystemVerilog concurrent assertions. Immediate assertions and coverage statements are not discussed here. Throughout this appendix, "assertion" is used to mean "concurrent assertion". The semantics is defined by a relation that determines when a finite or infinite word (i.e., trace) satisfies an assertion. Intuitively, such a word represents a sequence of valuations of SystemVerilog variables sampled at the finest relevant granularity of time (e.g., at the granularity of simulator cycles). The process by which such words are produced is closely related to the SytemVerilog scheduling semantics and is not defined here. In this appendix, words are assumed to be sequences of elements, each element being either a set of atomic propositions or one of two special symbols used as placeholders when extending finite words. The atomic propositions are not further defined. The meaning of satisfaction of a SystemVerilog boolean expression by a set of atomic propositions is assumed to be understood.

The semantics is based on an abstract syntax for SystemVerilog assertions. There are several advantages to using the abstract syntax rather than the full SystemVerilog Assertions BNF.

1. The abstract syntax facilitates separation of derived operators from basic operators. The satisfaction relation is defined explicitly only for assertions built from basic operators.
2. The abstract syntax avoids reliance on operator precedence, associativity, and auxiliary rules for resolving syntactic and semantic ambiguities.
3. The abstract syntax simplifies the assertion language by eliminating some features that tend to encumber the definition of the formal semantics.
   (a) The abstract syntax eliminates local variable declarations. The semantics of local variables is written with implicit types.
   (b) The abstract syntax eliminates instantiation of sequences and properties. The semantics of an assertion with an instance of a sequence or property is the same as the semantics of a related assertion obtained by replacing the sequence or property instance with an explicitly written sequence or property. The explicit sequence or property is obtained from the body of the associated declaration by substituting actual arguments for formal arguments.
   (c) The abstract syntax does not allow implicit clocks. Clocking event controls must be applied explicitly in the abstract syntax.
   (d) The abstract syntax does not allow explicit procedural enabling conditions for assertions. Procedural enabling conditions are utilized in the semantics definition (see Subsection 3.3.1), but the method for extracting such conditions is not defined in this appendix.

In order to use this appendix to determine the semantics of a SystemVerilog assertion, the assertion must first be transformed into an enabling condition together with an assertion in the abstract syntax. This transformation involves eliminating sequence and property instances by substitution, eliminating local variable declarations, introducing parentheses, determining the enabling condition, determining implicit or inferred event controls, and eliminating redundant event controls. For example, the following SystemVerilog assertion

```
sequence s(x,y); x ##1 y; endsequence
sequence t(z); @(c) z[*1:2] ##1 B; endsequence
always @(c) if (b) assert property (s(A,B) |=> t(A));
```

is transformed into the enabling condition “*b*” together with the assertion

```
always @(c) assert property ((A ##1 B) |=> (A[*1:2] ##1 B))
```

in the abstract syntax.

2 Abstract Syntax

2.1 Abstract grammars

In the following abstract grammars, *b* denotes a boolean expression, *v* denotes a local variable name, and *e* denotes an expression.

The abstract grammar for unclocked sequences is

```
R ::= b                                  // "boolean expression" form
    | ( 1, v = e )                       // "local variable sampling" form
    | ( R )                              // "parenthesis" form
    | ( R ##1 R )                        // "concatenation" form
    | ( R ##0 R )                        // "fusion" form
    | ( R or R )                         // "or" form
    | ( R intersect R )                  // "intersect" form
    | first_match ( R )                  // "first match" form
    | R [*0]                             // "null repetition" form
    | R [*1:$]                           // "unbounded repetition" form
```

The abstract grammar for clocked sequences is

```
S ::= @(b) R                                  // "clock" form
    | ( S ## S )                              // "concatenation" form
```

The abstract grammar for unclocked properties is

```
P ::= [disable iff ( b )] [not] R                      // "sequence" form
    | [disable iff ( b )] [not] ( R |-> [not] R )      // "implication" form
```

The abstract grammar for clocked properties is

```
Q ::= @(b) P                                       // "clock" form
    | [disable iff ( b )] [not] S                  // "sequence" form
    | [disable iff ( b )] [not] ( S |-> [not] S )  // "implication" form
```

The abstract grammar for assertions is

```
A ::= always assert property Q                // "always" form
    | always @(b) assert property P           // "always with clock" form
    | initial assert property Q               // "initial" form
    | initial @(b) assert property P          // "initial with clock" form
```

2.2 Notations

The following auxiliary notions will be used in defining the semantics.

