

# Assertion Syntax Conclusions

Version 0.9

ASWG (edited by Jay Lawrence)

## 1 Overview

This document presents a comprehensive BNF for SystemVerilog assertions reflecting the consensus reached in the Assertion Syntax Working Group. The remaining change bars reflect changes that occurred in the 3/21/03 meeting of the working group and are intended to inform the group of the final decisions.

## 2 Statements

This section covers the concurrent and procedural statements that are specifically related to assertions. Templates are included here because they were originally introduced by the sv-ac. They have been defined here in a significantly broader context. This context will need to be vetted with both the sv-ac and sv-ec committees.

The following statements are given in a concurrent context and should be added to the SystemVerilog `module_item` production.

```
module_assertion_item ::=
    | property_decl
    | sequence_decl
    | template_instance
    | template_decl
    | concurrent_assertion_item
```

The following `statement_assertion_item` production needs to be added to the `statement_item` production of SystemVerilog.

```
statement_assertion_item ::=
    concurrent_assertion_item
    | immediate_assert_statement

concurrent_assertion_item ::=
    concurrent_assert_statement
    | concurrent_cover_statement

immediate_assert_statement ::=
    [ identifier : ] assert ( expression ) action_block

concurrent_assert_statement ::=
    [ identifier : ] assert ( property_instance ) action_block
```

Note, the difference between an immediate and concurrent assertion statement is that an immediate assertion takes an expression as the argument, the concurrent assertion takes a property\_instance.

```
concurrent_cover_statement ::=
    [ identifier : ] cover ( property_instance ) stmt_or_null

action_block ::=
    stmt_or_null [ else stmt_or_null ]
```

```
stmt_or_null ::=
  statement
  | ;
```

```
Deleted: action_block ::=¶
pass stmt_or_null fail
stmt_or_null¶
¶
¶
```

Note, the production `stmt_or_null` is really the `statement_or_null` production of SystemVerilog. The shorter version has been used here to make this document easier to read.

### 3 Properties

The syntax of a property should resemble the declaration of a task or primitive and should live in the concurrent context as given above.

```
property_decl ::=
  property identifier [ property_formal_args ] ;
  { property_decl_item }
  property_spec
  endproperty

property_formal_args ::=
  ( identifier { , identifier } )

property_decl_item ::=
  sequence_decl
  | variable_decl
```

Variables declared in a property would be available to all sequences used in each `sequence_body` declared in a property either as a locally declared sequence, or directly as an embedded `sequence_body`. However there validity at different locations would be limited by the semantic rules of **and**, **or**, and **intersect**.

```
property_spec ::=
  [ disable iff ( expression ) ] [ not ] property_expr
  | property_identifier [ ( expression_list ) ]

property_expr ::=
  sequence_spec
  | implication

implication ::=
  sequence_spec => [ not ] sequence_spec
  | sequence_spec | => [ not ] sequence_spec
```

Multi-clock sequence implication is supported because each of these sequence specification terms can include its own event control.

### 4 Sequences

This section deals with the declarations of sequences.

```
sequence_decl ::=
  sequence identifier [ sequence_formal_args ] ;
  { sequence_decl_item }
  sequence_spec ;
  endsequence

sequence_formal_args ::=
  [ sequence_formal_var_list ] [ sequence_formal_arg_list ]
```

```

sequence_formal_var_list ::=
    %( sequence_formal_var_decl { , sequence_formal_var_decl }
    )

sequence_formal_var_decl ::=
    sequence_var_formal_mode data_type identifier

sequence_var_formal_mode ::=
    input
    | output
    | inout

sequence_formal_arg_list ::=
    ( identifier { , identifier } )

sequence_decl_item ::=
    variable_declaration
    | sequence_decl

sequence_spec ::=
    clocked_sequence_phrase
    | sequence_phrase

clocked_sequence_phrase ::=
    event_control sequence_phrase

sequence_phrase ::=
    [ delay_range ] sequence_element
    { delay_range sequence_element }

delay_range ::=
    ## const_range_expression
    | ## [ const_range_expression ]
    | ## [ const_range_expression : const_range_expression ]
    | ## [ const_range_expression : $ ]

sequence_element ::=
    sequence_element { , function_blocking_assign }
    | boolean_expr [ boolean_abbrev ]
    | sequence_expr
    | sequence_instance [ sequence_abbrev ]
    | ( sequence_phrase ) [ sequence_abbrev ]

sequence_expr ::=
    | sequence_element and sequence_element
    | sequence_element intersect sequence_element
    | sequence_element or sequence_element
    | first_match ( sequence_phrase )
    | boolean_expr throughout sequence_element
    | sequence_element within sequence_element

sequence_instance ::=
    identifier [ %( identifier_list ) ] [ ( expression_list ) ]

function_blocking_assign ::=
    identifier '=' expression

```

There are 3 different semantics for which an abbreviation is proposed:

- o repeat\_operator M [ : N ] - the expression is repeated M or M:N times (i.e. a, a, a, a)
  - o nth\_event\_operator M [ : N ] - stop at the specified occurrence of an event M or M:N (a, true, a, a, true, a)
  - o counting\_operator M [ : N ] - count the number of occurrences of an expression and return true if the sequence has that number (a, true, a, a, true, a, true, true).
  - o
- ```
boolean_abbrev ::=
    repeat_operator
    | nth_event_operator
    | counting_operator

sequence_abbrev ::=
    repeat_operator

repeat_operator ::=
    [* const_range_expression [ : constant_range_expression ] ]
```

The repeat\_operator is intended to cover the semantic case where an expression is repeated M or M:N times (e.g. a[\* 2] would match (a, a)).

```
counting_operator ::=
    [*= const_range_expression [ : const_range_expression ] ]
```

The counting\_operator M [ : N ] is intended to cover the semantic of counting the number of occurrences of an expression and return true if the sequence has that number (e.g. a[\*= 2] would match (a, true, a, true)).

```
nth_event_operator ::=
    [*> const_range_expression [ : const_range_expression ] ]
```

The nth\_event\_operator M [ : N ] is intended to cover the semantic which matches the nth occurrence of an event M or M:N (e.g. a[\*> 2] would match (a, true, a)).

