

## Overview

The purpose of this change is to define shortcuts as in PSL, TCL, and Shell where:

delay operator:        ##[+] is equivalent to    ##[1:\$]

consecutive repeat:    x[+] is equivalent to    x[\*1:\$]

delay operator:        ##[\*] is equivalent to    ##[0:\$]

consecutive repeat:    x[\*] is equivalent to    x[\*0:\$]

also not in PSL, but in TCL and shell:

delay operator:        ##[?] is equivalent to    ##[0:1]

consecutive repeat:    x[?] is equivalent to    x[\*0:1]

In addition, I have fixed the omission of the "]" in the text that describes the repetition operators to avoid confusion between "[\*]" and "[\*constant\_or\_range\_expression]"

## Details

REPLACE (In Syntax 16-2)

```
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
```

...

```
consecutive_repetition ::= [* const_or_range_expression ]
```

WITH

```
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
    | ##[*]
    | ##[+]
    | ##[?]
```

...

```
consecutive_repetition ::=
    [* const_or_range_expression ]
    | [*]
    | [+]
    | [?]
```

On page 312-313: REPLACE

The following is the syntax for sequence concatenation.

```
sequence_expr ::=
    cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    ...
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
cycle_delay_const_range_expression ::=
    constant_expression : constant_expression
    | constant_expression : $
```

*Syntax 16-3—Sequence concatenation syntax (excerpt from Annex A)*

In this syntax, the following statements apply:

- *constant\_expression* is computed at compile time and must result in an integer value.
- *constant\_expression* can only be 0 or greater.
- The \$ token is used to indicate the end of simulation. For formal verification tools, \$ is used to indicate a finite, but unbounded, range.
- When a range is specified with two expressions, the second expression must be greater than or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

WITH

The following is the syntax for sequence concatenation.

```
sequence_expr ::=
    cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    ...
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
    | ##[+]
```

```

    | ##[*]
    | ##[?]
cycle_delay_const_range_expression ::=
    constant_expression : constant_expression
    | constant_expression : $

```

*Syntax 16-3—Sequence concatenation syntax (excerpt from Annex A)*

In this syntax, the following statements apply:

- *constant\_expression* is computed at compile time and must result in an integer value.
- *constant\_expression* can only be 0 or greater.
- The \$ token is used to indicate the end of simulation. For formal verification tools, \$ is used to indicate a finite, but unbounded, range.
- ##[+] is used as an equivalent representation of ##[1:\$]
- ##[\*] is used as an equivalent representation of ##[0:\$]
- ##[?] is used as an equivalent representation of ##[0:1]
- When a range is specified with two expressions, the second expression must be greater than or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

REPLACE:

### 16.8.2 Repetition in sequences

Following is the syntax for sequence repetition.

```

sequence_expr ::=
    ...
    | expression_or_dist [ boolean_abbrev ]
    | sequence_instance [ sequence_abbrev ]
    | ( sequence_expr { , sequence_match_item } ) [ sequence_abbrev ]
    ...
boolean_abbrev ::=
    consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
    constant_expression
    | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
    constant_expression : constant_expression
    | constant_expression : $

```

*Syntax 16-5—Sequence repetition syntax (excerpt from Annex A)*

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression; and the maximum number of iterations either is defined by a non-negative integer constant expression or is \$, indicating a finite, but unbounded, maximum. If both the minimum and maximum numbers of iterations are defined by non-negative integer constant expressions, then the minimum number must be less than or equal to the maximum number.

Three kinds of repetition are provided:

— *Consecutive repetition* ( [ \* ] ): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

— *Goto repetition* ( [ -> ] ): Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

— *Nonconsecutive repetition* ( [ = ] ): Nonconsecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

WITH

### 16.8.2 Repetition in sequences

Following is the syntax for sequence repetition.

```
sequence_expr ::=
    ...
    | expression_or_dist [ boolean_abbrev ]
    | sequence_instance [ sequence_abbrev ]
    | ( sequence_expr { , sequence_match_item } ) [ sequence_abbrev ]
    ...
boolean_abbrev ::=
    consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::=
    [* const_or_range_expression ]
    | [*]
    | [+]
    | [?]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
    constant_expression
    | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
    constant_expression : constant_expression
    | constant_expression : $
```

Syntax 16-5—Sequence repetition syntax (excerpt from Annex A)

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression; and the maximum number of iterations either is defined by a non-negative integer constant expression or is \$, indicating a finite, but unbounded, maximum. If both the minimum and maximum numbers of iterations are defined by non-negative integer constant expressions, then the minimum number must be less than or equal to the maximum number.

Three kinds of repetition are provided:

— *Consecutive repetition* ( [ \*const\_or\_range\_expression ] ): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand. [\*] is an equivalent representation of [\*0:\$], [+] is an equivalent representation of [\*1:\$], and [?] is an equivalent representation of [\*0:1].