- φ is an *unclocked property fragment* provided "**disable iff** (*b*) φ" is an unclocked property.
- *N* is a *negation specifier* if *N* is either the empty token or "**not**".

Throughout the sequel, the following notational conventions will be used: *b*, *c* denote boolean expressions; *v* denotes a local variable name; *e* denotes an expression; φ denotes an unclocked property fragment; N, N_1, N_2 denote negation specifiers; R, R_1, R_2 denote unclocked sequences; S, S_1, S_2 denote clocked sequences; *P* denotes an unclocked property; *Q* denotes a clocked property; *A* denotes an assertion; *i*, *j*, *k*, *m*, *n* denote non-negative integer constants.

2.3 Derived forms

Internal parentheses are omitted in compositions of the (associative) operators "##1" and "**or**".

2.3.1 Derived non-overlapping implication operator

- (R_1 |=> N R_2) ≡ ((R_1 ##1 1) |-> N R_2) .
- (S_1 |=> N S_2) ≡ ((S1 ## @(1) 1) |-> N S_2) .

2.3.2 Derived consecutive repetition operators

- Let $m > 0$. R [*m] ≡ (R ##1 R ##1 ··· ##1 R) // m copies of R .
- R [*0:$] ≡ (R [*0] **or** R [*1:$]) .
- Let $m \leq n$. R [*m:n] ≡ (R [*m] **or** R [*m +1] **or** ··· **or** R [*n]) .
- Let $m > 1$. R [*m:$] ≡ (R [*m – 1] ##1 R [*1:$]) .

2.3.3 Derived delay and concatenation operators

Let $m \leq n$.

- (##[m:n] R) ≡ (1[*m:n] ##1 R) .
- (##[m:$] R) ≡ (1[*m:$] ##1 R) .
- (##m R) ≡ (1[*m] ##1 R) .
- Let $m > 0$. (R_1 ##[m:n] R_2) ≡ (R_1 ##1 1[*m – 1:n – 1] ##1 R_2) .
- Let $m > 0$. (R_1 ##[m:$] R_2) ≡ (R_1 ##1 1[*m – 1:$] ##1 R_2) .
- Let $m > 1$. (R_1 ##m R_2) ≡ (R_1 ##1 1[*m – 1] ##1 R_2) .
- (R_1 ##[0:0] R_2) ≡ (R_1 ##0 R_2) .
- Let $n > 0$. (R_1 ##[0:n] R_2) ≡ ((R_1 ##0 R_2) **or** (R_1 ##[1:n] R_2)) .
- (R_1 ##[0:$] R_2) ≡ ((R_1 ##0 R_2) **or** (R_1 ##[1:$] R_2)) .

2.3.4 Derived non-consecutive repetition operators

Let $m \leq n$.

- b [*->m:n] ≡ (!b [*0:$] ##1 b)[*m:n] .
- b [*->m:$] ≡ (!$b$ [*0:$] ##1 b)[*m:$] .
- b [*->m] ≡ (!b [*0:$] ##1 b)[*m] .
- b [*=m:n] ≡ (b [*->m:n] ##1 !b [*0:$]) .
- b [*=m:$] ≡ (b [*->m:$] ##1 !b [*0:$]) .
- b [*=m] ≡ (b [*->m] ##1 !b [*0:$]) .

2.3.5 Other derived operators

- (R_1 **and** R_2)

$\equiv$ (((R_1 ##1 1[*0:$]) **intersect** R_2) **or** (R_1 **intersect** (R_2 ##1 1[*0:$]))) .

