Proposal for formal semantics for property and sequence instances

SV-AC

August 6, 2007

F.1 REPLACE

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.
- **3)** The abstract syntax does not allow implicit clocks. Clocking event controls must be applied explicitly in the abstract syntax.
- **4)** The abstract syntax does not allow explicit procedural enabling conditions for assertions. Procedural enabling conditions are utilized in the semantics definition (see F.3.3.1), but the method for extracting such conditions is not defined in this annex.

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. 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. 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 nonrecursive properties in F.5. Once the semantics of the recursive property instances is understood, the placeholder functions are treated as properties with this semantics. Then the ordinary definitions can be applied to the transformed assertion in the abstract syntax together with placeholder functions.

WITH

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 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 instance of a named 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 expression. ~~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.~~ Subclause [Note to the editor: add link to the new sub section, which is described in the next item] defines a rewriting algorithm that replaces each instance of a named sequence or nonrecursive property with a flattened sequence or property expression. The semantics of an assertion that has one or more

instances of recursive properties is defined in subclause F.5. The definition is in terms of an infinite set of associated assertions, each of which may have instances of sequences and nonrecursive properties, but has no instances of recursive properties. The semantics of each associated assertion is obtained, as before, by using the rewriting algorithm.

3) The abstract syntax does not allow implicit clocks. Clocking event controls must be applied explicitly in the abstract syntax.

4) The abstract syntax does not allow explicit procedural enabling conditions for assertions. Procedural enabling conditions are utilized in the semantics definition (see F.3.3.1), but the method for extracting such conditions is not defined in this annex.

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 [Note to the editor: add link to the new sub section, which is described in the next item], determining the enabling condition, determining implicit or inferred clocking event controls, and eliminating redundant clocking 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));~~

property P(logic[3:0] a, property q);
   (a[1:0] == 2'b10) ##1 (a[3:2] == 2'b01) |=> q;
endproperty
property Q(r, logic[1:2] d);
   logic[1:2] v;
   (1, v = d) ##1 r |=> d == v;
endproperty
always @(c) if (b) assert property ( P(A, Q(R, D)) );
```

is transformed into the enabling condition "*b*" together with the assertion

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

always @(c) assert property (
   ($var((logic[3:0])'(A))[1:0] == 2'b10) ##1
   ($var((logic[3:0])'(A))[3:2] == 2'b01) |=>
   (
      logic[1:2] v;
      (1, v = $var((logic[1:2])'(D))) ##1 (R) |=>
```

```
        $var((logic[1:2])'(D)) == v
    )
);
```

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 nonrecursive properties in F.5. Once the semantics of the recursive property instances is understood, the placeholder functions are treated as properties with this semantics. Then the ordinary definitions can be applied to the transformed assertion in the abstract syntax together with placeholder functions.~~

F.2.1 CHANGE

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

TO

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

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; 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`;` 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`;` 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`;` 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`;` 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`;` U `)` // "local variable declaration" form

F.2.3.5 ADD the following item to the end of the list:

- `(` t_1 v_1`;...;` t_k v_k`;` X `)` $\equiv$ `(` t_1 v_1`;` `(` t_2 v_2`;...;` t_k v_k`;` X `)` `)` for $k > 1$ and X any of P, Q, R, S, T, U.

F.3.1 CHANGE

- `@(`c`)` $b \longmapsto$ `(` `!`c `[*0:$]` `##1` c `&` b `)` .

- `@(`c`) ( 1, `v` = `e` )` $\longmapsto$ `( @(`c`) 1 ##0 ( 1, `v` = `e` ) )` .

TO

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

F.3.4 CHANGE

- $sample(b) = \{\}$.
- $sample($`( 1, `v` = `e` )`$) = \{v\}$.

TO

- $sample(b) = \{\}$.
- $sample($`( `t v`; `R `)`$) = sample(R) - \{v\}$.
- $sample($`( 1, `v` = `e` )`$) = \{v\}$.

F.3.4 CHANGE

- $block(b) = \{\}$.
- $block($`( 1, `v` = `e` )`$) = \{\}$.

TO

- $block(b) = \{\}$.
- $block($`( `t v`; `R `)`$) = block(R) - \{v\}$.
- $block($`( 1, `v` = `e` )`$) = \{\}$.

