Portable Test and Stimulus Standard Version 2.0

April 2021

Abstract: The definition of the language syntax, C++ library API, and accompanying semantics for the specification of verification intent and behaviors reusable across multiple target platforms and allowing for the automation of test generation is provided. This standard provides a declarative environment designed for abstract behavioral description using actions, their inputs, outputs, and resource dependencies, and their composition into use cases including data and control flows. These use cases capture verification intent that can be analyzed to produce a wide range of possible legal scenarios for multiple execution platforms. It also includes a preliminary mechanism to capture the programmer's view of a peripheral device, independent of the underlying platform, further enhancing portability.

Keywords: behavioral model, constrained randomization, functional verification, hardware-software interface, portability, PSS, test generation.

Notices

Suggestions for improvements to the Portable Test and Stimulus Standard 2.0 are welcome. They should be posted to the PSS Community Forum at:

https://forums.accellera.org/forum/44-portable-stimulus-discussion/

The current Working Group web page is:

http://www.accellera.org/activities/working-groups/portable-stimulus

Introduction

The definition of a Portable Test and Stimulus Standard (PSS) will enable user companies to select the best tool(s) from competing vendors to meet their verification needs. Creation of a specification language for abstract use-cases is required. The goal is to allow stimulus and tests, including coverage and results checking, to be specified at a high level of abstraction, suitable for tools to interpret and create scenarios and generate implementations in a variety of languages and tool environments, with consistent behavior across multiple implementations.

This revision adds new features, corrects errors, clarifies aspects of the language and semantic definitions, and reorganizes some sections relative to version 1.0a of the Portable Test and Stimulus Standard (February 2019). The new features include (by section number):

—	4.8	Aggregate literals
—	8.8	Collection types and the operators and methods that each supports
—	8.9	Reference types
—	10.7	Pure components
—	11	Actions may now include more than one activity statement
—	12	Template types
—	13.3.2	Action handle array traversal
—	13.3.6	Fine-grained scheduling specifiers
—	13.5.1	*replicate* activity statement
—	14.4	Arrays of flow object references
—	15.2	Arrays of resource object references
—	17.1.8	*forall* constraints
—	17.1.10	Default value constraints
—	21.1.2	Nested packages
—	21.1.4	Package aliases
—	21.2	Added rules for declaration and reference ordering
—	22.1.3	*init_down* and *init_up* exec blocks
—	22.2.3	Default function parameter values
—	22.2.4	Generic and vararg function parameters
—	22.2.5	Pure functions
—	22.3	PSS native functions
—	22.7	Abstract procedural constructs in exec blocks
—	24	PSS core library

Participants

The Portable Stimulus Working Group (PSWG) is entity-based. At the time this standard was developed, the PSWG had the following active participants:

Faris Khundakjie, Intel Corporation, *Chair*
Tom Fitzpatrick, Siemens EDA, *Vice-Chair*
David Brownell, Analog Devices, Inc., *Secretary*
Shalom Bresticker, *Technical Editor*

Aedvices: Francois Cerisier

Agnisys, Inc: Rajat Bagga, Yash Bharal

AMD: Prabhat Gupta

AMIQ EDA: Tom Anderson, Adrian Simionescu

Analog Devices, Inc: Gaurav Bhatnagar, David Brownell

Breker Verification Systems, Inc.: Leigh Brady, Adnan Hamid, David Kelf

Cadence Design Systems, Inc.: Angelina Silver, Matan Vax

Intel Corporation: Ramon Chemel, Michael Chin, Faris Khundakjie

Qualcomm Incorporated: Sanjay Gupta, Kunal Kulnari, Santosh Kumar

Semifore, Inc.: Jamsheed Agahi, Josh Rensch

Siemens EDA: Matthew Ballance, Tom Fitzpatrick, Amit Garg

Synopsys, Inc.: Hillel Miller, Sumant Sathe

Vayavya Labs Pvt. Ltd.: Karthick Gururaj

Xilinx, Inc.: Georgios Karyotis

At the time of standardization, the PSWG had the following eligible voters:

Agnisys, Inc	**Qualcomm Incorporated**
AMIQ EDA	**Semifore, Inc.**
Analog Devices, Inc.	**Siemens EDA**
Breker Verification Systems, Inc.	**Synopsys, Inc.**
Cadence Design Systems, Inc.	**Vayavya Labs Pvt. Ltd.**
Intel Corporation	**Xilinx, Inc.**

Contents

List of figures

List of tables

List of syntax excerpts

List of examples

Portable Test and Stimulus Standard Version 2.0

1. Overview

This clause explains the purpose of this standard, describes its key concepts and considerations, details the conventions used, and summarizes its contents.

The Portable Test and Stimulus Standard syntax is specified using Backus-Naur Form (BNF). The rest of this standard is intended to be consistent with the BNF description. If any discrepancies between the two occur, the BNF formal syntax in Annex B shall take precedence. Similarly, the C++ class declarations in Annex C shall take precedence over the rest of this standard when C++ is used as the input format.

1.1 Purpose

The Portable Test and Stimulus Standard defines a specification for creating a single representation of stimulus and test scenarios, usable by a variety of users across different levels of integration under different configurations, enabling the generation of different implementations of a scenario that run on a variety of execution platforms, including, but not necessarily limited to, simulation, emulation, FPGA prototyping, and post-silicon. With this standard, users can specify a set of behaviors once, from which multiple implementations may be derived.

1.2 Language design considerations

The Portable Test and Stimulus Standard (PSS) describes a declarative domain-specific language (DSL), intended for modeling scenario spaces of systems, generating test cases, and analyzing test runs. Scenario elements and formation rules are captured in a way that abstracts from implementation details and is thus reusable, portable, and adaptable. This specification also defines a public interface to a C++ library 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. The portable stimulus specification captured either in DSL or C++ is herein referred to as *PSS*.

PSS borrows its core concepts from object-oriented programming languages, hardware-verification languages, and behavioral modeling languages. PSS features native constructs for system notions, such as data/control flow, concurrency and synchronization, resource requirements, and states and transitions. It also includes native constructs for mapping these to target implementation artifacts.

Introducing a new language has major benefits insofar as it expresses user intention that would be lost in other languages. However, user tasks that can be handled well enough in existing languages should be left to the language of choice, so as to leverage existing skill, tools, flows, and code bases. Thus, PSS focuses on the essential domain-specific semantic layer and links with other languages to achieve other related purposes. This eases adoption and facilitates project efficiency and productivity.

Finally, PSS builds on prevailing linguistic intuitions in its constructs. In particular, its lexical and syntactic conventions come from the C/C++ family, and its constraint and coverage language uses SystemVerilog (IEEE Std 1800)[1] as a reference.

1.3 Modeling basics

A PSS *model* is a representation of some view of a system's behavior, along with a set of abstract flows. It is essentially a set of class definitions augmented with rules constraining their legal instantiation. A model consists of two types of class definitions: elements of behavior, called *actions*; and passive entities used by actions, such as resources, states, and data flow items, collectively called *objects*. The behaviors associated with an action are specified as *activities*. Actions and object definitions may be encapsulated in *components* to form reusable model pieces. All of these elements may also be encapsulated and extended in a *package* to allow for additional reuse and customization.

A particular instantiation of a given PSS model is a called a *scenario*. Each scenario consists of a set of action instances and data object instances, as well as scheduling constraints and rules defining the relationships between them. The scheduling rules define a partial-order dependency relation over the included actions, which determines the execution semantics. A *consistent scenario* is one that conforms to model rules and satisfies all constraints.

Actions constitute the main abstraction mechanism in PSS. An action represents an element in the space of modeled behavior. Actions may correspond directly to operations of the underlying system under test (SUT) and test environment, in which case they are called *atomic actions*. Actions also use *activities* to encapsulate flows of simpler actions, constituting some joint activity or scenario intention. As such, actions can be used as top-level test intent or reusable test specification elements. Actions and objects have data attributes and data constraints over them.

Actions define the rules for legal combinations in general, not relative to a specific scenario. These are stated in terms of references to objects, having some role from the action's perspective. Objects thus serve as data, and control inputs and outputs of actions, or they are exclusively used as resources. Assembling actions and objects together, along with the scheduling and arithmetic constraints defined for them, produces a model that captures the full state-space of possible scenarios. A scenario is a particular solution of the constraints described by the model to produce an implementation consistent with the described intent.

1.4 Test realization

A key purpose of PSS is to automate the generation of test cases and test suites. Tests for electronic systems often involve code running on embedded controllers, exercising the underlying hardware and software layers. Tests may involve code in hardware-verification languages (HVLs) controlling bus functional models, as well as scripts, command files, data files, and other related artifacts. From the PSS model perspective, these are called *target files*, and *target languages*, which jointly implement the test case for a *target platform*.

[1]Information on references can be found in Clause 2.

The execution of a *consistent scenario* essentially consists of invoking its actions' implementations, if any, in their respective scheduling order. An action is invoked immediately after all its dependencies have completed, and subsequent actions wait for it to complete. Thus, actions that have the same set of dependencies are logically invoked at the same time. Mapping atomic actions to their respective implementation for a target platform is captured in several ways, defined in Clause 22.

PSS features a native mechanism for referring to the actual state of the system under test (SUT) and the environment. Runtime values accessible to the generated test can be sampled and fed back into the model as part of an action's execution. These external values are sampled and, in turn, affect subsequent generation, which can be checked against model constraints and/or collected as coverage. The system/environment state can also be sampled during pre-run processing utilizing models and during post-run processing, given a run trace.

Similarly, the generation of a specific test-case from a given scenario may require further refinement or annotations, such as the external computation of expected results, memory modeling, and/or allocation policies. For these, external models, software libraries, or dedicated algorithmic code in other languages or tools may need to be employed. In PSS, the execution of these pre-run computations is defined using the same scheme as described above, with the results linked in the target language of choice.

1.5 Conventions used

The conventions used throughout the document are included here.

1.5.1 Visual cues (meta-syntax)

The meta-syntax for the description of the syntax rules uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
bold	The **bold** font is used to indicate keywords and punctuation, text that shall be typed exactly as it appears. For example, in the following line, the keyword "state" and special characters "{" and "}" shall be typed exactly as they appear: **state** identifier [template_param_decl_list] [struct_super_spec] **{** { struct_body_item } **}**
plain text	The normal or plain text font indicates syntactic categories. For example, an identifier shall be specified in the following line (after the "state" keyword): **state** identifier [template_param_decl_list] [struct_super_spec] **{** { struct_body_item } **}**
italics	The *italics* font in running text indicates a definition. For example, the following line shows the definition of "activities": The behaviors associated with an action are specified as *activities*. The *italics* font in syntax definitions depicts a *meta-identifier*, e.g., *action*_identifier. See also 4.2.
`courier`	The `courier` font in running text indicates DSL or C++ code. For example, the following line indicates DSL code (for a `state`): `state power_state_s { int in [0..4] val; };`

Table 1—Document conventions (Continued)

Visual cue	Represents
[ ] square brackets	Square brackets indicate optional items. For example, the *struct_super_spec* is optional in the following line: **state** identifier [template_param_decl_list] [struct_super_spec] **{** { struct_body_item } **}**
{ } curly braces	Curly braces ({ }) indicate items that can be repeated zero or more times. For example, the following line shows that zero or more *struct_body_item*s can be specified in this declaration: **state** identifier [template_param_decl_list] [struct_super_spec] **{** { struct_body_item } **}**
| separator bar	The separator bar (|) character indicates alternative choices. For example, the following line shows that the `"input"` or `"output"` keywords are possible values in a flow object reference: flow_ref_field_declaration ::= (**input** | **output**) flow_object_type object_ref_field { **,** object_ref_field } **;**
() parentheses	Parentheses (( )) group together alternative choices. For example, the following line shows that a flow object reference begins with either an `"input"` or an `"output"` keyword: flow_ref_field_declaration ::= (**input** | **output**) flow_object_type object_ref_field { **,** object_ref_field } **;**

1.5.2 Notational conventions

The terms “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in the IETF Best Practices Document 14, RFC 2119.

1.5.3 Examples

Any examples shown in this standard are for information only and are only intended to illustrate the use of PSS.

Many of the examples use “`...`” to indicate code omitted for brevity. Where “`...`” is used in Annex C or in C++ syntax boxes, it indicates the use of variadic arguments in C++.

1.6 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text** when initially defined.

1.7 Contents of this standard

The organization of the remainder of this standard is as follows:

— Clause 2 provides references to other applicable standards that are assumed or required for this standard.

— Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.

— Clause 4 defines the lexical conventions used in PSS.

— Clause 5 defines the PSS modeling concepts.

— Clause 6 defines the PSS execution semantic concepts.

— Clause 7 details some specific C++ considerations in using PSS.

— Clause 8 highlights the PSS data types.

— Clause 9 describes the operators and operands that can be used in expressions and how expressions are evaluated.

— Clause 10 - Clause 21 describe the PSS abstract modeling constructs.

— Clause 22 describes the realization of PSS atomic actions.

— Clause 23 describes the process for conditional code processing.

— Clause 24 describes the PSS core library, which consists of portable functionality and utilities that PSS tools must implement.

— Annexes. Following Clause 24 is a series of annexes.

2. References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

ANSI X3.4-1986: Coded Character Sets—7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)[2] (ISO 646 International Reference Version)

IEEE Std 1800™-2017, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.[3, 4]

The IETF Best Practices Document (for notational conventions) is available from the IETF web site: https://www.ietf.org/rfc/rfc2119.txt.

ISO/IEC 14882:2011, Programming Languages—C++.[5]

[2]ANSI publications are available from the American National Standards Institute (https://www.ansi.org/).

[3]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[4]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (https://standards.ieee.org/).

[5]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (https://www.iso.org/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (https://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (https://www.ansi.org/).

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B1][6] should be referenced for terms not defined in this clause.

3.1 Definitions

action: An element of behavior.

activity: An abstract, partial specification of a **scenario** that is used in a **compound action** to determine the high-level intent and leaves all other details open.

atomic action: An **action** that corresponds directly to operations of the underlying system under test (SUT) and test environment.

component: A structural entity, defined per type and instantiated under other components.

compound action: An **action** which is defined in terms of one or more sub-actions.

constraint: An algebraic expression relating attributes of model entities used to limit the resulting scenario space of the **model**.

coverage: A metric to measure the percentage of possible **scenario**s that have actually been processed for a given **model**.

exec block: Specifies the mapping of PSS scenario entities to their non-PSS implementation.

field: A variable associated with an instance of a type.

inheritance: The process of deriving one model element from another of a similar type, but adding or modifying functionality as desired. It allows multiple types to share functionality that only needs to be specified once, thereby maximizing reuse and portability.

loop: A traversal region of an **activity** in which a set of sub-actions is repeatedly executed. Values for the fields of the **action** are selected for each traversal of the loop, subject to the active constraints and resource requirements present.

model: A representation of some view of a system's behavior, along with a set of abstract flows.

object: A passive entity used by an **action**, such as resources, states, and data flow items.

override: To replace one or all instances of an element of a given type with an element of a compatible type inherited from the original type.

package: A way to group, encapsulate, and identify sets of related definitions, namely type declarations and type extensions.

resource: A computational element available in the target environment that may be claimed by an **action** for the duration of its execution.

[6]The numbers in brackets correspond to those of the bibliography in Annex A.

root action: An **action** designated explicitly as the entry point for the generation of a specific **scenario**. Any **action** in a **model** can serve as the root action of some **scenario**.

scenario: A particular instantiation of a given PSS model.

solve platform: The platform on which the test scenario is solved and, where applicable, target test code is generated. In some generation flows, the solve and target platforms may be the same.

target file: Contains textual content to be used in realizing the test intent.

target language: The language used to realize a specific unit of test intent, e.g., ANSI C, assembly language, Perl.

target platform: The execution platform on which test intent is executed.

type extension: The process of adding additional functionality to a model element of a given type, thereby maximizing reuse and portability. As opposed to **inheritance**, extension does not create a new type.

3.2 Acronyms and abbreviations

API	Application Programming Interface
DSL	Domain-Specific Language
PI	Procedural Interface
PSS	Portable Test and Stimulus Standard
SUT	System Under Test
UVM	Universal Verification Methodology

4. Lexical conventions

PSS borrows its lexical conventions from the C language family.

4.1 Comments

The token `/*` introduces a comment, which terminates with the first occurrence of the token `*/`. The C++ comment delimiter `//` is also supported and introduces a comment which terminates at the end of the current line.

4.2 Identifiers

An *identifier* is a sequence of letters, digits, and underscores; it is used to give an object a unique name so that it can be referenced. In a given namespace, identifiers shall be unique. Identifiers are case-sensitive.

A *meta-identifier* can appear in syntax definitions using the form: *construct_name_*identifier, e.g., *action_*identifier. See also B.18.

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 **`0x21`** through **`0x7E`** 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}
    \a*(b+c)
```

4.4 Keywords

PSS reserves the keywords listed in Table 2.

Table 2—PSS keywords

abstract	**action**	**activity**	**array**	**as**	**assert**
bind	**bins**	**bit**	**body**	**bool**	**break**
buffer	**chandle**	**class**	**compile**	**component**	**const**
constraint	**continue**	**covergroup**	**coverpoint**	**cross**	**declaration**
default	**disable**	**do**	**dynamic**	**else**	**enum**
exec	**export**	**extend**	**false**	**file**	**forall**
foreach	**function**	**has**	**header**	**if**	**iff**
ignore_bins	**illegal_bins**	**import**	**in**	**init**	**init_down**
init_up	**inout**	**input**	**instance**	**int**	**join_branch**
join_first	**join_none**	**join_select**	**list**	**lock**	**map**
match	**null**	**output**	**override**	**package**	**parallel**
pool	**post_solve**	**pre_solve**	**private**	**protected**	**public**
pure	**rand**	**ref**	**repeat**	**replicate**	**resource**
return	**run_end**	**run_start**	**schedule**	**select**	**sequence**
set	**share**	**solve**	**state**	**static**	**stream**
string	**struct**	**super**	**symbol**	**target**	**true**
type	**typedef**	**unique**	**void**	**while**	**with**

4.5 Operators

Operators are single-, double-, and triple-character sequences and are used in expressions. *Unary operators* appear to the left of their operand. *Binary operators* appear between their operands. A *conditional operator* has two operator characters that separate three operands.

4.6 Numbers

Constant numbers are specified as integer constants. The formal syntax for numbers is shown in Syntax 1.

```
number ::=
    oct_number
  | dec_number
  | hex_number
  | based_bin_number
  | based_oct_number
  | based_dec_number
  | based_hex_number
bin_digit ::= [0-1]
oct_digit ::= [0-7]
dec_digit ::= [0-9]
hex_digit ::= [0-9] | [a-f] | [A-F]
oct_number ::= 0 { oct_digit | _ }
dec_number ::= [1-9] { dec_digit | _ }
hex_number ::= 0[x|X] hex_digit { hex_digit | _ }
BASED_BIN_LITERAL ::= '[s|S]b|B bin_digit { bin_digit | _ }
BASED_OCT_LITERAL ::= '[s|S]o|O oct_digit { oct_digit | _ }
BASED_DEC_LITERAL ::= '[s|S]d|D dec_digit { dec_digit | _ }
BASED_HEX_LITERAL ::= '[s|S]h|H hex_digit { hex_digit | _ }
based_bin_number ::= [ dec_number ] BASED_BIN_LITERAL
based_oct_number ::= [ dec_number ] BASED_OCT_LITERAL
based_dec_number ::= [ dec_number ] BASED_DEC_LITERAL
based_hex_number ::= [ dec_number ] BASED_HEX_LITERAL
```

Syntax 1—DSL: Integer constants

Integer literal constants can be specified in decimal, hexadecimal, octal, or binary format.

Four forms may be used to express an integer literal constant. The first form is a simple unsized decimal number, which is specified as a sequence of digits starting with `1` though `9` and containing the digits `0` through `9`.

The second form is an unsized hexadecimal number, which is specified with a prefix of `0x` or `0X` followed by a sequence of digits `0` through `9`, `a` through `f`, and `A` through `F`.

The third form is an unsized octal number, which is specified as a sequence of digits starting with `0` and containing the digits `0` through `7`.

The fourth form specifies a *based literal constant*, which is composed of up to three tokens:

— An optional size constant
— An apostrophe character (`'`) followed by a *base format* character
— Digits representing the value of the number.

The first token, a *size constant*, specifies the size of the integer literal constant in bits. This token shall be specified as an unsigned non-zero decimal number.

The second token, a *base format*, is a case-insensitive letter specifying the base for the number. The base is optionally preceded by the single character **s** (or **S**) to indicate a signed quantity. Legal base specifications are **d**, **D**, **h**, **H**, **o**, **O**, **b**, or **B**. These specify, respectively, decimal, hexadecimal, octal, and binary formats. The base format character and the optional sign character shall be preceded by an apostrophe. The apostrophe character and the base format character shall not be separated by white space.

The third token, an unsigned number, shall consist of digits that are legal for the specified base format. The unsigned number token immediately follows the base format, optionally separated by white space.

Simple decimal and octal numbers without the size and the base format shall be treated as *signed integers*. Unsized unbased hexadecimal numbers shall be treated as unsigned. Numbers specified with a base format shall be treated as signed integers only if the **s** designator is included. If the **s** designator is not included, the number shall be treated as an unsigned integer.

If the size of an unsigned number is smaller than the size specified for the literal constant, the unsigned number shall be padded to the left with zeros. If the size of an unsigned number is larger than the size specified for the literal constant, the unsigned number shall be truncated from the left.

The number of bits that compose an unsized number is tool-specific, but shall be at least 32. An unsized number that requires more than 32 bits shall have at least the minimum width needed to properly represent the value, including a sign bit if the number is signed. For example, `0x7_0000_0000`, an *unsigned* hexadecimal number, shall have at least 35 bits. `4294967296` (2**32), a positive *signed* integer, shall be represented by at least 34 bits.

The underscore character (**_**) shall be legal anywhere in a number except as the first character. The underscore character can be used to break up long integer literals to improve readability.

4.6.1 Using integer literals in expressions

A negative value for an integer with no base specifier shall be interpreted differently from an integer with a base specifier. An integer with no base specifier shall be interpreted as a signed value in two's-complement form. An integer with an unsigned base specifier shall be interpreted as an unsigned value.

The following example shows four ways to write the expression "minus 12 divided by 3." Note that `-12` and `-'d12` both evaluate to the same two's-complement bit pattern, but, in an expression, the `-'d12` loses its identity as a signed negative number.

```
int IntA;
IntA = -12 / 3;         // The result is -4.
IntA = -'d12 / 3;       // The result is 1431655761.
IntA = -'sd12 / 3;      // The result is -4.
IntA = -4'sd12 / 3;     // -4'sd12 is the negative of the 4-bit quantity 1100,
                        // which is -4. -(-4) = 4. The result is 1.
```

4.7 String literals

A *string literal* is a sequence of ASCII characters enclosed by a single pair of double quotes (`" ... "`), called a *quoted string*, or a triple pair of double quotes (`""" ... """`), called a *triple-quoted string*. There is no predefined limit to the length of a string literal. The formal syntax for string literals is shown in .

```
string_literal ::=
    QUOTED_STRING
  | TRIPLE_QUOTED_STRING
QUOTED_STRING ::= " { unescaped_character | escaped_character } "
TRIPLE_QUOTED_STRING ::= """{any_ASCII_character}"""
unescaped_character ::= any_printable_ASCII_character
escaped_character ::= \('|"|?|\|a|b|f|n|r|t|v|[0-7][0-7][0-7])
filename_string ::= QUOTED_STRING
```

Syntax 2—DSL: String literals

PSS also includes a **string** data type to which a string literal can be assigned or compared. Variables of type **string** have arbitrary length; they are dynamically resized to hold any string. String literals are implicitly converted to the **string** type when assigned to a **string** type or used in an expression involving **string** type operands.

The *empty string literal* (`""`) represents an empty, or null, string.

Quoted string literals may only contain printable ASCII characters (the decimal values `32` through `126`, or `0x20` through `0x7E` in hexadecimal). Certain characters can be used in quoted string literals when preceded by an *escape character* (a *backslash*). Table 3 lists these characters, with the escape sequence that represents them. A quoted string shall be contained in a single line.

Table 3—Specifying special characters in string literals

Escape sequence	ASCII hex value	Character produced by escape sequence
`\a`	`0x07`	Alert (Beep, Bell)
`\b`	`0x08`	Backspace
`\f`	`0x0C`	Formfeed
`\n`	`0x0A`	Newline
`\r`	`0x0D`	Carriage return
`\t`	`0x09`	Horizontal tab
`\v`	`0x0B`	Vertical tab
`\\`	`0x5C`	`\` character (backslash)
`\"`	`0x22`	`"` character (double quotation mark)
`\'`	`0x27`	`'` character (apostrophe, single quotation mark)
`\?`	`0x3F`	`?` character (question mark)
`\ddd`	`any`	A character specified in 3 octal digits (see Syntax 1). Implementations may issue an error if the character represented is greater than `\377`.

An escape sequence is considered a single character in the string literal. An escaped apostrophe or question mark is treated the same as an unescaped apostrophe or question mark, respectively, i.e., the backslash is ignored. The other escaped characters in the table have different meanings from their unescaped versions. It is illegal for an escape character in a quoted string literal to be followed by any character not appearing in the table above.

In contrast, a triple-quoted string literal may contain any ASCII character, printing or nonprinting. There is no escape character. All characters are passed as they are, unchanged. For example, triple-quoted strings may contain both single and double quotation marks (except for three consecutive double quotation marks) and newline characters.

Both quoted string literals and triple-quoted string literals may be used anywhere a string literal is desired or required, except for *filename_strings* (see *target_file_exec_block* in Syntax 138), where a quoted string is required.

In a string literal that appears in target-template code, *mustache notation* (`{{expression}}`) can be used to reference PSS variables. See 22.5.3 and 22.6 for details.

4.7.1 Examples

The following string literals are equivalent:

```
" \"Humpty Dumpty sat on a wall.\nHumpty Dumpty had a great fall.\" "

""" "Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall." """
```

4.8 Aggregate literals

Aggregate literals are used to specify the content values of collections and structure types. The different types of aggregate literals are described in the following sections. The use of aggregate literals in expressions is described in 9.4.2.

```
aggregate_literal ::=
    empty_aggregate_literal
  | value_list_literal
  | map_literal
  | struct_literal
```

Syntax 3—DSL: Aggregate literals

NOTE—Aggregate literals are not supported in PSS/C++.

4.8.1 Empty aggregate literal

```
empty_aggregate_literal ::= { }
```

Syntax 4—DSL: Empty aggregate literal

Aggregate literals with no values specify an empty *collection* (see 8.8) when used in the context of a variable-sized collection type (**list**, **set**, **map**).

4.8.2 Value list literals

```
value_list_literal ::= { expression { , expression } }
```

Syntax 5—DSL: Value list literal

Aggregate literals for use with **array**s, **list**s, and **set**s (see 8.8) use *value list literal*s. Each element in the list specifies an individual value. When used in the context of a variable-size data type (**list**, **set**), the number of elements in the value list literal specifies the size as well as the values. However, when used in the context of **set**s, each value is counted only once, even if it appears multiple times. When used in the context of **array**s and **list**s, the value list literal also specifies the order of elements, starting with element 0. The data types of the values must match the data type specified in the collection declaration.

When a value list literal is used in the context of an **array**, the value list literal must have the same number of elements as the **array**. It is an error if the value list literal has more or fewer elements than the **array**.