- (R_1 **within** R_2) $\equiv$ ((1[*0:$] ##1 R_1 ##1 1[*0:$]) **intersect** R_2) .
- (b **throughout** R) $\equiv$ ((b [*0:$]) **intersect** R) .
- (b, $v = e$) $\equiv$ (b ##0 (1, $v = e$)) .

3 Semantics

Let **P** be the set of atomic propositions.

The semantics of assertions and properties is defined via a relation of satisfaction by empty, finite, and infinite words over the alphabet $\Sigma = 2^{\mathbf{P}} \cup \{\top, \bot\}$. Such a word is an empty, finite, or infinite sequence of elements of Σ. The number of elements in the sequence is called the *length* of the word, and the length of word w is denoted $|w|$. Note that $|w|$ is either a non-negative integer or infinity.

The sequence elements of a word are called its *letters* and are assumed to be indexed consecutively beginning at zero. If $|w| > 0$, then the first letter of w is denoted w^0; if $|w| > 1$, then the second letter of w is denoted w^1; and so forth. $w^{i..}$ denotes the word obtained from w by deleting its first i letters. If $i < |w|$, then $w^{i..} = w^i w^{i+1}$.... If $i \geq |w|$, then $w^{i..}$ is empty.

If $i \leq j$, then $w^{i,j}$ denotes the finite word obtained from w by deleting its first i letters and also deleting all letters after its (j + 1)st. If $i \leq j < |w|$, then $w^{i,j} = w^i w^{i+1} ... w^j$.

If w is a word over Σ, define $\overline{w}$ to be the word obtained from w by interchanging $\top$ with $\bot$. More precisely, $\overline{w}^i = \top$ if $w^i = \bot$; $\overline{w}^i = \bot$ if $w^i = \top$; and $\overline{w}^i = w^i$ if w^i is an element in $2^{\mathbf{P}}$.

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. See the subsection on rewrite rules for clocks below.

It is assumed that the satisfaction relation $\zeta \models b$ is defined for elements ζ in $2^{\mathbf{P}}$ and boolean expressions b. For any boolean expression b, define

$\top \models b$ and $\bot \not\models b$.

3.1 Rewrite rules for clocks

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

- @(c) $b \longmapsto$ (!c [*0:$] ##1 c & b) .
- @(c)(1, $v = e$) $\longmapsto$ (@(c) 1 ##0 (1, $v = e$)) .
- @(c)(R) $\longmapsto$ @(c) R .
- @(c)(R_1 ##1 R_2) $\longmapsto$ (@(c) R_1 ##1 @(c) R_2) .
- @(c)(R_1 ##0 R_2) $\longmapsto$ (@(c) R_1 ##0 @(c) R_2) .
- @(c)(R_1 **or** R_2) $\longmapsto$ (@(c) R_1 **or** @(c) R_2) .
- @(c)(R_1 **intersect** R_2) $\longmapsto$ (@(c) R_1 **intersect** @(c) R_2) .
- @(c) **first_match** (R) $\longmapsto$ **first_match** (@(c) R) .
- @(c) R [*0] $\longmapsto$ (@(c) R)[*0] .

- `@(`c`)` R `[*1:$]` $\longmapsto$ `(` `@(`c`)` R `)[*1:$]` .
- `@(`c`)` **`disable iff`** `(` b `)` φ $\longmapsto$ **`disable iff`** `(` b `)` `@(`c`)` φ .
- `@(`c`)` **`not`** b $\longmapsto$ `@(`c`)` `!`b .
- `@(`c`)` **`not`** R $\longmapsto$ **`not`** `@(`c`)` R, provided R is not a boolean expression.
- `@(`c`)` N_1 `(` R_1 `|->` N_2 R_2 `)` $\longmapsto$ N_1 `(` `@(`c`)` R_1 `|->` `@(`c`)` N_2 R_2 `)` .
- `(` S_1 `##` S_2 `)` $\longmapsto$ `(` S_1 `##1` S_2 `)` .

