Proposal for Formal Semantics for Local Variable Declaration Assignments

SV-AC

August 19, 2007

Note to the Editor: These changes should be consistent with those in 1549. In the sections that say "consistent with 1549", the additions from 1549 are shown in blue, while the additions from 1668 are shown in purple.

F.1, CHANGE (consistent with 1549)

c) The abstract syntax simplifies the assertion language by eliminating some features that tend to encumber the definition of the formal semantics.

1) The abstract syntax eliminates local variable declarations. The semantics of local variables is written with implicit types.

TO

c) The abstract syntax simplifies the assertion language by modifying or eliminating some features that tend to encumber the definition of the formal semantics.

1) The abstract syntax ~~eliminates~~ modifies local variable declarations so that they are integrated with sequence and property expressions. This change supports the rewriting algorithm (see below) that replaces each instance of a named sequence or property with a flattened sequence or property expression. The local variable declarations that appeared in the named sequence or property declaration become part of the flattened expression. The abstract syntax also allows local variable declaration assignments. Local variable declaration assignments are eliminated by a rewriting procedure after sequence and property instances have been flattened (see [Note to the editor: Add reference to the appropriate Subclause]). The semantics of local variables ~~is written with implicit~~ does not explicitly refer to their types.

F.1, CHANGE (consistent with 1549)

In order to use this annex 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 do not involve recursive properties, this transformation involves eliminating sequence and nonrecursive 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.

TO

In order to use this annex 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 do not involve recursive properties, this transformation involves eliminating sequence and nonrecursive property instances by ~~substitution, eliminating local variable declarations, introducing parentheses~~ using the rewriting algorithm (see [Note to the editor: add link to the new sub section, which is described in 1549]), eliminating local variable declaration assignments (see [Note to the editor: add link to the new subsection described below]), determining the enabling condition, determining implicit or inferred clocking event controls, and eliminating redundant clocking event controls.

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for unclocked sequences is

R ::= b	// "boolean expression" form
| `(1,` v `=` e `)`	// "local variable sampling" form

TO

The abstract grammar for unclocked sequences is

R ::= b	// "boolean expression" form
| `(` t v `[ =` e `];` R `)`	// "local variable declaration" form
| `(1,` v `=` e `)`	// "local variable sampling" form

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for clocked sequences is

S ::= `@(`b`)` R // “clock” form
| `(` S `)` // “parenthesized” form

TO

The abstract grammar for clocked sequences is

S ::= `@(`b`)` R // “clock” form
| `(` t v `[` `=` e `]` `;` S `)` // “local variable declaration” form
| `(` S `)` // “parenthesized” form

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for unclocked properties is

P ::= R // “sequence” form
| `(` P `)` // “parenthesis” form

TO

The abstract grammar for unclocked properties is

P ::= R // “sequence” form
| `(` t v `[` `=` e `]` `;` P `)` // “local variable declaration” form
| `(` P `)` // “parenthesis” form

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for clocked properties is

Q ::= `@(`b`)` P // “clock” form
| S // “sequence” form
| `(` Q `)` // “parenthesis” form

TO

The abstract grammar for clocked properties is

Q ::= `@(`b`)` P // “clock” form
| S // “sequence” form
| `(` t v `[` `=` e `]` `;` Q `)` // “local variable declaration” form
| `(` Q `)` // “parenthesis” form

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for unclocked top-level properties is

T ::= P	// "plain" form
| `disable iff` (b) P	// "disable" form

TO

The abstract grammar for unclocked top-level properties is

T ::= P	// "plain" form
| `disable iff` (b) P	// "disable" form
| (t v [= e]; T)	// "local variable declaration" form

F.2.1 CHANGE (consistent with 1549)

The abstract grammar for clocked top-level properties is

U ::= Q	// "plain" form
| `disable iff` (b) Q	// "disable" form

TO

The abstract grammar for unclocked top-level properties is

U ::= Q	// "plain" form
| `disable iff` (b) Q	// "disable" form
| (t v [= e]; U)	// "local variable declaration" form

F.2.3.5 ADD the following item to the end of the list (consistent with 1549):

- (t_1 v_1 [= e_1];... ; t_k v_k [= e_k]; X) ≡
  (t_1 v_1 [= e_1]; (t_2 v_2 [= e_2];... ; t_k v_k [= e_k]; X))
  for $k > 1$ and X any of P, Q, R, S, T, U.