```
int c1[4] = {1, 2, 3, 4};       // OK
int c2[4] =  {1};               // Error: literal has fewer elements than array
int c3[4] = {1, 2, 3, 4, 5, 6}; // Error: literal has more elements than array
```

Example 1—DSL: Value list literals

Values in value list literals may be non-constant expressions.

4.8.3 Map literals

```
map_literal ::= { map_literal_item { , map_literal_item } }
map_literal_item ::= expression : expression
```

Syntax 6—DSL: Map literal

Aggregate literals for use with **map**s (see 8.8.4) use *map literals*. The first element in each colon-separated pair is the key. The second element is the value to be associated with the key. The data types of the expressions must match the data types specified in the **map** declaration. If the same key appears more than once, the last value specified is used.

In Example 2, a map literal is used to set the value of a **map** with integer keys and Boolean values.

```
struct t {
  map<int,bool> m = {1:true, 2:false, 4:true, 8:false};
  constraint m[1]; // True, since the value "true" is associated with key "1"
}
```

Example 2—DSL: Map literals

Both keys and values in map literals may be non-constant expressions.

4.8.4 Structure literals

```
struct_literal ::= { struct_literal_item { , struct_literal_item } }
struct_literal_item ::= . identifier = expression
```

Syntax 7—DSL: Structure literal

A *structure literal* explicitly specifies the name of the **struct** attribute that a given expression is associated with. **Struct** attributes whose value is not specified are assigned the default value of the attribute's data type. The order of the attributes in the literal does not have to match their order in the **struct** declaration. It shall be illegal to specify the same attribute more than once in the literal.

In Example 3, the initial value for the attributes of `s1` is explicitly specified for all attributes. The initial value for the attributes of `s2` is specified for a subset of attributes. The resulting value of both `s1` and `s2` is `{.a=1,.b=2,.c=0,.d=0}`. Consequently, the constraint `s1==s2` holds.

```
struct s {
  int a, b, c, d;
};
struct t {
  s s1 = {.a=1,.b=2,.c=0,.d=0};
  s s2 = {.b=2,.a=1};
  constraint s1 == s2;
}
```

Example 3—DSL: Structure literals

Values in structure literals may be non-constant expressions.

4.8.5 Nesting aggregate literals

Aggregate literals may be nested to form the value of data structures formed from nesting of aggregate data types.

In Example 4, an aggregate literal is used to form a list of **struct** values. Each structure literal specifies a subset of the **struct** attributes.

```
struct s {
  int a, b, c, d;
};
struct t {
  list<s>  my_l = {
    {.a=1,.d=4},
    {.b=2,.c=8}
  };
}
```

Example 4—DSL: Nesting aggregate literals

5. Modeling concepts

A PSS model is made up of a number of elements (described briefly in 1.3) that define a set of possible scenarios to be applied to the Design Under Test (DUT) via the associated test environment. *Scenarios* are composed of behaviors—ultimately executed on some combination of components that make up the DUT or on verification components that define the test environment—and the communication between them. This clause introduces the elements of a PSS model and defines their relationships.

The primary behavior abstraction mechanism in PSS is an *action*, which represents a particular behavior or set of behaviors. Actions combine to form the scenarios that represents the verification intent. Actions that correspond directly to operations performed by the underlying DUT or test environment are referred to as *atomic actions*, which contain an explicit mapping of the behavior to an implementation on the target platform in one of several supported forms. *Compound actions* encapsulate flows of other actions using an *activity* that defines the critical intent to be verified by specifying the relationships between specific actions.

The remainder of the PSS model describes a set of rules that are used by a PSS processing tool to create the scenarios that implements the critical verification intent while satisfying the data flow, scheduling, and resource constraints of the target DUT and associated test environment. In the case where the specification of intent is incomplete (partial), the PSS processing tool shall infer the execution of additional actions and other model elements necessary to make the partial specification complete and valid. In this way, a single partial specification of verification intent may be expanded into a variety of actual scenarios that all implement the critical intent, but might also include a wide range of other behaviors that may provide greater coverage of the functionality of the DUT as demonstrated in the example in Figure 1.

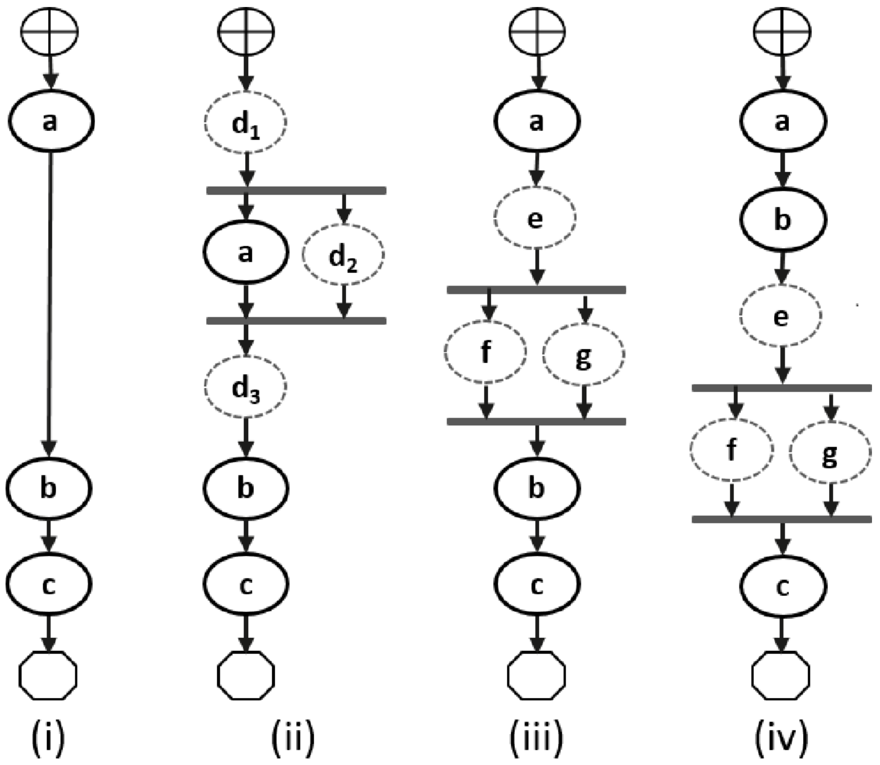

Figure 1—Partial specification of verification intent

In Figure 1, actions `a`, `b`, and `c` are specified to be traversed sequentially in an activity. Depending on the data flow between them, and on other constraints in the model, this may describe a complete scenario specification (see Figure 1(i)), or it may describe a partial specification, which may be expanded into multiple scenarios that infer other actions. All scenarios satisfy the critical intent defined by the activity, where `a` will be traversed, followed sometime later by `b`, followed sometime later by `c`. Figure 1 shows several possible scenarios that may be generated from the partial specification, depending on various factors to be discussed later in this section.

An *activity* primarily specifies the set of actions to be executed and the scheduling relationships between them. Actions may be scheduled sequentially, in parallel, or in various combinations based on conditional evaluation, looping, or randomization constructs. Activities may also include explicit data bindings between actions. An activity that traverses a compound action is evaluated hierarchically, i.e., when a compound sub-action is traversed in an activity, the sub-action activity is traversed fully at that point in the parent activity (see 5.3.2).

5.1 Modeling data flow

Actions may be declared to have inputs and/or outputs of a given data flow object type. The data flow object types define scheduling semantics for the given action relative to those with which it shares the object. Data flow objects may be declared directly or may inherit from user-defined data structures or other flow objects of a compatible type. An action that outputs a flow object is said to *produce* that object and an action that inputs a flow object is said to *consume* the object. Data flow objects are described in Clause 14.

5.1.1 Buffers

The first kind of data flow object is the *buffer* type. A buffer represents *persistent* data that can be written (output) by a producing action and may be read (input) by any number of consuming actions. As such, a buffer defines a strict scheduling dependency between the producer and the consumer that requires the producing action to complete its execution—and, thus, complete writing the buffer object—before execution of the consuming action may begin to read the buffer (see Figure 2). Note that other consuming actions may also input the same buffer object. While there are no implied scheduling constraints between the consuming actions, none of them may start until the producing action completes.

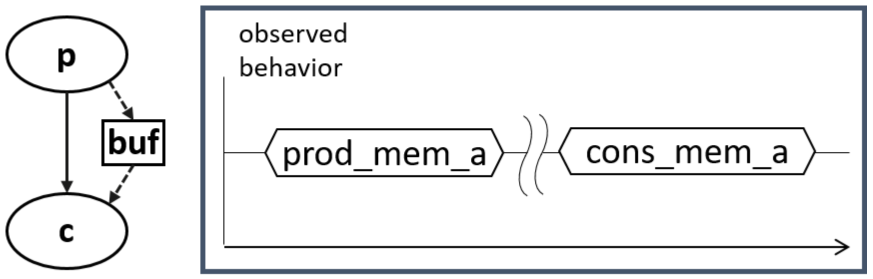

Figure 2—Buffer flow object semantics

Figure 2 illustrates the sequential scheduling semantics between the producer and consumer of a buffer flow object.

In Figure 1(i), assume that action `a` produces a buffer of a particular type, and `b` inputs a buffer object of a compatible type, In this case, we say that the buffer object is *bound* from the output of `a` to the input of `b`, since the semantics of the buffer object support the activity. Similarly, in Figure 1(ii), if, instead of action `a`, action d produced a buffer object of a compatible type for action b, action d could be inferred as the producer of the buffer for action b to consume. The buffer scheduling semantics allow action d to be inferred at any point in the schedule prior to the start of action b (shown in Figure 1(ii) as either d_1, d_2, or d_3), while the activity requires only that action a completes before action b starts. In this case, there is no explicit scheduling constraint between a and d.

5.1.2 Streams

The *stream* flow object type represents *transient* data exchanged between actions. The semantics of the stream flow object require that the producing and consuming actions execute in parallel (i.e., both activities shall begin execution when the same preceding actions complete; see Figure 3). In a stream object, there shall be a one-to-one connection between the producer and consumer.

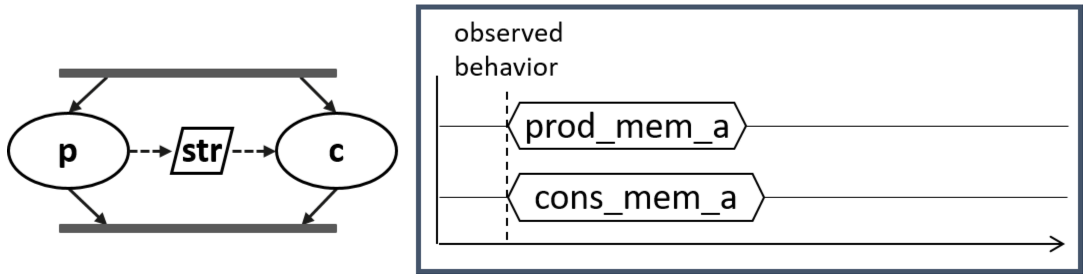

Figure 3—Stream flow object semantics

Figure 3 illustrates the parallel scheduling semantics between the producer and the consumer of a stream flow object.

In Figure 1(iii), the parallel execution of actions f and g dictates that any data exchanged between these actions shall be of the *stream* type. Again, assuming that action a does not output a compatible buffer for action b to input, then action f may be inferred to supply the buffer to action b. If action f inputs or outputs a stream object, then the one-to-one requirement of the stream object would require that action g, which has a compatible stream type, also be inferred to execute in parallel with f. Action e may be inferred if it is needed to supply a buffer input to either f or g.

NOTE—Figure 1(iv) shows an alternate inferred scenario that also satisfies the base scenario of sequential execution of actions a, b, and c, but in this case, the binding between a and b is legal, and action c requires a buffer input that can only be supplied by f or g.

5.1.3 States

The *state* flow object represents the state of some element in the DUT or test environment at a given time. Multiple actions may read or write the state object, but only one write action may execute at a time. Any number of read actions may execute in parallel, but read and write actions shall be sequential (see Figure 4).

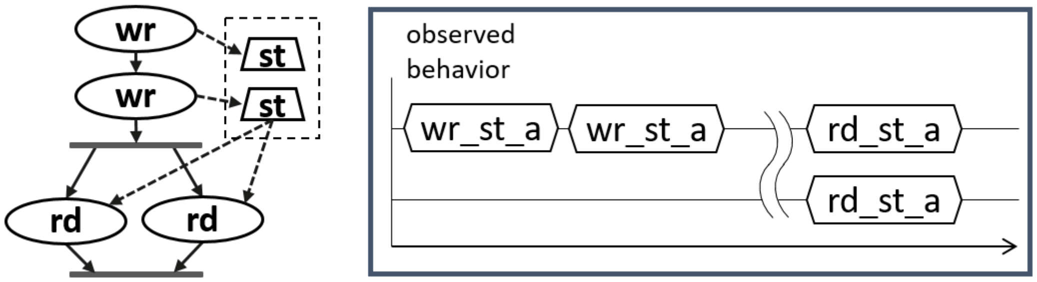

Figure 4—State flow object semantics

State flow objects have a built-in Boolean **`initial`** attribute that is automatically set to **true** initially and automatically set to **false** on the first write operation to the state object. This attribute can be used in constraint expressions to define the starting value for fields of the state object and then allow the values to be modified on subsequent writes of the state object.

5.1.4 Data flow object pools

Data flow objects are grouped into *pools*, which can be used to limit the set of actions that can communicate using objects of a given type. For buffer and stream types, the pool will contain the number of objects of the given type needed to support the communication between actions sharing the pool. For state objects, the pool will only contain a single object of the state type at any given time. Thus, all actions sharing a state object via a pool will see the same value for the state object at a given time. Pools are described in Clause 16.

5.2 Modeling system resources

5.2.1 Resource objects

In addition to declaring inputs and outputs, actions may require system resources that must be accessible in order to accomplish the specified behavior. The *resource* object is a user-defined data object that represents this functionality. Similar to data flow objects, a resource may be declared directly or may inherit from a user-defined data structure or another resource object. Resource objects are described in Clause 15.

5.2.2 Resource pools

Resource objects are also grouped into pools to define the set of actions that have access to the resources. A resource pool is defined to have an explicit number of resource objects in it (the default is `1`), corresponding to the available resources in the DUT and/or test environment. In addition to optionally randomizable data fields, the resource has a built-in non-negative integer attribute called **`instance_id`**, which serves to identify the resource and is unique for each resource in the given pool. Pools are described in Clause 16.

5.2.2.1 Locking resources

An action that requires exclusive access to a resource may *lock* the resource, which prevents any other action that claims the same resource instance from executing until the locking action completes. For a given pool of resource `R`, with size `S`, there may be `S` actions that lock a resource of type `R` executing at any given time. 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 **`instance_id`**. For example, if a DUT contains two DMA channels, the PSS model would define a pool containing two instances of the `DMA_channel` resource type.

In this case, no more than two actions that lock the `DMA_channel` resource could be scheduled concurrently.

5.2.2.2 Sharing resources

An action that requires non-exclusive access to a resource may *share* the resource. An action may not share a resource instance that is locked by another action, but may share the resource instance with other actions that also share the same resource instance. If all resources in a given pool are locked at a given time, then no sharing actions can execute until at least one locking action completes to free a resource in that pool.

5.3 Basic building blocks

5.3.1 Components and binding

A critical aspect of portability is the ability to encapsulate elements of verification intent into "building blocks" that can be used to combine and compose PSS models. A *component* is a structural element of the PSS model that serves to encapsulate other elements of the model for reuse. A component is typically associated with a structural element of the DUT or testbench environment, such as hardware engines, software packages, or testbench agents, and contains the actions that the element is intended to perform, as well as the data and resource pools associated with those actions. Each component declaration defines a unique type that can be instantiated inside other components. The component declaration also serves as a type namespace in which other types may be declared.

A PSS model is composed of one or more component instantiations constituting a static hierarchy beginning with the top-level or root component, called **`pss_top`** by default, which is implicitly instantiated. Components are identified uniquely by their hierarchical path. In addition to instantiating other components, a component may declare functions and class instances (see Clause 10).

NOTE—PSS/C++ does not provide a mechanism for making **`pss_top`**-rooted static references into the component tree. Relative references are supported.

When a component instantiates a pool of data flow or resource objects, it also shall *bind* the pool to a set of actions and/or subcomponents to define who has access to the objects in the pool. Actions may only communicate via an object pool with other actions that are bound to the same object pool. Object binding may be specified hierarchically, so a given pool may be shared across subcomponents, allowing actions in different components to communicate with each other via the pool.

5.3.2 Evaluation and inference

A PSS model is evaluated starting with the top-level *root action*, which shall be specified to a tool. The component hierarchy, starting with **`pss_top`** or a user-specified top-level component, provides the context in which the model rules are defined. If the root action is a compound action, its activity forms the root of a potentially hierarchical activity tree that includes all activities present in any sub-activities traversed in the activity. Additional actions may be inferred as necessary to support the data flow and binding requirements of all actions explicitly traversed in the activity, as well as those previously inferred. Resources add an additional set of scheduling constraints that may limit which actions actually get inferred, but resources do not cause additional actions to be inferred.

The semantics of data flow objects allow the tool to infer, for each action in the overall activity, connections to other actions already instantiated in the activity; or to infer and connect new action instances to conform to the scheduling constraints defined in the activity and/or by the data and resource requirements of the actions, including pool bindings. The model thus consists of a set of actions, with defined scheduling dependencies, along with a set of data flow objects that may be explicitly bound or inferred to connect between actions and a set of resources that may be claimed by the actions as each executes. Actions and flow objects and their bindings may only be inferred as required to make the (partial) activity specification legal. A PSS implementation shall not infer an action or object binding that is not required, either directly or indirectly, to make the activity specification legal. Clause 18 describes action inferencing in more detail.

Figure 5 demonstrates how actions can be inferred to generate multiple scenarios from a single activity.

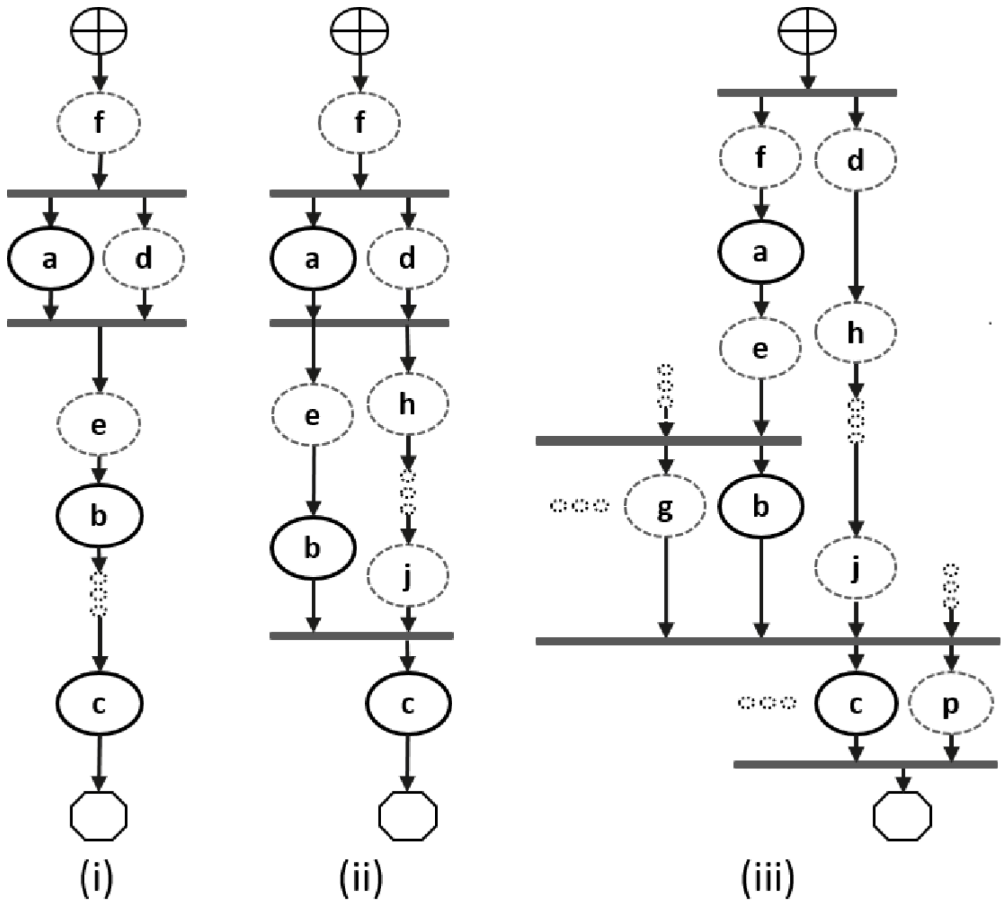

Figure 5—Single activity, multiple scenarios

Looking at Figure 5, actions `a`, `b`, and `c` are scheduled sequentially in an activity. The data flow and resource requirements specified in the model (which are not shown in Figure 5) allow for multiple scenarios to be generated. If action `a` has a buffer or state input, then an action, `f` in this case, is inferred to execute sequentially before `a` in order to provide the buffer or state object. If `a` does not have a buffer or state input, `f` may still be inferred in order to supply an input to `b` or `c`, and may ultimately be scheduled before `a` as shown, although the only real scheduling constraint is that `f` complete before the start of the action that requires the input flow object.

Once inferred, if `f` also has a buffer or state input, then another action shall be inferred to supply that object and so on until an action is inferred that does not have an input (or the tool's inferencing limit is reached, at which point an error shall be generated). For the purposes of this example, action `f` does not have an input.

In Figure 5(i), presume that action `a` produces (or consumes) a stream object. In this case, action `d` is inferred in parallel with `a` since stream objects require a one-to-one connection between actions. Actions `a` and `d` both start upon completion of action `f`. If action `d` also has a buffer input, then another action shall be inferred to provide that input. For Figure 5(i), action f can be presumed to have a second buffer output that gets bound to action d, although a second buffer-providing action could also have been inferred.

If action a produces a buffer object, the buffer may be connected to another action with a compatible input type. In the case where a.out and b.in are incompatible, action e (or a series of actions) may be inferred to receive the output of action a and produce the input to action b. If a.out and b.in are compatible, then the direct connection between a.out and b.in would be inferred here, in which case no action would be inferred between them, although an action inferred to supply the input to c (or for some other reason) could be scheduled between them.

Similarly, in the absence of an explicit binding of b.out to c.in, and if they are incompatible, a series of actions may be inferred prior to the start of action c in order to provide the input of action c. These inferred actions will be scheduled independent of b unless their data flow requirements create scheduling constraints relative to b. As the terminal action in the activity, no action may be inferred after action c however, even if action c produces a buffer object as an output.

If b.out and c.in are incompatible, it is possible to infer another action, j, to supply the buffer input to c.in, as shown in Figure 5(ii). In this case, there are two constraints on when the execution of action c may begin. The activity scheduling requires action b to complete before action c starts. The buffer object semantics also require action j to complete before action c starts. If action j requires a buffer input, a series of actions could be inferred to supply the buffer object. That inferred action chain could eventually be bound to a previously inferred action, such as action d as shown in Figure 5(ii), or it may infer an independent series of actions until it infers an initial action that only produces an output or until the inferencing limit is reached. Since the output of action b is not bound to action c, action b is treated as a terminating action, so no subsequent actions may be inferred after action b.

Finally, Figure 5(iii) shows the case where action c produces or consumes a stream object. In this case, even though action c is the terminating action of the activity, action p shall be inferred to satisfy the stream object semantics for action c. Here, action p is also treated as a terminating action, so no subsequent actions may be inferred. However, additional actions may be inferred either preceding or in parallel to action p to satisfy its data flow requirements. Each action thus inferred is also treated as a terminating action. Similarly, since action b is not bound to action c, b shall also be treated as a terminating action.

5.4 Constraints and inferencing

Data flow and resource objects may define constraint expressions on the values of their data fields (including **instance_id** in the case of resource objects). In addition, actions may also define constraint expressions on the data fields of their input/output flow objects and locked/shared resource objects. For data flow objects, 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. Similarly, the constraints defined for a resource object shall be combined with the constraints defined in all actions that claim the resource. Inferred actions or data flow objects that result in constraint contradictions are excluded from the legal scenario. At least one valid solution must exist for the scenario model for that model to be considered valid.

5.5 Summary

In portable stimulus, a single PSS model may be used to generate a set of scenarios, each of which may have different sets of inferred actions, data flow objects, and resources, while still implementing the critical verification intent explicitly specified in the activity. Each resulting scenario may be generated as a test implementation for the target platform by taking the behavior mapping implementation embedded in each resulting atomic action and generating output code that assembles the implementations and provides any other required infrastructure to ensure the behaviors execute on the target platform according to the scheduling semantics defined by the original PSS model.

6. Execution semantic concepts

6.1 Overview

A PSS test scenario is identified given a PSS model and an action type designated as the root action. The execution of the scenario consists essentially in executing a set of actions defined in the model, in some (partial) order. In the case of atomic actions, the mapped behavior of any **exec body** clauses (see 22.1.3) is invoked in the target execution environment, while for compound actions the behaviors specified by their **activity** statements are executed.

All action executions observed in a test run either correspond to those explicitly called by traversed activities or are implicitly introduced to establish flows that are correct with respect to the model rules. The order in which actions are executed shall conform to the flow dictated by the activities, starting from the root action, and shall also be correct with respect to the model rules. *Correctness* involves consistent resolution of actions' inputs, outputs, and resource references, as well as satisfaction of scheduling constraints. Action executions themselves shall reflect data attribute assignments that satisfy all constraints.

6.2 Assumptions of abstract scheduling

Guarantees provided by PSS are based on general capabilities that test realizations need to have in any target execution environment. The following are assumptions and invariants from the abstract semantics viewpoint.

6.2.1 Starting and ending action executions

PSS semantics assume that target-mapped behavior associated with atomic actions can be invoked in the execution environment at arbitrary points in time, unless model rules (such as state or data dependencies) restrict doing so. They also assume that target-mapped behavior of actions can be known to have completed.

PSS semantics make no assumptions on the duration of the execution of the behavior. They also make no assumptions on the mechanism by which an implementation would monitor or be notified upon action completion.

6.2.2 Concurrency

PSS semantics assume that actions can be invoked to execute concurrently, under restrictions of model rules (such as resource contentions).

PSS semantics make no assumptions on the actual threading framework employed in the execution environment. In particular, a target may have a native notion of concurrent tasks, as in SystemVerilog simulation; it may provide native asynchronous execution threads and means for synchronizing them, such as embedded code running on multi-core processors; or it may implement time sharing of native execution thread(s) in a preemptive or cooperative threading scheme, as is the case with a runtime operating system kernel. PSS semantics do not distinguish between these.

6.2.3 Synchronized invocation

PSS semantics assume that action invocations can be synchronized, i.e., logically starting at the same time. In practice there may be some delay between the invocations of synchronized actions. However, the "sync-time" overhead is (at worse) relative to the number of actions that are synchronized and is constant with respect to any other properties of the scenario or the duration of any specific action execution.

PSS semantics make no assumptions on the actual runtime logic that synchronizes native execution threads and put no absolute limit on the "sync-time" of synchronized action invocations.

6.3 Scheduling concepts

PSS execution semantics define the criteria for legal runs of scenarios. The criterion covered in this section is stated in terms of scheduling dependency—the fundamental scheduling relation between action executions. Ultimately, scheduling is observed as the relative order of behaviors in the target environment per the respective mapping of atomic actions. This section defines the basic concepts, leading up to the definition of sequential and parallel scheduling of action executions.

6.3.1 Preliminary definitions

a) An *action execution* of an atomic action type is the execution of its exec-body block,[7] 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 13.

   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 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).

   The difference between end-time and start-time of an action execution is its *duration*.

b) A *scheduling dependency* is the relation between two action executions, by which one necessarily starts after the other ends. Action execution `b` has a scheduling dependency on `a` if `b`'s start has to wait for `a`'s end. The temporal order between action executions with a scheduling dependency between them shall be guaranteed by the PSS implementation regardless of their actual duration or that of any other action execution in the scenario. Taken as a whole, scheduling dependencies constitute a partial order over action executions, which a PSS solver determines and a PSS scheduler obeys.

   Consequently, the lack of scheduling dependency between two action executions (direct or indirect) means neither one must wait for the other. Having no scheduling dependency between two action executions implies that they may (or may not) overlap in time.

c) Action executions are *synchronized* (scheduled to start at the same time) if they all have the exact same scheduling dependencies. No delay shall be introduced between their invocations, except a minimal constant delay (see 6.2.3).

d) Two or more sets of action executions are *independent* (scheduling-wise) if there is no scheduling dependency between any two action executions across the sets. Note that within each set, there may be scheduling dependencies.

e) Within a set of action executions, the *initial* ones are those without scheduling dependency on any other action execution in the set. The *final* action executions within the set are those in which no other action execution within the set depends.

[7]Throughout this section, exec-body block is referred to in the singular, although it may be the aggregate of multiple exec-body clauses in different locations in PSS source code (e.g., in different extensions of the same action type).

6.3.2 Sequential scheduling

Action executions `a` and `b` are scheduled in *sequence* if `b` has a scheduling dependency on `a`. Two sets of action executions, S_1 and S_2, are scheduled in sequence if every initial action execution in S_2 has a scheduling dependency on every final action execution in S_1. Generally, sequential scheduling of N action execution sets S_1 .. S_n is the scheduling dependency of every initial action execution in S_i on every final action execution in S_{i-1} for every i from 2 to N, inclusive.

For examples of sequential scheduling, see 13.3.3.3.

6.3.3 Parallel scheduling

N sets of action executions S_1 .. S_n 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).

— S_1 .. S_n are all scheduled independently with respect to one another (i.e., there are no scheduling dependencies across any two sets S_i and S_j).

For examples of parallel scheduling, see 13.3.4.3.

6.3.4 Concurrent scheduling

N sets of action executions S_1 .. S_n are scheduled *concurrently* if S_1 .. S_n are all scheduled independently with respect to one another (i.e., there are no scheduling dependencies across any two sets S_i and S_j).

7. C++ specifics

7.1 General

All PSS/C++ types are defined in the **pss** namespace and are the only types defined by this specification. Detailed header files for the C++ constructs introduced in the C++ syntax sections of this document (e.g., Syntax 8) are listed in Annex C.

The prototype for functions (constructors or other functions) in header files may specify a type within a comment, such as in the example below:

```
template <class... R> sequence(R&&... /* detail::Stmt */ r)
```

The type in the comment signifies what the specification allows to be passed as argument type for the function. This convention is typically used with C++ parameter packs.

Nested within the **pss** namespace is the **detail** namespace. Types defined within the **detail** namespace are documented only in order to capture the intended *user-visible* behavior of the PSS/C++ types. Any code that directly refers to types in the **detail** namespace shall be PSS implementation-specific. A PSS implementation is allowed to remove, rename, extend, or otherwise modify the types in the **detail** namespace—as long as user-visible behavior of the types in the **pss** namespace is preserved.

PSS/C++ object hierarchies are managed via the **scope** object, as shown in Syntax 8.

pss::scope

Defined in **pss/scope.h** (see C.43).

```
class scope;
```

Base class for scope.

Member functions

```
scope (const char* name)  : constructor
scope (const std::string& name)  : constructor
template <class T> scope (T* s)  : constructor
```

Syntax 8—C++: scope declaration

Most PSS/C++ class constructors take **scope** as their first argument; this argument is typically passed the name of the object as a string.

The constructor of any user-defined classes that inherit from a PSS class shall always take **const scope&** as an argument and propagate the **this** pointer to the parent scope. The class type shall also be declared using the **type_decl<>** template object, as shown in Syntax 9.

pss::type_decl

Defined in **`pss/type_decl.h`** (see C.49).

```
template <class T> class type_decl;
```

Declare a type.

Member functions

`type_decl ()` : constructor
`T* operator->()` : access underlying type
`T& operator*()` : access underlying type

Syntax 9—C++: type declaration

Example 5 shows an example of this usage.

```
class A1 : public action {
public:
  A1 (const scope& s) : action (this) {}
};
type_decl<A1> A1_decl;
```

Example 5—C++: type declaration

The **`PSS_CTOR`** convenience macro for constructors:

```
#define PSS_CTOR(C,P) public: C (const scope& p) : P (this) {}
```

can also be used to simplify class declarations, as shown in Example 6.

```
class A1 : public action {
  PSS_CTOR(A1,action);
};
type_decl<A1> A1_decl;
```

Example 6—C++: Simplifying class declarations

7.2 PSS/DSL features not supported in PSS/C++

When designing a new language such as PSS/DSL, the language can be tailored to address specific requirements as needed. Unfortunately, when trying to design a C++ library to implement the same feature set, certain limitations of the C++ language rendered supporting some features impossible. The following table outlines the PSS/DSL features that are not supported in PSS/C++, pointing to the relevant sections of the LRM.

Table 4—PSS/DSL features not supported in PSS/C++

Section(s)	Description
4.8	Aggregate literals are not supported in PSS/C++.
5.3.1	PSS/C++ does not provide a mechanism for making **`pss_top`**-rooted static references into the component tree. Relative references are supported.
8.1.1	PSS/C++ does not provide an explicit **`static const`** construct. C++ language features may be used to achieve similar functionality.
8.2.2	PSS/C++ non-**`rand`** attributes of **`bit`** and **`int`** types support specifying an initial value as an expression of other attributes. Other attribute types (e.g., **`enum`**, **`string`**) require the user to specify a C++ constant expression. PSS/C++ **`rand`** attributes do not support specification of an initial value.
8.8.2.3	PSS/C++ does not support array initialization or array methods except **`size()`** and **`sum()`**.
9.5.8	PSS/C++ does not support the conditional operator (<*cond*> **`?`** <*val_true*> **`:`** <*val_false*>).
9.5.9.2	PSS/C++ only supports the *open_range_list* use of the set membership operator.
9.6.1	PSS/C++ supports bit-selects on scalar fields, but does not support part-selects.
11.3.3	PSS/C++ does not support declaring abstract actions.
12.1	PSS parameterized types are not supported in PSS/C++. Referencing PSS parameterized types across PSS/C++ and PSS/DSL is not supported.
13.2.1	PSS/C++ does not support declaring action handles and activity data fields within an activity scope.
13.3.3.1	PSS/C++ requires the *sequence* block to be explicitly specified, while PSS/DSL interprets a curly brace-enclosed scope as a sequence.
13.3.6.1	There is no PSS/C++ equivalent for the fine-grained scheduling modifiers to either **`parallel`** or **`schedule`**.
13.4.3.2 17.1.7.2 22.7.8.2	Only iteration over arrays is supported in PSS/C++. **`foreach`** iteration over other collection types is not supported. Also, in PSS/C++, the index and iteration variables must be explicitly declared in the containing scope of the **`foreach`** loop.
13.6	PSS/C++ does not define a mechanism for referring to the set of activities declared in a super-type—in other words, an equivalent to *activity_super_stmt*.
13.8	Labeled activity statements are not supported in PSS/C++.
14.4.2 15.2.2	PSS/C++ does not support arrays of object references.
17.2.1	PSS/C++ does not support scheduling constraints.

Table 4—PSS/DSL features not supported in PSS/C++ (Continued)

Section(s)	Description
20.2.7	Since PSS/C++ does not support template types (see 12.1), type extension of template types is obviously not supported.
21.1.1.2	PSS/C++ does not provide features directly corresponding to the PSS/DSL **package** and **import** constructs. The C++ `namespace` and `using` constructs provide similar functionality.
22.1.5.2	PSS/C++ does not define a mechanism to invoke procedural blocks in a super-type—in other words, an equivalent to *exec_super_stmt*.
22.2.1.2	PSS/C++ functions declared outside a component are considered to be in the global PSS namespace even if they are declared within a C++ `namespace`.
22.2.2	PSS/C++ function parameters and return types support `int` and `bit` types with width and range information, but do not support associating range information with `string` or `enum` types. PSS/C++ function parameters of class `arg` support array types with width or range specifications, while the `in_arg`/`out_arg`/`inout_arg`/`result` classes do not.
22.2.3	PSS/C++ function parameter default values of type `int` and `bit` may be specified using the `in_arg` class. Parameters of other types, and parameters specified using `arg` and `inout_arg` may not be given default values.
22.3.2	PSS/C++ version only allows definition of native PSS target functions, not of solve functions. PSS/C++ does allow declaration of function prototypes for both target and solve functions.
22.7.2.2	PSS/C++ may require special handling for destruction of local variables declared in procedural scopes. The variables need to be destructed in the reverse order of construction. This is automatically achieved if all variables are on the stack. Otherwise, if they are allocated on the heap, the user must ensure correct order of destruction.
Clause 23	PSS/C++ does not provide features directly corresponding to PSS/DSL conditional code processing. Native C++ features (e.g., the pre-processor) provide similar features.
Clause 24	PSS/C++ does not provide an interface to the core library features—storage management, register models, and executors.

8. Data types

8.1 General

In this document, "*scalar*" means a single data item of type **bit**, **int**, **bool**, **enum**, **string**, or **chandle**, unless otherwise specified. A **struct** (see 8.7) or *collection* (see 8.8) is not a scalar. A **typedef** (see 8.10) of a scalar data type is also a scalar data type.

The term "*aggregate"* refers both to *collections* and to **structs**. The term "*aggregate*" does not include **actions**, **components**, *flow objects*, or *resource objects*. Aggregates may be nested. A **typedef** of an aggregate data type is also an aggregate data type.

A "*plain-data type*" is a scalar or an aggregate of scalars. Nested aggregates are also plain-data types. A **typedef** of a plain-data type is also a plain-data type.

Fields of all scalar types except **chandle** are *randomizable*. Array collections of randomizable types are also randomizable, but the **list**, **map**, and **set** collection types are not randomizable.

A field of randomizable type may be declared as *random* by preceding its declaration with the **rand** keyword. It shall be an error to declare a field of non-randomizable type as **rand**.

8.1.1 DSL syntax

The DSL syntax for data types and data declarations is shown in Syntax 10.

```
data_type ::=
    scalar_data_type
  | collection_type
  | reference_type
  | type_identifier
scalar_data_type ::=
    chandle_type
  | integer_type
  | string_type
  | bool_type
  | enum_type
data_declaration ::= data_type data_instantiation { , data_instantiation } ;
data_instantiation ::= identifier [ array_dim ] [ = constant_expression ]
array_dim ::= [ constant_expression ]
attr_field ::= [ access_modifier ] [ rand | static const ] data_declaration
access_modifier ::= public | protected | private
```

Syntax 10—DSL: Data types and data declarations

Scalar data types are described in 8.2 through 8.6, structure data types are described in 8.7, and collection data types are described in 8.8. Reference types are described in 8.9. Access protection and access modifiers are described in 20.4.

NOTE—PSS/C++ does not provide an explicit **`static const`** construct. C++ language features may be used to achieve similar functionality.

8.2 Numeric types

PSS supports two 2-state numeric data types. These fundamental numeric data types are summarized in Table 5, along with their default widths and value domains.

Table 5—Numeric data types

Data type	Default width	Default domain	Signed/Unsigned
int	32 bits	-2^31 .. (2^31-1)	Signed
bit	1 bit	0..1	Unsigned

4-state values are not supported. If 4-state values are passed into the PSS model via the *foreign procedural interface* (see 22.4), any **`X`** or **`Z`** values are converted to `0`.

8.2.1 DSL syntax

The DSL syntax for numeric types is shown in Syntax 11.

```
integer_type ::= integer_atom_type
   [ [ constant_expression [ : 0 ] ] ]
   [ in [ domain_open_range_list ] ]
integer_atom_type ::=
     int
   | bit
domain_open_range_list ::= domain_open_range_value { , domain_open_range_value }
domain_open_range_value ::=
    constant_expression [ .. constant_expression ]
  | constant_expression ..
  | .. constant_expression
```

Syntax 11—DSL: Numeric type declaration

The following also apply:

a) Numeric values of **bit** type are unsigned. Numeric values of **int** type are signed.

b) The default value of the **bit** and **int** types is `0`.

c) Widths should be specified with a single expression with a constant positive integer value (e.g., `bit[4]`). A specification of `[N]` is equivalent to `[N-1:0]`. A type specified using dual bounds shall use `0` as the lower bound and a constant non-negative integer value as the upper bound. Specifying a width using dual bounds is considered deprecated in PSS 2.0, and may be removed in a future version.

d) A value domain may be specified for the type. The domain specification consists of a list of one or more values and/or value ranges.

e) The width and value domain specifications are independent. A variable of the declared type can hold values within the intersection of the possible values determined by the specified width (or the default width, if not specified) and the explicit value domain specification, if present.

8.2.2 C++ syntax

The corresponding C++ syntax for Syntax 11 is shown in Syntax 12 through Syntax 16.

pss::bit

Defined in **`pss/bit.h`** (see C.7).

```
   using bit = unsigned int;
```

Declare a bit.

Syntax 12—C++: bit declaration

pss::width

Defined in **`pss/width.h`** (see C.52).

```
   class width;
```

Declare the width of an attribute.

Member functions

`width (const std::size_t& size)` : constructor, width in bits
`width (const std::size_t& lhs, const std::size_t& rhs)` : constructor, width as bit range

Syntax 13—C++: Numeric type width declaration

pss::range

Defined in **`pss/range.h`** (see C.41).

```
class range;
```

Declare a range of values.

Member functions

`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 14—C++: Numeric type range declaration

pss::attr

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

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

Declare a scalar non-random attribute.

Member functions

`attr (const scope& s)` : constructor
`attr (const scope& s, const T& init_val)` : constructor, with initial value
`attr (const scope& s, const width& a_width)` : constructor, with width (T = int or bit only)
`attr (const scope& s, const width& a_width, const T& init_val)` : constructor, with width and initial value (T = int or bit only)
`attr (const scope& s, const range& a_range)` : constructor, with range (T = int, bit, string, or enum)
`attr (const scope& s, const range& a_range, const T& init_val)` : constructor, with range and initial value (T = int, bit, string, or enum)
`attr (const scope& s, const width& a_width, const range& a_range)` : constructor, with width and range (T = int or bit only)
`attr (const scope& s, const width& a_width, const range& a_range, const T& init_val)` : constructor, with width, range and initial value (T = int or bit only)
`T& val()` : access to underlying data

Syntax 15—C++: Scalar non-rand declarations

NOTE—PSS/C++ non-**`rand`** attributes of **`bit`** and **`int`** types support specifying an initial value as an expression of other attributes. Other attribute types (e.g., **`enum`**, **`string`**) require the user to specify a C++ constant expression. PSS/C++ **`rand`** attributes do not support specification of an initial value.

pss::rand_attr

Defined in **`pss/rand_attr.h`** (see C.40).

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

Declare a scalar 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, bit, string, or enum)
`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

Syntax 16—C++: Scalar rand declarations

8.2.3 Examples

DSL and C++ numeric data type examples are shown in-line in this section.

Declare a signed variable that is 32 bits wide.

```
DSL:   int a;
C++:   attr<int> a {"a"};
```

Declare a signed variable that is 5 bits wide.

```
DSL:   int [4:0] a;
C++:   attr<int> a {"a", width (4,0)};
```

Declare an unsigned variable that is 5 bits wide and has the valid values `0..31`.

```
DSL:   bit [5] in [0..31] b;
C++:   attr<bit> b {"b", width(5), range (0,31)};
```

Declare an unsigned variable that is 5 bits wide and has the valid values `1`, `2`, and `4`.

```
DSL:   bit [5] in [1,2,4] c;
C++:   attr<bit> c {"c", width(5), range (1)(2)(4)};
```

Declare an unsigned variable that is 5 bits wide and has the valid values `0..10`.

```
DSL:   bit [5] in [..10] b; // 0 <= b <= 10
C++:   attr<bit> b {"b", width(5), range(lower,10)};
```

Declare an unsigned variable that is 5 bits wide and has the valid values `10..31`.

```
DSL:   bit [5] in [10..] b; // 10 <= b <= 31
C++:   attr<bit> b {"b", width(5), range(10, upper)};
```

8.3 Booleans

The PSS language supports a built-in Boolean type, with the type name **bool**. The **bool** type has two enumerated values **true** (=`1`) and **false** (=`0`). When not initialized, the default value of a **bool** type is **false**.

C++ uses **`attr<bool>`** or **`rand_attr<bool>`** to represent Boolean types. These are template specializations of **`attr<T>`** and **`rand_attr<T>`**, respectively (see Syntax 15 and Syntax 16).

8.4 Enumeration types

An *enumeration type* is a distinct user-defined type whose value is restricted to a specified set of integral named constants. Enumeration data types also can be easily referenced or displayed using the enumeration constant names as opposed to their numeric values.

The syntax for declaration of enumeration types is shown in Syntax 17 and Syntax 18.

8.4.1 DSL syntax

The DSL syntax for declaration of enumeration types is shown in Syntax 17.

```
enum_declaration ::= enum enum_identifier { [ enum_item { , enum_item } ] }
enum_identifier ::= identifier
enum_item ::= identifier [ = constant_expression ]
enum_type_identifier ::= type_identifier
enum_type ::= enum_type_identifier [ in [ domain_open_range_list ] ]
```

Syntax 17—DSL: enum declaration

An enumeration type declaration (*enum_declaration*) consists of the keyword **enum** followed by the name of the type (*enum_identifier*) and a list in curly braces of constant names (*enum items*) with optional constant integer value assignments.

The following also apply:

a) *enum_items* are considered static constant members of the enumeration type in which they are declared.

b) The first *enum_item* in the list, if not explicitly assigned a value, is by default assigned the value `0`. Each following *enum_item*, if not explicitly assigned a value, is assigned a value of the previous *enum_item* **+** `1`.

c) *enum_item* values need not be contiguous, nor need they be in ascending arithmetic order. An *enum_item* may be assigned a negative value.

d) Each *enum_item* must have a distinct integer value. No two *enum_items* may have the same value.

e) Enumeration types may be *extended* with the **extend** statement. See 20.2, particularly 20.2.5.

f) *enum_item* identifiers must be unique in the scope of the enumeration type across its initial definition and extensions, if any. However, they need not be unique across different enumeration types declared in the same namespace.

g) *enum_items* can be referenced using their *qualified name* in the form `'`*`enum-type-name::enum-item-name`*`'`.

h) In expression contexts where the expected type is an enumeration type, *enum_items* of that type can be referenced without qualification (see 9.4.3 for the definition of the expected type in expression contexts).
i) An *enum_declaration* may contain an empty set of *enum_item*s, and then have *enum_item*s added in extensions. It shall be illegal to declare an enumeration variable whose type contains no *enum_items* across its initial definition and extensions.
j) When not initialized, the default value of an **enum** field shall be the first *enum_item* in the list. This is not necessarily the value `0` nor the *enum_item* with the minimum value.

Like numeric types, an enumeration type can be restricted to a range of values specified by a *domain_open_range_list* (see 8.2.1 and 8.2.3). The domain specification cannot be specified in the *enum_declaration* itself. See examples of use in 8.4.3.

An **enum** attribute or *enum_item* may be used to assign values to an attribute of the same enumeration type or in an equality comparison.

An **enum** attribute or *enum_item* of one enumeration type may be cast to another enumeration type using the cast operator (see 8.11) if its value is valid for the target enumeration type. An **enum** attribute or *enum_item* may be cast to integer and Boolean data types using the cast operator. Similarly, an integer or Boolean value may be cast to an enumeration type if its value is valid for the target enumeration type.

8.4.2 C++ syntax

The corresponding C++ syntax for Syntax 17 is shown in Syntax 18.

The **`PSS_ENUM`** macro is used to declare an enumeration type. As in C++, enum_items may optionally define values.

The **`PSS_EXTEND_ENUM`** macro is used to extend an enumeration type. Again, enum_items may optionally define values.

pss::enumeration

Defined in **`pss/enumeration.h`** (see C.24).

```
#define PSS_ENUM(enum_name, enum_item, enum_item=value, ...) // 1
#define PSS_EXTEND_ENUM(ext_name, base_name,
                        enum_item, enum_item=value, ...) // 2
```

1) Declare an enumeration type with a name and a list of items (values optional)
2) Extend an enumeration type with a name and a list of items (values optional)

Member functions

`template <class T> enumeration& operator=(const T& t)` : assign an enum value

Syntax 18—C++: enum declaration

In PSS/C++, enum_items are referenced using the name of the enumeration type and the scope operator. enum_items declared in an extension are referenced using the name of the extension and the scope operator.

8.4.3 Examples

Examples of enum usage are shown in Example 7 and Example 8.

```
enum config_modes_e {UNKNOWN, MODE_A=10, MODE_B=20, MODE_C=35, MODE_D=40};

component uart_c {
  action configure {
    rand config_modes_e mode;
    constraint {mode != UNKNOWN;}
  }
};
```

Example 7—DSL: enum data type

The corresponding C++ example for Example 7 is shown in Example 8.

```
PSS_ENUM(config_modes_e, UNKNOWN, MODE_A=10, MODE_B=20, MODE_C=35,
   MODE_D=40);

class uart_c : public component { ...
  class configure : public action { ...
    PSS_CTOR(configure, action);
    rand_attr<config_modes_e> mode {"mode"};
    constraint c {"c", mode != config_modes_e::UNKNOWN};
  };
  type_decl<configure> configure_decl;
};
...
```

Example 8—C++: enum data type

See an example of extending an enumeration in 20.2.5.

Examples of domain specifications for enumeration types are shown below:

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

```
DSL: rand config_modes_e in [MODE_A..MODE_C] mode_ac;
C++: rand_attr<config_modes_e> mode_ac {"mode_ac",
       range (config_modes_e::MODE_A, config_modes_e::MODE_C)};
```

Declare an enum of type `config_modes_e` with values `MODE_A` or `MODE_C`.

```
DSL: rand config_modes_e in [MODE_A, MODE_C] mode_ac;
C++: rand_attr<config_modes_e> mode_ac {"mode_ac",
       range (config_modes_e::MODE_A)(config_modes_e::MODE_C)};
```

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

```
DSL: rand config_modes_e in [..MODE_B] mode_ub;
C++: rand_attr<config_modes_e> mode_ub {"mode_ub",
       range (lower, config_modes_e::MODE_B)};
```

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

```
DSL: rand config_modes_e in [MODE_B..] mode_bd;
C++: rand_attr<config_modes_e> mode_bd {"mode_bd",
        range (config_modes_e::MODE_B, upper)};
```

Note that an *open_range_list* of enums may be used in set membership (**in**) expressions (see 9.5.9) and as a *match_choice* expression in **match** statements (see 13.4.6 and 22.7.10).

8.5 Strings

The PSS language supports a built-in string type with the type name **string**. When not initialized, the default value of a **string** shall be the empty string literal (`""`). See also Syntax 19, Syntax 20, and Syntax 21.

8.5.1 DSL syntax

string_type ::= **string** [**in** **[** string_literal { **,** string_literal } **]**]

Syntax 19—DSL: string declaration

Comma-separated domain specifications are allowed for string data types (see 8.2.1).

8.5.2 C++ syntax

C++ uses **`attr<std::string>`** (see Syntax 20) or **`rand_attr<std::string>`** (see Syntax 21) to represent strings. These are template specializations of **`attr<T>`** and **`rand_attr<T>`**, respectively (see Syntax 15 and Syntax 16).

pss::attr

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

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

Declare a non-rand string attribute.

Member functions

`attr(const scope& name)` : constructor
`attr(const scope& name, const std::string& init_val)` : constructor, with initial value
`std::string& val()` : access to underlying data

Syntax 20—C++: Non-rand string declaration

pss::rand_attr

Defined in **`pss/rand_attr.h`** (see C.40).

```
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 21—C++: Rand string declaration

8.5.3 Examples

The value of a random string-type field can be constrained with equality constraints and can be compared using equality operators, as shown in Example 9 and Example 10.

```
struct string_s {
  rand bit      a;
  rand string   s;

  constraint {
    if (a == 1) {
      s == "FOO";
    } else {
      s == "BAR";
    }
  }
}
```

Example 9—DSL: String data type

The corresponding C++ example for Example 9 is shown in Example 10.

```
struct string_s : public structure { ...
  rand_attr<bit> a {"a"};
  rand_attr<std::string> s {"s"};

  constraint c1 { "c1",
    if_then_else {
      cond (a == 1),
      s == "FOO",
      s == "BAR"
    }
  };
};
...
```

Example 10—C++: String data type

Declare string with values `"Hello"`, `"Hallo"`, or `"Ni Hao"`.

```
DSL: rand string in ["Hello", "Hallo", "Ni Hao"] hello_s;
C++: rand_attr<std::string>
        hello_s {"hello_s", range ("Hello")("Hallo")("Ni Hao")};
```

Note that an *open_range_list*, composed solely of individual string literals, may also be used in set membership (**in**) expressions (see 9.5.9) and as a *match_choice* expression in **match** statements (see 13.4.6 and 22.7.10). Ranges of string literals (e.g., `"a".."b"`) are not permitted.

8.6 Chandles

The **chandle** type (pronounced “see-handle”) represents an opaque handle to a foreign language pointer as shown in Syntax 22 and Syntax 23. A **chandle** is used with the foreign procedural interface (see 22.4) to store foreign language pointers in the PSS model and pass them to foreign language functions. See Annex E for more information about the foreign procedural interface.

A **chandle** has the following restrictions:

— The **rand** qualifier may not be applied to it.

— The only logical operators it may be used with are **==** and **!=**.

— The only literal value with which it may be compared is **0**, which is equivalent to a null handle in the foreign language.

When not initialized, the default value of a **chandle** shall be **0**.