3.2 Tight satisfaction without local variables

Tight satisfaction is denoted by $\models\!\!\!\models$. For unclocked sequences without local variables, tight satisfaction is defined as follows. w, x, y, z denote finite words over Σ.

- $w \models\!\!\!\models b$ iff $|w| = 1$ and $w \models b$.
- $w \models\!\!\!\models$ `(` R `)` iff $w \models\!\!\!\models R$.
- $w \models\!\!\!\models$ `(` R_1 `##1` R_2 `)` iff there exist x, y such that $w = xy$ and $x \models\!\!\!\models R_1$ and $y \models\!\!\!\models R_2$.
- $w \models\!\!\!\models$ `(` R_1 `##0` R_2 `)` iff there exist x, y, z such that $w = xyz$ and $|y| = 1$, and $xy \models\!\!\!\models R_1$ and $yz \models\!\!\!\models R_2$.
- $w \models\!\!\!\models$ `(` R_1 **`or`** R_2 `)` iff either $w \models\!\!\!\models R_1$ or $w \models\!\!\!\models R_2$.
- $w \models\!\!\!\models$ `(` R_1 **`intersect`** R_2 `)` iff both $w \models\!\!\!\models R_1$ and $w \models\!\!\!\models R_2$.
- $w \models\!\!\!\models$ **`first_match`** `(` R `)` iff both

  — $w \models\!\!\!\models R$ and

  — if there exist x, y such that $w = xy$ and $x \models\!\!\!\models R$, then y is empty.
- $w \models\!\!\!\models R$ `[*0]` iff $|w| = 0$.
- $w \models\!\!\!\models R$ `[*1:$]` iff there exist words $w_1, w_2, ..., w_j$ ($j \geq 1$) such that $w = w_1 w_2 ... w_j$ and for every i such that $1 \leq i \leq j$, $w_i \models\!\!\!\models R$.

If S is a clocked sequence, then $w \models\!\!\!\models S$ iff $w \models\!\!\!\models S'$, where S' is the unclocked sequence that results from S by applying the rewrite rules.

3.3 Satisfaction without local variables

3.3.1 Satisfaction by infinite words

w denotes an infinite word over Σ. Assume that all properties, sequences, and unclocked property fragments do not involve local variables.

Assertion Satisfaction:

For the definition of assertion satisfaction, b denotes the boolean expression representing the enabling condition for the assertion. Intuitively, b is derived from the conditions in the context of a procedural assertion, while b is "`1`" for a declarative assertion.

- $w, b \models$ **`always`** `@(`c`)` **`assert property`** P iff for every $i \geq 0$ such that $w^i \models c$, if $\overline{w}^i \models b$ then $w^{i..} \models$ `@(`c`)` P.
- $w, b \models$ **`always assert property`** Q iff for every $i \geq 0$, if $\overline{w}^i \models b$ then $w^{i..} \models Q$.
- $w, b \models$ **`initial`** `@(`c`)` **`assert property`** P iff (if there exists $i \geq 0$ such that $w^i \models c$, then for the first

such i, if $\overline{w}^{\,i} \models b$ then $w^{\,i..} \models \texttt{@(}c\texttt{)}\ P$).

- $w, b \models$ **`initial assert property`** Q iff (if $\overline{w}^{\,0} \models b$ then $w \models Q$) .

Property Satisfaction:

- $w \models Q$ iff $w \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
- $w \models$ **`disable iff (`**b**`)`** φ iff either $w \models \varphi$ or there exists $k \geq 0$ such that $w^{\,k} \models b$ and $w^{\,0,\,k-1}\top^{\omega} \models \varphi$. Here, $w^{\,0,\,-1}$ denotes the empty word.
- $w \models$ **`not`** φ iff $\overline{w} \not\models \varphi$.
- $w \models R$ iff there exists $j \geq 0$ such that $w^{\,0,\,j} \models R$.
- $w \models$ (R_1 `|->` $N\,R_2$) iff for every $j \geq 0$ such that $\overline{w}^{\,0,\,j} \models R_1$, $w^{\,j..} \models N\,R_2$.

