Conventions used in this Errata document

This document shows errata, and their corrections, reported in the Portable Test and Stimulus Standard Version 1.0, released by Accellera in June, 2018.

- All section numbers are provided to assist the reader in locating the accompanying change(s) in the original document
- All deletions will be marked using strikethrough text: removed text
- All insertions will be marked using blue text.
  - NOTE that in certain cases documenting BNF definitions, where keywords have been inserted, these keywords are shown in red, to show that they are, in fact, keywords. The surrounding text will still be blue to indicate the insertion.
1.2 Language design considerations
This specification also defines a public interface to a C++ library input format that is semantically equivalent to the DSL, as shown in the following clauses (see also Annex C). The PSS C++ and DSL input formats are designed with the intent that tool implementations may combine source files of either format in a single overall stimulus representation, allowing declarations in one format to be referenced in the other.

2. References
ISO/IEC 14882:2011, Programming Languages—C++.4
US ASCII, ANSI X3.4-1986 (ISO 646 International Reference Version)

3.1 Definitions
identifier: Uniquely name an object so it can be referenced.

3.2 Acronyms and abbreviations
API a Application Programming Interface
DSL d Domain-Specific Language
HSI Hardware/Software Interface
PI p Procedural Interface
PSS Portable Test and Stimulus Standard
SUT s System Under Test

4.1 Comments
The token /* introduces a comment, which terminates with the first occurrence of the token */.

4.3 Escaped Identifiers
Escaped identifiers shall start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable non-whitespace ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier \cpu3 is treated the same as a non-escaped identifier cpu3.

Some examples of legal escaped identifiers are shown here:
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b\}
4.4 Keywords

Table 2—PSS keywords

<table>
<thead>
<tr>
<th>abstract</th>
<th>action</th>
<th>activity</th>
<th>assert</th>
<th>bind</th>
<th>bins</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>body</td>
<td>bool</td>
<td>buffer</td>
<td>handle</td>
<td>class</td>
</tr>
<tr>
<td>compile</td>
<td>component</td>
<td>const</td>
<td>constraint</td>
<td>covergroup</td>
<td>coverpoint</td>
</tr>
<tr>
<td>cross</td>
<td>declaration</td>
<td>default</td>
<td>do</td>
<td>dynamic</td>
<td>else</td>
</tr>
<tr>
<td>enum</td>
<td>exec</td>
<td>export</td>
<td>extend</td>
<td>false</td>
<td>file</td>
</tr>
<tr>
<td>foreach</td>
<td>function</td>
<td>has</td>
<td>header</td>
<td>if</td>
<td>iff</td>
</tr>
<tr>
<td>ignore_bins</td>
<td>illegal_bins</td>
<td>import</td>
<td>in</td>
<td>init</td>
<td>inout</td>
</tr>
<tr>
<td>input</td>
<td>instance</td>
<td>int</td>
<td>lock</td>
<td>match</td>
<td>memory</td>
</tr>
<tr>
<td>option</td>
<td>output</td>
<td>override</td>
<td>package</td>
<td>parallel</td>
<td>pool</td>
</tr>
<tr>
<td>post_solve</td>
<td>pre_solve</td>
<td>private</td>
<td>protected</td>
<td>public</td>
<td>rand</td>
</tr>
<tr>
<td>repeat</td>
<td>resource</td>
<td>run_end</td>
<td>run_start</td>
<td>schedule</td>
<td>select</td>
</tr>
<tr>
<td>sequence</td>
<td>share</td>
<td>solve</td>
<td>state</td>
<td>static</td>
<td>stream</td>
</tr>
<tr>
<td>string</td>
<td>struct</td>
<td>super</td>
<td>symbol</td>
<td>target</td>
<td>true</td>
</tr>
<tr>
<td>type</td>
<td>type_option</td>
<td>typedef</td>
<td>unique</td>
<td>void</td>
<td>while</td>
</tr>
<tr>
<td>with</td>
<td>with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.1.3 States

Figure 4 reinforces that writing a state flow object shall be sequential; reading the state flow object may occur in parallel.

5.2.2.1 Locking resources

Each action that locks a resource in a given pool at a given time shall have access to a unique instance of the resource, identified by the integer attribute and the instance_id value for each instance shall be unique.

5.4 Constraints and inferencing

All constraints defined in the object and in all actions that are bound to the object are combined to define the legal set of values available for the object field.

6.3.1 Preliminary Definitions

a) An action-execution of an atomic action type is the execution of its exec-body block, with values assigned to all of its parameters (reachable attributes). The execution of a compound action consists in executing the set of atomic actions it contains, directly or indirectly. For more on execution semantics of compound actions and activities, see Clause 11.