— *Goto repetition* ( [ ->const\_or\_range\_expression ] ): Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

— *Nonconsecutive repetition* ( [ =const\_or\_range\_expression ] ): Nonconsecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

REPLACE in A.2.10

```
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
```

WITH

```
cycle_delay_range ::=
    ## integral_number
    | ## identifier
    | ## ( constant_expression )
    | ## [ cycle_delay_const_range_expression ]
    | ## [*]
    | ## [+]
    | ## [?]
```

REPLACE (Also in A.2.10)

```
consecutive_repetition ::= [* const_or_range_expression ]
```

WITH

```
consecutive_repetition ::=
```

$[* \text{ const\_or\_range\_expression } ]$   
 $[[*]$   
 $[+]$   
 $[?]$

REPLACE

### F.2.3.2 Derived consecutive repetition operators

- Let  $m > 0$ .  $R[*m] \equiv ( R \#\#1 R \#\#1 \dots \#\#1 R )$  //  $m$  copies of  $R$ .
- $R[*0:\$] \equiv ( R[*0] \text{ or } R[*1:\$] )$ .
- Let  $m < n$ .  $R[*m:n] \equiv ( R[*m] \text{ or } R[*m+1] \text{ or } \dots \text{ or } R[*n] )$ .
- Let  $m > 1$ .  $R[*m:\$] \equiv ( R[*m-1] \#\#1 R[*1:\$] )$ .

### F.2.3.3 Derived delay and concatenation operators

Let  $m < n$ .

- $( \#\#[m:n] R ) \equiv ( 1[*m:n] \#\#1 R )$ .
- $( \#\#[m:\$] R ) \equiv ( 1[*m:\$] \#\#1 R )$ .
- $( \#\#m R ) \equiv ( 1[*m] \#\#1 R )$ .
- Let  $m > 0$ .  $( R1 \#\#[m:n] R2 ) \equiv ( R1 \#\#1 1[*m-1:n-1] \#\#1 R2 )$ .
- Let  $m > 0$ .  $( R1 \#\#[m:\$] R2 ) \equiv ( R1 \#\#1 1[*m-1:\$] \#\#1 R2 )$ .
- Let  $m > 1$ .  $( R1 \#\#m R2 ) \equiv ( R1 \#\#1 1[*m-1] \#\#1 R2 )$ .
- $( R1 \#\#[0:0] R2 ) \equiv ( R1 \#\#0 R2 )$ .
- Let  $n > 0$ .  $( R1 \#\#[0:n] R2 ) \equiv ( ( R1 \#\#0 R2 ) \text{ or } ( R1 \#\#[1:n] R2 ) )$ .
- $( R1 \#\#[0:\$] R2 ) \equiv ( ( R1 \#\#0 R2 ) \text{ or } ( R1 \#\#[1:\$] R2 ) )$ .

WITH

### F.2.3.2 Derived consecutive repetition operators

- Let  $m > 0$ .  $R[*m] \equiv ( R \#\#1 R \#\#1 \dots \#\#1 R )$  //  $m$  copies of  $R$ .
- $R[*0:\$] \equiv ( R[*0] \text{ or } R[*1:\$] )$ .
- Let  $m < n$ .  $R[*m:n] \equiv ( R[*m] \text{ or } R[*m+1] \text{ or } \dots \text{ or } R[*n] )$ .
- Let  $m > 1$ .  $R[*m:\$] \equiv ( R[*m-1] \#\#1 R[*1:\$] )$ .
- $R[*] \equiv ( R[*0] \text{ or } R[*1:\$] )$ .
- $R[+] \equiv ( R[*1:\$] )$ .
- $R[?] \equiv ( R[*0:1] )$ .

### F.2.3.3 Derived delay and concatenation operators

Let  $m < n$

- $( \#\#[m:n] R ) \equiv ( 1[*m:n] \#\#1 R )$ .
- $( \#\#[m:\$] R ) \equiv ( 1[*m:\$] \#\#1 R )$ .
- $( \#\#m R ) \equiv ( 1[*m] \#\#1 R )$ .
- $( \#\#[*] R ) \equiv ( \#\#[0:\$] R )$ .
- $( \#\#[+] R ) \equiv ( \#\#[1:\$] R )$ .
- $( \#\#[?] R ) \equiv ( \#\#[0:1] R )$ .
- Let  $m > 0$ .  $( R1 \#\#[m:n] R2 ) \equiv ( R1 \#\#1 1[*m-1:n-1] \#\#1 R2 )$ .
- Let  $m > 0$ .  $( R1 \#\#[m:\$] R2 ) \equiv ( R1 \#\#1 1[*m-1:\$] \#\#1 R2 )$ .
- Let  $m > 1$ .  $( R1 \#\#m R2 ) \equiv ( R1 \#\#1 1[*m-1] \#\#1 R2 )$ .
- $( R1 \#\#[0:0] R2 ) \equiv ( R1 \#\#0 R2 )$ .
- Let  $n > 0$ .  $( R1 \#\#[0:n] R2 ) \equiv ( ( R1 \#\#0 R2 ) \text{ or } ( R1 \#\#[1:n] R2 ) )$ .
- $( R1 \#\#[0:\$] R2 ) \equiv ( ( R1 \#\#0 R2 ) \text{ or } ( R1 \#\#[1:\$] R2 ) )$ .