Remark: It can be proved that $w \models$ **`not`** b iff $w \models$ `!`b.

3.3.2 Satisfaction by finite words

This subsection defines weak and strong satisfaction, denoted $\models^{-}$ and $\models^{+}$ (respectively) of an assertion A by a finite word w over Σ. These relations are defined in terms of the relation of satisfaction by infinite words as follows:

- $w \models^{-} A$ iff $w\top^{\omega} \models A$.
- $w \models^{+} A$ iff $w\bot^{\omega} \models A$.

A tool checking for satisfaction of A by the finite word w should return

- "true" if $w \models^{+} A$.
- "false" if $w \not\models^{-} A$.
- "unknown" otherwise.

3.4 Local variable flow

This subsection defines inductively how local variable names flow through unclocked sequences. Below, "$\cup$" denotes set union, "$\cap$" denotes set intersection, "–" denotes set difference, and "{}" denotes the empty set.

The function "*sample*" takes a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that are sampled in the sequence.

The function "*block*" takes a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that are blocked from flowing out of the sequence.

The function "*flow*" takes a set X of local variable names and a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that flow out of the sequence given the set X of local variable names that flow into the sequence.

The function "*sample*" is defined by

- *sample* (b) = {} .
- *sample* ((`1,` $v = e$)) = {v} .
- *sample* ((R)) = *sample* (R) .
- *sample* ((R_1 `##1` R_2)) = *sample* (R_1) $\cup$ *sample* (R_2) .
- *sample* ((R_1 `##0` R_2)) = *sample* (R_1) $\cup$ *sample* (R_2) .

- *sample* ((R_1 **or** R_2)) = *sample* (R_1) ∪ *sample* (R_2) .
- *sample* ((R_1 **intersect** R_2)) = *sample* (R_1) ∪ *sample* (R_2) .
- *sample* (**first_match** (R)) = *sample* (R) .
- *sample* (R `[*0]`) = {} .
- *sample* (R `[*1:$]`) = *sample* (R) .

The function "*block*" is defined by

- *block* (b) = {} .
- *block* ((`1,` $v = e$)) = {} .
- *block* ((R)) = *block* (R) .
- *block* ((R_1 `##1` R_2)) = (*block* (R_1) − *flow* ({}, R_2)) ∪ *block* (R_2) .
- *block* ((R_1 `##0` R_2)) = (*block* (R_1) − *flow* ({}, R_2)) ∪ *block* (R_2) .
- *block* ((R_1 **or** R_2)) = *block* (R_1) ∪ *block* (R_2) .
- *block* ((R_1 **intersect** R_2))
  = *block* (R_1) ∪ *block* (R_2) ∪ (*sample* (R_1) ∩ *sample* (R_2)) .
- *block* (**first_match** (R)) = *block* (R) .
- *block* (R `[*0]`) = {} .
- *block* (R `[*1:$]`) = *block* (R) .

The function "*flow*" is defined by

- *flow* (X, b) = X .
- *flow* (X, (`1,` $v = e$)) = X ∪ {v} .
- *flow* (X, (R)) = *flow* (X, R) .
- *flow* (X, (R_1 `##1` R_2)) = *flow* (*flow* (X, R_1), R_2) .
- *flow* (X, (R_1 `##0` R_2)) = *flow* (*flow* (X, R_1), R_2) .
- *flow* (X, (R_1 **or** R_2)) = *flow* (X, R_1) ∩ *flow* (X, R_2) .
- *flow* (X, (R_1 **intersect** R_2))
  = (*flow* (X, R_1) ∪ *flow* (X, R_2)) − *block* ((R_1 **intersect** R_2)) .
- *flow* (X, **first_match**(R)) = *flow* (X, R) .
- *flow* (X, R `[*0]`) = X .
- *flow* (X, R `[*1:$]`) = *flow* (X, R) .