8.6.1 DSL syntax

```
chandle_type ::= chandle
```

Syntax 22—DSL: chandle declaration

8.6.2 C++ syntax

pss::chandle

Defined in **`pss/chandle.h`** (see C.9).

```
class chandle;
```

Declare a chandle.

Member functions

`chandle& operator= (detail::AlgebExpr val)` : assign to chandle

Syntax 23—C++: chandle declaration

8.6.3 Example

Example 11 shows a **struct** containing a **chandle** field that is initialized by the return of a foreign language function.

```
function chandle do_init();

struct info_s {
  chandle ptr;

  exec pre_solve {
    ptr = do_init();
  }
}
```

Example 11—DSL: chandle data type

8.7 Structs

A **struct** type is an aggregate of data items, as shown in Syntax 24 and Syntax 25.

8.7.1 DSL syntax

```
struct_declaration ::= struct_kind struct_identifier [ template_param_decl_list ]
    [ struct_super_spec ] { { struct_body_item } }
struct_kind ::=
    struct
  | object_kind
object_kind ::=
    buffer
  | stream
  | state
  | resource
struct_super_spec ::= : type_identifier
struct_body_item ::=
    constraint_declaration
  | attr_field
  | typedef_declaration
  | exec_block_stmt
  | attr_group
  | compile_assert_stmt
  | covergroup_declaration
  | covergroup_instantiation
  | struct_body_compile_if
  | stmt_terminator
```

Syntax 24—DSL: struct declaration

A **struct** is a plain-data type (see 8.1). That is, a **struct** may contain scalar data items and aggregates thereof. A **struct** declaration may specify a *struct_super_spec*, a previously defined **struct** type from which the new type inherits its members, by using a colon (**:**), as in C++. In addition, **struct**s may

— include **constraint**s (see 17.1) and **covergroup**s (see 19.1 and 19.2);

— include **exec** blocks of any kind other than **init_down**, **init_up**, and **body** (see 22.1**).**

Data items in a **struct** shall be of plain-data types (whether randomizable or not). Declarations of randomizable data items may optionally include the **rand** keyword to indicate that the element shall be randomized when the overall **struct** is randomized (see Example 12 and Example 13). 17.4.1 describes **struct** randomization in detail.

Note that **chandle**s are a non-randomizable scalar data type. **List**s, **map**s, and **set**s are non-randomizable collection types (see 8.8).

8.7.2 C++ syntax

In C++, structures shall derive from the **`structure`** class.

The corresponding C++ syntax for Syntax 24 is shown in Syntax 25, Syntax 26, and Syntax 27.

pss::structure

Defined in **`pss/structure.h`** (see C.47).

```
class structure;
```

Base class for declaring a structure.

Member functions

`structure (const scope& s)` : constructor
`virtual void pre_solve()` : in-line pre_solve exec block
`virtual void post_solve()` : in-line post_solve exec block

Syntax 25—C++: struct declaration

pss::attr

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

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

Declare a non-random struct attribute.

Member functions

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

Syntax 26—C++: Struct non-rand declarations

pss::rand_attr

Defined in **`pss/rand_attr.h`** (see C.40).

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

Declare a 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 27—C++: Struct rand declarations

8.7.3 Examples

Struct examples are shown in Example 12 and Example 13.

```
struct axi4_trans_req {
   rand bit[31:0]   axi_addr;
   rand bit[31:0]   axi_write_data;
   bit              is_write;
   rand bit[3:0]    prot;
   rand bit[1:0]    sema4;
}
```

Example 12—DSL: Struct with rand qualifiers

```
struct axi4_trans_req : public structure { ...
  rand_attr<bit> axi_addr {"axi_addr", width (31, 0)};
  rand_attr<bit> axi_write_data {"axi_write_data", width (31, 0)};
  attr<bit>      is_write {"is_write"};
  rand_attr<bit> prot  {"prot", width (3, 0)};
  rand_attr<bit> sema4 {"sema4", width (1, 0)};
};
type_decl<axi4_trans_req> axi4_trans_req_decl;
```

Example 13—C++: Struct with rand qualifiers

8.8 Collections

Collection types are built-in data types. PSS supports fixed-size **array** and variable-size **list**, **map**, and **set** collections of plain-data types (see 8.1). Each kind of collection has its own keyword, and its declaration specifies the data type of the collection elements (and for **map**s, also the data type of the *key*).

PSS also has limited support for fixed-sized arrays of action handles, **component**s, and flow and resource object references, as described in 8.8.2. These are not considered plain-data types. All other collections are plain-data types.

8.8.1 DSL syntax

```
collection_type ::=
     array < data_type , array_size_expression >
   | list < data_type >
   | map < data_type , data_type >
   | set < data_type >
array_size_expression ::= constant_expression
```

Syntax 28—DSL: Collection data types

In an **array**, each element is initialized to the default initial value of the element type, unless the **array** declaration contains an initialization assignment. A **list**, **map** or **set** is initialized as an empty collection unless the declaration contains an initialization assignment. A collection that is empty is as if it was assigned an *empty aggregate literal* ({}). See 4.8 for more information on literal syntax and semantics used to initialize collection types.

Collection store both scalar and aggregate elements by value. This means that an element's value is captured when it is added or assigned to a collection. Modifying the value of an element in a collection does not modify the element originally added to the collection. In the example below, `v1`, a **struct** with two integer values, is assigned as the first element of `my_list`. Modifying `a` in that element does not modify `v1`. (See 8.8.3 for more details on **list** operators and methods.)

```
struct my_s1 {
  int a, b;
}

struct my_s2 {
  list<my_s1> my_list;

  exec pre_solve {
    my_s1 v1 = {.a=1,.b=2};
    my_list.push_back(v1);
    my_list[0].a = 10; // my_list == {{.a=10,.b=2}}, v1 == {.a=1,.b=2}
  }
}
```

Example 14—DSL: Modifying collection contents

Collection variables can be operated on with built-in operators using standard operator symbols (e.g., **[]**, **=**, **==**, etc.) or with built-in methods using a method name and an argument list in parentheses.

Operators and methods that modify the contents of a collection shall not be used in activities, constraints, or **covergroup**s. These are allowed only in **exec** blocks (see 22.1) and native functions (see 22.3). Operators and methods that do not modify collection contents may be used in activities, constraints, and **covergroup**s.

Arrays of randomizable types are randomizable. **List**s, **map**s and **set**s are non-randomizable. It is legal to have a **rand** struct field that contains non-randomizable collection types.

Collection types may be nested to describe more complex collections.

```
struct my_s {
  list<map<string, int>> m_list_of_maps;
  map<string, list<int>> m_map_of_lists;
}
```

Example 15—DSL: Nested collection types

8.8.2 Arrays

PSS supports fixed-sized arrays of plain-data types. Arrays may be declared with two different syntaxes, the classical syntax where arrays are declared by adding square brackets with the array size (**[** *constant_expression* **]**) after the array name, referred to as the *square array* syntax, and the syntax that is aligned to the other collection types, using angle brackets, referred to as the *template array* syntax.

```
int my_int_arr1[20];        // Square array declaration syntax
array<int,20> my_int_arr2;  // Template array declaration syntax
```

Example 16—DSL: Array declarations

The same operators and methods may be applied to arrays declared using both syntaxes. However, the template array syntax may be used where a *data_type* is required, enabling such capabilities as use as a function return type, nested array types, and more.

An array with `N` elements, is ordered, with the first element accessed using **`0`** as an index value with the **[]** operator, and the last element accessed using `N-1` as an index value.

The square array syntax can also be used to declare fixed-size arrays of *action handles*, **components**, and *flow and resource object references*. Individual elements of such arrays may be accessed using the **[]** operator. However, other operators and methods do not apply to these arrays, unless otherwise specified. Action handle arrays are described in 13.3.1.1 and 13.3.2, component arrays are described in 10.2 and 10.5, and object reference arrays are described in 14.4 and 15.2. Note that the elements of action handle arrays and object reference arrays have reference semantics (see 8.9).

8.8.2.1 Array operators

The following operators are defined for **array**s:

Index operator **[]**

Used to access a specific element of an array, given an index into the array. The index shall be an integral value. See 9.6.2 for more information on the index operator.

Assignment operator =

Creates a copy of the array-type expression on the RHS and assigns it to the array on the LHS. See 9.3 for more information on the assignment operator.

Equality operator ==

Evaluates to *true* if all elements with corresponding indexes are equal. Two arrays of different element types or different sizes are incomparable. See 9.5.3 for more information on the equality operator.

Inequality operator **!=**

Evaluates to *true* if not all elements with corresponding indexes are equal. Two arrays of different element types or different sizes are incomparable. See 9.5.3 for more information on the inequality operator.

Set membership operator **in**

The set membership operator can be applied to an array to check whether a specific element is currently within the array. It evaluates to *true* if the element specified on the left of the operator exists in the **array** collection on the right of the operator. The type of the element shall be the same as the array's element data type. See 9.5.9 for more information on the set membership operator.

foreach *statement*

The **foreach** statement can be applied to an array to iterate over the array elements within an activity, a constraint or native exec code. See 13.4.3, 17.1.7, and 22.7.8, respectively, for more information on the **foreach** statements in these contexts.

8.8.2.2 Array methods

The following methods are defined for **array**s:

function int *size()***;**

Returns the number of elements in the array. Since arrays have fixed sizes, the returned value is considered a constant expression.

function int *sum()***;**

Returns the sum of all elements currently stored in the array. This function can only be used on arrays of a numeric data type (**int** or **bit**). The method can be used in a constraint to constrain an array of random elements to have a sum of a certain value.

function list<*data_type*> *to_list()***;**

Returns a **list** containing the elements of the array. The **list**'s element data type is the same as the data type of the array elements. The **list** elements are ordered in the same order as the array.

function set<*data_type*> *to_set()***;**

Returns a **set** containing the elements of the array. Each element value will appear once. The **set**'s element data type is the same as the data type of the array elements. The **set** is unordered.

8.8.2.3 C++ syntax

The corresponding C++ syntax for arrays is shown in Syntax 29 and Syntax 30.

pss::attr_vec

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

```
template <class T> using vec = std::vector <T>;
template <class T> using attr_vec = attr<vec<T>>;
```

Declare array of non-random attributes.

Member functions

`attr_vec(const scope& name, const std::size_t count)` : constructor
`attr_vec(const scope& name, const std::size_t count, const width& a_width)` : constructor, with element width (T = int or bit only)
`attr_vec(const scope& name, const std::size_t count, const range& a_range)` : constructor, with element range (T = int or bit only)
`attr_vec(const scope& name, const std::size_t count, const width& a_width, const range& a_range)` : constructor, with element width and range (T = int or bit only)
`attr<T>& operator[](const std::size_t idx)` : access to a specific element
`std::size_t size()` : get size of array
`detail::AlgebExpr operator[](const detail::AlgebExpr& idx)` : constrain an element
`detail::AlgebExpr sum()` : constrain sum of array (T = int or bit only)

Syntax 29—C++: Arrays of non-random attributes

pss::rand_attr_vec

Defined in **`pss/rand_attr.h`** (see C.40).

```
template <class T> using vec = std::vector <T>;
template <class T> using rand_attr_vec = rand_attr< vec <T> >;
```

Declare array of random attributes.

Member functions

`rand_attr_vec(const scope& name, const std::size_t count)` : constructor
`rand_attr_vec(const scope& name, const std::size_t count, const width& a_width)` : constructor, with element width (T = int or bit only)
`rand_attr_vec(const scope& name, const std::size_t count, const range& a_range)` : constructor, with element range (T = int or bit only)
`rand_attr_vec(const scope& name, const std::size_t count, const width& a_width, const range& a_range)` : constructor, with element width and range (T = int or bit only)
`rand_attr<T>& operator[](const std::size_t idx)` : access to a specific element
`std::size_t size()` : get size of array
`detail::AlgebExpr operator[](const detail::AlgebExpr& idx)` : constrain an element
`detail::AlgebExpr sum()` : constrain sum of array (T = int or bit only)

Syntax 30—C++: Arrays of random attributes

NOTE—PSS/C++ does not support array initialization or array methods except **`size()`** and **`sum()`**.

8.8.2.4 Examples

Examples of fixed-size array declarations are shown in Example 17 and Example 18.

```
int fixed_sized_arr [16];         // array of 16 signed integers
array<bit[7:0],256> byte_arr;     // array of 256 bytes
array<route,8>      east_routes;  // array of 8 route structs
```

Example 17—DSL: Fixed-size arrays

In Example 17, individual elements of the `east_routes` array are accessed using the index operator **[]**, i.e., `east_routes[0]`, `east_routes[1]`,….

```
// array of 16 signed integers
attr_vec <int>   fixed_sized_arr {"fixed_size_arr", 16};
// array of 256 bytes
attr_vec <bit>   byte_arr        {"byte_arr", 256, width(7,0)};
// array of 8 route structs
attr_vec <route> east_routes     {"east_routes", 8};
```

Example 18—C++: Fixed-size arrays

In C++, individual elements of the array are accessed using the index operator **[]**, as in DSL.

The following example shows use of array operators and methods. In this example, action type `A` is traversed six times, once for each element in `foo_arr`, and once more since `foo_arr[0]` is greater than `3`.

```
    component pss_top {
      array<bit[15:0],5> foo_arr;
      set  <bit[15:0]>   foo_set;

      exec init_up {
        foo_arr = {1, 2, 3, 4, 4};    // Array initialization assignment
        foo_arr[0] = 5;               // Use of [] to select an array element
        foo_set = foo_arr.to_set();   // Use of to_set() method
      }

      action A{ rand bit[15:0] x; }
      action B{}
      action C{}

      action traverse_array_a {

        // foo_arr has 5 elements and foo_set has 4
        rand int in [1..] y;
        constraint y < comp.foo_arr.size(); // Use of size() method in constraint

        activity {
          foreach (elem: comp.foo_arr)           // "foreach" used on an array
            do A with { x == elem; };

            if (comp.foo_arr[0] > 3)
              do A;
            else if (4 in comp.foo_arr)          // Use of "in" operator
              do B;
            else if (comp.foo_arr.size() < 4)    // Use of size() method
              do C;
        }
      }
    }
```

Example 19—DSL: Array operators and methods

8.8.2.5 Array properties

Arrays provide the properties **`size`** and **`sum`**, which may be used in expressions. These properties are deprecated and have matching methods that should be used instead. They are used as follows:

```
int data[4];
... data.size ... // same as data.size()
... data.sum  ... // same as data.sum()
```

8.8.3 Lists

The **list** collection type is used to declare a variable-sized ordered list of elements. Using an index, an element in the list can be assigned or used in an expression. A list with `N` elements, is ordered, with the first element accessed using **`0`** as an index value with the **[]** operator, and the last element accessed using `N-1` as an index value.

A **list** is initialized as an empty collection unless the declaration contains an initialization assignment. A **list** that is empty is as if it was assigned an *empty aggregate literal* ({}). **List** elements can be added or removed in **exec** blocks; therefore the size of a list is not fixed like an array.

A **list** declaration consists of the keyword **list**, followed by the data type of the **list** elements between angle brackets, followed by the name(s) of the **list**(s). **List**s are non-randomizable.

```
struct my_s {
  list<int> my_list;
}
```

Example 20—DSL: Declaring a list in a struct

8.8.3.1 List operators

The following operators are defined for **list**s:

Index operator **[]**

Used to access a specific element of a **list**, given an index into the **list**. The index shall be an integral value. See 9.6.2 for more information on the index operator.

Assignment operator =

Creates a copy of the list-type expression on the RHS and assigns it to the **list** on the LHS. See 9.3 for more information on the assignment operator.

Equality operator ==

Evaluates to *true* if the two **list**s are the same size and all elements with corresponding indexes are equal. Two **list**s of different element types are incomparable. See 9.5.3 for more information on the equality operator.

Inequality operator **!=**

Evaluates to *true* if the two **list**s are not the same size or not all elements with corresponding indexes are equal. Two **list**s of different element types are incomparable. See 9.5.3 for more information on the inequality operator.

Set membership operator **in**

The set membership operator can be applied to a **list** to check whether a specific element is currently in the **list**. It evaluates to *true* if the element specified on the left of the operator exists in the **list** collection on the right of the operator. The type of the element shall be the same as the **list**'s element data type. See 9.5.9 for more information on the set membership operator.

foreach *statement*

The **foreach** statement can be applied to a **list** to iterate over the **list** elements within an activity, a constraint or native exec code. See 13.4.3, 17.1.7, and 22.7.8, respectively, for more information on the **foreach** statements in these contexts.

8.8.3.2 List methods

The following methods are defined for **list**s:

function int *size()***;**

Returns the number of elements in the **list**.

function void *clear()***;**

Removes all elements from the **list**.

function *data_type delete(***int** *index)***;**

Removes an element at the specified index of type integer and returns the element value. The return value data type is the same as the data type of the **list** elements. If the index is out of bounds, the operation is illegal.

function void *insert(***int** *index, data_type element)***;**

Adds an element to the **list** at the specified index of type integer. If the index is equal to the size of the **list**, *insert* is equivalent to *push_back()*. If the index is less than the size of the **list**, then elements at and beyond the index are moved by one. If the index is greater than the size of the **list**, the operation is illegal. The inserted element's data type shall be the same as the data type of the **list** elements.

function *data_type pop_front()***;**

Removes the first element of the **list** and returns the element value. This is equivalent to *delete(0)*.

function void *push_front(data_type element)***;**

Inserts an element at the beginning of the **list**. This is equivalent to *insert(0, element)*.

function *data_type pop_back()***;**

Removes the last element of the **list** and returns the element value. This is equivalent to *delete(size()-1)*.

function void *push_back(data_type element)***;**

Appends an element to the end of the **list**. This is equivalent to *insert(size(), element)*.

function set<*data_type*> *to_set()***;**

Returns a **set** containing the elements of the **list**. Each element value will appear once. The **set**'s element data type is the same as the data type of the **list** elements. The **set** is unordered.

8.8.3.3 Examples

The following example shows use of **list** operators and methods. In this example, an action of type B will be traversed six times. There are six elements in foo_list3, foo_list2[0] is 1 and 4 is in comp.foo_list1. Action A and action C are never traversed.

```
component pss_top {
  list<bit[15:0]> foo_list1, foo_list2;

  exec init_up {
    foo_list1 = {1, 2, 3, 4};  // List initialization with aggregate literal
    foo_list2.push_back(1);   // List initialization with push_back
    foo_list2.push_back(4);
  }

  action A{}
  action B{}
  action C{}

  action traverse_list_a {
    list <bit[15:0]> foo_list3;
    bit[15:0] deleted;

    exec pre_solve {
      foo_list3 = pss_top.foo_list1; // foo_list3 = {1, 2, 3, 4}
      foo_list3.push_front(0);       // foo_list3 = {0, 1, 2, 3, 4}
      foo_list3.push_back(5);        // foo_list3 = {0, 1, 2, 3, 4, 5}
      foo_list3.insert(0, 1);        // foo_list3 = {1, 0, 1, 2, 3, 4, 5}
      foo_list3[0] = 6;              // foo_list3 = {6, 0, 1, 2, 3, 4, 5}
      deleted = foo_list3.delete(0); // foo_list3 = {0, 1, 2, 3, 4, 5}
    }

    activity {
      if (comp.foo_list1 == comp.foo_list2) // Use of == operator on list
        do A;
      else foreach (e: foo_list3)           // Use of "foreach" on list
        if (comp.foo_list2[0] > 3)          // Use of [] operator on list
          do A;
        else if (4 in comp.foo_list1)       // Use of "in" operator on list
          do B;
        else
          do C;
    }

    exec post_solve {
      foo_list3.clear();                    // foo_list3 = {}
    }
  }
}
```

Example 21—DSL: List operators and methods

8.8.4 Maps

The **map** collection type is used to declare a variable-sized associative array that associates a *key* with an element (or *value*). The keys serve as indexes into the **map** collection. Using a key, an element in the **map** can be assigned or used in an expression. A **map** is unordered.

A **map** is initialized as an empty collection unless the declaration contains an initialization assignment. A **map** that is empty is as if it was assigned an *empty aggregate literal* (**{}**). **Map** elements can be added or removed within **exec** blocks.

A **map** declaration consists of the keyword **map**, followed by the data type of the **map** keys and the data type of **map** elements, between angle brackets, followed by the name(s) of the **map**(s). Both keys and element values may be of any plain-data type. **Maps** are non-randomizable.

```
struct my_s {
  map<int, string> my_map;
}
```

Example 22—DSL: Declaring a map in a struct

8.8.4.1 Map operators

The following operators are defined for **map**s:

Index operator **[]**

Used to access a specific element of a **map**, given a key of the specified data type. When used on the LHS in an assignment, the index operator sets the element value associated with the specified key. If the key already exists, the current value associated with the key is replaced with the value of the expression on the RHS. If the key does not exist, then a new key is added to the **map** collection and the value of the expression on the RHS is assigned to the new key's associated **map** entry. Use of a key that does not exist in the **map** to reference an element in the **map** is illegal. See 9.6.2 for more information on the index operator.

Assignment operator **=**

Creates a copy of the map-type expression on the RHS and assigns it to the **map** on the LHS. If the same key appears more than once in the expression on the RHS, the last value specified is used. See 9.3 for more information on the assignment operator.

Equality operator **==**

Evaluates to *true* if the two **map**s are the same size, have the same set of keys, and all elements with corresponding keys are equal. Two **map**s of different key or element types are incomparable. See 9.5.3 for more information on the equality operator.

Inequality operator **!=**

Evaluates to *true* if the two **map**s are not the same size, do not have the same set of keys, or not all elements with corresponding keys are equal. Two **map**s of different key or element types are incomparable. See 9.5.3 for more information on the inequality operator.

foreach *statement*

The **foreach** statement can be applied to a **map** to iterate over the **map** elements within an activity, a constraint or native exec code. See 13.4.3, 17.1.7, and 22.7.8, respectively, for more information on the **foreach** statements in these contexts.

The set membership operator (**in**) cannot be applied directly to a map. However, it may be applied to the set of keys or the list of values produced by the *keys()* and *values()* methods, respectively, described below.

8.8.4.2 Map methods

The following methods are defined for **map**s:

function int *size()***;**

Returns the number of elements in the map.

function void *clear()***;**

Removes all elements from the map.

function *data_type delete(data_type key)***;**

Removes the element associated with the specified key from the **map** and returns the element value. The return value data type is the same as the data type of the **map** elements. The key argument shall have the same type as specified in the **map** declaration. If the specified key does not exist in the **map**, the operation is illegal.

function void *insert(data_type key, data_type value)***;**

Adds the specified key/value pair to the **map**. If the key currently exists in the **map**, then the current value is replaced with the new value. The arguments shall have the same types as specified in the **map** declaration.

function set<*data_type*> *keys()***;**

Returns a **set** containing the **map** keys. The **set**'s element data type is the same as the data type of the map keys. Since each key is unique and no order is defined on the keys, the method returns a **set** collection.

function list<*data_type*> *values()***;**

Returns a **list** containing the **map** element values. The **list**'s element data type is the same as the data type of the map elements. Since element values may not be unique, the method returns a **list** collection. However, the order of the **list** elements is unspecified.

8.8.4.3 Example

The following example shows use of map operators and methods. In this example, an action of type `B` will be traversed four times: `foo_map1` is not equal to `foo_map2`, `foo_map3` has four elements, `foo_map2["a"]` is `1` which is not greater than `3`, and `"b"` exists in `foo_map1`.

```
component pss_top {
  map<string, bit[15:0]> foo_map1, foo_map2;
  list<bit[15:0]>        foo_list1;

  exec init_up {
    foo_map1 = {"a":1,"b":2,"c":3,"d":4};  // Map initialization
                                           // with key/value literal
    foo_map2["a"] = 1;
    foo_map2["b"] = 4;
    foo_list1 = foo_map1.values();
    foreach (foo_map2[i]) foo_list1.push_back(foo_map2[i]);
  }

  action A{}
  action B{}
  action C{}

  action traverse_map_a {
    rand int lower_size;
    map <string, bit[15:0]> foo_map3;
    set <string> foo_set1;

    exec pre_solve {
      foo_map3 = pss_top.foo_map1; // foo_map3 = {"a":1,"b":2,"c":3,"d":4}
      foo_map3.insert("z",0); // foo_map3 = {"a":1,"b":2,"c":3,"d":4,"z":0}
      foo_map3.insert("d",5); // foo_map3 = {"a":1,"b":2,"c":3,"d":5,"z":0}
      foo_map3.delete(“d”);   // foo_map3 = {"a":1,"b":2,"c":3,"z":0}
      foo_set1 = foo_map3.keys();
    }
    constraint lower_size < comp.foo_map3.size() + comp.foo_list1.size();
    activity {
      if (comp.foo_map1 == comp.foo_map2)  // Use of == operator on maps
         do A;
      else foreach (foo_map3.values()[i])  // Use of "foreach" on a map
                                           // converted to a list of values
         if (comp.foo_map2["a"] > 3)       // Usage of operator[] on a map
           do A;
         else if ("b" in comp.foo_map1.keys()) // Check whether a key
                                               // is in the map
           do B;
         else
           do C;
    }
    exec post_solve {
      foo_map3.clear();                       // foo_map3 = {}
    }
  }
}
```

Example 23—DSL: Map operators and methods

8.8.5 Sets

The **set** collection type is used to declare a variable-sized unordered set of unique elements of plain-data type. Sets can be created, modified, and queried using the operators and methods described below.

A **set** is initialized as an empty collection unless the declaration contains an initialization assignment. A **set** that is empty is as if it was assigned an *empty aggregate literal* (**{}**). **Set** elements can be added or removed within **exec** blocks; therefore the size of a list is not fixed like an array.

A **set** declaration consists of the keyword **set**, followed by the data type of the **set** elements between angle brackets, followed by the name(s) of the **set**(s). **Set**s are non-randomizable.

```
    struct my_s {
      set<int> my_set;
    }
```

Example 24—DSL: Declaring a set in a struct

8.8.5.1 Set operators

The following operators are defined for **set**s:

Assignment operator =

Creates a copy of the set-type expression on the RHS and assigns it to the **set** on the LHS. The same value may appear more than once in the expression on the RHS, but it will appear only once in the **set**. See 9.3 for more information on the assignment operator.

Equality operator ==

Evaluates to *true* if the two **set**s have exactly the same elements. Note that sets are unordered. Two **set**s of different element types are incomparable. See 9.5.3 for more information on the equality operator.

Inequality operator **!=**

Evaluates to *true* if the two **set**s do not have exactly the same elements. Two **set**s of different element types are incomparable. See 9.5.3 for more information on the inequality operator.

Set membership operator **in**

The set membership operator can be applied to a **set** to check whether a specific element is currently within the **set**. It evaluates to *true* if the element specified on the left of the operator exists in the **set** collection on the right of the operator. The type of the element shall be the same as the **set**'s element data type. See 9.5.9 for more information on the set membership operator.

foreach *statement*

The **foreach** statement can be applied to a **set** to iterate over the **set** elements within an activity, a constraint or native exec code. When applied to a set, the **foreach** statement shall specify an *iterator variable* and shall not specify an *index variable*. See 13.4.3, 17.1.7, and 22.7.8, respectively, for more information on the **foreach** statements in these contexts.