Note to the editor: Add the following section after the section for the rewriting algorithm from 1549.

Rewriting local variable declaration assignments

After replacing instances of named sequences and properties as described in Subclause [Note to the editor: Add a reference to the subclause "Rewriting property and sequence instances" from Mantis 1549], local variable declaration assignments are eliminated from the resulting sequences and properties. Corresponding local variable assignments are added within the sequences and properties using the procedure described below. Only after this step is completed are the clock rewrite rules used.

At several points, the procedure for rewriting local variable declaration assignments queries whether a sequence admits an empty match. The queries allow splitting of cases in order to avoid changing the empty match behavior. Formally, a sequence admits an empty match if and only if it is tightly satisfied by the empty word. The tight satisfaction relation is defined in Subclauses F.3.2 and F.3.5, but those subclauses assume that the clock rewrite rules have already been applied to eliminate clocking operators. The current procedure requires that the clocking operators remain in the syntax. Therefore, an independent definition of admission of an empty match is given below by the function *admits_empty*, which maps sequences to $\{0, 1\}$. It can be proved that for a sequence r, $admits\_empty(r) = 1$ if and only if the empty word tightly satisfies r', where r' is the sequence that results from r by eliminating local variable declaration assignments and by applying the clock rewrite rules.

- $admits\_empty(b) = 0$.
- $admits\_empty((t\ v\ [= e\ ];\ r)) = admits\_empty(r)$.
- $admits\_empty((1,\ v = e)) = 0$.
- $admits\_empty((r)) = admits\_empty(r)$.
- $admits\_empty((r_1\ \#\#1\ r_2)) = admits\_empty(r_1)\ \&\&\ admits\_empty(r_2)$.
- $admits\_empty((r_1\ \#\#0\ r_2)) = 0$.
- $admits\_empty((r_1\ \texttt{or}\ r_2)) = admits\_empty(r_1)\ ||\ admits\_empty(r_2)$.
- $admits\_empty((r_1\ \texttt{intersect}\ r_2)) = admits\_empty(r_1)\ \&\&\ admits\_empty(r_2)$.
- $admits\_empty(\texttt{first\_match}(r)) = admits\_empty(r)$.
- $admits\_empty(r[*0]) = 1$.
- $admits\_empty(r[*1:\$]) = admits\_empty(r)$.
- $admits\_empty(@(b)r) = admits\_empty(r)$.

Let r denote a sequence, and let c denote the unique semantic leading clock of r (semantic leading clocks are defined in 16.15.1). If $c = \mathit{inherited}$, then let $\kappa(r)$ be the empty string. Otherwise, let $\kappa(r) = \texttt{@(}c\texttt{)}$.

The procedure first eliminates all local variable declaration assignments that are attached to sequences. In general, `(` $t\ v$ `=` e`;` r `)` is replaced by

$$\texttt{(}\ t\ v\texttt{;}\ \kappa(r)\ \texttt{(}\ \texttt{((1,}\ v\ \texttt{=}\ e\texttt{)}\ \texttt{\#\#0}\ \texttt{(}r\texttt{))}\ \texttt{or}\ \texttt{((}r\texttt{)}\ \texttt{intersect}\ \texttt{1[*0])}\ \texttt{)}\ \texttt{)}$$

If $\mathit{admits\_empty}(r) = \texttt{0}$, then the replacement may be simplified to

$$\texttt{(}\ t\ v\texttt{;}\ \kappa(r)\ \texttt{((1,}\ v\ \texttt{=}\ e\texttt{)}\ \texttt{\#\#0}\ \texttt{(}r\texttt{))}\ \texttt{)}$$

If $\mathit{admits\_empty}(r) = \texttt{1}$, then the replacement may be simplified to

$$\texttt{(}\ t\ v\texttt{;}\ \kappa(r)\ \texttt{(}\ \texttt{((1,}\ v\ \texttt{=}\ e\texttt{)}\ \texttt{\#\#0}\ \texttt{(}r\texttt{))}\ \texttt{or}\ \texttt{1[*0]}\ \texttt{)}\ \texttt{)}$$

After this step, local variable declaration assignments remain only attached to properties. In order to ensure that the declaration assignments are executed after advancing to the alignment points with the appropriate semantic leading clocks, the procedure next pushes these assignments down in the syntax using the function *push* defined below. *push* takes a list of local variable declaration assignments as its first argument and a property as its second argument. The property may be a top-level property. For clarity of notation, concatenations of lists are enclosed in angle brackets ($\langle, \rangle$), and the empty list is denoted by $\langle\rangle$.