Remark: It can be proved that *flow* (X, R) = (X ∪ *flow* ({}, R)) − *block* (R) . It follows that *flow* ({}, R) and *block* (R) are disjoint. It can also be proved that *flow* ({}, R) is a subset of *sample* (R).

3.5 Tight satisfaction with local variables

A *local variable context* is a function that assigns values to local variable names. If L is a local variable context, then dom(L) denotes the set of local variable names that are in the domain of L. If $D \subseteq$ dom(L), then $L|_D$

means the local variable context obtained from L by restricting its domain to D.

In the presence of local variables, tight satisfaction is a four-way relation defining when a finite word w over the alphabet Σ together with an input local variable context L_0 satisfies an unclocked sequence R and yields an output local variable context L_1. This relation is denoted

$$w, L_0, L_1 \models R .$$

and is defined below. It can be proved that the definition guarantees that

$$w, L_0, L_1 \models R \quad \text{implies} \quad \text{dom}(L_1) = \mathit{flow}\,(\text{dom}(L_0), R) .$$

- $w, L_0, L_1 \models$ `(` `1,` $v = e$ `)` iff $|w| = 1$ and $w^0 \models$ `1` and

  $$L_1 = \{(v, e[L_0, w^0])\} \cup L_0|_D ,$$

  where $e[L_0, w^0]$ denotes the value obtained from e by evaluating first according to L_0 and second according to w^0 and $D = \text{dom}(L_0) - \{v\}$.

- $w, L_0, L_1 \models b$ iff $|w| = 1$ and $w^0 \models b[L_0]$ and $L_1 = L_0$. Here $b[L_0]$ denotes the expression obtained from b by substituting values from L_0 .

- $w, L_0, L_1 \models$ `(` R `)` iff $w, L_0, L_1 \models R$.

- $w, L_0, L_1 \models$ `(` R_1 `##1` R_2 `)` iff there exist x, y, L' such that $w = xy$ and $x, L_0, L' \models R_1$ and $y, L', L_1 \models R_2$.

- $w, L_0, L_1 \models$ `(` R_1 `##0` R_2 `)` iff there exist x, y, z, L' such that $w = xyz$ and $|y| = 1$, and $xy, L_0, L' \models R_1$ and $yz, L', L_1 \models R_2$.

- $w, L_0, L_1 \models$ `(` R_1 **`or`** R_2 `)` iff there exists L' such that both of the following hold:

  — either $w, L_0, L' \models R_1$ or $w, L_0, L' \models R_2$, and

  — $L_1 = L'|_D$, where $D = \mathit{flow}\,(\text{dom}(L_0),$ `(` R_1 **`or`** R_2 `))` .

- $w, L_0, L_1 \models$ `(` R_1 **`intersect`** R_2 `)` iff there exist L', L'' such that $w, L_0, L' \models R_1$ and $w, L_0, L'' \models R_2$ and $L_1 = L'|_{D'} \cup L''|_{D''}$, where

  $D' = \mathit{flow}\,(\text{dom}(L_0), R_1) - (\mathit{block}\,((\ R_1$ **`intersect`** $R_2\ )) \cup \mathit{sample}\,(R_2))$
  $D'' = \mathit{flow}\,(\text{dom}(L_0), R_2) - (\mathit{block}\,((\ R_1$ **`intersect`** $R_2\ )) \cup \mathit{sample}\,(R_1))$

  Remark: It can be proved that if $w, L_0, L' \models R_1$ and $w, L_0, L'' \models R_2$, then $L'|_{D'} \cup L''|_{D''}$ is a function.

- $w, L_0, L_1 \models$ **`first_match`** `(` R `)` iff both

  — $w, L_0, L_1 \models R$ and

  — if there exist x, y, L' such that $w = xy$ and $x, L_0, L' \models R$, then y is empty.

- $w, L_0, L_1 \models R$ `[*0]` iff $|w| = 0$ and $L_1 = L_0$.