## 5 Boolean Expression Extensions

```
boolean_expr_op ::=
    expression
    | seq_name.ended
    | seq_name.matched
    | value_change_functions
    | $past ( expression [ , number_of_ticks ] )
    | $countones ( expression )

value_change_functions ::=
    $rose ( expression )
    | $fell ( expression )
    | $stable ( expression )
```

## 6 Templates

Templates have been added by the sv-ac to allow creation of libraries of properties that are easily reused. The Assertion Syntax Working Group has performed a preliminary attempt at integrated this concept into the entire SystemVerilog language. This should be reviewed by the entire technical committee and moved out of the assertion specific chapter.

A template is an extremely powerful macro capability. It differs from a macro in the following ways:

- it is bounded by `template/endtemplate` instead of using `\` for line continuation
- it allows passing of formal parameters by position or by name
- the contents must be syntactically legal SystemVerilog code

When a template is expanded a syntactic inline expansion of the template body is performed, substituting the actual arguments for the formal arguments. This expansion is performed as a preprocessing step during parsing in the same fashion as macros.

See

```

template_decl ::=
    template identifier [ template_formal_arg_list ] ;
    { template_body }
    endtemplate [ : identifier ]

template_body ::=
    { generate_item }

template_formal_arg_list ::=
    ( template_formal_arg { , template_formal_arg } )

template_formal_arg ::=
    identifier [ = actual_arg_expr ]

template_instance ::=
    template_identifier identifier [ template_actual_arg_list
];

template_actual_arg_list ::=
    ( template_actual_arg { , template_actual_arg } )

template_actual_arg ::=
    actual_arg_expr
    | .formal_identifier( actual_arg_expr )

actual_arg_expr ::=
    expression
    | identifier
    | event_control

```

## 7 Usage Intent

Some of the syntax specified may not have obvious usage. This section was included to provide brief examples of a couple of the constructs for illustrative purposes.

### 7.1 Usage of Property and Sequence Variables

The ability to declare sequences and variables inside of a property is the first step in allowing a variable name to be defined as spanning multiple sequences through an implication operator. If a variable is declared in a property then it can be used in any sequences also declared in the property and represents the same variable in all these sequences. For example, a property that uses an embedded Request and Grant sequence that specifies that a request should be followed by a grant within 100 cycles and uses a 'tag' variable to ensure that the address is consistent between the request and grant would look something like:

```

property P (address, req, rdy, ack);

    int tag;

```

```

sequence Request;
    tag = address, req ; ##1 rdy ;
endsequence

sequence Acknowledge;
    ack && address == tag;
endsequence

Request | => [0:100] Acknowledge;

endproperty

```

Alternatively, the sequences could be declared as non-local to the property and parameterized with variable arguments.

```

sequence Request %( output int X ) (address, req, rdy);
    X = address, req ; ##1 rdy ;
endsequence

sequence Acknowledge %( input int Y ) (ack, address);
    ack && address == Y;
endsequence

property P (address, req, rdy, ack);

    int tag;
    Request %(tag)(address, req, rdy) | =>
        [0:100] Acknowledge %(tag) (ack, address);

endproperty

```

Alternatively this could be specified without using embedded sequences at all and directly using sequence bodies as follows.

```

property P ;

    int tag;

    tag = addr, req ; ##1 rdy | => [0:100] (ack, addr == tag);

endproperty

```

## 7.2 Templates

The declaration of a template would be something like:

```

template T ( MSG="Hello, world", EDGE_SPEC=@(negedge clk) );

generate : begin
    int count;

    initial
        count = 0;

    always EDGE_SPEC

```

```
begin
    count++;
    $display($time, "%m\n", MSG);
end

endgenerate

endtemplate
```

Note that the embedded generate statement in the template creates a new block so that names declared in the template will be unique if multiple instances of this template are used.

An instance of this template would be:

```
T T1 ("Goodnight, Irene", @(posedge clk));
```

## 8 SystemVerilog Productions Modified/Referenced

The following is a list of grammar elements that must be modified to include assertion productions.

- o module\_item
- o statement\_item

The following is a list of grammar elements that are referenced in this document that are defined in SystemVerilog.

- o function\_blocking\_assignment
- o [statement\\_or\\_null](#)
- o [event\\_control](#)
- o [generate\\_item](#)
- o [identifier](#)

## 9 Outstanding Issues

This section is provided to be a placeholder to keep a list of outstanding issues; primarily those that ask questions about whether we have provided sufficient syntax to support all the semantics required by sv-ac and DWG.

### 9.1 Inference of conditions and clocks

Specific rules for clock and precondition inference from concurrent\_assert\_statement in a procedural context need to be fully specified.