Annex H
SystemVerilog Concurrent Assertions Semantics

H.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 SystemVerilog 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 non-recursive property is the same as the semantics of a related assertion obtained by replacing the sequence or non-recursive 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. A separate section defines the semantics of instances of recursive properties in terms of the semantics of instances of non-recursive properties. LRM 130

   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.

4) The abstract syntax eliminates the distinction between *property_expr* and *property_spec* from the full BNF. Without the distinction, **`disable iff`** is a general, nestable property-building operator, while in the full BNF **`disable iff`** can be attached only at the top level of a property. Semantically, there is no need for this restriction on the placement of **`disable iff`**. The abstract syntax thus eliminates an unnecessary semantic layer while maintaining the simple inductive form for the definition of the semantics of properties. As a result, semantics are given for some properties that do not correspond to forms from the full BNF, but this does not degrade the definitions for the properties that do correspond to forms from the full BNF. LRM 130

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. For assertions that LRM 130

do not involve recursive properties, this transformation involves eliminating sequence and non-recursive property instances by substitution, eliminating local variable declarations, introducing parentheses, determining the enabling condition, determining implicit or inferred clocking event controls, and eliminating redundant clocking event controls. For example, the following SystemVerilog assertion

LRM 131

```
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.

If the SystemVerilog assertion involves instances of recursive properties, then the transformation replaces these instances with placeholder functions of the actual arguments. The semantics of an instance of a recursive property is defined in terms of associated non-recursive properties in Section H.5. Once the semantics of the recursive property instances are understood, the placeholder functions are treated as properties with these semantics. Then the ordinary definitions can be applied to the transformed assertion in the abstract syntax together with placeholder functions.

LRM 130

H.2 Abstract Syntax

H.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*

```
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

LRM 131
JH3
JH4

```
~~P~~ ::= ~~R~~R                     // "sequence" form
    | ( ~~P~~P )                     // "parenthesis" form
    | not ~~P~~P                     // "negation" form
    | ( ~~P~~P or ~~P~~P )           // "or" form
    | ( ~~P~~P and ~~P~~P )          // "and" form
    | ( ~~R~~R |-> ~~P~~P )          // "implication" form
    | disable iff ( ~~b~~b ) ~~P~~P  // "reset" form
```

Each instance of *R* in this production must be a non-degenerate unclocked sequence. See H.3.2 and H.3.5 for the definition of non-degeneracy.

JH5
SVA 1

The abstract grammar for clocked properties is

```
~~Q~~Q ::= @( ~~b~~b ) ~~P~~P          // "clock" form
```

LRM 131


```
    | S                          // "sequence" form
    | ( Q )                      // "parenthesis" form
    | not Q                      // "negation" form
    | ( Q or Q )                 // "or" form
    | ( Q and Q )                // "and" form
    | ( S |-> Q )                // "implication" form
    | disable iff ( b ) Q        // "reset" form
```

Each instance of *S* in this production must be a non-degenerate clocked sequence. See H.3.2 and H.3.5 for the definition of non-degeneracy.