The procedure finishes by applying the function *push* with $\langle\rangle$ as first argument to each top-level property and descending recursively.

Let E denote an ordered list of local variable assignments; let p, q denote properties, which may be top-level properties with `disable iff` clauses; let b denote a boolean expression; and let d denote a clocking event.

- $\mathit{push}(E, \texttt{(}\ t\ v\texttt{;}\ p\ \texttt{)}) \ = \ \texttt{(}\ t\ v\texttt{;}\ \mathit{push}(E, p)\ \texttt{)}$.
- $\mathit{push}(E, \texttt{(}\ t\ v\ \texttt{=}\ e\texttt{;}\ p\ \texttt{)}) \ = \ \texttt{(}\ t\ v\texttt{;}\ \mathit{push}(\langle E,\ v\ \texttt{=}\ e\rangle, p)\ \texttt{)}$.
- $\mathit{push}(\langle\rangle, r) = r$. If E is non-empty, then

  $$\mathit{push}(E, r) \ = \ \kappa(r)\ \texttt{(1,}\ E\texttt{)}\ \texttt{\#\#0}\ \texttt{(}r\texttt{)}$$

  In this case, r is a sequence used as a property. According to 16.12.6, $\mathit{admits\_empty}(r) = \texttt{0}$.
- $\mathit{push}(\langle\rangle, r\ \texttt{|->}\ p) = r\ \texttt{|->}\ \mathit{push}(\langle\rangle, p)$. If E is non-empty, then

  $$\mathit{push}(E, r\ \texttt{|->}\ p) \ = \ \kappa(r)\ \texttt{(1,}\ E\texttt{)}\ \texttt{\#\#0}\ \texttt{(}r\texttt{)}\ \texttt{|->}\ \mathit{push}(\langle\rangle, p)$$

- $push(\langle\rangle, r \texttt{ |=> } p) \;=\; r \texttt{ |=> } push(\langle\rangle, p)$. If E is non-empty and $admits\_empty(r) = \texttt{0}$, then

  $$push(E, r \texttt{ |=> } p) \;=\; \kappa(r) \texttt{ (1, } E\texttt{) ##0 (} r\texttt{) |=> } push(\langle\rangle, p)$$

  If E is non-empty and $admits\_empty(r) = \texttt{1}$, then

  $push(E, r \texttt{ |=> } p) \;=\; \texttt{(}\kappa(r) \texttt{ (1, } E\texttt{) ##0 (} r\texttt{) |=> } push(\langle\rangle, p)\texttt{) and (}push(E, p)\texttt{)}$

- $push(\langle\rangle, \texttt{if (} b \texttt{) } p \texttt{ [else } q \texttt{]}) \;=\; \texttt{if (} b \texttt{) } push(\langle\rangle, p) \texttt{ [else } push(\langle\rangle, q) \texttt{]}$.
  If E is non-empty, then

  $push(E, \texttt{if (} b \texttt{) } p \texttt{ [else } q \texttt{]}) \;=\; \texttt{(1, } E\texttt{) |-> if (} b \texttt{) } push(\langle\rangle, p) \texttt{ [else } push(\langle\rangle, q) \texttt{]}$

- $push(E, \texttt{disable iff (} b \texttt{) } p) \;=\; \texttt{disable iff (} b \texttt{) } push(E, p)$.
- $push(E, \texttt{@(} d \texttt{) } p) \;=\; \texttt{@(} d \texttt{) } push(E, p)$.
- $push(E, \texttt{(} p \texttt{)}) = \texttt{ (} push(E, p) \texttt{) }$.
- $push(E, \texttt{not } p) = \texttt{not } push(E, p)$.
- $push(E, p \texttt{ or } q) = push(E,\ p) \texttt{ or } push(E, q)$.
- $push(E, p \texttt{ and } q) = push(E,\ p) \texttt{ and } push(E, q)$.