Clause: 17.11, 17.13.1, 17.13.3, 29.4.3, 28.3.2, Annex I

Description

Currently, LRM mentions that if disable iff condition becomes true, the evaluation of property spec is true.

This makes it difficult for the users to identify the difference in match due to disable iff condition vs. match due to complete property evaluation. Therefore we propose to introduce another evaluation condition called “disabled”. This type of evaluation only occurs when disable iff condition of the property becomes true.

Suggested Resolution

Clause 17.11, replace the following on page 267:

~~For an evaluation of the property_spec, there is an evaluation of the underlying property_expr. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the property_spec is true.~~

By

For an evaluation of the property_spec, there is an evaluation of the underlying property_expr. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the property results in disabled. A property has disabled evaluation if it was preempted due to disable iff condition. A disabled evaluation of a property does not result in success or failure.

Clause 17.13.1, add/change the following on page 285:

The **assert** statement is used to enforce a **property** as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. ~~Otherwise~~ When the property for the **`assert`** statement is evaluated to be false, the fail statements of the *action_block* are executed. When the property for the **`assert`** statement is evaluated to be disabled, no `action_block` statement is executed.

Clause 17.13.3, add/change the following on page 288:

The results of coverage statement for a property shall contain the following:
— Number of times attempted
— Number of times succeeded
— Number of times failed
— Number of times succeeded because of vacuity

In addition, `statement_or_null` is executed every time a property ~~succeeds~~ is being evaluated to true.

The coverage counters for success, failure or vacuous success do not include disabled evaluations. The attempt counter includes the attempts which result in disabled evaluation.

Clause 29.4.3, add the following on page 481:

— Assertion-specific coverage status properties
`vpiAssertAttemptCovered`

```
vpiAssertSuccessCovered
vpiAssertFailureCovered
vpiAssertVacuousSuccessCovered
vpiAssertDisableCovered
```

Clause 29.4.3, add the following on page 482:

`vpi_get(vpiAssertVacuousSuccessCovered, assertion_handle)`
returns the number of times the assertion has succeeded vacuously. Refer to 17.11.2 and 17.13 for the definition
of vacuity.

`vpi_get(vpiAssertDisableCovered, assertion_handle)`
returns the number of times the assertion has reached the disabled state (e.g. as a result of disable iff condition becoming true or if an attempt starts when the disable iff is true). Refer to 17.11 for the definition of disabled evaluation.

For any assertion, the number of attempts that have not
yet reached any conclusion (success or failure) can be derived from the formula:
`in progress = attempts - (successes + vacuous success + disabled + failures)`
The example below illustrates some of these queries:

Annex I, add the following:

```
#define vpiAssertAttemptCovered 770
#define vpiAssertSuccessCovered 771
#define vpiAssertFailureCovered 772
#define vpiAssertVacuousSuccessCovered 773
#define vpiAssertDisableCovered 774
```

Clause 28.3.2, add the following:

cbAssertionSuccess
An assertion attempt reaches a success state.
cbAssertionVacuousSuccess
An assertion attempt reaches a vacuous success state.
cbAssertionDisabledEvaluation
An assertion attempt reaches the disabled state (e.g. as a result of disable iff condition becoming true or if an attempt starts when the disable iff is true).

Details:
a) In a failing transition, there shall always be at least one element in the expression array.
b) Placing a step callback results in the same callback function being invoked for both success and failure steps.
c) The content of the `cb_time` field depends on the reason identified by the reason field, as follows:
— `cbAssertionStart: cb_time` is the time when the assertion attempt has been started.
— `cbAssertionSuccess` and `cbAssertionFailure: cb_time` is the time when the assertion succeeded nonvacuously or failed.
— `cbAssertionVacuousSuccess: cb_time` is the time when the assertion succeeded vacuously.
— `cbAssertionDisabledEvaluation: cb_time` is the time when the assertion reached the disabled state.

Annex E.1

Remove:

d) 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 is 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.