An atomic action-execution has a specific start-time—the time in which its exec-body block is entered, and end-time—the time in which its exec-body block exits (the test itself does not complete successfully before until all actions that have started complete themselves). The start-time of an atomic action-execution is assumed to be under the direct control of the PSS implementation. In contrast, the end-time of an atomic action-execution, once started, depends on its implementation in the target environment, if any (see 6.2.1).
6.3.2 Sequential Scheduling
Two sets of action-executions, S₁ and S₂, are scheduled in sequence if every initial action-execution in S₂ has a scheduling dependency on every final action-execution in S₂. Generally, sequential scheduling of N action-execution sets S₁...Sₙ is the scheduling dependency of every initial action-execution in Sᵢ on every final action-execution in Sᵢ,j for every i from 2 to N, inclusive.

6.3.3 Parallel scheduling
N sets of action-executions S₁...Sₙ are scheduled in parallel if the following two conditions hold.
- All initial action-executions in all N sets are synchronized (i.e., all have the exact same set of scheduling dependencies).
- S₁...Sₙ are all scheduled independently scheduling-wise with respect to one another (i.e., there are no scheduling dependencies across any two sets Sᵢ and Sⱼ).

6.3.4 Concurrent scheduling
N sets of action-executions S₁...Sₙ are scheduled concurrently if S₁...Sₙ are all scheduled independently with respect to one another (i.e., there are no scheduling dependencies across any two sets Sᵢ and Sⱼ).

7 C++ specifics
[NOTE: Insert after Example 2]
In this Errata document, for brevity in the C++ examples, the use of the type_decl and PSS_CTOR constructs is replaced by ‘...’. Note that this use of elipses is distinct from the use of ‘...’ for C++ variadic arguments.

8.1.2 C++ syntax

```cpp
define pss::range

Defined in pss/range.h (see C.36).

```template <class T = int> class range;

Declare a range of values.

**Member functions**

```cpp
range (const detail::AlgebExpr value): constructor, single value
range (const detail::AlgebExpr lhs, const detail::AlgebExpr rhs): constructor, value range
range (const Lower& lhs, const detail::AlgebExpr rhs): constructor, Lower bounded value range
range (const detail::AlgebExpr lhs, const Upper& rhs): constructor, Upper bounded value range
range& operator() (const detail::AlgebExpr lhs, const detail::AlgebExpr rhs): function chaining to declare additional value ranges
range& operator() (const detail::AlgebExpr value): function chaining to declare additional values
```

Syntax 6—C++: Scalar range declaration
pss::rand_attr

Defined in pss/rand_attr.h (see C.35).

    template <class T> class rand_attr;

Declare a random attribute.

Member functions

    rand_attr (const scope& name): constructor
    rand_attr (const scope& name, const width& a_width): constructor, with width (T = int or bit only)
    rand_attr (const scope& name, const range& a_range): constructor, with range (T = int or bit only)
    rand_attr (const scope& name, const width& a_width, const range& a_range): constructor, with width and range (T = int or bit only)
    T& val(): access randomized data
    T* operator->(): access underlying structure
    T& operator*(): access underlying structure

Syntax 8—C++: Scalar rand declarations

8.3.2 C++ syntax

The PSS_EXTEND_ENUM macro is used when extending an enumeration. Again, enumeration values may optionally define values.

8.3.3 Examples

Declare an enum of type config_modes_e with values UNKNOWN, MODE_A, or MODE_B.

    DSL: config_modes_e in [..MODE_B] mode_ub; C++:
    C++: rand_attr<config_modes_e>
         mode_ub("mode_ub", range<config_modes_e>(minlower(),MODE_B));

Declare an enum of type config_modes_e with values MODE_B, MODE_C, or MODE_D.

    DSL: config_modes_e in [MODE_B..] mode_bd; C++:
    C++: rand_attr<config_modes_e>
         mode_bd("mode_bd", range<config_modes_e>(MODE_B, maxupper()));
8.4.2 C++ Syntax

C++ uses attr<std::string> (see Syntax12) or rand_attr<std::string> (see Syntax13) to represent strings. These are template specializations of attr<T> and rand_attr<T>, respectively (see Syntax 7).

```cpp
pss::attr

Defined in pss/attr.h (see C.36).

    template<> class attr<std::string>;

Declare a non-rand string attribute.

Member functions

    attr(const scope& name) : constructor
    std::string& val() : Access to underlying data
```

Syntax 12—C++: Scalar string declaration

```cpp
pss::rand_attr

Defined in pss/rand_attr.h (see C.35).

    template<> class rand_attr<std::string>;

Declare a randomized string.

Member functions

    rand_attr(const scope& name) : constructor
    std::string& val() : Access to underlying data
```

Syntax 13—C++: Scalar rand string declaration

8.6.1 DSL syntax

A `struct` is a pure-data type; `it does not declare an operation sequence`, `it may include exec blocks of any kind other than exec body or exec init`. 
8.6.2 C++ syntax

**pss::attr**

Defined in `pss/attr.h` (see C.5).

```cpp
template <class T> class attr;
```

Declare a scalar non-random struct attribute.