8.8.5.2 Set methods

The following methods are defined for **set**s:

function int *size()***;**

Returns the number of elements in the **set**.

function void *clear()***;**

Removes all elements from the **set**.

function void *delete(data_type element)***;**

Removes the specified element from the **set**. The element argument data type shall be the same as the data type of the **set** elements. If the element does not exist in the **set**, the operation is illegal.

function void *insert(data_type element)***;**

Adds the specified element to the **set**. The inserted element's data type shall be the same as the data type of the **set** elements. If the element already exists in the **set**, the method shall have no effect.

function list<*data_type*> *to_list()***;**

Returns a **list** containing the elements of the **set** in an arbitrary order. The **list**'s element data type is the same as the data type of the **set** elements.

8.8.5.3 Examples

The following example shows use of **set** operators and methods. In this example, A is traversed two times and B is traversed three times: foo_set1 is not equal to foo_set2, there are five elements in foo_set3, two of the foo_set3 elements are in foo_set2, and "b" is in foo_set1.

```
component pss_top {
  set <string> foo_set1, foo_set2;
  list<string> foo_list1;

  exec init_up {
    foo_set1 = {"a","b","c","d"}; // Set initialization with aggregate literal
    foo_set2.insert("a");
    foo_set2.insert("b");
    foo_list1 = foo_set1.to_list();
    foreach (e:foo_set2) foo_list1.push_back(e);
  }

  action A{}
  action B{}
  action C{rand string character;}

  action traverse_set_a {
    rand int lower_size;
    set <string> foo_set3;
    list<string> foo_list2;

    exec pre_solve {
      foo_set3 = pss_top.foo_set1;
      foo_set3.insert("z");
      foo_set3.insert("e");
      foo_set3.delete("d");
      foo_list2 = foo_set3.to_list();
    }

    constraint lower_size < foo_set3.size() + comp.foo_list1.size();

    activity {
      if (comp.foo_set1 == comp.foo_set2)   // Use == operator on sets
        do A;
      else foreach (e:foo_set3)             // Use "foreach" on set
        if (e in comp.foo_set2)             // Use [] operator on set
          do A;
        else if ("b" in comp.foo_set1)      // Use "in" operator on set
          do B;
        else
          replicate (j:foo_list2.size())
            do C with {character == foo_list2[j];};
    }
  }
}
```

Example 25—DSL: Set operators and methods

8.9 Reference types

PSS supports a limited form of *reference types* for actions, components, and flow/resource objects, but does not support references to plain-data types. *References* in PSS are similar in their semantics to class variables in such languages as Java and SystemVerilog. Variables of reference types can be assigned and compared (see more in 9.3 and 9.5.3).

8.9.1 DSL syntax

```
reference_type ::= ref entity_type_identifier
entity_type_identifier ::=
     action_type_identifier
   | component_type_identifier
   | flow_object_type
   | resource_object_type
null_ref ::= null
```

Syntax 31—DSL: ref declaration

The following also apply:

a) The **ref** modifier can be used in the declaration of local variables, fields of components, function parameters, and function return values. The **ref** modifier shall not be used in the declaration of fields in the scope of actions, flow/resource objects, and structs. Nor shall it be used to declare static constants or the key or element type in collections.
b) Fields and instance functions can be accessed through a reference expression in the same way as through an instance path, using the dot ('.') operator.
c) An expression of reference type may evaluate to the special value **null**, indicating that it does not reference any entity. It shall be an error to access members of an entity through a null reference. See also 9.3 and 9.5.3.
d) When not initialized, the default value of a reference variable is **null**.

Note that PSS supports special reference fields that are automatically resolved as part of the solving process. They are:

— The context component reference **`comp`** (see 10.6)
— Action handles to sub-actions within compound actions (see 13.3.1.1)
— The previous state reference **`prev`** (see 14.3.1)
— Input and output reference fields of actions (see 14.4)
— Resource claim reference fields (see 15.2)

8.9.2 Examples

Example 26 demonstrates the use of a reference as a local variable and as a return type of a function. In the body of **action** `call_foo`, a reference to `A` is stored in a local variable, and then used to call function `foo()`. In addition, a reference to `A` is returned from function `choose_A()`, and it is used in turn to call `foo()` on the chosen instance of `A`.

```
component A {
   function void foo();
};

component B {
   A a_arr[5];

   function ref A choose_A(int code) {
         return a_arr[code % 5];
   }

   action call_foo {
      exec body {
         ref A aref = comp.a_arr[3];
         aref.foo();
         comp.choose_A(123).foo();
      }
   };
};
```

Example 26—DSL: Use of reference as local variable and function return value

In Example 27, a reference field is declared under **component** `my_comp`. After the construction of the component instance tree, the attribute `sibling_size` of `c2` is equal to `10`, having been assigned in the **init_down** block through the `sibling` reference field. However, the attribute `sibling_size` of `c1` is still equal to its default value 0, because for `c1`, reference field `sibling` was not initialized, and therefore `c1.sibling` is equal to **null**.

```
component my_comp {
   ref my_comp sibling;
   int size, sibling_size;

   exec init_down {
      if (sibling != null) {
         sibling_size = sibling.size;
      }
   }
};

component pss_top {
   my_comp c1, c2;
   exec init_down {
      c1.size = 10;
      c2.sibling = c1;
   }
};
```

Example 27—DSL: Use of reference field and null value

8.10 User-defined data types

The **typedef** statement declares a user-defined type name in terms of an existing data type, as shown in Syntax 32.

8.10.1 DSL syntax

```
typedef_declaration ::= typedef data_type identifier ;
```

Syntax 32—DSL: User-defined type declaration

8.10.2 C++ syntax

C++ uses the built-in **`typedef`** construct.

8.10.3 Examples

typedef examples are shown in Example 28 and Example 29.

```
typedef bit[31:0] uint32_t;
```

Example 28—DSL: typedef

```
typedef unsigned int uint32_t;
```

Example 29—C++: typedef

8.11 Data type conversion

Expressions of types **int**, **bit**, **bool**, or **enum** in DSL can be changed to another type in this list by using a *cast operator*. C++ casting is handled using the existing C++ mechanism.

8.11.1 DSL syntax

Syntax 33 defines the *cast operator*.

```
cast_expression ::= ( casting_type ) expression
casting_type ::=
    integer_type
  | bool_type
  | enum_type
  | type_identifier
```

Syntax 33—DSL: cast operation

In a *cast_expression*, the *expression* to be cast shall be preceded by the casting data type enclosed in parentheses. The *cast* shall return the value of the *expression* represented as the *casting_type*. A *type_identifier* specified as a *casting_type* shall refer to an integer, boolean, or enumeration type.

The following also apply:

a) Any non-zero value *cast* to a **bool** type shall evaluate to *true*. A zero value cast to a **bool** type shall evaluate to *false*. When casting a **bool** type to another type, *false* evaluates to **0** and *true* evaluates to **1**.

b) When casting a value to a **bit** type, the *casting_type* shall include the width specification of the resulting bit vector. The *expression* shall be converted to a bit vector of sufficient width to hold the value of the *expression*, and then truncated or left-zero-padded as necessary to match the *casting_type*.

c) When casting a value to a user-defined **enum** type, the value shall correspond to a valid integral value for the resulting **enum** type. When used in a constraint, the resulting domain is the intersection of the value sets of the two **enum** types.

d) All numeric expressions (**int** and **bit** types) are type-compatible, so an explicit *cast* is not required from one to another.

8.11.2 Examples

Example 30 shows the overlap of possible **enum** values (from 8.11.1 (c)) when used in constraints.

```
enum config_modes_e {UNKNOWN, MODE_A=10, MODE_B=20};
enum foo_e {A=10, B, C};

action my_a {
   rand config_modes_e cfg;
   rand foo_e foo;
   constraint cfg == (config_modes_e)11;  // illegal
   constraint cfg == (config_modes_e)foo;
                                 // cfg==MODE_A, the only value in the
                                 // numeric domain of both cfg and foo
   ...
}
```

Example 30—DSL: Overlap of possible enum values

Example 31 shows the casting of `al` from the `align_e` enum type to a 4-bit vector to pass into the `alloc_addr` imported function.

```
package external_fn_pkg {
   enum align_e {byte_aligned=1, short_aligned=2, word_aligned=4};
   function bit[31:0] alloc_addr(bit[31:0] size, bit[3:0] align);
   buffer mem_seg_s {
      rand bit[31:0]  size;
      bit[31:0] addr;
      align_e al;
      exec post_solve {
         addr = alloc_addr(size, (bit[3:0])al);
      }
   }
}
```

Example 31—DSL: Casting of variable to a bit vector

9. Operators and expressions

This section describes the operators and operands available in PSS and how to use them to form expressions.

An *expression* is a construct that can be evaluated to determine a specific value. Expressions may be *primary expressions*, consisting of a single term, or *compound expressions*, combining *operators* with sub-expressions as their *operands.*

The various types of primary expressions are specified in 9.6.

9.1 DSL syntax

```
expression ::=
     primary
   | unary_operator primary
   | expression binary_operator expression
   | conditional_expression
   | in_expression
unary_operator ::= - | ! | ~ | & | | | ^
binary_operator ::= * | / | % | + | - | << | >> | == | != | < | <= | > | >= | || | && | | | ^ | & | **
assign_op ::= = | += | -= | <<= | >>= | |= | &=
primary ::=
     number
   | aggregate_literal
   | bool_literal
   | string_literal
   | null_ref
   | paren_expr
   | cast_expression
   | ref_path
   | compile_has_expr
paren_expr ::= ( expression )
cast_expression ::= ( casting_type ) expression
```

Syntax 34—DSL: Expressions and operators

9.2 Constant expressions

Some constructs require an expression to be a *constant expression*. The operands of a constant expression consist of numeric and string literals, aggregate literals with constant values, named constants (e.g., **static const**, template parameters), bit-selects and part-selects of named constants, enum items, and calls of **pure** functions with constant arguments.

9.3 Assignment operators

The assignment operators defined by the PSS language are listed in the table below.

Table 6—Assignment operators and data types

Operator token	Operator name	Operand data types
`=`	Binary assignment operator	Any plain-data type or reference type
`+=` `-=`	Binary arithmetic assignment operators	Numeric
`&=` `|=`	Binary bitwise assignment operators	Numeric
`>>=` `<<=`	Binary shift assignment operators	Numeric

The *assignment* (**`=`**) operator is used in the context of attribute initializers and procedural statements.

The *arithmetic assignment* (**`+=`**, **`-=`**), *shift assignment* (**`<<=`**, **`>>=`**), and *bitwise assignment* (**`|=`**, **`&=`**) operators are used in the context of procedural statements. These compound assignment operators are equivalent to assigning to the left-hand operand the result of applying the leading operator to the left-hand and right-hand operands. For example, **`a <<= b`** is equivalent to **`a = a << b`**.

While these operators may not be used as a part of an expression, they are documented here for consistency.

The type of the right-hand side of an assignment shall be assignment-compatible with the type of the left-hand side. In an aggregate assignment, assignment is performed element by element. In an assignment of a fixed-size array, the left-hand and right-hand sides of the assignment shall have the same size.

In assignment of reference types, the right-hand side shall be one of the following:
- A reference expression of the same type as the left-hand side or a derived type of it
- An instance path to a component of the same type as the left-hand side or a derived type of it
- The value **null**

Following the assignment of a reference, the left-hand side variable shall point to (be an alias to) the same entity (component, action, flow/resource object) referred to by the right-hand side (or have the value **null** in case the right-hand side evaluates to **null**).

9.4 Expression operators

The expression operators defined by the PSS language are listed in the table below.

Table 7—Expression operators and data types

Operator token	Operator name	Operand data types	Result data type
`?:`	Conditional operator	Any plain-data type or reference type (condition is Boolean)	Same as operands
`-`	Unary arithmetic negation operator	Numeric	Same as operand
`~`	Unary bitwise negation operator	Numeric	Same as operand
`!`	Unary Boolean negation operator	Boolean	Boolean

Table 7—Expression operators and data types

Operator token	Operator name	Operand data types	Result data type
& | ^	Unary bitwise reduction operators	Numeric	1-bit
+ - * / % **	Binary arithmetic operators	Numeric	1-bit
& | ^	Binary bitwise operators	Numeric	1-bit
>> <<	Binary shift operators	Numeric	Same as left operand
&& ||	Binary Boolean logical operators	Boolean	Same as operands
< <= > >=	Binary relational operators	Numeric	Boolean
== !=	Binary logical equality operators	Any plain-data type or reference type	Boolean
cast	Data type conversion operator	Numeric, Boolean, enum	Casting type
in	Binary set membership operator	Any plain-data type	Boolean
[*expression*]	Index operator	Array, list, map	Same as element of collection
[*expression*]	Bit-select operators	Numeric	Numeric
[*expression*: *expression*]	Part-select operator	Numeric	Numeric

9.4.1 Operator precedence and associativity

Operator precedence and associativity are listed in Table 8. The highest precedence is listed first.

Table 8—Operator precedence and associativity

Operator	Associativity	Precedence
() []	Left	1 (Highest)
cast	Right	2
- ! ~ & | ^ (unary)		2
**	Left	3
* / %	Left	4
+ - (binary)	Left	5
<< >>	Left	6
< <= > >= in	Left	7
== !=	Left	8
& (binary)	Left	9
^ (binary)	Left	10
| (binary)	Left	11

Table 8—Operator precedence and associativity

`&&`	Left	12
`||`	Left	13
`?: (conditional operator)`	Right	14 (Lowest)

Operators shown in the same row in the table shall have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *****, **/**, and **%** all have the same precedence, which is higher than that of the binary **+** and **–** operators.

All operators shall associate left to right with the exception of the conditional (**?:**) and cast operators, which shall associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example, `B` is added to `A`, and then `C` is subtracted from the result of `A+B`.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence shall associate first. In the following example, `B` is divided by `C` (division has higher precedence than addition), and then the result is added to `A`.

```
A + B / C
```

Parentheses can be used to change the operator precedence, as shown below.

```
(A + B) / C              // not the same as A + B / C
```

9.4.2 Using aggregate literals in expressions

Aggregate literals (i.e., value list, map, and structure literals, see 4.8) can be used as expression operands. For example, aggregate literals can be used to initialize the contents of aggregate types as part of a variable declaration, in constraint contexts, as foreign language function parameters, and as template-type value parameters. An aggregate literal may not be the target of an assignment.

When the operands of an assignment or equality operator are a structure aggregate literal and a **struct**-type variable, any elements not specified by the literal are given the default values of the data type of the element. When the operands of an assignment or equality operator are a value list literal and an array, the number of elements in the aggregate literal must be the same as the number of elements in the array.

In Example 32, a **struct** type is declared that has four integer fields. A non-random instance of that **struct** is created where all field values are explicitly specified. A constraint compares the fields of this **struct** with an aggregate literal in which only the first two struct fields are specified explicitly. Because a **struct** is a fixed-size data structure, the fields that are not explicitly specified in the aggregate literal are given default values– in this case 0. Consequently, the constraint holds.

```
struct s {
  int a, b, c, d;
};
struct t {
  s s1 = {.a=1,.b=2,.c=0,.d=0};
  constraint s1 == {.b=2,.a=1};
}
```

Example 32—DSL: Using a structure literal with an equality operator

When an aggregate literal is used in the context of a variable-sized data type, the aggregate literal specifies both size and content.

In Example 33, a **set** variable is compared with an aggregate literal using a constraint. The size of the **set** variable is three, since there are three unique values in the initializing literal, while the size of the aggregate literal in the constraint is two. Consequently, the constraint does not hold.

```
struct t {
  set<int>   s =  {1, 2, 0, 0};
  constraint s == {1, 2}; // False: s has 3 elements, but the literal has 2
}
```

Example 33—DSL: Using an aggregate literal with a set

Values in aggregate literals may be non-constant expressions. Example 34 shows use of a **repeat**-loop index variable and a function call in a value list literal.

```
function int get_val(int idx);
import solve function get_val;
struct S {
  list<array<int,2>>  pair_l;

  exec pre_solve {
    repeat(i : 4) {
      array<int,2> pair = {i, get_val(i)};
      pair_l.push_back(pair);
    }
  }
}
```

Example 34—DSL: Using non-constant expressions in aggregate literals

9.4.3 Type inference rules

The *expected type* of an expression shall be inferred according to the rules below. The expected type is used in the resolution of unqualified *enum_item* names (see 8.4) and in the interpretation of aggregate literals (see 9.4.2).

— The type of the expression on the left-hand side of an assignment determines the expected type of the expression on the right-hand side. This includes initialization assignments.

— The type of the formal parameter of a **function** determines the expected type of the respective actual parameter expression (see 22.2). This is true also for **covergroup** instantiations (see 19.2).

— The return type of a **function** determines the expected type of the expression in its **return** statement (see 22.7.5).

— An expression of a known type on the left-hand side of an equality operator (**==**, **!=**) determines the expected type of the right-hand side (see 9.5.3).

— The expected type of a *conditional_expression* (**?:**) determines the expected type of the second and third operands of the expression (see 9.5.8).

— The type of the expression on the left-hand side of a set membership (**in**) operator determines the expected type of the expressions in the *open_range_list*, or the elements of the *collection_expression*, on the right-hand side (see 9.5.9).

— An explicit data type of a **coverpoint** determines the expected type of the coverpoint expression (see 19.3).

— The type (explicit or implicit) of a **coverpoint** determines the expected type of its bin values (see 19.3.4).

— In a *cast_expression*, the specified target type (*casting_type*) determines the expected type of the expression to be cast (see 8.11).

For the purposes of this section, all numeric types are considered to be a single type, as all numeric expressions are type-compatible (see 8.11). See more on the evaluation of numeric expressions in 9.7 and 9.8.

In Example 35, contextual typing is required to interpret structure literals. Based on the type of the left operand of an equality operator, the structure literal on the right-hand side is interpreted differently in two different constraints within the same **action**.

```
component my_ip_c {
  struct my_struct { rand int a; };
  action my_op {
    rand my_struct s;
  }
}

component pss_top {
  my_ip_c my_ip;
  struct your_struct { rand int a; };

  action test {
    rand your_struct s;
    constraint s == {.a = 2};    // pss_top::your_struct literal

    my_ip_c::my_op op;
    constraint op.s == {.a = 3}; // my_ip_c::my_struct literal

    activity {
      op;
    }
  }
}
```

Example 35—DSL: Contextual typing in structure literal interpretation

Example 36 shows two cases of unqualified *enum item* resolution based on contextual typing—an assignment and a function call. Note that in calling function `print_num()`, whose formal parameter is declared with type **int**, the identifier `ORANGE` cannot be resolved, because the expected type is an **int**. The *enum_item* must be qualified in this case.

```
enum color_e {RED, GREEN, ORANGE};

function void print_color(color_e c);
function void print_num(int n);

component pss_top {
  enum fruit_e {APPLE, ORANGE};

  exec init_down {
    color_e c = ORANGE;               // OK - expected type is color_e
    print_color(RED);                 // OK - same as above
    print_num((int)ORANGE);           // Error - 'ORANGE' unresolved -
                                      // no enum type expected here
    print_num((int)fruit_e::ORANGE); // OK - qualified reference
  }
}
```

Example 36—DSL: Contextual typing in enum_item resolution

9.4.4 Operator expression short-circuiting

The logical operators (**&&**, **||**) and the conditional operator (**?:**) shall use *short-circuit evaluation*. In other words, operand expressions that are not required to determine the final value of the operation shall not be evaluated. All other operators shall not use short-circuit evaluation. In other words, all of their operand expressions are always evaluated.

9.5 Operator descriptions

The following sections describe each of the operator categories. The legal operand types for each operator are listed in Table 7.

9.5.1 Arithmetic operators

The binary arithmetic operators are given in Table 9.

Table 9—Binary arithmetic operators

a + b	a plus b
a – b	a minus b
a * b	a multiplied by b (or a times b)
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

Integer division shall truncate the fractional part toward zero. The modulus operator (for example, `a % b`) gives the remainder when the first operand is divided by the second, and thus zero when `b` divides `a` exactly.

The result of a modulus operation shall take the sign of the first operand. Division or modulus by zero shall be considered illegal.

The result of the power operator is unspecified if the first operand is zero and the second operand is negative.

Table 10—Power operator rules

	op1 is < −1	**op1 is −1**	**op1 is 0**	**op1 is 1**	**op1 is > 1**
op2 is positive	op1 ** op2	op2 is odd -> −1 op2 is even -> 1	0	1	op1 ** op2
op2 is zero	1	1	1	1	1
op2 is negative	0	op2 is odd -> −1 op2 is even -> 1	undefined	1	0

NOTE—The power operator is not supported directly in PSS/C++. Instead, the function **`pow(`**`a,b`**`)`** implements `a` to the power of `b`.

The unary arithmetic negation operator (**-**) shall take precedence over the binary operators.

9.5.1.1 Arithmetic expressions with unsigned and signed types

bit-type variables are unsigned, while **int**-type variables are signed.

A value assigned to an unsigned variable shall be treated as an *unsigned* value. A value assigned to a signed variable shall be treated as *signed*. Signed values shall use two's-complement representation. Conversions between signed and unsigned values shall keep the same bit representation. Only the bit interpretation changes.

9.5.2 Relational operators

Table 11 lists and defines the relational operators. Relational operators may be applied only to numeric operands.

Table 11—Relational operators

a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

An expression using these *relational operators* shall yield the Boolean value *true* if the specified relation holds, or the Boolean value *false* if the specified relation does not hold.

When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.

When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand.

All the relational operators have the same precedence, and have lower precedence than arithmetic operators.

9.5.3 Equality operators

The *equality operators* rank lower in precedence than the relational operators. Table 12 defines the equality operators.

Table 12—Equality operators

a == b	a equal to b
a != b	a not equal to b

Both equality operators have the same precedence. When the operands are numeric, these operators compare operands bit for bit. As with the relational operators, the result shall be *false* if the comparison fails and *true* if it succeeds.

When one or both operands are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.

When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand.

When the operands of an equality operator are of **string** type, both the sizes and the values of the string operands are compared.

Aggregate data (**struct**s and collections) may be compared using equality operators. When the equality operators are applied to aggregate data, both operands shall be of the same type. Aggregate operands are compared element-by-element to assess equality.

The following rules apply to comparison of collections:

— It shall be illegal to compare two fixed-size arrays of different sizes. Variable-sized collections of the same type may be compared, but they shall be considered not equal if they have different sizes.

— Two fixed-size arrays are considered equal if they have the same elements in the same order.

— Two **list**s are considered equal if they have the same size and they have the same elements in the same order.

— Two **map**s are considered equal if they have the same size and the same *key-value* pairs, regardless of order (maps are unordered).

— Two **set**s are considered equal if they have the same size and the same elements, regardless of order (sets are unordered).

The right-hand side of an equality operator may be an aggregate literal of the same type as the left-hand side. The left-hand side of an equality operator may not be an aggregate literal. See more details about collections in 8.8 and about aggregate literals in 4.8 and 9.4.2.

References can be compared with equality operators. The operands may be one of the following:

— Two expressions of the same reference type, or one expression of a reference to a derived type of the other

— One expression of a component reference type, and the other an instance path to a component of the same type, or a derived type of it

— An expression of a reference type and the value **null**

The expression evaluates to *true* if both operands refer to the same entity (component, action, flow/resource object) or if both evaluate to **null**. Otherwise it evaluates to *false*. Note that these rules apply to variables declared with the **ref** modifier, the built-in **`comp`** reference, and other reference fields (see 8.9).

9.5.4 Logical operators

The binary operators *logical AND* (**`&&`**) and *logical OR* (`||`) are logical connective operators and have a Boolean result. The precedence of **`&&`** is greater than that of **`||`**, and both have a lower precedence than the relational and equality operators.

The unary *logical negation* operator (**`!`**) converts a *true* operand to *false* and a *false* operand to *true*.

In procedural contexts, the **`&&`** and **`||`** operators shall use short-circuit evaluation as follows:

— The first operand expression shall always be evaluated.

— For **`&&`**, if the first operand evaluates to *false*, then the second operand shall not be evaluated.

— For **`||`**, if the first operand evaluates to *true*, then the second operand shall not be evaluated.

9.5.5 Bitwise operators

The *bitwise operators* perform bitwise manipulations on the operands. Specifically, the binary bitwise operators combine a bit in one operand with the corresponding bit in the other operand to calculate one bit for the result. The following truth tables show the result for each operator and input operands.

Table 13—Bitwise binary AND operator

&	**0**	**1**
0	0	0
1	0	1

Table 14—Bitwise binary OR operator

|	**0**	**1**
0	0	1
1	1	1

Table 15—Bitwise binary XOR operator

^	**0**	**1**
0	0	1
1	1	0

The bitwise unary negation operator (~) negates each bit of a single operand.

Table 16—Bitwise unary negation operator

~	
0	1
1	0

These operators may be applied only to numeric operands.

9.5.6 Reduction operators

The *unary reduction operators* perform bitwise operations on a single operand to produce a single-bit result.

The *unary AND* operator (**`&`**) returns **`1'b1`** if all the bits of the operand are **`1`**, and returns **`1'b0`** otherwise. The *unary OR* operator (**`|`**) returns **`1'b1`** if any bit of the operand is **`1`**, and returns **`1'b0`** otherwise. The *unary XOR* operator (**`^`**) returns **`1'b1`** if an odd number of bits of the operand are **`1`**, and returns **`1'b0`** otherwise. These operators are implemented in PSS/C++ using the **`and_reduce(operand)`**, **`or_reduce(operand)`**, and **`xor_reduce(operand)`** functions, respectively.

These operators (and functions in PSS/C++) may be applied only to numeric operands.

The table below shows the results of applying the three reduction operators to four example bit patterns.

Table 17—Results of unary reduction operations

Operand	&	|	^	Comments
4'b0000	0	0	0	No bits set
4'b1111	1	1	0	All bits set
4'b0110	0	1	0	Even number of bits set
4'b1000	0	1	1	Odd number of bits set

9.5.7 Shift operators

PSS provides two bitwise *shift operators*: shift-left (**<<**) and shift-right (**>>**). The left shift operator shifts the left operand to the left by the number of bit positions given by the right operand. The vacated bit positions shall be filled with zeros. The right shift operator shifts the left operand to the right by the number of bit positions given by the right operand. If the left operand is unsigned or if the left operand has a non-negative value, the vacated bit positions shall be filled with zeros. If the left operand is signed and has a negative value, the vacated bit positions shall be filled with ones. The right operand shall be a non-negative number. These operators may be applied only to numeric operands.

9.5.8 Conditional operator

The *conditional operator* (**?:**) is right-associative and is composed of three operands separated by two operators as shown in Syntax 35. The first operand (the *cond_predicate*) shall be of Boolean type. The second and third operands shall be of the same type, and may be of any plain-data or reference type.

```
conditional_expression ::= cond_predicate ? expression : expression
cond_predicate ::= expression
```

Syntax 35—DSL: Conditional operator

If *cond_predicate* is *true*, then the operator evaluates to the first *expression* without evaluating the second *expression*. If *false*, then the operator evaluates to the second *expression* without evaluating the first *expression*.