JH6
SVA 2

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
```

H.2.2 Notations

JH7
LRM 131

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

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; *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*, P_1, P_2 denotes an unclocked property; *Q* denotes a clocked property; *A* denotes an assertion; *i*, *j*, *k*, *m*, *n* denote non-negative integer constants.

JH8
JH9

H.2.3 Derived forms

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

H.2.3.1 Derived non-overlapping implication operator

LRM 131

- (R_1 |=> P) ≡ ((R_1 ##1 1) |-> P) .
- (S_1 |=> Q) ≡ ((S_1 ## @(1) 1) |-> Q) .

H.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 $m \le n$. R [*m:n] ≡ (R [*m] **or** R [*m+1] **or** ··· **or** R [*n]) .
- Let $m > 1$. R [*m:$] ≡ (R [*m – 1] ##1 R [*1:$]) .

H.2.3.3 Derived delay and concatenation operators

Let $m \le n$.

- (##[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)) .

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

H.2.3.5 Other derived operators

- (R_1 **and** R_2)
  ≡ (((R_1 ##1 1[*0:$]) **intersect** R_2) **or** (R_1 **intersect** (R_2 ##1 1[*0:$]))) .
- (R_1 **within** R_2) ≡ ((1[*0:$] ##1 R_1 ##1 1[*0:$]) **intersect** R_2) .
- (b **throughout** R) ≡ ((b [*0:$]) **intersect** R) .
- (R, $v = e$) ≡ (R ##0 (1, $v = e$)) . LRM 129
- (R, $v_1 = e_1$,... ,$v_k = e_k$) ≡ ((R, $v_1 = e_1$) ##0 (1, $v_2 = e_2$,... , $v_k = e_k$)) **for** $k > 1$ SVA 3
- (**if** (b) P) ≡ (b |-> P) LRM 131
- (**if** (b) P_1 **else** P_2) ≡ ((b |-> P_1) **and** (!b |-> P_2))

H.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 $\Sigma = 2^{\mathbf{P}} \cup \{\mathsf{T}, \bot\}$. 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. SVA 4

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 T with ⊥. More precisely, $\overline{w}^i = \mathsf{T}$ if $w^i = \bot$; $\overline{w}^i = \bot$ if $w^i = \mathsf{T}$; and $\overline{w}^i = w^i$ if w^i is an element in $2^{\mathbf{P}}$. SVA 5,6,7

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

- $\mathsf{T} \models b$ and $\bot \not\models b$. SVA 8

H.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`) (`P`)` $\longmapsto$ `(@(`c`)` P `)`. LRM 131, JH10
- `@(`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`)` $P \longmapsto$ **disable iff** `(`b`) @(`c`)` P. LRM 131
- `@(`c`)` **not** $P \longmapsto$ **not** `@(`c`)` P. LRM 131
- `@(`c`) (`R `|->` P `)` $\longmapsto$ `(@(`c`)` R `|-> @(`c`)` P `)`. LRM 131
- `(`S_1 `##` S_2 `)` $\longmapsto$ `(`S_1 `##1` S_2 `)`.
- `@(`c`) (`P_1 **or** P_2 `)` $\longmapsto$ `(@(`c`)` P_1 **or** `@(`c`)` P_2 `)`. LRM 131
- `@(`c`) (`P_1 **and** P_2 `)` $\longmapsto$ `(@(`c`)` P_1 **and** `@(`c`)` P_2 `)`. JH11

H.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 $w^0 \models b$. SVA 9
- $w \models\!\!\models$ `(`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 $w \models\!\!\models R_2$.

- $w \models$ **first_match** (R) iff both
  - — $w \models R$ and
  - — if there exist x, y such that $w = xy$ and $\overline{x} \models R$, then y is empty.
- $w \models R$ [*0] iff $|w| = 0$.
- $w \models R$ [*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 $1 \leq i \leq j$, $w_i \models R$.

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

An unclocked sequence R is *non-degenerate* iff there exists a non-empty finite word w over Σ such that $w \models R$. A clocked sequence S is *non-degenerate* iff the unclocked sequence S' that results from S by applying the rewrite rules is non-degenerate.

H.3.3 Satisfaction without local variables

H.3.3.1 Neutral satisfaction

w denotes a non-empty finite or infinite word over Σ. Assume that all properties, sequences, and unclocked property fragments do not involve local variables.

Neutral satisfaction of assertions:

For the definition of neutral satisfaction of assertions, 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 $0 \leq i < |w|$ such that $\overline{w}^{\,i} \models c$ and ~~$w^{\,i}$~~ $\{\overline{w}\}^{\,i} \models b$, $w^{\,i..} \models$ @(c) P.
- $w, b \models$ **always assert property** Q iff for every $0 \leq i < |w|$, if $\overline{w}^{\,i} \models b$ then $w^{\,i..} \models Q$.
- $w, b \models$ **initial** @(c) **assert property** P iff ~~(~~for every $0 \leq i < |w|$ such that $\overline{w}^{\,0,\,i} \models$!c [*0:$] ##1 c and $\overline{w}^{\,i} \models b$, ~~then~~ $w^{\,i..} \models$ @(c) P ~~)~~.
- $w, b \models$ **initial assert property** Q iff (if $\overline{w}^{\,0} \models b$ then $w \models Q$) .

Neutral satisfaction of properties:

- $w \models$ (P) iff $w \models P$.
- $w \models Q$ iff $w \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