*Member functions*

- `attr (const scope& name): constructor`
- `T& val(): Access randomized data`
- `T* operator->(): access underlying structure`
- `T& operator*(): access underlying structure`

Syntax 16a—C++: Struct non-rand declarations

**pss::attr**

Defined in `pss/rand_attr.h` (see C.35).

```cpp
template <class T> class rand_attr;
```

Declare a scalar non-random struct attribute. *Member functions*

- `rand_attr (const scope& name): constructor`
- `T& val(): Access randomized data`
- `T* operator->(): access underlying structure`
- `T& operator*(): access underlying structure`

Syntax 16b—C++: Struct rand declarations

8.8.2

In C++, the name of each array element is obtained by appending `_[N]` to the array name, where `N` is the index of the element in the array. In Example 13, the names of the individual elements of the `east_routes` array are `east_routes_[0]`, `east_routes_[1]` ... `east_routes_[7]`, respectively.

9.3 Examples

```cpp
class uart_c : public component { ... };
```

*Example 23—C++: Component*

9.4 Components as namespaces

Component types serve as a namespace for their nested types, i.e., action and struct types defined under them. Actions, but not structs, may be thought of as non-static inner classes of the component (for example, as in Java), since each action is associated with a specific component instance.

10.3.1 Atomic actions

Examples of an **atomic action** declaration are shown in Example30 and Example31.

```cpp
action write {
```

Copyright © 2018 Accellera. All rights reserved.
Example 30—DSL: atomic action

11.5.4.1 DSL syntax

c) Formally, each evaluation of a select statement corresponds to the evaluation of just one of the \texttt{activity_labeled_stmts} select_branch statements. All scheduling requirements shall hold for the selected activity statement.
11.5.6.3 Examples

```cpp
class my_test : public action{
    input<security_data> in_security_data {"in_security_data"};
    action_handle<my_action> action1 {"action1"};
    action_handle<my_action> action2 {"action2"};
    action_handle<my_action> action3 {"action3"};

    activity act {
        match {
            cond(in_security_data->val),
            choice {
                range(security_level_e::LEVEL2)
                    (security_level_e::LEVEL4), action1
            },
            choice {
                range(security_level_e::LEVEL3)
                    (security_level_e::LEVEL5), action2
            },
            default_choice { action3 }
        }
    }
};
...
```

Example 70—C++: match statement
11.8.3 Examples

```plaintext
component top{
    buffer B { rand int a; }
    action P1 {
        output B out;
    }
    action P2 {
        output B out;
    }
    action C {
        input B inp;
    }

    pool B B_p;
    bind B { *; }

    action T {
        P1 p1;
        P2 p2;
        C c;
        activity {
            p1;
            p2;
            c;
            bind p1.out c.inp; // c.inp.a == p1.out.a
        }
    }
}
```

Example 77—DSL: bind statement
class B : public buffer { ...
    rand_attr<int> a {"a"}
};
...
class P1 : public action { ...
    output<B> out {"out"};
};
...
class P2 : public action { ...
    output<B> out {"out"};
};
...
class C : public action { ...
    input<B> inp {"inp"};
};
...
class T : public action { ...
    action_handle<P> p {"p"}; action_handle<C> c {"c"};

    activity act {
        p1, p2, c,
        bind b1 {p1->out, c->in}; // c.inp.a == p1.out.a
    }
};
...

Example 78—C++: bind statement

11.9 Hierarchical flow object binding
In the case of a buffer object input to the compound action, the action that produces the buffer object needs to must complete before the activity of the compound action begins, regardless of where within the activity the sub-action to which the input buffer is bound begins. Similarly, the compound action’s activity needs to complete before the compound action’s output buffer is available, regardless of where in the compound action’s activity the sub-action that produces the buffer object executes. The corollary to this statement is no other sub-action in the compound action’s activity may have an input explicitly hierarchically bound to the compound action’s buffer output object. Similarly, no sub-action in the compound action’s activity may have an output that is explicitly hierarchically bound to the compound action’s input object. Consider example 79 and example 80.
Example 79—DSL: Hierarchical flow binding
Example 80—C++ Hierarchical flow binding