NOTE—The conditional operator is not supported in PSS/C++.

9.5.9 Set membership operator

PSS supports the *set membership operator* **in**, as applied to value sets and collection data types. Syntax 36 and Syntax 37 show the syntax for the set membership operator.

9.5.9.1 DSL syntax

```
in_expression ::=
     expression in [ open_range_list ]
   | expression in collection_expression
open_range_list ::= open_range_value { , open_range_value }
open_range_value ::= expression [ .. expression ]
collection_expression ::= expression
```

Syntax 36—DSL: Set membership operator

The set membership operator returns *true* if the value of the *expression* on the left-hand side of the **in** operator is found in the *open_range_list* or *collection_expression* on the right-hand side of the operator, and *false* otherwise.

The expression on the left-hand side shall have a self-determined type; in particular, the left-hand side shall not be an unqualified *enum_item* (see 8.4) or an aggregate literal (see 4.8). The elements of the right-hand side of the **in** operator shall have a type compatible with the *expression* on the left-hand side.

If the *expression* on the left-hand side is of a scalar type, the right-hand side may be an *open_range_list* or a *collection_expression*. If the *expression* on the left-hand side is of a collection type, the right-hand side shall be a *collection_expression*.

An *open_range_list* on the right-hand side of the **in** operator shall be a comma-separated list of scalar value expressions or ranges. When specifying a range, the expressions shall be of a numeric or enumeration type. If the left-hand bound of the range is greater than the right-hand bound of the range, the range is considered empty. Values can be repeated; therefore, values and value ranges can overlap. The evaluation order of the expressions and ranges within the *open_range_list* is nondeterministic.

A *collection_expression* on the right-hand side of the **in** operator shall evaluate to an **array**, **list**, or **set** type that contains elements whose type is compatible with the type of the *expression* on the left-hand side. For example, the *collection_expression* may be a value_list_literal or a hierarchical reference to a **set**.

9.5.9.2 C++ syntax

The corresponding C++ syntax for Syntax 36 is shown in Syntax 37.

pss::in

Defined in **`pss/in.h`** (see C.33).

```
template <class T> class in;
```

Set membership.

Member functions

```
template <class T> in(const      attr<T>& a_var, const range& a_range) :
```
attribute constructor for bit and int
```
template <class T> in(const rand_attr<T>& a_var, const range& a_range) :
```
random attribute constructor for bit and int

Syntax 37—C++: Set membership operator

NOTE—PSS/C++ only supports the *open_range_list* use of the set membership operator. Since PSS/C++ does not support collection types, the “**in** *collection_expression*” form is not supported.

9.5.9.3 Examples

Example 37 and Example 38 constrain the `addr` attribute field to the range `0x0000` to `0xFFFF`.

```
constraint addr_c {
  addr in [0x0000..0xFFFF];
}
```

Example 37—DSL: Value range constraint

```
    constraint addr_c {"addr_c",
      in (addr, range(0x0000, 0xFFFF))
    };
```

Example 38—C++: Value range constraint

In the example below, `v` is constrained to be in the combined value set of `values` and the values specified directly in the *open_range_list* `1, 2`. In other words, the value of `v` will be in `[1,2,3,4,5]`. The variable `values` of type **list** may not be referenced in an *open_range_list*.

```
    struct s {
        list<int> values = {3, 4, 5};
        rand int v;
        constraint v in [1,2] || v in values;
    }
```

Example 39—DSL: Set membership in collection

In the example below, `v` is constrained to be in the range `1, 2`, and between `a` and `b`. The range `a..b` may overlap with the values `1` and `2`.

```
    struct s {
        rand int v, a, b;
        constraint a < b;
        constraint v in [1,2,a..b];
    }
```

Example 40—DSL: Set membership in variable range

9.6 Primary expressions

There are several types of primary expressions (or *simple operands*).

The simplest type of primary expression is a reference (simple or hierarchical) to a variable, constant, or template parameter.

In order to select a single bit of a numeric variable or numeric named constant (e.g., **static const** or template parameter), a *bit-select* shall be used. In order to select a bit range of a numeric variable or numeric named constant, a *part-select* shall be used.

A collection variable of plain-data type can be referenced as a primary expression. In order to select an element within a collection, an *index operator* shall be used.

A **struct** variable can be referenced as a primary expression.

A function call is a primary expression.

There are additional types of primary expressions. Formally, an expression is a primary expression if it is a *primary* as defined in B.17 and unparenthesized.

9.6.1 Bit-selects and part-selects

Bit-selects select a particular bit from a named numeric variable or constant using the syntax

```
    identifier [ expression ]
```

The index may be any integer expression and may be non-constant.

Part-selects select a fixed range of contiguous bits using the syntax

```
    identifier [ constant_expression : constant_expression ]
```

The value of the first *constant_expression* shall be greater than or equal to the value of the second *constant_expression*.

Bit-selects and part-selects may be used as operands of other operators and as targets of assignments. It shall be illegal for a bit-select or a part-select to access an out-of-bounds bit index.

NOTE—PSS/C++ supports bit-selects on scalar fields, but does not support part-selects.

9.6.2 Selecting an element from a collection (indexing)

The *index operator* [] is applied to an **array**, **list**, or **map** collection to select a single element. In the case of an **array** or a **list**, the index shall be an integer expression whose value is between 0 and the size of the **array**/**list** - 1. In the case of a **map**, the index shall be of the same type as that of the key in the **map** declaration.

An indexed collection may be used as an operand of other operators and as a target of assignments.

In the case of an **array** or a **list**, it shall be illegal to access an out-of-bounds index. In the case of a **map**, it shall be illegal to read an element whose key does not appear in the **map**. An assignment to a **map** element whose key does not currently appear in the **map** shall add that key and value pair to the **map**.

9.7 Bit sizes for numeric expressions

The size, in bits, of a numeric expression is determined by the operands involved in the expression and the context in which the expression appears. Casting can be used to set the size context of an intermediate value (see 8.11).

9.7.1 Rules for expression bit sizes

A *self-determined expression* is one where the size of the expression is solely determined by the expression itself. A *context-determined expression* is one where the size of the expression is determined both by the expression itself and by the fact that it is part of another expression. For example, the size of the right-hand expression of an assignment depends on itself and the size of the left-hand side.

Table 18 shows how the form of an expression determines the sizes of the results of the expression. In Table 18, **`i`**, **`j`**, and **`k`** represent operands of an expression, and **`L(i)`** represents the size of the operand represented by **`i`**.

Table 18—Bit sizes resulting from self-determined expressions

Expression	Bit size	Comments
Unsized constant number	At least 32	
Sized constant number	As specified	
`i op j`, where **`op`** is: **`+ - * / % &	^`**	max(L(i),L(j))	
`op i`, where **`op`** is: **`+ - ~`**	L(i)	
`op i`, where **`op`** is: **`&	^`**	1	
`i op j`, where **`op`** is: **`>> << **`**	L(i)	**`j`** is self-determined
`i ? j : k`	max(L(j),L(k))	**`i`** must be Boolean
cast, where *casting_type* is numeric	L(*casting_type*)	

9.8 Evaluation rules for numeric expressions

9.8.1 Rules for expression signedness

The following apply when determining the signedness of an expression:

a) Expression signedness depends only on the operands. In an assignment, the signedness does not depend on the left-hand side.

b) Unsized unbased decimal and octal numbers are signed. Unsized unbased hexadecimal numbers are unsighed.

c) Based numbers are unsigned, except when they are designated as signed with the **`'s`** notation (e.g., `4'sd12`).

d) Bit-select results are unsigned, regardless of the operands.

e) Part-select results are unsigned, regardless of the operands, even if the part-select specifies the entire width.

f) The signedness and size of a self-determined operand are determined by the operand itself, independent of the remainder of the expression.

g) If any operand of an expression is unsigned, the result is unsigned regardless of the operators.

h) If all operands of an expression are signed, the result is signed regardless of the operators, unless specified otherwise.

9.8.2 Steps for evaluating a numeric expression

The following are the steps for evaluating a numeric expression:

a) Determine the expression size based on the expression size rules (see 9.7.1).

b) Determine the signedness of the expression using the rules described above.

c) Propagate the signedness and size of the expression to the context-determined operands of the expression. Context-determined operands of an operator shall have the same signedness and size as the result of the operator.

d) When propagation reaches a simple operand (see 9.6), that operand shall be converted to the propagated signedness and size. If the operand must be size-extended, it shall be sign-extended if the propagated type is signed and zero-extended if the propagated type is unsigned.

9.8.3 Steps for evaluating an assignment

The following are the steps for evaluating an assignment when the operands are of numeric type:

a) Determine the size of the right-hand side of the assignment using the size determination rules described in 9.7.1.

b) If required, extend the size of the right-hand side, using sign extension if the type of the right-hand side is signed and zero-extension if the type of the right-hand side is unsigned.

10. Components

Components serve as a mechanism to encapsulate and reuse elements of functionality in a portable stimulus model. Typically, a model is broken down into parts that correspond to roles played by different actors during test execution. Components often align with certain structural elements of the system and execution environment, such as hardware engines, software packages, or testbench agents.

Components are structural entities, defined per type and instantiated under other components (see Syntax 38, Syntax 39 and Syntax 40). Component instances constitute a hierarchy (tree structure), beginning with the top or root component, called **`pss_top`** by default, which is implicitly instantiated. Each component instance has a unique hierarchical path name, and may also contain data attributes, but not constraints. Components may also encapsulate function declarations (see 22.2.1) and imported class instances (see 22.4.2). In addition, components may be derived from other components via inheritance, or a component may be extended to add elements to the component type (see Clause 20).

10.1 DSL syntax

```
component_declaration ::= [ pure ] component component_identifier [ template_param_decl_list ]
   [ component_super_spec ] { { component_body_item } }
component_super_spec ::= : type_identifier
component_body_item ::=
    override_declaration
   | component_data_declaration
   | component_pool_declaration
   | action_declaration
   | abstract_action_declaration
   | object_bind_stmt
   | exec_block
   | struct_declaration
   | enum_declaration
   | covergroup_declaration
   | function_decl
   | import_class_decl
   | procedural_function
   | import_function
   | target_template_function
   | export_action
   | typedef_declaration
   | import_stmt
   | extend_stmt
   | compile_assert_stmt
   | attr_group
   | component_body_compile_if
   | stmt_terminator
```

Syntax 38—DSL: component declaration

10.2 C++ syntax

The corresponding C++ syntax for Syntax 38 is shown in Syntax 39 and Syntax 40.

Components are declared using the **`component`** class (see Syntax 39).

pss::component

Defined in **`pss/component.h`** (see C.11) .

```
class component;
```

Base class for declaring a component.

Member functions

`component (const scope& name)` : constructor
`virtual void init_down()` : in-line exec init_down block
`virtual void init_up()` : in-line exec init_up block
`virtual void init()` : in-line exec init block

Syntax 39—C++: component declaration

Components are instantiated using the **`comp_inst<>`** or **`comp_inst_vec<>`** class (see Syntax 40).

pss::comp_inst

Defined in **`pss/comp_inst.h`** (see C.10).

```
template <class T> class comp_inst;
```

Instantiate a component.

Member functions

`comp_inst (const scope& name)` : constructor
`T* operator-> ()` : access fields of component instance
`T& operator* ()` : access fields of component instance

pss::comp_inst_vec

Defined in **`pss/comp_inst.h`** (see C.10) .

```
template <class T> class comp_inst_vec;
```

Instantiate an array of components.

Member functions

`comp_inst<T>& operator[](const std::size_t index)` : access element of component array
`std::size_t size()` : returns number of components in array

Syntax 40—C++: component instantiation

10.3 Examples

For examples of how to declare a component, see Example 41 and Example 42.

```
component uart_c { ... };
```

Example 41—DSL: Component

The corresponding C++ example for Example 41 is shown in Example 42.

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

Example 42—C++: Component

10.4 Components as namespaces

Component types serve as namespaces 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. The fully-qualified name of **action** and object types is of the form '`*package-namespace*::*component-type*::*class-type*`'.

Within a given component type, references can be left unqualified. However, referencing a nested type from another component requires the component namespace qualification.

For an example of how to use a component as a namespace, see Example 43 and Example 44.

```
component usb_c {
  action write {...}
}
component uart_c {
  action write {...}
}
component pss_top {
  uart_c s1;
  usb_c s2;
  action entry {
    uart_c::write wr; //refers to the write action in uart_c
    ...
  }
}
```

Example 43—DSL: Namespace

The corresponding C++ example for Example 43 is shown in Example 44.

```
class usb_c : public component { ...
  class write : public action {...};
  type_decl<write> write_decl;
};
...

class uart_c : public component { ...
  class write : public action {...};
  type_decl<write> write_decl;
};
...

class pss_top : public component { ...
  comp_inst<uart_c> s1{"s1"};
  comp_inst<usb_c> s2{"s2"};
  class entry : public action { ...
    action_handle<uart_c::write> wr{"wr"};
    ...
  };
  type_decl<entry> entry_decl;
};
...
```

Example 44—C++: Namespace

In Example 45 below, a **component** `C1` is declared in a **package**. That **component** is instantiated in **component** **`pss_top`**, and an **action** within **component** `C1` is traversed in **action** `pss_top::entry`. In the traversal of **action** `P::C1::A`, the qualified name elements are the following:

— *package-namespace*: `P`

— *component-type*: `C1`

— *class-type*: `A`

```
package P {
    component C1 {
        action A {}
    }
}

component pss_top {
    P::C1 c1;

    action entry {
        activity {
            do P::C1::A;
        }
    }
}
```

Example 45—DSL: Component declared in package

10.5 Component instantiation

Components are instantiated under other components as their fields, much like data fields of structs, and may be arrays thereof.

10.5.1 Semantics

a) Component fields are non-random; therefore, the **rand** modifier shall not be used. Component data fields represent configuration data that is accessed by actions declared in the component. To avoid infinite component instantiation recursion, a component type and all template specializations thereof shall not be instantiated under its own sub-tree.

b) In any model, the component instance tree has a predefined root component, called **`pss_top`** by default, but this may be user-defined. There can only be one root component in any valid scenario.

c) Other components or actions are instantiated (directly or indirectly) under the root component. See also Example 46 and Example 47.

d) Plain-data fields may be initialized using a constant expression in their declaration. Data fields may also be initialized via an **exec init_down** or **init_up** block (see 22.1.3), which overrides the value set by an initialization assignment. The component tree is elaborated to instantiate each component and then the **exec init_down** and **init_up** blocks are evaluated hierarchically. See also Example 297, Example 298, and Example 299 in 22.1.4.

e) Component data fields are considered immutable once construction of the component tree is complete. Actions can read the value of these fields, but cannot modify their value. Component data fields are accessed from actions relative to the **`comp`** field, which is a handle to the component context in which the action is executing. See also Example 300 and Example 301 (and 22.1).

10.5.2 Examples

Example 46 and Example 47 depict a component tree definition. In total, there is one instance of `multimedia_ss_c` (instantiated in **`pss_top`**), four instances of `codec_c` (from the array declared in `multimedia_ss_c`), and eight instances of `vid_pipe_c` (two in each element of the `codec_c` array).

```
component vid_pipe_c { ... };

component codec_c {
   vid_pipe_c pipeA, pipeB;
   action decode { ... };
};

component multimedia_ss_c {
   codec_c codecs[4];
};

component pss_top {
   multimedia_ss_c multimedia_ss;
};
```

Example 46—DSL: Component instantiation

```
class vid_pipe_c : public component { ... };
...
class codec_c : public component {...
  comp_inst<vid_pipe_c> pipeA{"pipeA"}, pipeB{"pipeB"};

  class decode : public action { ... };
  type_decl<decode> decode_decl;
};
...
class multimedia_ss_c : public component { ...
  comp_inst_vec<codec_c> codecs{ "codecs", 4};
};
...
class pss_top : public component { ...
  comp_inst<multimedia_ss_c> multimedia_ss{"multimedia_ss"};
};
...
```

Example 47—C++: Component instantiation

10.6 Component references

Each action instance is associated with a specific component instance of its containing component type, the component-type scope where the action is defined. The component instance is the “actor” or “agent” that performs the action. Only actions defined in the scope of instantiated components can legally participate in a scenario.

The component instance with which an **action** is associated is referenced via the built-in field **comp**. The value of the **comp** field can be used for comparisons of references (see 9.5.3). Unlike user-defined reference variables, the **comp** field is assigned automatically as part of the solving process (see 17.4.4) and may not be assigned by the user. The static type of the **comp** field is the **ref** type of the action’s context **component**. Consequently, attributes and sub-components of the containing component may be referenced via the **comp** field using relative paths.

10.6.1 Semantics

A compound action can only instantiate sub-actions that are defined in its containing component or defined in component types that are instantiated in its containing component's instance sub-tree. In other words, compound actions cannot instantiate actions that are defined in components outside their context component hierarchy.

10.6.2 Examples

Example 48 and Example 49 demonstrate the use of the **comp** reference. The constraint within the `decode` action forces the value of the action’s `mode` bit to be `0` for the `codecs[0]` instance, while the value of `mode` is randomly selected for the other instances. The sub-action type `program` is available on both sub-component instances, `pipeA` and `pipeB`, but in this case is assigned specifically to `pipeA` using the **comp** reference.

See also 17.1.3.

```
component vid_pipe_c { ... };
component codec_c {
  vid_pipe_c pipeA, pipeB;
  bit mode1_enable;
  action decode {
    rand bit mode;
    constraint set_mode {
      comp.mode1_enable==0 -> mode == 0;
    }
    activity {
      do vid_pipe_c::program with { comp == this.comp.pipeA; };
    }
  };
};
component multimedia_ss_c {
  codec_c codecs[2];
  exec init_up {
    codecs[0].mode1_enable = 0;
    codecs[1].mode1_enable = 1;
  }
};
```

Example 48—DSL: Constraining a comp attribute

```
class vid_pipe_c : public component {...};
...
class codec_c : public component { ...
  comp_inst<vid_pipe_c> pipeA{"pipeA"}, pipeB{"pipeB"};
  attr<bit> mode1_enable {"mode1_enable"};

  class decode : public action { ...
    rand_attr<modes_e> mode {"mode"};
    action_handle<codec_c::decode> codec_c_decode{"codec_c_decode"};

    action_handle<vid_pipe_c::program> pipe_prog_a{"pipe_prog_a"};

    activity act {
      pipe_prog_a.with(
        pipe_prog_a->comp() == comp<codec_c>()->pipeA
        )
    };
  };
  type_decl<decode> decode_decl;
};
...
class multimedia_ss_c : public component { ...
  comp_inst_vec<codec_c> codecs{ "codecs", 2};
  exec e { exec::init_up,
    codecs[0]->mode1_enable = 0,
    codecs[1]->mode1_enable = 1
  };
};
...
```

Example 49—C++: Constraining a comp attribute

10.7 Pure components

Pure components are restricted types of components that provide PSS implementations with opportunities for significant optimization of storage and initialization. Pure components are used to encapsulate realization-level functionality and cannot contain scenario model features. *Register structures* are one possible application for pure components (see 24.5).

The following rules apply to pure components, that is, component types declared with the **pure** modifier:

a) In the scope of a **pure component**, it shall be an error to declare **action** types, **pool** instances, **pool**-binding directives, non-static data attributes, instances of non-pure **component** types, or **exec** blocks.

b) A **pure component** may be instantiated under a non-pure **component**. However, a non-pure **component** may not be instantiated under a **pure component**.

c) A **pure component** may not be derived from a non-pure **component**. However, both a **pure component** and a non-pure **component** may be derived from a **pure component**.

An example of the use of pure components is shown in Example 50.

```
pure component my_register {
   function bit[32] read();
   function void write(bit[32] val);
};

pure component my_register_group {
   my_register regs[10];
};

component my_ip {
   my_register_group reg_groups[100]; // sparsely-used large structure
};
```

Example 50—DSL: Pure components

11. Actions

Actions are a key abstraction unit in PSS. Actions serve to decompose scenarios into elements whose definitions can be reused in many different contexts. Along with their intrinsic properties, actions also encapsulate the rules for their interaction with other actions and the ways to combine them in legal scenarios. Atomic actions may be composed into higher-level actions, and, ultimately, to top-level test actions, using activities (see Clause 13). The *activity* of a compound action specifies the intended schedule of its sub-actions, their object binding, and any constraints. Activities are a partial specification of a scenario: determining their abstract intent and leaving other details open.

Actions prescribe their possible interactions with other actions indirectly, by using flow (see Clause 14) and resource (see Clause 15) objects. *Flow object references* specify the action's inputs and outputs and *resource object references* specify the action's resource claims.

By declaring a reference to an object, an action determines its relation to other actions that reference the very same object without presupposing anything specific about them. For example, one action may reference a data flow object of some type as its input, which another action references as its output. By referencing the same object, the two actions necessarily agree on its properties without having to know about each other. Each action may constrain the attributes of the object. In any consistent scenario, all constraints shall hold; thus, the requirements of both actions are satisfied, as well as any constraints declared in the object itself.

Actions may be *atomic*, in which case their implementation is supplied via an **exec body** block (see 22.1.3), or they may be *compound*, in which case they contain an **activity** (see Clause 13) that instantiates and schedules other actions. A single action can have multiple implementations in different packages, so the actual implementation of the action is determined by which package is used.

An action is declared using the **action** keyword and an *action_identifier*, as shown in Syntax 41. See also Syntax 42.

11.1 DSL syntax

```
action_declaration ::= action action_identifier [ template_param_decl_list ] [ action_super_spec ]
  { { action_body_item } }
abstract_action_declaration ::= abstract action_declaration
action_super_spec ::= : type_identifier
action_body_item ::=
    activity_declaration
  | override_declaration
  | constraint_declaration
  | action_field_declaration
  | symbol_declaration
  | covergroup_declaration
  | exec_block_stmt
  | activity_scheduling_constraint
  | attr_group
  | compile_assert_stmt
  | covergroup_instantiation
  | action_body_compile_if
  | stmt_terminator
action_field_declaration ::=
    attr_field
  | activity_data_field
  | action_handle_declaration
  | object_ref_field_declaration
```

Syntax 41—DSL: action declaration

An **action** declaration optionally specifies an *action_super_spec*, a previously defined action type from which the new type inherits its members.

The following also apply:

a) The *activity_declaration* and **body** *exec_block_stmt* (see 22.1.3) action body items are mutually exclusive. An atomic action may specify **body** *exec_block_stmt* items; it shall not specify *activity_-declaration* items. A compound action, which contains instances of other actions and *activity_declaration* items, shall not specify **body** *exec_block_stmt* items.

b) An *abstract action* may be declared as a template that defines a base set of field attributes and behavior from which other actions may inherit. Non-abstract derived actions may be instantiated like any other action. Abstract actions shall not be instantiated directly.

c) An abstract action may be derived from another abstract action, but not from a non-abstract action.

d) Abstract actions may be extended, but the action remains abstract and may not be instantiated directly.

e) PSS/C++ does not support declaring abstract actions.

11.2 C++ syntax

Actions are declared using the **`action`** class.

The corresponding C++ syntax for Syntax 41 is shown in Syntax 42.

pss::action

Defined in **`pss/action.h`** (see C.2).

```
class action;
```

Base class for declaring an action.

Member functions

`action (const scope& name)` : constructor
`virtual void pre_solve()` : in-line pre_solve exec block
`virtual void post_solve()` : in-line post_solve exec block
`template <class T=component> detail::comp_ref<T> comp();` : refer to action's context component instance

Syntax 42—C++: action declaration

11.3 Examples

11.3.1 Atomic actions

Examples of an *atomic action* declaration are shown in Example 51 and Example 52.

```
action write {
  output data_buf data;
  rand int size;
  //implementation details
  ...
};
```

Example 51—DSL: atomic action

The corresponding C++ example for Example 51 is shown in Example 52.

```
class write : public action { ...
  output <data_buf> data {"data"};
  rand_attr<int> size {"size"};
  // implementation details
  ...
};
...
```

Example 52—C++: atomic action

11.3.2 Compound actions

Compound actions instantiate other actions within them and use **activity** statements (see Clause 13) to define the relative scheduling of these sub-actions.

Examples of compound action usage are shown in Example 53 and Example 54.

```
action sub_a {...};

action compound_a {
  sub_a a1, a2;
  activity {
    a1;
    a2;
  }
}
```

Example 53—DSL: compound action

The corresponding C++ example for Example 53 is shown in Example 54.

```
class sub_a : public action { ... };
...
class compound_a : public action { ...
  action_handle<sub_a> a1{"a1"}, a2{"a2"};
  activity act {
    a1,
    a2
  };
};
...
```

Example 54—C++: compound action

11.3.3 Abstract actions

Abstract action types are used to capture common features of different actions, including actions of different components. Abstract actions may not be traversed directly. Rather, they are used through inheritance, as base types for non-abstract action types. Abstract action types may be declared outside the scope of a component, unlike non-abstract actions, which may only be declared in a component scope.

An example of abstract action usage is shown in Example 55. In this example, abstract action `base` is declared outside a component scope, in package `mypkg`, and subsequently extended in the same package. Action `derived` is declared as a non-abstract subtype of action `base`.

```
package mypkg {
  abstract action base {
    rand int i;
    constraint i>5 && i<10;
  }

// action base remains abstract
  extend action base {
    rand int j;
  }
}

component pss_top {
  import mypkg::*;

  action derived : base {
    constraint i>6;
    constraint j>9;
  }
}
```

Example 55—DSL: abstract action

NOTE—PSS/C++ does not support declaring abstract actions.

12. Template types

12.1 General

Template types in PSS/DSL provide a way to define generic parameterized types.

In many cases, it is useful to define a generic parameterizable type (struct/flow object/resource object/action/component) that can be instantiated with different parameter values (e.g., array sizes or data types). Template types maximize reuse, avoid writing similar code for each parameter value (value or data type) combination, and allow a single specification to be used in multiple places.

Template types must be explicitly instantiated by the user, and only an explicit instantiation of a template type represents an actual type.

The following sections describe how to define, use, and extend a template type when using the PSS/DSL input.

NOTE—PSL parameterized types are not supported in PSL/C++. Referencing PSS parameterized types between PSS/C++ and PSS/DSL is not supported.

12.2 Template type declarations

A *template type* (**struct**, **action**, **component**, etc.) declaration specifies a list of formal *type* or *value template parameter* declarations. The parameters are provided as a comma-separated list enclosed in angle brackets (<>) following the name of the template type.

A template type may inherit from another template or non-template data type. A non-template type may inherit from a template type instance. In both cases, the same inheritance rules and restrictions as for the corresponding non-template type of the same type category are applied (e.g., a template **struct** may inherit from a **struct**, or from a template **struct**).

The DSL syntax specified in the corresponding **struct**/**action**/**component** sections contains the *template_param_decl_list* nonterminal marked as optional. When the parameter declaration list enclosed in angle brackets is provided on a **struct**/**action**/**component** declaration, it denotes that the **struct**/**action**/**component** type is a template generic type.