F.3.4 CHANGE

- $\mathit{flow}(X,\, b) = X$.
- $\mathit{flow}(X,\, \texttt{(}\ \texttt{1,}\ v\ \texttt{=}\ e\ \texttt{)}) = X \cup \{v\}$.

TO

- $\mathit{flow}(X,\, b) = X$.
- $\mathit{flow}(X,\, \texttt{(}\ t\ v\texttt{;}\ R\ \texttt{)}) = (X \cap \{v\}) \cup (\mathit{flow}(X - \{v\}, R) - \{v\})$.
- $\mathit{flow}(X,\, \texttt{(}\ \texttt{1,}\ v\ \texttt{=}\ e\ \texttt{)}) = X \cup \{v\}$.

F.3.5 CHANGE

A local variable context is a function that assigns values to local variable names. If L is a local variable context, then $\mathrm{dom}(L)$ denotes the set of local variable names that are in the domain of L. If $D \subseteq \mathrm{dom}(L)$, then $L|_D$ means the local variable context obtained from L by restricting its domain to D.

TO

A local variable context is a function that assigns values to local variable names. If L is a local variable context, then $\mathrm{dom}(L)$ denotes the set of local variable names that are in the domain of L. If $D \subseteq \mathrm{dom}(L)$, then $L|_D$ means the local variable context obtained from L by restricting its domain to D. If v is a local variable name, then $L\backslash v$ denotes $L|_{\mathrm{dom}(L)-\{v\}}$ and $L[v]$ denotes $L|_{\{v\}}$.

F.3.5 CHANGE

It can be proved that the definition guarantees that $w, L_0, L_1 \models R$ implies $\mathrm{dom}(L_1) = \mathit{flow}(\mathrm{dom}(L_0), R)$.

- $w, L_0, L_1 \models \texttt{(}\ \texttt{1,}\ v\ \texttt{=}\ e\ \texttt{)}$ iff $|w| = 1$ and $w^0 \models \texttt{1}$ 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 $D = \mathrm{dom}(L_0) - \{v\}$. In case $w^0 \in \{\top, \bot\}$, $e[L_0, \top]$ and $e[L_0, \bot]$ can be any constant values of the type of e.

TO

It can be proved that the definition guarantees that $w, L_0, L_1 \models\!\models R$ implies $\mathrm{dom}(L_1) = \mathit{flow}(\mathrm{dom}(L_0), R)$.

- $w, L_0, L_1 \models\!\models$ `(` t v`;` R `)` iff there exists L such that $w, L_0 \backslash v, L \models\!\models R$ and $L_1 = L_0[v] \cup (L \backslash v)$.
- $w, L_0, L_1 \models\!\models$ `(` `1,` v `=` e `)` iff $|w| = 1$ and $w^0 \models$ `1` and ~~$L_1 = \{(v, e[L_0, w^0])\} \cup L_0 | D$~~ $L_1 = \{(v, e[L_0, w^0])\} \cup (L_0 \backslash v)$, 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 $D = \mathrm{dom}(L_0) \quad \{v\}$~~. In case $w^0 \in \{\top, \bot\}$, $e[L_0, \top]$ and $e[L_0, \bot]$ can be any constant values of the type of e.

F.3.6.1 CHANGE

Neutral satisfaction of top-level properties is defined as follows:

- For $T = P$, $w, L_0 \models T$ iff $w, L_0 \models P$.
- For $T =$ `disable iff (` b `)` P, $w, L_0 \models T$ iff either
  - $w, L_0 \models P$ and no letter of w satisfies b, or
  - Some letter of w satisfies b and $w^{0,i-1}\bot^\omega, L_0 \models P$ for i the least index such that $w^i \models b$, $0 \le i < |w|$.

TO

Neutral satisfaction of top-level properties is defined as follows:

- For $T = P$, $w, L_0 \models T$ iff $w, L_0 \models P$.
- For $T =$ `disable iff (` b `)` P, $w, L_0 \models T$ iff either
  - $w, L_0 \models P$ and no letter of w satisfies b, or
  - Some letter of w satisfies b and $w^{0,i-1}\bot^\omega, L_0 \models P$ for i the least index such that $w^i \models b$, $0 \le i < |w|$.
- For $T =$ `(` t v`;` T' `)`, $w, L_0 \models T$ iff $w, L_0 \backslash v \models T'$.