- $w \models$ **disable iff** (b) P iff either $w \models P$ or there exists $0 \leq k < |w|$ such that $w^{\,k} \models b$ and $w^{\,0,\,k-1}$ ~~$\vee$~~ $\top^{\omega} \models P$. Here, $w^{\,0,\,-1}$ denotes the empty word.
- $w \models$ **not** P iff $\overline{w} \not\models P$.
- $w \models R$ iff there exists $0 \leq j < |w|$ such that $w^{\,0,\,j} \models R$.
- $w \models$ (R |-> P) iff for every $0 \leq j < |w|$ such that $\overline{w}^{\,0,\,j} \models R$, $w^{\,j..} \models P$.
- $w \models$ (P_1 **or** P_2) iff $w \models P_1$ or $w \models P_2$.
- $w \models$ (P_1 **and** P_2) iff $w \models P_1$ and $w \models P_2$.

Remark: ~~It~~ Since w is non-empty, it can be proved that $w \models$ **not** b iff $w \models$!b.

H.3.3.2 Weak and strong satisfaction by finite words

SVA 23

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

SVA 24

SVA 25

JH17

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

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

JH18

- ~~"true"~~ "holds strongly" if $w \models^+ A$.
- ~~"false"~~ "fails" if $w \not\models^- A$.
- ~~"unknown"~~ "pending" or "holds weakly" otherwise.

H.3.4 Local variable flow

SVA 26

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 (i.e., assigned) in the sequence.

JH19

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) $\cup$ *sample* (R_2) .
- *sample* (`(` R_1 **`intersect`** R_2 `)`) = *sample* (R_1) $\cup$ *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)) $\cup$ *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).

H.3.5 Tight satisfaction with local variables

SVA 27

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 $w, L_0, L_1 \models R$ implies dom(L_1) = *flow* (dom(L_0), R) .

JH20

~~$w, L_0, L_1 \models R$ implies dom(L_1) = *flow* (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 = dom(L_0) – {v}~~ and D = dom(L_0) – {v}. In case $w^0 \in \{\mathsf{T}, \bot\}$, $e[L_0, \mathsf{T}]$ and $e[L_0, \bot]$ can be any constant values of the type of e.

JH21

SVA 28

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

~~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 = dom(L_0) – {v}. In case $w^0 \in \{\mathsf{T}, \bot\}$, $e[L_0, \mathsf{T}]$ and $e[L_0, \bot]$ can be any constant values of the type of e.~~

SVA 28

- $w, L_0, L_1 \models\!\!\models b$ 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 .
- $w, L_0, L_1 \models\!\!\models (\ R\ )$ iff $w, L_0, L_1 \models\!\!\models R$.
- $w, L_0, L_1 \models\!\!\models (\ R_1\ \texttt{\#\#1}\ R_2\ )$ iff there exist x, y, L' such that $w = xy$ and $x, L_0, L' \models\!\!\models R_1$ and $y, L', L_1 \models\!\!\models R_2$.
- $w, L_0, L_1 \models\!\!\models (\ R_1\ \texttt{\#\#0}\ R_2\ )$ iff there exist x, y, z, L' such that $w = xyz$ and $|y| = 1$, and $xy, L_0, L' \models\!\!\models R_1$ and $yz, L', L_1 \models\!\!\models R_2$.
- $w, L_0, L_1 \models\!\!\models (\ R_1\ \texttt{or}\ R_2\ )$ iff there exists L' such that both of the following hold:
  - either $w, L_0, L' \models\!\!\models R_1$ or $w, L_0, L' \models\!\!\models R_2$, and
  - $L_1 = L'|_D$, where $D = \mathit{flow}\ (\mathrm{dom}(L_0), (\ R_1\ \texttt{or}\ R_2\ ))$.
- $w, L_0, L_1 \models\!\!\models (\ R_1\ \texttt{intersect}\ R_2\ )$ iff there exist L', L'' such that $w, L_0, L' \models\!\!\models R_1$ and $w, L_0, L'' \models\!\!\models R_2$ and $L_1 = L'|_{D'} \cup L''|_{D''}$, where

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

  Remark: It can be proved that if $w, L_0, L' \models\!\!\models R_1$ and $w, L_0, L'' \models\!\!\models R_2$, then $L'|_{D'} \cup L''|_{D''}$ is a function.
- $w, L_0, L_1 \models\!\!\models \texttt{first\_match}\ (\ R\ )$ iff both
  - $w, L_0, L_1 \models\!\!\models R$ and
  - if there exist x, y, L' such that $w = xy$ and $\bar{x}, L_0, L' \models\!\!\models R$, then y is empty.
- $w, L_0, L_1 \models\!\!\models R\ \texttt{[*0]}$ iff $|w| = 0$ and $L_1 = L_0$.
- $w, L_0, L_1 \models\!\!\models R\ \texttt{[*1:\$]}$ 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 $1 \leq i \leq j$, $w_i, L_{(i-1)}, L_{(i)} \models\!\!\models R$.

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

An unclocked sequence R is *non-degenerate* iff there exist a non-empty finite word w over Σ and local variable contexts L_0, L_1 such that $w, L_0, L_1 \models\!\!\models R$. A clocked sequence S is *non-degenerate* iff the unclocked sequence S' that results from S by applying the rewrite rules is non-degenerate.

H.3.6 Satisfaction with local variables

H.3.6.1 Neutral satisfaction

w denotes a non-empty finite or infinite word over Σ. L_0, L_1 denote local variable contexts.

The rules defining neutral satisfaction of an assertion ~~satisfaction~~ are identical to those without local variables, but with the understanding that the underlying properties can have local variables.