For stream objects, the compound action’s activity shall execute in parallel with the action that produces the input stream object to the compound action or consumes the stream object output by the compound action, regardless of where within the activity the sub-action to which the stream object is bound actually executes. The corollary to this statement is all sub-actions within the activity of a compound action that are bound to a stream input/output of the compound activity shall execute in parallel as the first statement in the activity be an initial action in the activity of the compound action. Consider examples 79a and 80a.
action sub_a {
    input data_str din;
    output data_buf dout;
}

action compound_a {
    input data_str data_in;
    output data_buf data_out;
    sub_a a1, a2;
    activity {
        a1;
        a2;
        bind data_in a1.din; // hierarchical bind
        // The following bind statement would be illegal
        // bind data_in a2.din; // a2 is not scheduled in parallel with
        // compound_a
    }
}

Example 79a—DSL: Hierarchical flow binding

class sub_a : public action {
    input<data_str> din{"din"};
    output<data_buf> dout{"dout"};
};

class compound_a : public action {
    input<data_str> data_in{"data_in"};
    output<data_buf> data_out{"data_out"};
    action_handle<sub_a> a1{"a1"}, a2{"a2"};

    activity act{
        a1,
        a2,
        bind b2 {data_in, a1->din}; // hierarchical bind
        // The following bind statement would be illegal
        // bind b4 {data_in, a2->din}; // a2 is not scheduled in parallel with
        // compound_a
    }
};

Example 80a—C++: Hierarchical flow binding

For state object outputs of the compound action, the activity shall complete before any other action may write to or read from the state object, regardless of where in the activity the sub-action executes within the activity. Only one sub-action may be bound to the compound action’s state object output. Any number of sub-actions may have input state objects bound to the compound action’s state object input.

The same hierarchical binding shown in Example79 and Example80 may be used for any type of data flow object.
11.10 Hierarchical resource object binding

```cpp
class sub_a : public action {
    lock <reslk_r> rlkA("rlkA"), rlkB("rlkB");
    share <resshr_r> rshA("rshA"), rshB("rshB");
};
...

class compound_a : public action {
    lock <reslk_r> crlkA("crlkA"), crlkB("crlkB"); share
    <resshr_r> crshA("crshA"), crshB("crshB");
    action_handle<sub_a> a1("a1"), a2("a2");

    activity act {
        schedule {
            a1, a2
        }

        bind b1 {crlkA, a1->rlkA, a2->rlkB}; bind
        b2 {crshA, a1->rshA, a2->rshB}; bind b3
        {crlkB, a1->rlkB, a2->rshB};
        bind b4 {crshB, a1->shB, a2->rlkB}; //illegal
    }
};
...
```

Example 82—C++: Hierarchical resource binding

12.3.3 Examples

```cpp
class mode_e : public enumeration {...};
PSS_ENUM (mode_e,...);
...
struct config_s : public state {
    PSS_CTOR(config_s, state);
    rand_attr<mode_e> mode "{mode}"
};

type_decl<config_s> config_s_decl;
```

Example 88—C++: state object

13.2.3 Examples

Example 95 and Example 96 demonstrate resource claims in lock and share mode. Action
two_dma_chan_transfer claims exclusive access to two different DMA_channel_s instances. It also
claims one CPU_core_s instance in non-exclusive share mode. While two_chan_transfer executes,
no other action may claim either instance of the DMA_channel_s resource, nor may any other action lock the
CPU_core_s resource instance.
15.1.4.1 DSL syntax

```
logical_inequality_expr ::= binary_shift_expr {logical_inequality_rhs}
logical_inequality_rhs ::= 
  inequality_expr_term
| inside_expr_term

inequality_expr_term ::= logical_inequality_op binary_shift_expr
logical_inequality_op ::= < | <= | > | >=
inside_expr_term ::= in [ open_range_list ]
open_range_list ::= open_range_value { , open_range_value }
open_range_value ::= expression [ .. expression ]
```

Syntax 72—DSL: Set membership expression

15.1.8.1 DSL Syntax

```
unique_constraint_item ::= unique { open_range_list hierarchical_id_list } ;
hierarchical_id_list ::= hierarchical_id { , hierarchical_id }
```

Syntax 79—DSL: unique constraint

15.4.3 Randomization of resource objects

```
component top {
  enum rsrckind_e { A, B, C, D};

  resource rsrck_obj {
    rand rsrckind_e kind;
  }

  pool[2] rsrck_obj rsrck_pool;
  bind rsrck_pool *;

  action do_something {
    share rsrck_obj myrsrck_inst;
    constraint myrsrck_inst.kind != A;
  }

  action do_something_else {
    lock rsrck_obj myrsrck_inst;
  }

  action test {
    activity {
      parallel {
        do do_something with { myrsrck_inst.kind != B; };
        do do_something with { myrsrck_inst.kind != C; };
        do do_something_else;
      }
    }
  }
```

Example 140—DSL: Randomizing resource object attributes
Example 141—C++: Randomizing resource object attributes

16.1 Implicit binding and action inferences

a) An input of any kind is not explicitly bound to an output, or an output of stream kind is not explicitly bound to an input.

