Proposal for Formal Semantics for Local Variable Declaration Assignments

SV-AC

July 12, 2007

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.

2) The abstract syntax eliminates instantiation of sequences and properties. The semantics of an assertion with an instance of a sequence or nonrecursive property is the same as the semantics of a related assertion obtained by replacing the sequence or nonrecursive 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 subclause defines the semantics of instances of recursive properties in terms of the semantics of instances of nonrecursive properties.

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 that replaces instances of named sequences and properties with flattened sequence and property expressions. The local varible 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.

2) The abstract syntax eliminates instantiation of sequences and properties. The semantics of an assertion with ~~an~~ instances of a sequence or nonrecursive property is the same as the semantics of a related assertion obtained by replacing the sequence or nonrecursive 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 subclause defines the semantics of instances of recursive properties in terms of the semantics of instances of nonrecursive properties.~~ The definition of rewriting an assertion without property and sequence instances is given in clause [Note to the editor: add link to the new subsection, which is described in 1549]. The semantics of an assertion that has one or more recursive property instances is defined in clause **F.5** in terms of an infinite series of assertions with no instances of recursive properties, but possibly with instances of nonrecursive properties and sequences. The semantics of each assertion in the series is defined as the semantics for other assertions with no instances of recursive properties.

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 using the rewriting algorithm [Note to the editor: add link to the new subsection, which is described in 1549] ~~by substitution, eliminating local variable declarations, introducing parentheses~~, eliminating local variable declaration assignments [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

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

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

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

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

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

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:

- `(` 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.

F.3.1 CHANGE

- `@(`c`)` b ⟼ `(` `!`c `[*0:$] ##1` c `&` b `)` .
- `@(`c`)` `(` `1,` v `=` e `)` ⟼ `(` `@(`c`)` `1 ##0 (` `1,` v `=` e `)` `)` .

TO

- `@(`c`)` b ⟼ `(` `!`c `[*0:$] ##1` c `&` b `)` .
- `@(`c`)` `(` t v `[ = ` e `];` X `)` ⟼ `(` t v `[ = ` e `];` `@(`c`)` X `)` , where X is either R or P.
- `@(`c`)` `(` `1,` v `=` e `)` ⟼ `(` `@(`c`)` `1 ##0 (` `1,` v `=` e `)` `)` .

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

Rewriting local variable declaration assignments

After flattening instances of named sequences and properties as described above, local variable declaration assignments are eliminated from the resulting sequence and property expressions and corresponding local variable assignments are added within the expressions using the procedure described below. Only after this step is completed are the clock rewrite rules used.

Let r denote a sequence expression and let c denote the unique semantic leading clock of r. Let $\kappa(r)$ be the empty string if $c = \mathit{inherited}$, and otherwise let $\kappa(r) = \texttt{@(}c\texttt{)}$.

The procedure first eliminates all local variable declaration assignments that are attached to sequence expressions by replacing each occurrence of

$$\texttt{(}\ t\ v\ \texttt{=}\ e\texttt{;}\ r\ \texttt{)}$$

with

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

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 semanatic 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 expression or top-level property as its second argument. 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 property expressions or top-level properties, possibly 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

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

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

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

- $push(\langle\rangle, r \texttt{ |=> } p) \;=\; r \texttt{ |=> } push(\langle\rangle, p)$. If E is non-empty and the empty word tightly satisfies r, then

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

  If E is non-empty and the empty word does not tightly satisfy r, then

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

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

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

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