Annex E.2.1

Replace

The abstract grammar for unclocked properties is

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

Each instance of *R* in this production must be a nondegenerate unclocked sequence. In the “sequence” form,
R must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

The abstract grammar for clocked properties is

```
Q ::= @( b ) P // "clock" form
| 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 nondegenerate clocked sequence. In the “sequence” form, *S* must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

with

The abstract grammar for unclocked properties is

```
P ::= R // "sequence" form
| ( P ) // "parenthesis" form
| not P // "negation" form
```

```
| ( P or P ) // "or" form
| ( P and P ) // "and" form
| ( R |-> P ) // "implication" form
| disable iff ( b ) P // "reset" form
```

Each instance of *R* in this production must be a nondegenerate unclocked sequence. In the “sequence” form,
R must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

The abstract grammar for clocked properties is

```
Q ::= @( b ) P // "clock" form
| 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 nondegenerate clocked sequence. In the “sequence” form, *S* must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

The abstract grammar for unclocked reset properties is:

```
T ::= P // no reset form
| disable iff ( b ) P // "reset" form
```

The abstract grammar for clocked reset properties is:

```
U ::= Q // no reset form
| disable iff ( b ) Q // "reset" form
```

Annex E.3.3

Replace

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 $0 < i < |w|$ so that $\hat{\mathbf{w}}^{i} \models c$ and $\hat{\mathbf{w}}^{i} \models b$, $w^{i\cdot\cdot} \models$ `@(c)` *P*.
— $w, b \models$ **always assert property** *Q* iff for every $0 < i < |w|$, if $\hat{\mathbf{w}}^{i} \models b$ then $w^{i\cdot\cdot} \models Q$.
— $w, b \models$ **initial** `@(c)` **assert property** *P* iff for every $0 < i < |w|$ so that $\hat{\mathbf{w}}^{0,i} \models$!*c* [*0:$] ##1 *c* and $\hat{\mathbf{w}}^{i} \models b$, $w^{i\cdot\cdot} \models$ `@(c)` *P* .
— $w, b \models$ **initial assert property** *Q* iff (if $\hat{\mathbf{w}}^{0} \models b$ then $w \models Q$) .

Neutral satisfaction of properties is as follows:

— $w \models$ (*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 $0 < k < |w|$ so that $w^{k} \models b$ and $w^{0,k-1}\mathsf{T}^{\omega} \models P$. Here, $w^{0,-1}$ denotes the empty word.

— $w \models$ **not** P iff $\hat{\mathbf{w}} \models / P$.
— $w \models R$ iff there exists $0 < j < |w|$ so that $w^{0,j} \models R$.
— $w \models$ (R | -> P) iff for every $0 < j < |w|$ so that $\hat{\mathbf{w}}^{0,j} \models R$, $w^{j..} \models P$.
— $w \models (P1$ **or** $P2)$ iff $w \models P1$ or $w \models P2$.
--- $w \models (P1$ **and** $P2)$ iff $w \models P1$ and $w \models P2$.

Remark: Because w is nonempty, it can be proved that $w \models$ **not** b iff $w \models$!b.

With

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** T iff for every $0 < i < |w|$ so that $\hat{\mathbf{w}}^{i} \models c$ and $\hat{\mathbf{w}}^{i} \models b$, either $w^{i..} \models$ @(c) T *or* $w^{i..} \models^{d}$ @(c)T.
— $w, b \models$ **always assert property** U iff for every $0 < i < |w|$, if $\hat{\mathbf{w}}^{i} \models b$ then either $w^{i..} \models U$ *or* $w^{i..} \models^{d} U$
.
— $w, b \models$ **initial** @(c) **assert property** T iff for every $0 < i < |w|$ so that $\hat{\mathbf{w}}^{0,i} \models$!c [*0:$] ##1 c and $\hat{\mathbf{w}}^{i} \models b$, either $w^{i..} \models$ @(c)T *or* $w^{i..} \models^{d}$ @(c)T.
— $w, b \models$ **initial assert property** U iff (if $\hat{\mathbf{w}}^{0} \models b$ then either $w \models U$ *or* $w \models^{d} U$) .

Neutral satisfaction of reset properties is as follows:

— $w \models P$ **iff** $w \models P$.

— $w \models$ **disable iff** (b)P if there i ≥0 such that for every 0≤j≤i, $w^{j} \models$/b and $w^{0..i}\ \perp^{\omega} \models$P.

Disabled satisfaction of reset properties is as follows:

— $w \models /^{d} P$.

— $w \models^{d}$ **disable iff** (b)P if there i ≥0 such that $w^{i} \models$b and $w^{0..i-1}$ T $^{\omega} \models$P and $w^{0..i-1} \perp^{\omega} \models$/P.