F.3.6.1 CHANGE

Disabling of top-level properties is defined as follows:

- $w, L_0 \not\models^{\mathbf{d}} P$.

- $w, L_0 \models^{\mathbf{d}}$ `disable iff ( b )` P iff some letter of w satisfies b and both $w^{0,i-1}\top^\omega, L_0 \models P$ and $w^{0,i-1}\bot^\omega, L_0 \not\models P$ for i the least index such that $w^i \models b$, $0 \leq i < |w|$.

TO

Disabling of top-level properties is defined as follows:

- $w, L_0 \not\models^{\mathbf{d}} P$.
- $w, L_0 \models^{\mathbf{d}}$ `disable iff ( b )` P iff some letter of w satisfies b and both $w^{0,i-1}\top^\omega, L_0 \models P$ and $w^{0,i-1}\bot^\omega, L_0 \not\models P$ for i the least index such that $w^i \models b$, $0 \leq i < |w|$.
- $w, L_0 \models^{\mathbf{d}}$ `( t v; T )` iff $w, L_0 \backslash v \models^{\mathbf{d}} T$.

F.3.6.1 CHANGE

Neutral satisfaction of properties is defined as follows:

- $w \models Q$ 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.
- $w, L_0 \models$ `not` P iff $\bar{w}, L_0 \not\models P$.

TO

Neutral satisfaction of properties is defined as follows:

- $w \models Q$ 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.
- $w, L_0 \models$ `( t v; P )` iff $w, L_0 \backslash v \models^{\mathbf{d}} P$.
- $w, L_0 \models$ `not` P iff $\bar{w}, L_0 \not\models P$.

F.3.6.3 CHANGE

The definition is identical to that without local variables (see F.3.3.3), but with the understanding that the underlying properties can have local variables.

TO

The definition is identical to that without local variables (see F.3.3.3), but with the understanding that the underlying properties can have local variables and that $w, L_0 \models^{\mathbf{non}} (\ t\ v;\ P\ )$ iff $w, L_0 \backslash v \models^{\mathbf{non}} P$.

Note to the editor: Add the following section after F.3.1

Rewriting property and sequence instances

This section describes an algorithm for rewriting a property that contains one or more instances of named sequences or nonrecursive properties. The result of the algorithm is one flattened property without instances. The semantics of a hierarchical property is defined to be the semantics of the flattened property resulting from the rewriting algorithm. In particular, if the result is illegal, then so is the source. The property rewritten in the algorithm may be the top-level property of a concurrent assertion.

For the rewriting algorithm, an auxiliary function `$var` is defined as follows. The function `$var` may be applied to any SystemVerilog expression that may appear as an actual argument expression in an instance of a named sequence or property. If e is such an expression, then `$var(`e`)` behaves like e in all respects except that operations allowed on a reference to or instance of a named item declared with the same type as e are also allowed on `$var(`e`)`. In particular, any operation that is allowed on a reference to a variable declared with the same type as e is allowed on `$var(`e`)`. Also any operation that is allowed on an instance of a named sequence (resp., property) is allowed on `$var` applied to a sequence (resp., property, including a top-level property).

The function `$var` is not a SystemVerilog function, and it is introduced only in the rewriting algorithm. The rewriting algorithm uses `$var` because operations that are legal on a reference to a formal argument within the body of a declaration might no longer be legal when an actual argument expression is substituted for the reference to the formal argument. For example, let `a` and `b` be variables of type `logic[0:1]`, let `v` be a variable of type `logic[0:3]`, and let e be the cast expression `(logic[0:3])'({a,b})`. If `v` is a formal argument, then the part select expression `v[1:2]` is legal within the body of the declared item. However, if e is an actual argument expression passed to `v` in an instance, then the part select operation cannot be applied when e is substituted for `v` because `((logic[0:3])'({a,b}))[1:2]` is illegal. Using the `$var` function, the form `$var((logic[0:3])'({a,b}))[1:2]` is legal. For expressions with undefined type, `$var` does not enable additional operations.