12.2.1 DSL syntax

struct_declaration ::= struct_kind identifier [template_param_decl_list]
  [struct_super_spec] **{** { struct_body_item } **}**

component_declaration ::= **component** *component*_identifier [template_param_decl_list]
  [component_super_spec] **{** { component_body_item } **}**

action_declaration ::= **action** *action*_identifier [template_param_decl_list]
  [action_super_spec] **{** { action_body_item } **}**

template_param_decl_list ::= **<** template_param_decl { **,** template_param_decl } **>**

template_param_decl ::= type_param_decl | value_param_decl

Syntax 43—DSL: Template type declaration

12.2.2 Examples

Generic template-type declaration for various type categories are shown in Example 56.

```
struct my_template_s <type T> {
  T  t_attr;
}

buffer my_buff_s <type T> {
  T  t_attr;
}

action my_consumer_action <int width, bool is_wide> {
  compile assert (width > 0);
}

component eth_controller_c <struct ifg_config_s, bool full_duplex = true> {
}
```

Example 56—DSL: Template type declarations

12.3 Template parameter declarations

A template parameter is declared as either a type or a value parameter. All template parameters have a name and an optional default value. All parameters subsequent to the first one that is given a default value shall also be given default values. Therefore, the parameters with defaults shall appear at the end of the parameter list. Specifying a parameter with a default value followed by a parameter without a default value shall be reported as an error.

A template parameter can be referenced using its name inside the body and the super-type specification of the template type and all subsequent generic template type extensions, including the template type instance extensions. A template parameter may not be referenced from within subtypes that inherit from the template type that originally defined the parameter.

12.3.1 Template value parameter declarations

Value parameters are given a data type and optionally a default value, as shown below.

12.3.1.1 DSL syntax

```
value_param_decl ::= data_type identifier [ = constant_expression ]
```

Syntax 44—DSL: Template value parameter declaration

The following also apply:

a) A value parameter can be referenced using its name anywhere a constant expression is allowed or expected inside the body and the super-type specification of the template type.

b) Valid data types for a *value_param_decl* are restricted to the scalar **int**, **bit**, **bool**, **string**, and **enum** types.

c) The default value, if provided, may also reference one or more of the previously defined parameters.

d) To avoid parsing ambiguity, a Boolean greater-than (>) or less-than (<) expression provided as a default value shall be enclosed in parentheses.

12.3.1.2 Examples

An example of declaring an action type that consumes a varying number of resources is shown in Example 57.

```
action my_consumer_action <int n_locks = 4> {
  compile assert (n_locks in [1..16]);
  lock my_resource res[n_locks];
}
```

Example 57—DSL: Template value parameter declaration

Example 58 contains a Boolean greater-than expression that must be enclosed in parentheses and depends on a previous parameter:

```
action my_consumer_action <int width, bool is_wide = (width > 10) > {
  compile assert (width > 0);
}
```

Example 58—DSL: Another template value parameter declaration

12.3.2 Template type parameter declarations

Type parameters are prefixed with either the **type** keyword or a type-category keyword in order to identify them as type parameters.

When the **type** keyword is used, the parameter is fully generic. In other words, it can take on any type.

Specifying category type parameters provides more information to users of a template type on acceptable usage and allows tools to flag usage errors earlier. A category type parameter enforces that a template instance parameter value must be of a certain category/class of type (e.g., **struct**, **action**, etc.). A category type parameter can be further restricted such that the specializing type (the parameter value provided on instantiation) must be related via inheritance to a specified base type.

The syntax for declaring a type parameter is shown below.

12.3.2.1 DSL syntax

```
type_param_decl ::= generic_type_param_decl | category_type_param_decl
generic_type_param_decl ::= type identifier [ = type_identifier ]
category_type_param_decl ::= type_category identifier [ type_restriction ] [ = type_identifier ]
type_restriction ::= : type_identifier
type_category ::=
   action
  | component
  | struct_kind
```

Syntax 45—DSL: Template type parameter declaration

The following also apply:

a) A type parameter can be referenced using its name anywhere inside the body of the template type where a type is allowed or expected.

b) The default value, if provided, may also reference one or more of the previously defined parameters.

12.3.2.2 Examples

Examples of a generic type and a category type parameter are shown in Example 59.

```
struct my_container_s <struct T> {
  T t_attr;
}

struct my_template_s <type T> {
  T t_attr;
}
```

Example 59—DSL: Template generic type and category type parameters

In the example above, the template parameter `T` of `my_container_s` must be of **struct** type, while in the case of `my_template_s`, the template parameter `T` may take on any type.

An example of how to use type restrictions in the case of a type-category parameter is shown in Example 60.

```
struct base_t {
  rand bit[3:0] core;
}

struct my_sub1_t : base_t {
  rand bit[3:0] add1;
}

struct my_sub2_t : base_t {
  rand bit[3:0] add2;
}

buffer b1 : base_t { }
buffer b2 : base_t { }

action my_action_a <buffer B : base_t> {
}

struct my_container_s  <struct T : base_t = my_sub1_t> {
  T t_attr;
  constraint t_attr.core >= 1;
}
```

Example 60—DSL: Template parameter type restriction

In the example above, the template parameter **T** of `my_container_s` must be of type `base_t` or one of its **struct** subtypes (`my_sub1_t` or `my_sub2_t`, but not `b1` or `b2`). This allows `my_container_s` to reasonably assume that **T** contains an attribute named '`core`', and communicates this requirement to users of this type and to the PSS processing tool. The template parameter `B` of `my_action_a` must be of one of the **buffer** subtypes of `base_t` (`b1` or `b2`).

The base type of the template type may also be a type parameter. In this way, the inheritance can be controlled when the template type is instantiated.

In Example 61, the `my_container_s` template **struct** inherits from the **struct** type template type parameter.

```
struct my_base1_t {
  rand int attr1;
}

struct my_base2_t {
  rand int attr2;
}

struct my_container_s <struct T> : T {
}

struct top_s {
  rand my_container_s <my_base1_t> cont1;
  rand my_container_s <my_base2_t> cont2;
  constraint cont1.attr1  == cont2.attr2;
}
```

Example 61—DSL: Template parameter used as base type

12.4 Template type instantiation

A template type is instantiated using the name of the template type followed by the parameter value list (specialization) enclosed in angle brackets (<>). Template parameter values are specified positionally.

The explicit instantiation of a template type represents an actual type. All explicit instantiations provided with the same set of parameter values are the same actual type.

12.4.1 DSL syntax

```
type_identifier ::= [ :: ] type_identifer_elem { :: type_identifer_elem }
type_identifier_elem ::= identifier [ template_param_value_list ]
template_param_value_list ::= < [ template_param_value { , template_param_value } ] >
template_param_value ::= constant_expression | data_type
```

Syntax 46—DSL: Template type instantiation

The following also apply:

a) Parameter values must be specified for all parameters that were not given a default value.
b) An instance of a template type must always specify the angle brackets (<>), even if no parameter value overrides are provided for the defaults.
c) The specified parameter values must comply with parameter categories and parameter type restrictions specified for each parameter in the original template declaration, or an error shall be generated.
d) To avoid parsing ambiguity, a Boolean greater-than (>) or less-than (<) expression provided as a parameter value must be enclosed in parentheses.

12.4.2 Examples

```
struct base_t {
  rand bit[3:0] core;
}

struct my_sub1_t : base_t {
  rand bit[3:0] add1;
}

struct my_sub2_t : base_t {
  rand bit[3:0] add2;
}

struct my_container_s <struct T : base_t = my_sub1_t> {
  T t_attr;
  constraint t_attr.core >= 1;
}

struct top_s {
  my_container_s<>          my_sub1_container_attr;
  my_container_s<my_sub2_t> my_sub2_container_attr;
}
```

Example 62—DSL: Template type instantiation

In Example 62 above, two attributes of `my_container_s` type are created. The first uses the default parameter value. The second specifies the `my_sub2_t` type as the value for the **`T`** parameter.

Type qualification for an action declared in a template component is shown in Example 63 below.

```
component my_comp1_c <int bus_width = 32> {
  action my_action1_a { }
  action my_action2_a <int nof_iter = 4> { }
}

component pss_top {
  my_comp1_c<64> comp1;
  my_comp1_c<32> comp2;

  action test {
    activity {
      do my_comp1_c<64>::my_action1_a;
      do my_comp1_c<64>::my_action2_a<>;
      do my_comp1_c::my_action1_a;  // Error -  my_comp1_c must be specialized
      do my_comp1_c<>::my_action1_a;
    }
  }
}
```

Example 63—DSL: Template type qualification

Example 64 depicts various ways of overriding the default values. In the example below, the `my_struct_t<2>` instance overrides the parameter `A` with `2`, and preserves the default values for parameters `B` and `C`. The `my_struct_t<2, 8>` instance overrides the parameter `A` with `2`, parameter `B` with `8`, and preserves the default value for `C`.

```
struct my_s_1 { }
struct my_s_2 { }

struct my_struct_t <int A = 4, int B = 7, int C = 3> {  }

struct container_t {
  my_struct_t<2>   a; // instantiated with <2, 7, 3>
  my_struct_t<2,8> b; // instantiated with <2, 8, 3>
}
```

Example 64—DSL: Overriding the default values

12.5 Template type user restrictions

A generic template type may not be used in the following contexts:

- As a root component
- As a root action
- As an inferred action to complete a partially specified scenario

Template types are explicitly instantiated by the user, and only an explicit instantiation of a template type represents an actual type. Only action actual types can be inferred to complete a partially specified scenario. The root component and the root action must be actual types.

Template types may not be used as parameter types or return types of imported functions.

13. Activities

When a *compound action* includes multiple operations, these behaviors are described within the **action** using one or more **activity** statements. An *activity* specifies the set of actions to be executed and the scheduling relationship(s) between them. If more than one activity is specified in an action, the execution semantics are the same as if the activity statements were combined in a **schedule** statement (see 13.3.5 and 13.6). A reference to an action within an activity is via an *action handle*, and the resulting *action traversal* causes the referenced action to be evaluated and randomized (see 13.3.1).

An activity, on its own, does not introduce any scheduling dependencies for its containing action. However, flow object or resource scheduling constraints of the sub-actions may introduce scheduling dependencies for the containing action relative to other actions in the system.

13.1 Activity declarations

Because activities are explicitly specified as part of an action, activities themselves do not have a separate name. Relative to the sub-actions referred to in the activity, the action that contains the activity is referred to as the *context action*.

13.2 Activity constructs

Each node of an activity represents an action, with the activity specifying the temporal, control, and/or data flow between them. These relationships are described via activity rules, which are explained herein. See also Syntax 47 and Syntax 48.

13.2.1 DSL syntax

```
activity_declaration ::= activity { { [ label_identifier : ] activity_stmt } }
activity_stmt ::=
    [ label_identifier :] labeled_activity_stmt
  | activity_data_field
  | activity_bind_stmt
  | action_handle_declaration
  | activity_constraint_stmt
  | activity_scheduling_constraint
  | stmt_terminator
labeled_activity_stmt ::=
    activity_action_traversal_stmt
  | activity_sequence_block_stmt
  | activity_parallel_stmt
  | activity_schedule_stmt
  | activity_repeat_stmt
  | activity_foreach_stmt
  | activity_select_stmt
  | activity_if_else_stmt
  | activity_match_stmt
  | activity_replicate_stmt
  | activity_super_stmt
  | symbol_call
```

Syntax 47—DSL: activity statement

NOTE—PSS/C++ does not support declaring action handles and activity data fields within an activity scope.

13.2.2 C++ syntax

In C++, an activity is declared by instantiating the **`activity`** class.

The corresponding C++ syntax for Syntax 47 is shown in Syntax 48.

pss::action::activity

Defined in **`pss/action.h`** (see C.2).

```
template <class... R> class activity;
```

Declare an activity.

Member functions

```
template <class... R> activity(R&&... /*detail::Stmt*/ r) : constructor
```

Syntax 48—C++: activity statement

13.3 Action scheduling statements

By default, statements in an activity specify sequential behaviors, subject to data flow constraints. In addition, there are several statements that allow additional scheduling semantics to be specified. Statements within an activity may be nested, so each element within an activity statement is referred to as a sub-activity.

13.3.1 Action traversal statement

An *action traversal statement* designates the point in the execution of an activity where an action is randomized and evaluated (see Syntax 49 and Syntax 50). The action being traversed may be specified via an action handle referring to an action field that was previously declared or the action being traversed may be specified by type, in which case the action instance is anonymous.

13.3.1.1 DSL syntax

```
activity_action_traversal_stmt ::=
    identifier [ [ expression ] ] inline_constraints_or_empty
  | do type_identifier inline_constraints_or_empty
inline_constraints_or_empty ::=
    with constraint_set
  | ;
```

Syntax 49—DSL: Action traversal statement

identifier names a unique action handle or variable in the context of the containing action type. If *identifier* refers to an *action handle array* (see 13.3.2), then a specific array element may be specified with the optional array subscript. The alternative form is an *anonymous action traversal*, specified by the keyword **do**, followed by an action-type specifier and an optional in-line constraint.

The following also apply:

a) The action variable is randomized and evaluated at the point in the flow where the statement occurs. The variable may be of an action type or a data type declared in the context action with the **action** modifier. In the latter case, it is randomized, but has no observed execution or duration (see Example 195 and Example 196).

1) An action handle is considered uninitialized until it is first traversed. The fields within the **action** cannot be referenced in an **exec** block or conditional activity statement until after the action is first traversed. The steps that occur as part of the *action traversal* are as follows:
    i) The **pre_solve** block (if present) is executed.
    ii) Random values are selected for **rand** fields.
    iii) The **post_solve** block (if present) is executed.
    iv) The **body exec** block (if present) is executed.
    v) The **activity** block (if present) is evaluated.
    vi) The validity of the constraint system is confirmed, given any changes by the **post_solve** or **body exec** blocks.
2) Upon entry to an **activity** scope, all action handles traversed in that scope are reset to an uninitialized state.

b) The *anonymous action traversal* statement is semantically equivalent to an action traversal with the exception that it does not create an action handle that may be referenced from elsewhere in the stimulus model.

c) A named action handle may only be traversed once in the following scopes and nested scopes thereof:
    1) sequential activity scope (e.g., **sequence** or **repeat**)
    2) parallel
    3) schedule

d) Formally, a *traversal statement* is equivalent to the sub-activity of the specified action type, with the optional addition of in-line constraints. The sub-activity is scheduled in accordance with the scheduling semantics of the containing activity or sub-activity.

e) Other aspects that impact action-evaluation scheduling, are covered via binding inputs or outputs (see 14.4), resource claims (see 15.2), or attribute value assignment.

13.3.1.2 C++ syntax

The corresponding C++ syntax for Syntax 49 is shown in Syntax 50.

pss::action_handle

Defined in **`pss/action_handle.h`** (see C.4).

```
template <class T> action_handle;
```

Declare an action handle.

Member functions

```
action_handle(const scope& name) : constructor
template <class... R> action_handle<T> with
   (const R&... /*detail ::AlgebExpr*/ constraints) : add constraint to action handle
T* operator->() : access underlying action type
T& operator*() : access underlying action type
```

Syntax 50—C++: Action traversal statement

13.3.1.3 Examples

Example 65 and Example 66 show an example of traversing an atomic action variable. Action `A` is an atomic action that contains a 4-bit random field `f1`. Action `B` is a compound action encapsulating an activity involving two invocations of action `A`. The default constraints for `A` apply to the evaluation of `a1`. An additional constraint is applied to `a2`, specifying that `f1` shall be less than `10`. Execution of action `B` results in two sequential evaluations of action `A`.

```
action A {
  rand bit[3:0]   f1;
  ...
}

action B {
  A a1, a2;

  activity {
    a1;
    a2 with {
      f1 < 10;
    };
  }
}
```

Example 65—DSL: Action traversal

```
class A : public action { ...
  rand_attr<bit> f1 {"f1", width(3, 0) };
};
...
class B : public action { ...
  action_handle<A> a1{"a1"}, a2{"a2"};
  activity a {
    a1,
    a2.with(a2->f1 < 10)
  };
};
...
```

Example 66—C++: Action traversal

Example 67 and Example 68 show an example of anonymous action traversal, including in-line constraints.

```
action A {
  rand bit[3:0]   f1;
  ...
}

action B {
  activity {
    do A;
    do A with {f1 < 10;};
  }
}
```

Example 67—DSL: Anonymous action traversal

```
class A : public action { ...
  rand_attr<bit> f1 {"f1", width(3, 0) };
  ...
};
...

class B : public action { ...
  activity a {
    sequence {
      action_handle<A>(),
      action_handle<A>().with(action_handle<A>()->f1 < 10)
    }
  };
};
...
```

Example 68—C++: Anonymous action traversal

Example 69 and Example 70 show an example of traversing a compound action as well as a random action variable field. The activity for action C traverses the random action variable field max, then traverses the action-type field b1. Evaluating this activity results in a random value being selected for max, then the sub-activity of b1 being evaluated, with a1.f1 constrained to be less than or equal to max.

```
action A {
  rand bit[3:0]   f1;
  ...
}

action B {
  A a1, a2;

  activity {
    a1;
    a2 with {
      f1 < 10;
    };
  }
}

action C {
  action bit[3:0] max;
  B b1;

  activity {
    max;
    b1 with {
      a1.f1 <= max;
    };
  }
}
```

Example 69—DSL: Compound action traversal

```
class A : public action { ...
  rand_attr<bit> f1 {"f1", width(3, 0)};
};
...

class B : public action { ...
  action_handle<A> a1{"a1"}, a2{"a2"};

  activity a {
    a1,
    a2.with(a2->f1 < 10)
  };
};
...

class C : public action { ...
  action_attr<bit> max {"max", width(3, 0)};
  action_handle<B> b1{"b1"};

  activity a {
    sequence {
      max,
      b1.with(b1->a1->f1 <= max)
    }
  };
};
...
```

Example 70—C++: Compound action traversal

13.3.2 Action handle array traversal

Arrays of action handles may be declared within an action. These *action handle arrays* may be traversed as a whole or traversed as individual elements.

The semantics of traversing individual action handle array elements are the same as those of traversing individually-declared action handles.

Example 71 below shows traversing an individual action handle array element and one action handle. The semantics of both action traversal statements are the same.

```
component pss_top {
    action A { }
    action entry {
        A   a_arr[4];
        A   a1, a2, a3, a4;
        activity {
            a_arr[0];
            a1;
        }
    }
}
```

Example 71—DSL: Individual action handle array element traversal

When an action handle array is traversed as a whole, each array element is traversed independently according to the semantics of the containing scope.

Example 72 below shows an action that traverses the elements of the `a_arr` action handle array in two ways, depending on the value of a **rand** action attribute. Both ways of traversing the elements of `a_arr` have identical semantics.

```
component pss_top {
    action A { }
    action entry {
        rand bit traverse_arr;
        A      a_arr[2];
        activity {
            if (traverse_arr) {
                a_arr;
            } else {
                a_arr[0];
                a_arr[1];
            }
        }
    }
}
```

Example 72—DSL: Action handle array traversal

The contexts in which action handle arrays may be traversed, and the resulting semantics, are described in the table below.

Table 19—Action handle array traversal contexts and semantics

Context	Semantics
parallel	All array elements are scheduled for traversal in parallel.
schedule	All array elements are scheduled for traversal independently.
select	One array element is randomly selected and traversed.
sequence	All array elements are scheduled for traversal in sequence from 0 to N-1.

13.3.3 Sequential block

An *activity sequence block* statement specifies sequential scheduling between sub-activities (see Syntax 51 and Syntax 52).

13.3.3.1 DSL syntax

```
activity_sequence_block_stmt ::= [ sequence ] { { activity_stmt } }
```

Syntax 51—DSL: Activity sequence block

The following also apply:

a) Statements in a sequential block execute in order so that one sub-activity completes before the next one starts.

b) Formally, a sequential block specifies sequential scheduling between the sets of action executions per the evaluation of *activity_stmt*$_1$ .. *activity_stmt*$_n$, keeping all scheduling dependencies within the sets and introducing additional dependencies between them to obtain sequential scheduling (see 6.3.2).

c) Sequential scheduling does not rule out other inferred dependencies affecting the nodes in the sequence block. In particular, there may be cases where additional action executions must be scheduled in between sub-activities of subsequent statements.

NOTE—PSS/C++ requires the *sequence* block to be explicitly specified, while PSS/DSL interprets a curly brace-enclosed scope as a sequence.

13.3.3.2 C++ syntax

The corresponding C++ syntax for Syntax 51 is shown in Syntax 52.

pss::action::sequence

Defined in **pss/action.h** (see C.2).

```
template <class... R> class sequence;
```

Declare a sequence block.

Member functions

```
template <class... R> sequence(R&&... /*detail::Stmt*/r) : constructor
```

Syntax 52—C++: Activity sequence block

13.3.3.3 Examples

Assume `A` and `B` are **action** types that have no rules or nested activity (see Example 73 and Example 74).

Action `my_test` specifies one execution of action `A` and one of action `B` with the scheduling dependency `(A) -> (B)`; the corresponding observed behavior is `{start A, end A, start B, end B}`.

Now assume action `B` has a state precondition which only action `C` can establish. `C` may execute before, concurrently to, or after `A`, but it shall execute before `B`. In this case the scheduling dependency relation would include `(A) -> (B)` and `(C) -> (B)` and multiple behaviors are possible, such as `{start C, start A, end A, end C, start B, end B}`.

Finally, assume also `C` has a state precondition which only `A` can establish. Dependencies in this case are `(A) -> (B)`, `(A) -> (C)` and `(C) -> (B)` (note that the first pair can be reduced) and, consequently, the only possible behavior is `{start A, end A, start C, end C, start B, end B}`.

```
action my_test {
   A a;
   B b;
   activity {
      a;
      b;
   }
};
```

Example 73—DSL: Sequential block

```
class my_test : public action { ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};
  activity act {
    a,
    b
  };
};
...
```

Example 74—C++: Sequential block

Example 75 and Example 76 show all variants of specifying sequential behaviors in an activity. By default, statements in an activity execute sequentially. The **sequence** keyword is optional, so placing sub-activities inside braces (**{}**) is the same as an explicit **sequence** statement, which includes sub-activities inside braces. The examples show a total of six sequential actions: `A`, `B`, `A`, `B`, `A`, `B`.

```
action my_test {
   A a1, a2, a3;
   B b1, b2, b3;
   activity {
      a1;
      b1;
      {a2; b2;};
      sequence{a3; b3;};
   }
};
```

Example 75—DSL: Variants of specifying sequential execution in activity

```
class my_test : public action {
  ...
  action_handle<A> a1{"a1"}, a2{"a2"}, a3{"a3"};
  action_handle<B> b1{"b1"}, b2{"b2"}, b3{"b3"};
  activity act {
    a1, b1,
    {a2, b2},
    sequence {a3, b3}
  };
};
...
```

Example 76—C++: Variants of specifying sequential execution in activity

13.3.4 parallel

The *parallel statement* specifies sub-activities that execute concurrently (see Syntax 53 and Syntax 54).

13.3.4.1 DSL syntax

```
activity_parallel_stmt ::= parallel [ activity_join_spec ] { { activity_stmt } }
```

Syntax 53—DSL: Parallel statement

The following also apply:

a) Parallel activities are invoked in a synchronized way and then proceed without further synchronization until their completion. Parallel scheduling guarantees that the invocation of an action in one sub-activity branch does not wait for the completion of any action in another.

b) Formally, the **parallel** statement specifies parallel scheduling between the sets of action executions per the evaluation of $activity\_stmt_1$ .. $activity\_stmt_n$, keeping all scheduling dependencies within the sets, ruling out scheduling dependencies across the sets, and introducing additional scheduling dependencies to initial action executions in each of the sets in order to obtain a synchronized start (see 6.3.2).

c) In the absence of an *activity_join_spec* (see 13.3.6), execution of the activity statement following the **parallel** block is scheduled to begin after all parallel branches have completed. When an *activity_join_spec* is specified, execution of the activity statement following the **parallel** block is scheduled based on the *join* specification.

13.3.4.2 C++ syntax

The corresponding C++ syntax for Syntax 53 is shown in Syntax 54.

pss::action::parallel

Defined in **`pss/action.h`** (see C.2).

```
template <class... R> class parallel;
```

Declare a parallel block.

Member functions

```
template <class... R> parallel (R&&... /*detail::Stmt*/ r) : constructor
```

Syntax 54—C++: Parallel statement

13.3.4.3 Examples

Assume `A`, `B`, and `C` are **action** types that have no rules or nested activity (see Example 77 and Example 78).

The activity in action `my_test` specifies two dependencies `(a) -> (b)` and `(a) -> (c)`. Since the executions of both `b` and `c` have the exact same scheduling dependencies, their invocation is synchronized.

Now assume action type `C` inputs a buffer object and action type `B` outputs the same buffer object type, and the input of `c` is bound to the output of `b`. According to buffer object exchange rules, the inputting action shall be scheduled after the outputting action. But this cannot satisfy the requirement of parallel scheduling, according to which an action in one branch cannot wait for an action in another. Thus, in the presence of a separate scheduling dependency between `b` and `c`, this activity shall be illegal.

```
action my_test {
   A a;
   B b;
   C c;
   activity {
      a;
      parallel {
         b;
         c;
      }
   }
};
```

Example 77—DSL: Parallel statement

```
class my_test : public action { ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};
  action_handle<C> c{"c"};
  activity act {
    a,
    parallel {
      b,
      c
    }
  };
};
...
```

Example 78—C++: Parallel statement

In Example 79 and Example 80, the semantics of the **parallel** construct require the sequences `{A,B}` and `{C,D}` to start execution at the same time. The semantics of the sequential block require that the execution of `B` follows `A` and `D` follows `C`. It is illegal to have any scheduling dependencies between sub-activities in a **parallel** statement, so neither `A` nor `B` may have any scheduling dependencies relative to either `C` or `D`.

Even though actions `A` and `D` lock the same resource type from the same pool, the pool contains a sufficient number of resource instances such that there are no scheduling dependencies between the actions. If `pool_R` contained only a single instance, there would be a scheduling dependency in that `A` and `D` could not overlap, which would violate the rules of the **parallel** statement.

```
resource R{...}
pool [4] R R_pool;
bind R_pool *;
action A { lock R r; }
action B {}
action C {}
action D { lock R r; }

action my_test {
   activity {
      parallel {
         {do A; do B;}
         {do C; do D;}
      }
   }
}
```

Example 79—DSL: Another parallel statement

```
struct R : public resource { ... };
...

pool<R> R_pool {"R_pool", 4};
bind R_bind {R_pool};

class A : public action { ... lock<R> r{"r"}; };
class B : public action { ... };
class C : public action { ... };
class D : public action { ... lock<R> r{"r"}; };
...

class my_test : public action {...
  activity act {
    parallel {
      sequence {
        action_handle<A>(),
        action_handle<B>()
      },
      sequence {
        action_handle<C>(),
        action_handle<D>()
      }
    }
  };
};
...
```

Example 80—C++: Another parallel statement

13.3.5 schedule

The **schedule** statement specifies that the PSS processing tool shall select a legal order in which to evaluate the sub-activities, provided that one exists. See Syntax 55 and Syntax 56.