Note that action inferences may be more than one level deep. The scenario executed by an implementation shall be the transitive closure of the specified scenario per the flow-object dependency relations.
17.2.1 DSL syntax

```plaintext
covergroup_instantiation ::=  
    covergroup_type_instantiation  
    | inline_covergroup  

inline_covergroup ::= covergroup { {covergroup_body_item} } identifier ;  

data_declaration ::= data_type data_instantiation { , data_instantiation } ;  

data_instantiation ::=  
    data_declaration  
    | plain_data_instantiation  

plain_data_instantiation covergroup_instantiation ::=  
    covergroup_identifier [ ( covergroup_portmap_list ) ] [with { covergroup_option } ]  

plain_data_instantiation ::= identifier [array_dim] [= constant_expression]  

covergroup_type_instantiation ::= covergroup_type covergroup_identifier  
    [ ( covergroup_portmap_list ) ] [with { covergroup_option } ] ;  

covergroup_type ::= type_identifier  

covergroup_portmap_list ::=  
    covergroup_portmap { , covergroup_portmap }  
    | hierarchical_id { , hierarchical_id }  

covergroup_portmap ::= .identifier(hierarchical_id)  
```

Syntax 84—DSL: covergroup instantiation

17.2.3 Examples

```plaintext
enum color_e {red, green, blue};  

struct s {  
    rand color_e color;  
    
    covergroup cs1 (color_e color) {  
        c : coverpoint color;  
    }  
    
    cs1 cs1_inst (color) with {  
        option.at_least = 2;  
    }  
}  
```

Example 176—DSL: Creating a covergroup instance with instance options
PSS_ENUM(color_e, red, blue, green);

class s : public structure { ...  
  rand_attr<color_e> color {"color"};

  class cs1 : public covergroup { ...  
    attr<color_e> c {"c"};  
    coverpoint c_cp { "c", c_cp_color };  
  };
  type_decl<cs1> _cs1_t;

  covergroup_inst<cs1> cs1_inst {"cs1_inst",  
    options {  
      at_least(2)  
    },  
    color  
  };  
};  
...

Example 177—C++: Creating a covergroup instance with instance options

17.3.4.1 DSL syntax

covergroup_coverpoint_binspec ::= bins_keyword identifier  
  coverpoint_bins ::=  
    [ covergroup_range_list ] [with ( covergroup_expression )] ;  
    | coverpoint_identifier with ( covergroup_expression ) ;  
    | default ;  
covergroup_range_list ::= covergroup_value_range {, covergroup_value_range}  
covergroup_value_range ::=  
  expression  
  | expression .. [expression]  
  | [expression] .. expression  

bins_keyword ::= bins | illegal_bins | ignore_bins

Syntax 91—C++: bins declaration

Copyright © 2018 Accellera. All rights reserved.
17.3.4.2 C++ syntax

**pss:bins**

Defined in `pss/covergroup_bins.h` (see C.15).

```cpp
template <class T> class bins;
```

Class for capturing coverpoint bins with template parameter of bit or int.

**Member functions**

```cpp
bins(const std::string &name): constructor for default bins
bins(
    const std::string &name,
    const range<T> &ranges): constructor for specified ranges
bins(
    const std::string &name,
    const coverpoint &cp): constructor for coverpoint-bounded bins
const bins<T> &with(const detail::AlgebExpr &expr)
    : apply with expression
```

*Syntax 92—C++: coverpoint bins with template parameter of bit or int*
**pss::bins**

Defined in `pss/covergroup_bins.h` (see C.15).

```cpp
template <class T> class bins;
```

Class for capturing coverpoint bins with template parameter of `vec<bit>` or `vec<int>`.

