Proposal for formal semantics for property and sequence instances

SV-AC

July 10, 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 4.3.1), 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 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 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 sub section, which is described in the next item]. The semantics of an assertion that have recursive properties 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.

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 4.3.1), 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 using the rewriting algorithm [Note to the editor: add link to the new sub section, which is described in the next item] ~~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

```
property p1( logic [0:2] a1, property a2);  (a1 == 3'b010) |=> a2; endproperty
property p2( a1, logic [1:3] a2); a1 ##1 (a2 == 3'b010); endproperty
always  @(c) if (b) assert property ( p1(g1, p2(g2, g3)));
```

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

```
always @(c) assert property ( ($var((logic [2:0])'g1) == 3'b010) |=>
                                $var( g1) ##1 ($var((logic [1:3])'g3) == 3'b010));
```

~~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.~~

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

F.3.4 CHANGE

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

TO

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

F.3.4 CHANGE

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

TO

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

F.3.5 CHANGE

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

- $w, L_0, L_1 \models\!\models \texttt{(}\ 1\texttt{,}\ v\ \texttt{=}\ e\ \texttt{)}$ 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 $\mathrm{dom}(L_1) = \mathit{flow}(\mathrm{dom}(L_0), R)$.

- $w, L_0, L_1 \models\!\models \texttt{(}\ t\ v\texttt{;}\ R\ \texttt{)}$ iff $w, L_0|_D, L_1 \models\!\models R$, where $D = \mathrm{dom}(L_0) - \{v\}$.
- $w, L_0, L_1 \models\!\models \texttt{(}\ 1\texttt{,}\ v\ \texttt{=}\ e\ \texttt{)}$ 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.

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 $w^i \models b$, $0 \leq i < |w|$.

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 $w^i \models b$, $0 \leq i < |w|$.
- For $T =$ `( t v ; T' )`, $w, L_0 \models T$ iff $w, L_0|_D \models T'$, where $D = \mathrm{dom}(L_0) - \{v\}$.

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 $w, L_0|_D \models^{\mathbf{d}} T$, where $D = \mathrm{dom}(L_0) - \{v\}$.

F.3.6.1 CHANGE

Neutral satsifaction 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.
- $w, L_0 \models \texttt{not}\ P$ iff $\bar{w}, L_0 \not\models P$.

TO

Neutral satsifaction 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 $w, L_0|_D \models^{\mathbf{d}} P$, where $D = \mathrm{dom}(L_0) - \{v\}$.
- $w, L_0 \models \texttt{not}\ P$ 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 $w, L_0|_D \models^{\mathbf{non}} P$, where $D = \mathrm{dom}(L_0) - \{v\}$.

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

.0.1 Rewriting property and sequence instances

This section describes an algorithm for rewriting an assertion that contains property and/or sequence instances. The result of the algorithm is an assertion with one "flat" property expression. The semantics of a hierarchical property expression is then defined to be the semantics of the flat property_spec resulting from the rewriting algorithm. In particular, if the result is illegal, then so is the source.

For the rewriting algorithm, an auxiliary function *$var()* is defined as follows. The function `$var` may be applied on every system verilog expression. For every system verilog expression e, $var(`e`) is the expression `e` with the exception that every operation that is allowed on a variable with the same type of e is allowed on `$var(`e`)`.

In particular, any operation that is allowed on a declared property or declared sequence, is allowed on `$var` applied on a property expression or a sequence expression respectively. Note that for some expressions e, there are operations that are allowed on variables v with the same type but not on e. For example, let `a, b` be variables of type `logic [0:1]`, let `v` be a variable of type `logic [0:3]`, and e the expression `(logic [0:3])'{a,b}`. Then, the result of applying a bit select operation on `v v[1:2]` is a valid expression, but `(logic [3:0])'{a,b}[0:1]` is not. Introducing the `$var` function enables usage of the form `$var((logic [3:0])'{a,b})[1:2]`. For expressions with undefined type, `$var` does not enable additional operations. `$var` is not a system verilog function and its only use is in the rewriting algorithm.

The rewrite algorithm
Given a property_spec expression inside an assertion declaration:

While there are property instances in the property_spec do:
begin
    Select an arbitrary property instance `P`
    flatten_property(`P`)
end

While there are sequence instances in the property_spec do:
begin
    Select an arbitrary sequence instance `R`
    flatten_sequence(`R`)
end

flatten_property(`P`)
begin

1. Create a copy `P'` of the declaration of `P`.
2. For every local variable declared in `P'`, make its identifier unique.
3. Replace every typed formal argument `f` in the property_expr of `P'` that is not on the left hand side of an assignment, with `$var(f)`.
4. For every occurrence of a typed formal argument `f` of `P'` with type `t` in the body of `P'`, replace `f` with `(t)'f` (cast `f` to type `t`). See cast definition at (4.14).
5. Replace every formal argument `f` in the property_expr of `P'`, with `(a)` where `a` is the corresponding actual argument expressions in the instance of `P`.

6. In the property_expr of the assertion, replace the instance of P with the local variable declarations followed by the property_expr of P', all wrapped in parenthesis.

end

flatten_sequence(R)
begin

1. Create a copy R' of the declaration of R.
2. For every local variable declared in R', make its identifier unique.
3. Replace every typed formal argument f in the sequence_expr of R' that is not on the left hand side of an assignment, with $var(f).
4. For every occurrence of a typed formal argument f of R' with type t in the sequence_expr of R', replace f with (t)'f (cast f to type t). See cast definition at (4.14).
5. Replace every formal argument f in the sequence_expr of R', with (a) where a is the corresponding actual argument expressions in the instance of R'.
6. In the property_expr of the assertion, replace the instance of R with local variable declarations followed by the sequence_expr of R', all wrapped in parenthesis.

end

Clause F.5:

replace

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.

with

This subclause defines the neutral semantics of assertions with instances of recursive properties in terms of the neutral semantics of assertions 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 sub section, which is described in the previous item]; therefore, their semantics is assumed to be understood.

replace

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 $\models p[k](X)$.

with

Let A be an assertion with instances of a recursive properties. Let $p(X)$ be ~~an instance~~a declaration of a ~~recursive~~ property p, where X denotes the ~~actual~~formal arguments of ~~the instance~~ p. For $k \geq 0$, the k-fold approximation to A and $p(X)$, denoted $A[k]$ and $p[k](X)$ respectively, is ~~an instance of~~ a non-recursive property declaration $p[k](A[k])$ defined inductively as follows:

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

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