- $w, L_0, L_1 \models R$ `[*1:$]` iff there exist $L_{(0)} = L_0, w_1, L_{(1)}, w_2, L_{(2)}, \ldots, w_j, L_{(j)} = L_1$ ($j \geq 1$) such that $w = w_1 w_2 \ldots w_j$ and for every i such that $1 \leq i \leq j$, $w_i, L_{(i-1)}, L_{(i)} \models R$.

If S is a clocked sequence, then $w, L_0, L_1 \models S$ iff $w, L_0, L_1 \models S'$, where S' is the unclocked sequence that results from S by applying the rewrite rules.

3.6 Satisfaction with local variables

3.6.1 Satisfaction by infinite words

w denotes an infinite word over Σ. L_0, L_1 denote local variable contexts.

The rules defining assertion satisfaction are identical to those without local variables, but with the understand-

ing that the underlying properties can have local variables.

Property Satisfaction:

- $w \models Q$ iff $w, \{\} \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
- $w, \{\} \models$ `disable iff (`b`)` φ iff either $w, \{\} \models \varphi$ or there exists $k \geq 0$ such that $w^{0,k-1}\top^{\omega}, \{\} \models \varphi$. Here, $w^{0,-1}$ denotes the empty word.
- $w, L_0 \models$ `not` φ iff $\overline{w}, L_0 \not\models \varphi$.
- $w, L_0 \models R$ iff there exist $j \geq 0$ and L_1 such that $w^{0,j}, L_0, L_1 \models R$.
- $w, L_0 \models$ `(` R_1 `|->` $N R_2$ `)` iff for every $j \geq 0$ and L_1 such that $\overline{w}^{0,j}, L_0, L_1 \models R_1, w^{j..}, L_1 \models N R_2$.

3.6.2 Satisfaction by finite words

The definition is identical to that without local variables, but with the understanding that the underlying properties can have local variables.

4 Extended Expressions

This section describes the semantics of several constructs that are used like expressions, but whose meaning at a point in a word can depend both on the letter at that point and on previous letters in the word. By abuse of notation, the meanings of these extended expressions are defined for letters denoted "w^j" even though they depend also on letters w^i for $i \leq j$. The reason for this abuse is to make clear the way these definitions should be used in combination with those in preceding sections.

4.1 Extended booleans

w denotes an infinite word over Σ, T denotes a clocked or unclocked sequence.

- $w^j \models T$`.ended` iff there exist $i \leq j$ and L such that $w^{i,j}, \{\}, L \models T$.
- $w^j \models$ `@(`c`)(`T`.matched)` iff there exists $i < j$ such that $w^i \models T$`.ended` and $w^{i+1,j}, \{\}, \{\} \models$ `(!`c `[*0:$] ##1` c`)` .
- $w^j \models$ `@(`c`)$stable(`e`)` iff there exists $i < j$ such that $w^{i,j}, \{\}, \{\} \models$ `(`c `##1` c `[*->1])` and $e[w^i] = e[w^j]$.
- $w^j \models$ `@(`c`)$rose(`e`)` iff $b[w^j] = 1$ and (if there exists $i < j$ such that $w^{i,j}, \{\}, \{\} \models$ `(`c `##1` c `[*->1])` then $b[w^i] \neq 1$), where b is the least-signifcant bit of e.
- $w^j \models$ `@(`c`)$fell(`e`)` iff $b[w^j] = 0$ and (if there exists $i < j$ such that $w^{i,j}, \{\}, \{\} \models$ `(`c `##1` c `[*->1])` then $b[w^i] \neq 0$), where b is the least-signiffcant bit of e.

4.2 Past

w denotes an infinite word over Σ.

- Let $n \geq 1$. If there exist $i < j$ such that $w^{i,j}, \{\}, \{\} \models$ `(`c `##1` c `[*->`$n-1$`])`, then `@(`c`)$past(`e`,` n`)`$[w^j] = e[w^i]$. Otherwise, `@(`c`)$past(`e`,` n`)`$[w^j]$ has the value `x`.
- `$past(`e`)` $\equiv$ `$past(`e`,1)` .