**Member functions**

- `bins(const std::string &name)`: constructor for default bins
- `bins(const std::string &name, uint32_t size)`: constructor for specified count default bins
- `bins(const std::string &name, uint32_t size, const range<int> &ranges)`: constructor for specified count bins
- `bins(const std::string &name, uint32_t size, const coverpoint &cp)`: constructor for specified count on coverpoint
- `bins(const std::string &name, const range<int> &ranges)`: constructor for unbounded count ranges
- `const bins<T> &with(const detail::AlgebExpr &expr)`: apply with expression

**Syntax 93—C++: coverpoint bins with template parameter of vec<bit> or vec<int>**

### 17.3.4.3 Examples

In [Example182](#) and [Example183](#), the first `bins` construct associates bin `a` with the values of `v_a`, between 0 and 63 and the value 65. The second `bins` construct creates a set of 65 bins `b[127], b[128], ... b[191]`. Likewise, the third `bins` construct creates 3 bins: `c[200], c[201],` and `c[202]`. The fourth `bins` construct associates bin `d` with the values between 1000 and 1023 (the trailing .. represents the maximum value of `v_a`). Every value that does not match `a, b[], c[],` or `d` is added into its own distinct bin.
struct s {
    rand bit[10] v_a;

covergroup ↔ {
    coverpoint v_a {
        bins a = [0..63, 65];
        bins b[] = [127..150, 148..191];
        bins c[] = [200, 201, 202];
        bins d = [1000..];
        bins others[] = default;
    }
} cs;
}

Example 182—DSL: Specifying bins

17.3.5 coverpoint bin with covergroup expressions

struct s {
    rand bit[8] x;

covergroup ↔ {
    a: coverpoint x {
        bins mod3[] = a with ((a % 3) == 0);
    }
} cs;
}

Example 186—DSL: Using with in a coverpoint

17.3.8 Specifying illegal coverage point values

class s : public structure {...
    rand_attr<bit> a {"a", width(4)};

covergroup Instantiate cs23 { "cs23", [6]() {
    coverpoint a_cp { a,
        ignore illegal_bins<bit> {"ignore illegal_vals", range(7)(8)}
    };
}];
}];
...

Example 191—C++: Specifying illegal coverage point values
17.3.9 Value resolution

```plaintext
struct s {
    rand bit[3] p1;
    int [3] p2;

covergroup c1 {
    coverpoint p1 {
        bins b1 = [1, 2..5, 6..10];
        bins b2 = [-1, 1..10, 15];
    }
    coverpoint p2 {
        bins b3 = [1, 2..5, 6..10];
        bins b4 = [-1, 1..10, 15];
    }
} c1;
}
```

Example 192—DSL: Value resolution

17.7 covergroup sampling

Table 5—covergroups sampling

<table>
<thead>
<tr>
<th>Instantiation context</th>
<th>Sampling point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow objects</td>
<td>Sampled when the outputting action completes traversal.</td>
</tr>
<tr>
<td>Resource objects</td>
<td>Sampled before the first action referencing them begins traversal.</td>
</tr>
<tr>
<td>Action</td>
<td>Sampled when the instantiating action completes traversal.</td>
</tr>
<tr>
<td>Data structures</td>
<td>Sampled along with the context in which the data structure is instantiated, e.g., if a data structure is instantiated in an action, the covergroup instantiated in the data structure is sampled when the action completes traversal.</td>
</tr>
<tr>
<td>Memory segments</td>
<td>Sampled along with the context in which the memory segment is instantiated, e.g., if a memory segment is instantiated in an action, the covergroup instantiated in the memory segment is sampled when the action completes traversal.</td>
</tr>
</tbody>
</table>
18.1.1 DSL syntax

```
extend_stmt ::= extend type_category type_identifier { action_body_item } [ ];

   type_category ::= action
   | component
   | buffer
   | stream
   | state
   | buffer
   | resource
   | struct
   | component
   | enum

   extend action type_identifier { action_body_item } [ ; ]
   | extend component type_identifier { component_body_item } [ ; ]
   | extend struct_kind type_identifier { struct_body_item } [ ; ]
   | extend enum type_identifier { enum_item [ , enum_item ] } [ ; ]
```

Syntax 105—DSL: type extension
18.2.3 Examples

```cpp
action axi_write_action { ... };  
action xlator_action { axi_write_action  
    axi_action; axi_write_action  
    other_axi_action; activity {  
        axi_action; // overridden by instance  
        other_axi_action; // overridden by type  
    }  
};  
action axi_write_action_x : axi_write_action_x { ... };  
action axi_write_action_x2 : axi_write_action_x { ... };  
action axi_write_action_x3 : axi_write_action_x { ... };  
action reg2axi_top {  
    override {  
        type axi_write_action with axi_write_action_x;  
        instance xlator.axi_action with axi_write_action_x2;  
    }  
    xlator_action xlator;  
    activity {  
        repeat (10) {  
            xlator; // override applies equally to all 10 traversals  
        }  
    }  
};  
action reg2axi_top_x : reg2axi_top {  
    override {  
        instance xlator.axi_action with axi_write_action_x3;  
    }  
};
```