Neutral satisfaction of properties:

- $w \models Q$ iff $w, \{\} \models Q$.
- $w, L_0 \models Q$ iff $w, L_0 \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
- $w, L_0 \models \texttt{disable iff}\ (b)\ P$ iff either $w, L_0 \models P$ or there exists $0 \leq k < |w|$ such that $w^k \models b[L_0]$ and $w^{0,k-1}\mathsf{T}^\omega, L_0 \models P$. Here, $w^{0,-1}$ denotes the empty word.

- $w, L_0 \models$ **not** P iff $\overline{w}, L_0 \not\models P$.
- $w, L_0 \models R$ iff there exist $0 \le j < |w|$ and L_1 such that $w^{0,j}, L_0, L_1 \models\!\!\models R$.
- $w, L_0 \models$ (R `|->` P) iff for every $0 \le j < |w|$ and L_1 such that $\overline{w}^{0,j}, L_0, L_1 \models\!\!\models R, w^{j..}, L_1 \models P$.
- $w, L_0 \models$ (P) iff $w, L_0 \models P$.
- $w, L_0 \models$ (P_1 **or** P_2) iff $w, L_0 \models P_1$ or $w, L_0 \models P_2$.
- $w, L_0 \models$ (P_1 **and** P_2) iff $w, L_0 \models P_1$ and $w, L_0 \models P_2$.

SVA 40
SVA 41
LRM 131
LRM 131

H.3.6.2 Weak and strong satisfaction by finite words

SVA 42

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

H.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 $i \le j$. The reason for this abuse is to make clear the way these definitions should be used in combination with those in preceding sections.

H.4.1 Extended booleans

LRM 131
SVA 43
JH26
SVA 44
SVA 153
SVA 45
LRM 128
LRM 128
SVA 46
SVA 47
SVA 48

w denotes a non-empty finite or infinite word over Σ, j denotes an integer such that $0 \le j < |w|$, and $T(V)$ denotes an instance of a clocked or unclocked sequence that is passed the local variables ~~$\forall$~~ V as actual arguments.

- $w^j, L_0, L_1 \models\!\!\models\!\!\models T(V)$`.ended` iff there exist $0 \le i \le j$ and L such that both $w^{i,j}, \{\}, L \models\!\!\models T(V)$ and $L_1 = L_0 |_{\text{~~D~~}D} \cup L|_{\text{~~T~~}V}$, where ~~D~~ D = dom(L_0) – (dom(L) ∩ ~~$\forall$~~ V).
- $w^j, L_0, L_1 \models$ `@(`c`)(`$T(V)$`.matched)` iff there exists $0 \le i < j$ such that $w^i, L_0, L_1 \models T(V)$`.ended` and $w^{i+1,j}, \{\}, \{\} \models\!\!\models$ `(!`c `[*0:$] ##1` c`)` .
- $w^j \models$ `@(`c`)$stable(`e`)` iff there exists $0 \le i < j$ such that $w^{i,j}, \{\}, \{\} \models\!\!\models$ `(`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 $0 \le i < j$ such that $w^{i,j}, \{\}, \{\} \models\!\!\models$ `(`c `##1` c `[*->1])` then $b[w^i] \ne 1$), where b is the least-significant bit of e.
- $w^j \models$ `@(`c`)$fell(`e`)` iff $b[w^j] = 0$ and (if there exists $0 \le i < j$ such that $w^{i,j}, \{\}, \{\} \models\!\!\models$ `(`c `##1` c `[*->1])` then $b[w^i] \ne 0$), where b is the least-significant bit of e.

H.4.2 Past

SVA 49
SVA 50

w denotes a non-empty finite or infinite word over Σ, and j denotes an integer such that $0 \le j < |w|$.

- Let $n \ge 1$. If there exist $0 \le i < j$ such that $w^{i,j}, \{\}, \{\} \models\!\!\models$ `(`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`.
- `$past(`e`)` ≡ `$past(`e`,1)` .

H.5 Recursive Properties

LRM 130

This section defines the neutral semantics of instances of recursive properties in terms of the neutral semantics of instances of non-recursive properties. The latter can be expanded to properties in the abstract syntax by

appropriate substitutions, and so their semantics is assumed to be understood.

According to Restriction 1 in Section 17.11.1, it is understood below that the negation operator ~~not~~ `not` cannot be applied to any property expression that instantiates a recursive property. Restriction 2 in Section 17.11.1 is not represented here because ~~disable iff~~ `disable iff` is treated as a general property-building operator in this appendix. A precise version of Restriction 3 is given below. JH27

Defined property p is said to ~~depend~~ *depend* on defined property q if there exist $n \geq 0$ and defined properties $p_0,\ldots,p_{\cancel{n}n}$ such that $p_0 = p$, $p_{\cancel{n}n} = q$, and for all $0 \leq i < n$, the definition of property $p_{\cancel{i}i}$ instantiates property $p_{\cancel{i}i+1}$. In particular, by taking $q = p$ and $n = 0$, it follows that property p depends on property p. JH28 JH29

A defined property p has an associated ~~dependency digraph~~ *dependency digraph*. The nodes of the digraph are all the defined properties on which p depends. If q and r are nodes of the digraph, then there is an arc from q to r for each instance of r in the definition of q. Such an arc is labelled by the minimum number of timesteps that are guaranteed from the beginning of the definition of q until the particular instance r. For example, if q is defined by: JH30