The rewriting algorithm

Given ϖ a property, possibly a top-level property:

While there are property instances in ϖ do:
begin
  Select an arbitrary property instance `P` and replace it by flatten_property(`P`).
end

While there are sequence instances in ϖ do:
begin
  Select an arbitrary sequence instance `R` and replace it by flatten_sequence(`R`).
end

flatten_property(`P`)
begin

1. Create a copy `P'` of the declaration of `P`.
2. In `P'`, replace every reference to typed formal argument `f` that is not on the left hand side of an assignment by `$var(f)`.
3. In `P'`, replace every reference to typed formal argument `f` of type `t`, including those within `$var(f)` introduced in the preceding step, by `t'(f)`, the cast of `f` to type `t`. Casting is defined in Subclause 6.24.
4. In `P'`, replace every reference to formal argument `f`, including those within casting expressions `t'(f)` introduced in the preceding step, by `(a)`, where `a` is the actual argument expression corresponding to `f` in the instance `P`. The parentheses around `a` may be omitted if the reference to `f` is itself already enclosed in parentheses.
5. Return the expression obtained by copying the local variable declarations and body property_spec from `P'` and enclosing the result in parentheses.

end

flatten_sequence(`R`)
begin

1. Create a copy `R'` of the declaration of `R`.
2. In `R'`, replace every reference to typed formal argument `f` that is not on the left hand side of an assignment by `$var(f)`.
3. In `R'`, replace every reference to typed formal argument `f` of type `t`, including those within `$var(f)` introduced in the preceding step, by `t'(f)`, the cast of `f` to type `t`. Casting is defined in Subclause 6.24.
4. In `R'`, replace every reference to formal argument `f`, including those within casting expressions `t'(f)` introduced in the preceding step, by `(a)`, where `a` is the actual argument expression corresponding to `f` in the instance `R`. The parentheses around `a` may be omitted if the reference to `f` is itself already enclosed in parentheses.

5. Return the expression obtained by copying the local variable declarations and body sequence_expr from `R'` and enclosing the result in parentheses.

end

F.5 CHANGE

This subclause defines the neutral semantics of instances of recursive properties in terms of the neutral semantics of instances of nonrecursive properties. The latter can be expanded to properties in the abstract syntax by appropriate substitutions; therefore, their semantics is assumed to be understood.

TO

This subclause defines the neutral semantics of properties, including top-level properties, with instances of recursive properties in terms of the neutral semantics of properties with instances of nonrecursive properties. The latter can be expanded to properties in the abstract syntax by ~~appropriate substitutions~~ applying the rewriting algorithm [Note to the editor: add link to the new subsection, which is described in the previous item]; therefore, their semantics is assumed to be understood.

F.5 CHANGE

Let $p(X)$ be an instance of a recursive named 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 declaration of $p[0]$ is obtained from the declaration of p by replacing the body `property_expr` with the literal `1'b1`.
- For $k > 0$, the declaration of $p[k]$ is obtained from the declaration 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, $w, L \models p(X)$ iff for all $k \geq 0$, $w, L \models p[k](X)$.

TO

~~Let $p(X)$ be an instance of a recursive named property p, where X denotes the actual arguments of the instance.~~ Let p be a named property. For $k \geq 0$, the k-fold approximation to ~~$p(X)$~~ p, denoted ~~$p[k](X)$~~ $p[k]$, is ~~an instance of~~ a ~~non recursive~~ named property ~~$p[k]$~~ without instances of recursive properties defined inductively as follows:

- The declaration of $p[0]$ is obtained from the declaration of p by replacing the body ~~`property_expr`~~ `property_spec` with the literal `1'b1`.
- For $k > 0$, the declaration of $p[k]$ is obtained from the declaration of p by replacing each instance of a recursive property by the corresponding instance of its $(k-1)$-fold approximation and by replacing each instance of a nonrecursive property by the corresponding instance of its k-fold approximation.

Let ϖ be a property, possibly the top-level property of a concurrent assertion. The k-fold approximation to ϖ, denoted $\varpi[k]$, is obtained from ϖ by replacing each instance of a named property by the corresponding instance of its k-fold approximation. The semantics of ~~the instance $p(X)$~~ ϖ is then defined as follows: for any word w over Σ and local variable context L, ~~$w, L \models p(X)$~~ $w, L \models \varpi$ iff for all ~~$k \geq 0$, $w, L \models p[k](X)$~~ $k > 0$, $w, L \models \varpi[k]$. Since $\varpi[k]$ does not have instances of recursive properties, its semantics is obtained using the rewriting algorithm[Note to the editor: add link to the new sub section, which is described in the previous item].