Example 207—DSL: Type inheritance and overrides

19.1.2 C++ Syntax
C++ uses native namespaces to provide equivalent functionality.

19.1.32 Examples
For examples of package usage, see 20.4.7.
## 20.1.1 DSL syntax

| exec_block_stmt ::= |
| exec_block |
| target_code_exec_block |
| target_file_exec_block |

exec_block ::= exec exec_kind_identifier { { exec_body_stmt } } [ ; ]

exec_kind_identifier ::= 
  | pre_solve |
  | post_solve |
  | body |
  | header |
  | declaration |
  | run_start |
  | run_end |
  | init |

target_code_exec_block ::= exec exec_kind_identifier language_identifier = string ;
target_file_exec_block ::= exec file filename_string = string ;

### Syntax 110—DSL: exec block declaration
20.1.2 C++ syntax

```cpp
pss::exec

Defined in pss/exec.h (see C.22).

class exec;

/// Types of exec blocks
enum ExecKind {
    run_start,
    header,
    declaration,
    init,
    pre_solve,
    post_solve,
    body,
    run_end,
    file
};

Declare an exec block.

Member functions

exec ( ExecKind kind, const std::initializer_list<
detail::AttrCommonAttrCommon>& write_vars ): declare in-line exec
exec ( ExecKind kind,
    const char* language_or_file,
    const char* target_template ): declare target template exec
exec ( ExecKind kind,
    const std::string&
    const char* language_or_file,
    const std::string&
    const char* target_template ): declare target template exec
exec ( ExecKind kind,
    const std::string&
    const char* language_or_file,
    const std::string&
    const char* target_template ): declare target template exec
template <class... R>
class exec(ExecKind kind, R&&... /*detail::ExecStmt8 r): declare native exec
template <class... R> class exec(ExecKind kind, R&&...
/*detail::ExecStmt8 r): declare native exec
eexec ( ExecKind kind,
    std::function<void()> genfunc ): declare procedural-interface exec
eexec ( ExecKind kind,
    std::function<void(std::ostream& code_stream)> genfunc ): declare
generative target-template exec
```

20.4.1 Function declaration

A PI function prototype is declared in a package or component scope within a PSS description. The PI function prototype specifies the function name, return type, and function parameters. See also Syntax112 or Syntax113.
20.4.2 DSL syntax
Insert the following after Syntax 112:

The following also apply.

a) Functions declared in global and package scope are considered static, and are called optionally using package qualification with scope operator (: :).
b) Functions declared in component scope are considered instance (non-static) functions, and are called optionally using dot operator ( . ) on a component instance expression.

20.6.1 DSL syntax

| function_qualifiers ::= import [ import_function_qualifiers ] function type_identifier ;
| import_function_qualifiers ::=  
| method_qualifiers [ language_identifier ]
| | language_identifier
| method_qualifiers ::=  
| target
| solve

Syntax 114—DSL: PI function qualifiers

20.8 Target-template implementation for functions

The target-template form of PI functions (see Syntax116 or Syntax117) allow non-functional languages, such as assembly, to be targeted in an efficient manner. The target-template form of PI functions are always target implementations. Variable references may only be used in expression positions. Function return values shall not be provided, i.e., only functions that return void are supported. Target-template functions declared under components are instance (non-static) functions (see 20.4.2). PSS expressions embedded in the target code (using mustache notation) can make reference to the instance attributes, optionally using this.

20.8.1 DSL syntax

| import_method_target_template_function ::= target [ method_qualifiers ] language_identifier
| function method_prototype = string ;

Syntax 116—DSL: Target-template function implementation

21.2.3 Examples

Example241 shows an example of conditional processing is if PSS were to use C pre-processor directives.

21.3.1 DSL syntax

Syntax125 shows the grammar for a compile has expression.

| compile_has_expr ::= compile has ( constant_expression static_ref )
| static_ref ::= [:] identifier [ :: identifier ]

Syntax 125—DSL: compile has declaration
B.1 Package declarations

package_body_item ::= 
  abstract_action_declaration  
  | struct_declaration  
  | enum_declaration  
  | covergroup_declaration  
  | function_decl  
  | import_class_decl  
  | function_qualifiers  
  | target_template_function  
  | export_action  
  | typedef_declaration  
  | import_stmt  
  | extend_stmt  
  | const_field_declaration // In package scope only  
  | static_const_field_declaration // In component scope only  
  | compile_assert_stmt  
  | package_body_compile_if