```
property ~~q~~q;
   (a |-> ~~r~~r)
   and
   ((b ##1 c[*0:3]) |=> ~~r~~r);
endproperty
```
JH31

where `a`, `b`, `c` are boolean expressions, then there is one arc from q to r labeled by "0" due to `a |-> r` and there is a second arc from q to r labeled by "2" due to `(b ##1 c[*0:3]) |=>` r.

A defined property ~~p~~p is called ~~recursive~~ *recursive* if its node appears on a cycle in the dependency digraph of p. JH32

The following is a precise version of Restriction 3:

RESTRICTION 3: The sum of the arc labels around any cycle of the dependency digraph of a recursive property must be positive.

Let $p(X)$ be an instance of a recursive defined property p, where X denotes the actual arguments of the instance. For $k \geq 0$, the k-fold approximation to $p(X)$, denoted $p[k](X)$, is an instance of a non-recursive property $p[k]$ defined inductively as follows.

— The definition of $p[0]$ is obtained from the definition of p by replacing the body *property_expr* with the literal `1'b1`.

— For $k > 0$, the definition of $p[k]$ is obtained from the definition of p by replacing each instance of a recursive property by its $(k – 1)$-fold approximation.

The semantics of the instance $p(X)$ is then defined as follows. For any word w over Σ and local variable context L, w,~~L~~ L $\models p(X)$ iff for all $k \geq 0$, $w,L \models p[k](X)$. JH33