T is said to pass on w, if $w \models$ T. T is said to be disabled on w, if $w \models^{d}$ T. T is said to fail on w, if $w \models /$ T and $w \models /^{d}$ T. It can be proved that T cannot both pass and be disabled on w.

Neutral satisfaction of properties is as follows:

— $w \models$ (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 $0 < k < |w|$ so that $w^{*} \models b$ and $w^{0,k-1}$T$^{\omega} \models P$. Here, $w^{0,-1}$ denotes the empty word.~~
— $w \models$ **not** P iff $\hat{\mathbf{w}} \models / P$.
— $w \models R$ iff there exists $0 < j < |w|$ so that $w^{0,j} \models R$.
— $w \models$ (R | -> P) iff for every $0 < j < |w|$ so that $\hat{\mathbf{w}}^{0,j} \models R$, $w^{j..} \models P$.
— $w \models (P1$ **or** $P2)$ iff $w \models P1$ or $w \models P2$.
--- $w \models (P1$ **and** $P2)$ iff $w \models P1$ and $w \models P2$.

Remark: Because w is nonempty, it can be proved that $w \models$ **not** b iff $w \models$!b.

Annex E.3.6.1

Replace

Neutral satisfaction of properties is as follows:

— $w,L_0 \models (P)$ **iff** $w,L_0 \models P$.
— $w,L_0 \models Q$ iff $w,L_0 \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
— $w,L_0 \models$ **disable iff** (b) P iff either $w,L_0 \models P$ or there exists $0 < k < |w|$ so that $w^k \models b$ and $w^{0,k-1}\top^\omega,L_0 \models P$. Here, $w^{0,-1}$ denotes the empty word.
— $w,L_0 \models$ **not** P iff $\hat{\mathbf{w}},L_0 \not\models P$.
— $w,L_0 \models R$ iff there exists $0 < j < |w|$ and L_1 so that $w^{0,j},L_0,L_1 \models\!\models R$.
— $w,L_0 \models (R \mid\!\!-\!\!> P)$ iff for every $0 < j < |w|$ and L_1 so that $\hat{\mathbf{w}}^{0,j},L_0,L_1 \models\!\models R$, $w^{j..},L_1 \models P$.
— $w,L_0 \models (P1$ **or** $P2)$ iff $w,L_0 \models P1$ or $w,L_0 \models P2$.
--- $w,L_0 \models (P1$ **and** $P2)$ iff $w,L_0 \models P1$ and $w,L_0 \models P2$.

With

Neutral satisfaction of reset properties is as follows:

— $w,L_0 \models P$ **iff** $w,L_0 \models P$.

— $w,L_0 \models$ **disable iff** $(b)P$ if there $i \geq 0$ such that for every $0 \leq j \leq i$, $w^j \not\models b$ and $w^{0..i}\,\bot^\omega,L_0 \models P$.

Disabled satisfaction of reset properties is as follows:

— $w,L_0 \not\models^{d} P$.

— $w,L_0 \models^{d}$ **disable iff** $(b)P$ if there $i \geq 0$ such that $w^i \models b$ and $w^{0..i-1}\top^\omega,L_0 \models P$ and $w^{0..i-1}\,\bot^\omega,L_0 \not\models P$.

Neutral satisfaction of properties is as follows:

— $w,L_0 \models (P)$ **iff** $w,L_0 \models P$.
— $w,L_0 \models Q$ iff $w,L_0 \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.
~~— $w,L_0 \models$ **disable iff** (b) P iff either $w,L_0 \models P$ or there exists $0 < k < |w|$ so that $w^k \models b$ and $w^{0,k-1}\top^\omega,L_0 \models P$. Here, $w^{0,-1}$ denotes the empty word.~~
— $w,L_0 \models$ **not** P iff $\hat{\mathbf{w}},L_0 \not\models P$.
— $w,L_0 \models R$ iff there exists $0 < j < |w|$ and L_1 so that $w^{0,j},L_0,L_1 \models\!\models R$.
— $w,L_0 \models (R \mid\!\!-\!\!> P)$ iff for every $0 < j < |w|$ and L_1 so that $\hat{\mathbf{w}}^{0,j},L_0,L_1 \models\!\models R$, $w^{j..},L_1 \models P$.
— $w,L_0 \models (P1$ **or** $P2)$ iff $w,L_0 \models P1$ or $w,L_0 \models P2$.
--- $w,L_0 \models (P1$ **and** $P2)$ iff $w,L_0 \models P1$ and $w,L_0 \models P2$.