B.2 Action declarations

action_body_item ::= 
  activity_declaration  
  | overrides_declaration  
  | constraint_declaration  
  | action_field_declaration  
  | symbol_declaration  
  | covergroup_declaration  
  | exec_block_stmt  
  | static_const_field_declaration  
  | action_scheduling_constraint  
  | attr_group  
  | compile_assert_stmt  
  | inline_covergroup_instantiation  
  | action_body_compile_if

action_handle_declaration ::= action_type_identifier [ array_dim ]

action_handle_declaration ::= 
  action_type_identifier action_instantiation  

action_instantiation ::= 
  action_identifier [array_dim] {, action_identifier[array_dim] } 

exec_body_stmt ::= expression [ assign_op expression ]  

exec_body_method_call_stmt  
  | exec_body_super_stmt  
  | exec_body_assign_stmt  

exec_body_method_call_stmt ::= 
  method_function_symbol_call  

exec_body_super_stmt ::=  super  

exec_body_assign_stmt ::= 
  variable_ref_path assign_op assign_op expression  

B.3 Struct declarations
struct_body_item ::= 
  | constraint_declaration
  | attr_field
  | typedef_declaration
  | exec_block_stmt
  | static_const_field_declaration
  | attr_group
  | compile_assert_stmt
  | inline_covergroup_instantiation
  | struct_body_compile_if

B.4 Procedural interface (PI)
function_qualifiers ::= import [ import_function_qualifiers ]
  | function type_identifier ;
import_method_target_template_function ::= target[method_qualifiers]
language_identifier
  | function method_prototype = string ;

B.8 Data declarations
data_instantiation ::= identifier [ array_dim ] [= constant_expression]
  | covergroup_instantiation
  | plain_data_instantiation

B.9 Data types
action_data_type ::= scalar_data_type
  | user_defined_datatype
  | action_type_identifier
  | action_type = type_identifier

B.10 Constraint
unique_constraint_item ::= unique { open_range_list hierarchical_id_list } ;

B.11 Coverage specification
covergroup_declaration ::= covergroup covergroup_identifier
  ( covergroup_port {, covergroup_port } ) { { covergroup_body_item } } [ ; ]

covergroup_port ::= data_type identifier

covergroup_body_item ::= covergroup_option
  | covergroup_coverpoint
  | covergroup_cross

covergroup_option ::= option . identifier = constant_expression

covergroup_instantiation ::=
covergroup_type_instantiation
| inline_covergroup

inline_covergroup ::= covergroup { { covergroup_body_item } } identifier ;

data_declaration ::= data_type data_instantiation [ , data_instantiation ] ;

covergroup_type_instantiation ::= covergroup_type_identifier covergroup_identifier
    + ( covergroup_portmap_list ) + [ with [ { covergroup_option } ] ]

plain_data_instantiation ::= identifier [ array_dim ] [ = constant_expression ]

covergroup_portmap_list ::= covergroup_portmap{ , covergroup_portmap}
    | hierarchical_id{ , hierarchical_id}

covergroup_portmap ::= .identifier(hierarchical_id)

covergroup_coverpoint ::= [ { data_type } coverpoint_identifier : ] coverpoint
    expression [ iff ( expression ) ] bins_or_empty

covergroup_coverpoint_binspec ::= bins_keyword identifier
    [ [ constant_expression ] ] = coverpoint_bins

B.12 Conditional-compile
compile_has_expr ::= compile has ( constant_expression static_ref )

static_ref ::= [:] identifier { :: identifier }

B.13 Expression
inside_expr_term ::= in [ open_range_list ] +
    variable_ref_path ::= hierarchical_id[[expression[:expression]]]variable_ref +
    variable_ref

    variable_ref ::= identifier [ [ expression [: expression ] ] ]

static_ref_path ::= [ identifier ] :: identifier { :: identifier }

B.14 Identifiers and literals
hierarchical_id_list ::= hierarchical_id { , hierarchical_id }

hierarchical_id ::= identifier hierarchical_id_elem [ .
    identifier hierarchical_id_elem ]

hierarchical_id_elem ::= identifier [ [ expression ] ]

package_identifier ::= hierarchical_identifier

component_action_identifier ::= identifier
covercross_identifier ::= identifier
covergroup_identifier ::= identifier
coverpoint_identifier ::= identifier
buffer_type_identifier ::= type_identifier
covergroup_type_identifier ::= type_identifier
resource_type_identifier ::= type_identifier

Annex E

b)1)v)1. If an action locks a resource instance, no other action claiming that same resource instance may be scheduled \textit{in parallel} concurrently with the locking action.
b)1)v)2. If actions scheduled \textit{in parallel} concurrently collectively attempt to lock more resource instances than are available in the pool, an error shall be generated.