13.3.5.1 DSL syntax

```
activity_schedule_stmt ::= schedule [ activity_join_spec ] { { activity_stmt } }
```

Syntax 55—DSL: Schedule statement

The following also apply:

a) All activities inside the **schedule** block shall execute, but the PSS processing tool is free to execute them in any order that satisfies their other scheduling requirements.

b) Formally, the **schedule** statement specifies that any scheduling of the combined sets of action executions per the evaluation of *activity_stmt*$_1$ .. *activity_stmt*$_n$ is permissible, as long as it keeps all scheduling dependencies within the sets and introduces (at least) the necessary scheduling dependencies across the sets in order to comply with the rules of input-output binding of actions and resource assignments.

c) In the absence of an *activity_join_spec* (see 13.3.6), execution of the activity statement following the **schedule** block is scheduled to begin after all statements within the block have completed. When an *activity_join_spec* is specified, execution of the activity statement following the **schedule** block is scheduled based on the *join* specification.

13.3.5.2 C++ syntax

The corresponding C++ syntax for Syntax 55 is shown in Syntax 56.

pss::action::schedule

Defined in **`pss/action.h`** (see C.2).

```
template <class... R> class schedule;
```

Declare a schedule block.

Member functions

`template <class... R> schedule(R&&... /*detail::Stmt*/ r)` : constructor

Syntax 56—C++: Schedule statement

13.3.5.3 Examples

Consider the code in Example 81 and Example 82, which are similar to Example 77 and Example 78, but use a **schedule** block instead of a **parallel** block. In this case, the following executions are valid:

a) The sequence of action nodes `a`, `b`, `c`.

b) The sequence of action nodes `a`, `c`, `b`.

c) The sequence of action node `a`, followed by `b` and `c` run in any order, subject to other scheduling constraints.

```
action my_test {
   A a;
   B b;
   C c;
   activity {
      a;
      schedule {
         b;
         c;
      }
   }
};
```

Example 81—DSL: Schedule statement

```
class my_test : public action { ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};
  action_handle<C> c{"c"};

  activity act {
    a,
    schedule {
      b,
      c
    }
  };
};
...
```

Example 82—C++: Schedule statement

Note that neither b nor c may start execution until after the completion of a, and the start of execution for either may be subject to additional scheduling constraints. In contrast to b and c executing in parallel, as in Example 77, there may be scheduling dependencies between b and c in the **schedule** block. The scheduling graph for the activity is shown here:

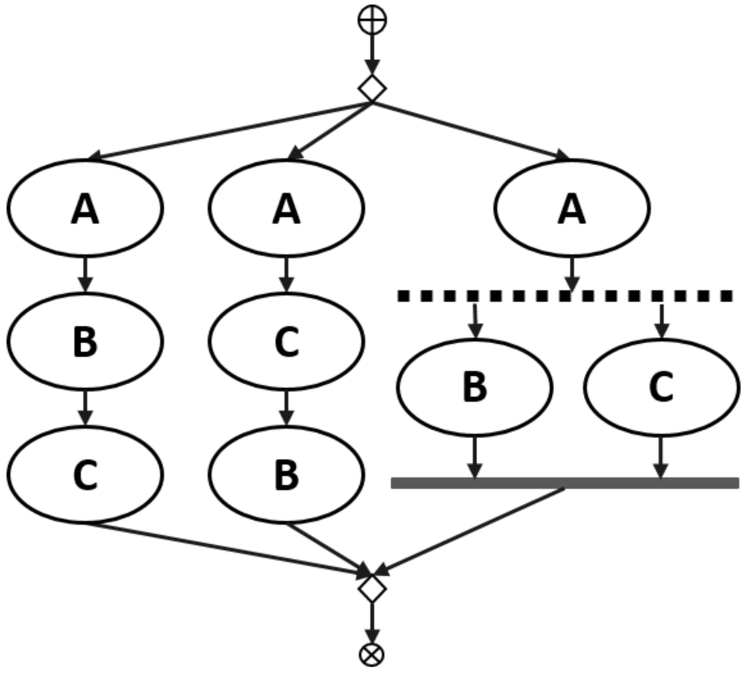

Figure 6—Scheduling graph of activity with schedule block

For the case where b and c overlap, the runtime behaviors will execute as shown here:

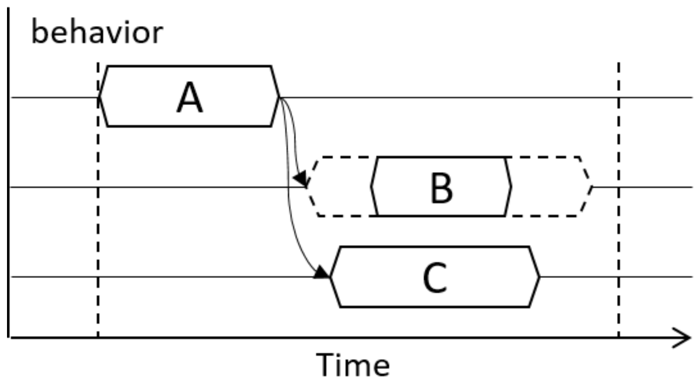

Figure 7—Runtime behavior of activity with schedule block

In contrast, consider the code in Example 83 and Example 84. In this case, any execution order in which both B comes after A and D comes after C is valid.

If both A and D wrote to the same state variable, they would have to execute sequentially. This is in addition to the sequencing of A and B and of C and D. In the case where D writes before A, the sequence would be {C, D, A, B}. In the case where A writes before D, the runtime behavior would be as shown in Figure 8.

```
action A {}
action B {}
action C {}
action D {}

action my_test {
   activity {
      schedule {
         {do A; do B;}
         {do C; do D;}
      }
   }
}
```

Example 83—DSL: Scheduling block with sequential sub-blocks

```
class A : public action { ... };
class B : public action { ... };
class C : public action { ... };
class D : public action { ... };
...

class my_test : public action { ...
  activity act {
    schedule {
      sequence {
        action_handle<A>(),
        action_handle<B>()
      },
      sequence {
        action_handle<C>(),
        action_handle<D>()
      }
    }
  };
};
...
```

Example 84—C++: Scheduling block with sequential sub-blocks

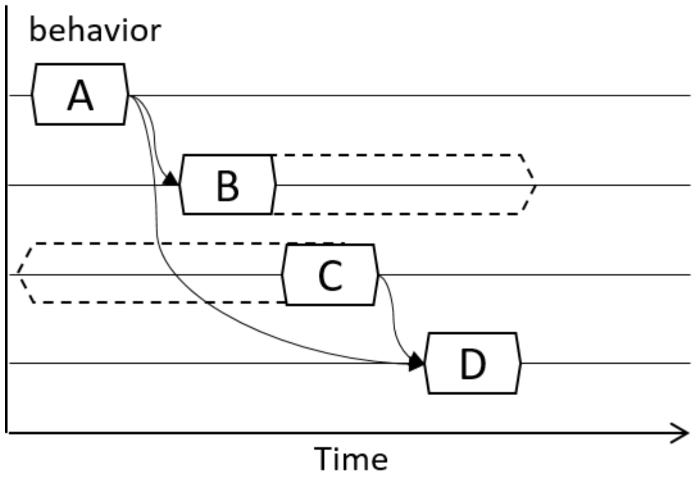

Figure 8—Runtime behavior of scheduling block with sequential sub-blocks

13.3.6 Fine-grained scheduling specifiers

Fine-grained scheduling specifiers modify the termination semantics for **parallel** and **schedule** blocks (see Syntax 53, Syntax 55, and Syntax 57). The semantics of fine-grained scheduling are defined strictly at the activity scheduling level. The semantics do not assume that any runtime execution information is incorporated by the PSS processing tool in the scheduling process. Activity scheduling in the presence of a fine-grained scheduling specifier is still subject to all other scheduling rules.

13.3.6.1 DSL syntax

```
activity_join_spec ::=
    activity_join_branch
  | activity_join_select
  | activity_join_none
  | activity_join_first
activity_join_branch ::= join_branch ( label_identifier { , label_identifier } )
activity_join_select ::= join_select ( expression )
activity_join_none ::= join_none
activity_join_first ::= join_first ( expression )
```

Syntax 57—DSL: Activity join specification

The following also apply:

a) **join_branch** accepts a list of labels referring to labeled activity statements. The activity statement following the fine-grained scheduling block is scheduled after all the listed activity statements have completed.
   1) The *label_identifier* used in the **join_branch** specification must be the label of a top-level branch within the **parallel** or **schedule** block to which the **join_branch** specification is applied.
   2) When the *label_identifier* used in the **join_branch** specification applies to traversal of an array, the activity statement following the fine-grained scheduling block is scheduled after all actions in the array have completed.

b) **join_select** accepts an *expression* specifying the number of top-level activity statements within the fine-grained scheduling block on which to condition execution of the activity statement following the fine-grained scheduling block. The specific activity statements shall be selected randomly. Execution of the activity statement following the fine-grained scheduling block is scheduled after the selected activity statements.
   1) The *expression* shall be of a numeric type. The value of the *expression* must be determinable at solve time. If the value is 0, the **join_select** is equivalent to **join_none**.
   2) When an action array is traversed, each element of the array is considered a separate action that may be selected independently.

c) **join_none** specifies that the activity statement following the fine-grained scheduling block has no scheduling dependency on activity statements within the block.

d) **join_first** specifies that the activity statement following the fine-grained scheduling block has a run-time execution dependency on the first N activity statements within the fine-grained scheduling block to complete execution. The activity statement following the fine-grained scheduling block has no scheduling dependency on activity statements within the block, only a runtime dependency.

1) The *expression* shall be of a numeric type. The value of the *expression* must be determinable at solve time. If the value is 0, the **join_first** is equivalent to **join_none**.
2) When an action array is traversed, each element of the array is considered a separate action that may be selected independently.

The application scope of a fine-grained scheduling block is bounded by the sequential block that contains it. In other words, all activity statements that start within the fine-grained scheduling block must complete before the statement following the containing sequential block begins. Activities started, but not joined, within a fine-grained scheduling block are not implicitly waited for by any containing parallel or schedule blocks. Only the containing sequential block causes a join on activities started within it.

NOTE—There is no PSS/C++ equivalent for the fine-grained scheduling modifiers to either **parallel** or **schedule**.

13.3.6.2 Examples

In Example 85, the innermost parallel block (`L4`) starts two activities (`L5` and `L6`), while only waiting for one (`L5`) to complete before continuing. Since `L5` traverses the action array `b`, all elements of `b` must complete before continuing. The next level of parallel block (`L2`) waits for its two branches to complete (`L3` and `L4`), but does not wait for `L6` to complete. The outermost parallel block (`L1`) waits for one of its branches (`L2`) to complete before proceeding. This means that both `L7` and `L6` may be in-flight when `L8` is traversed.

```
B b[2];
activity {
   L1: parallel join_branch(L2) {
      L2: parallel {
         L3: do A;
         L4: parallel join_branch (L5) {
            L5: b;
            L6: do C;
         }
      }
      L7: do D;
   }
   L8: do F;
}
```

Example 85—DSL: join_branch

The scheduling graph of the activity is shown in Figure 9.

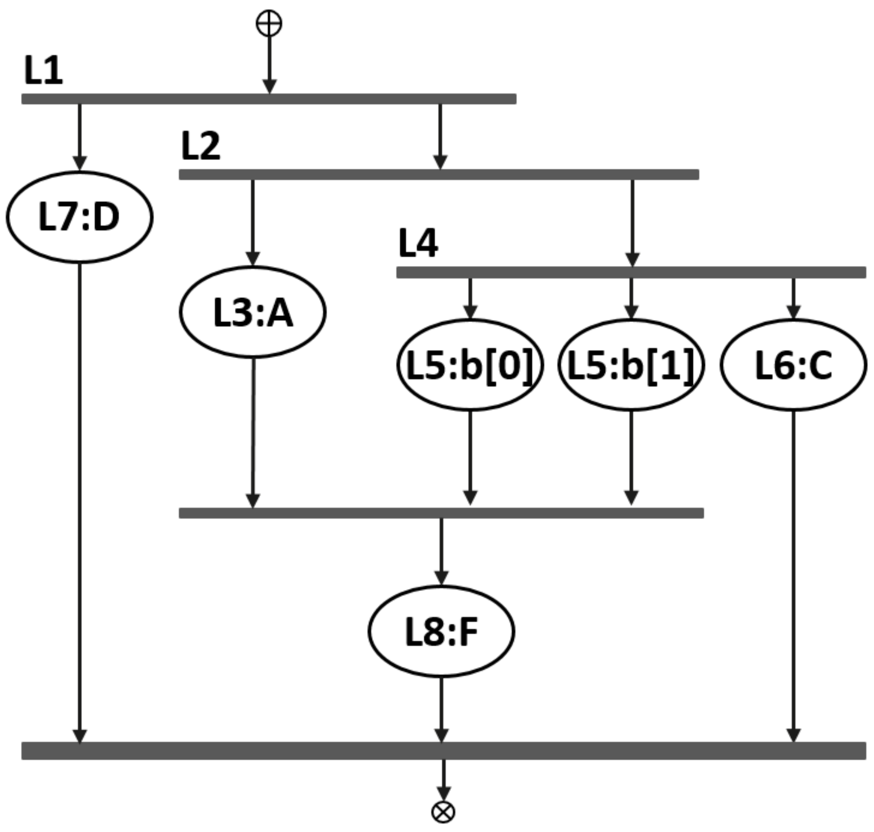

Figure 9—join_branch scheduling graph

The runtime behavior is shown in Figure 10.

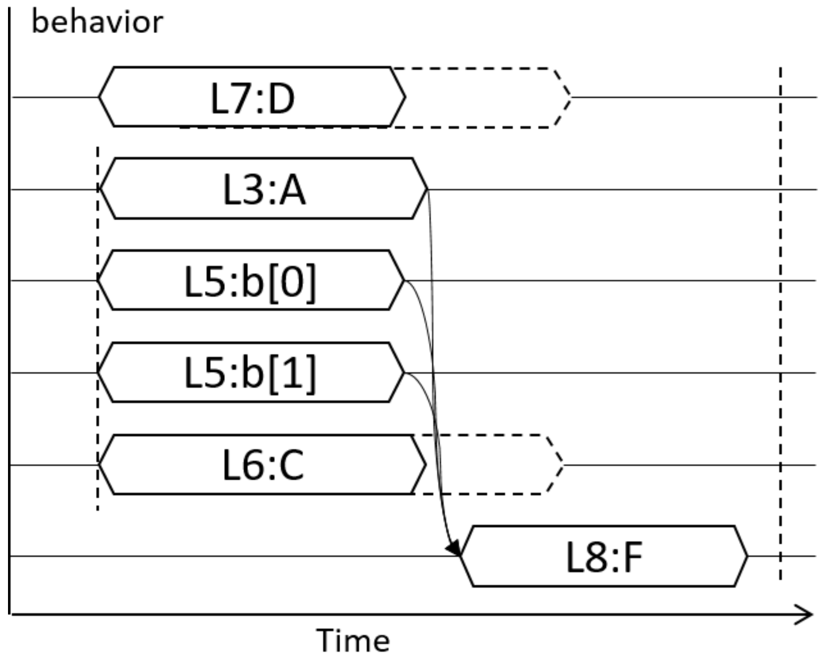

Figure 10—join_branch runtime behavior

Activity scheduling in the presence of a fine-grained scheduling block is still subject to all other scheduling rules. For example, if both `L6` and `L8` in the example above contend for the same single resource, they must be scheduled sequentially in order to avoid a resource conflict.

For the following four examples, assume that each of the three actions in the activity locks a resource from the same pool.

In Example 86, the **parallel** block causes traversal of branches `L1` and `L2` to be scheduled in parallel. The **join_branch** specifier causes traversal of action `C` to be scheduled with a sequential dependency on the activity statement labeled `L2`. Traversal of action `C` may not begin until the activity statement labeled `L2` has completed. To avoid adding additional scheduling dependencies, the resource pool would need a minimum of two resource instances. Actions `A` and `B` would each lock a resource instance, and `C`, since it is guaranteed not to start until `A` completes, would lock the same resource instance as that assigned to `A`. Note that this allocation is handled at solve-time, and is independent of whether `B` completes before or after `A` completes.

```
activity {
  L1 : parallel join_branch(L2) {
    L2: do A;
    L3: do B;
  }
  L4: do C;
}
```

Example 86—DSL: join_branch with scheduling dependency

The scheduling graph of the activity is shown in Figure 11.

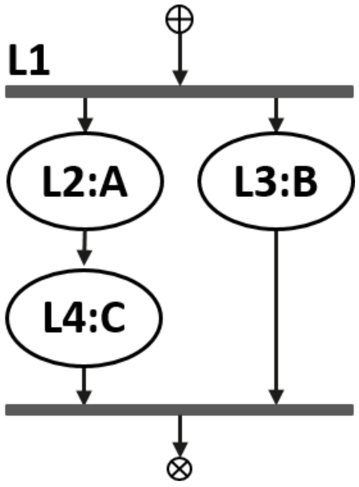

Figure 11—Scheduling graph of join_branch with scheduling dependency

The runtime behavior is shown in Figure 12.

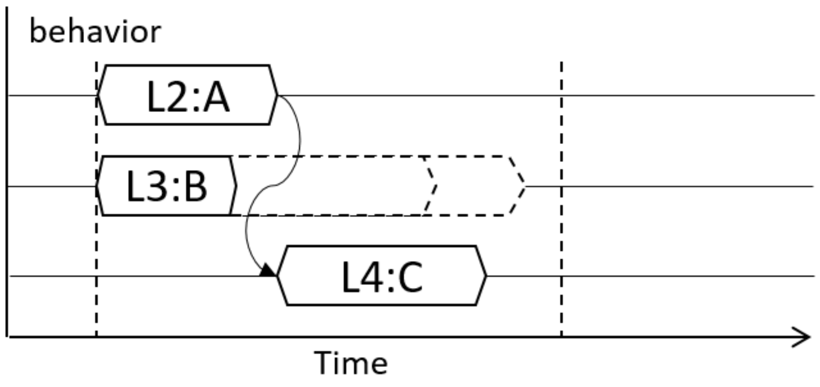

Figure 12—Runtime behavior of join_branch with scheduling dependency

In Example 87, the **parallel** block causes traversal of the branches labeled L2 and L3 to be scheduled in parallel. The **join_select** specifier causes traversal of action C to be scheduled with a sequential dependency on a random selection of either the branch labeled L2 or L3. This means that traversal of C may not begin until after the selected target activity statement has completed. The tool randomly selects N (in this case, 1) target branch(es) from the candidate branches on which to make traversal of the following activity statement dependent.

In this example, the resource pool would need a minimum of two resource instances. Because the tool may not know which of A or B will complete first, it must choose one and assign the same resource instance to action C. If the tool selected L2 as the branch on which C depends, the behavior would be identical to the previous example.

```
activity {
  L1 : parallel join_select(1) {
    L2: do A;
    L3: do B;
  }
  L4: do C;
}
```

Example 87—DSL: join_select

In Example 88, the **join_none** specifier causes traversal of action C to be scheduled with no dependencies. To avoid additional scheduling dependencies, the minimum size of the resource pool must be three, since each action traversed in the activity must have a unique resource instance.

Actions A and B are scheduled in parallel, and action C is scheduled concurrently with both of them. This means that C *could* start at the same time as A and B, but it may not. While the **parallel** statement precludes any dependencies between A and B, the **join_none** qualifier allows action C to be scheduled concurrently, but there may be additional dependencies between action C and action A and/or B.

```
activity {
  L1 : parallel join_none {
    L2: do A;
    L3: do B;
  }
  L4: do C;
}
```

Example 88—DSL: join_none

The scheduling graph of the activity is shown in Figure 13.

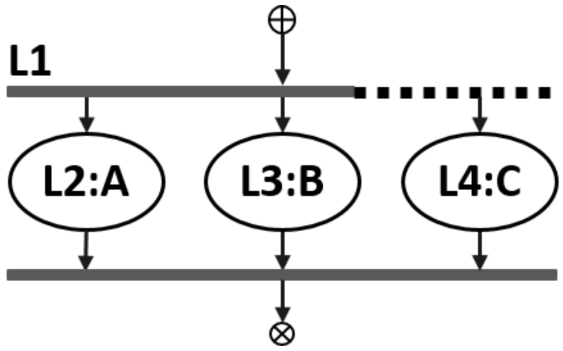

Figure 13—join_none scheduling graph

In Example 89, the **join_first** specifier causes the PSS processing tool to condition execution of action C on runtime execution completion of the first of either action `A` or `B`. Since the scheduling tool may not know which action will complete first, there must be a minimum of three resource instances in the pool in order to guarantee that `C` may execute immediately after whichever of `A` or `B` completes first. If there are two instances in the pool, the tool may assign either resource instance to `C` at solve-time. If the other action assigned the same resource instance completes last, then action `C`, because it starts execution after the previous action completes, will also start its execution after the completion of the first action.

```
activity {
  L1 : parallel join_first(1) {
    L2: do A;
    L3: do B;
  }
  L4: do C;
}
```

Example 89—DSL: join_first

The runtime behavior is shown in Figure 14.

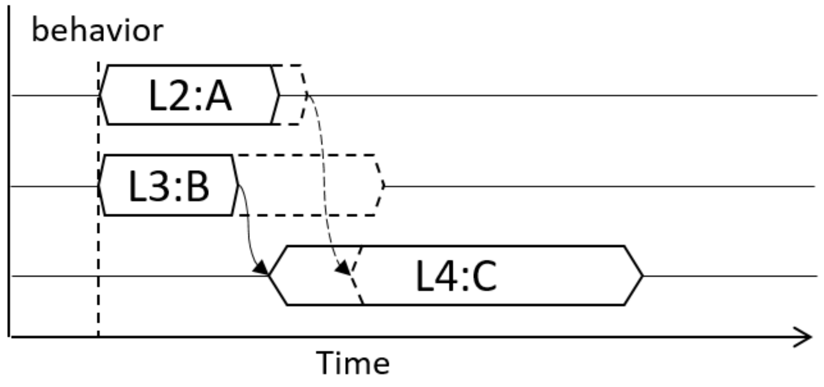

Figure 14—join_first runtime behavior

Example 90 illustrates how a **sequence** block bounds the impact of the fine-grained scheduling specifier. The execution of `L5` is scheduled in sequence with `L3`. `L4` and `L5` may be scheduled concurrently. `L6` is scheduled strictly sequentially to all statements inside `L1`, the **sequence** block.

```
activity {
   L1: sequence {
      L2: parallel join_branch(L3) {
         L3: do A;
         L4: do B;
      }
      L5: do C;
   }
   L6: do D;
}
```

Example 90—DSL: Scope of join inside sequence block

The scheduling graph is shown in Example 15.

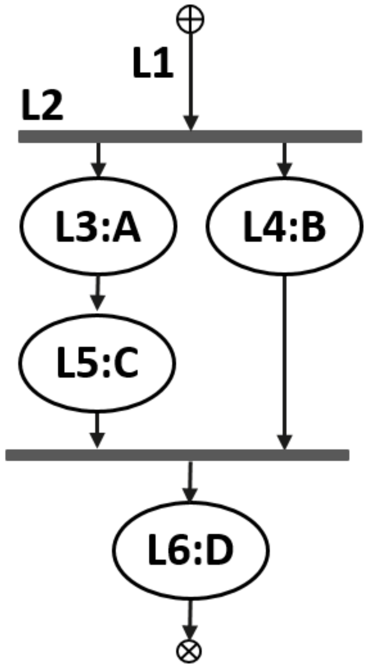

Figure 15—Scheduling graph of join inside sequence block

The runtime behavior is shown in Figure 16.

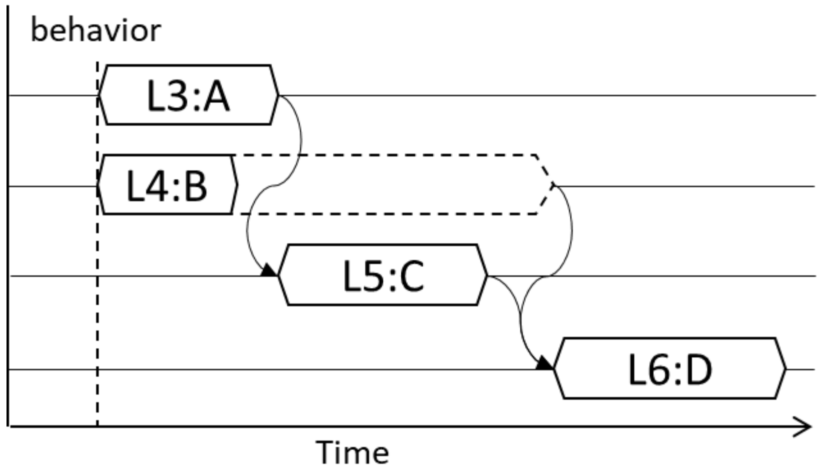

Figure 16—Runtime behavior of join inside sequence block

Example 91 shows how the **join** specification may also be used with the **schedule** block.

```
    activity {
      L1 : schedule join_branch(L2) {
        L2: do A;
        L3: do B;
      }
      L4: do C;
    }
```

Example 91—DSL: join with schedule block

Assuming there are no scheduling dependencies between actions `A` and `B`, the scheduling graph of **schedule** block `L1` is shown in Figure 17.

In all cases, action `C` is scheduled subsequent to action `A`. If `A` is scheduled before `B`, then `B` and `C` may–or may not–be scheduled concurrently, although there may be additional dependencies between them. If `B` is scheduled before `A`, the actions are executed in the order `B`, `A`, `C`. If `A` and `B` are scheduled concurrently, then `C` is still scheduled after `A`, but again may be concurrent with `B`, subject to any dependencies between `B` and `C`.

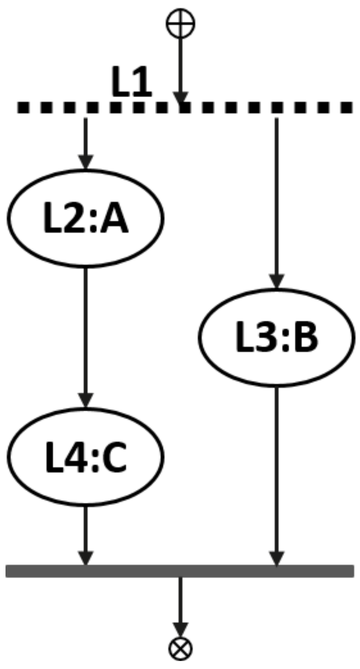

Figure 17—Scheduling graph join with schedule block

13.4 Activity control flow constructs

In addition to defining sequential and parallel blocks of action execution, repetition and branching statements can be used inside the **activity** clause.

13.4.1 repeat (count)

The **repeat** statement allows the specification of a loop consisting of one or more actions inside an activity. This section describes the *count-expression* variant (see Syntax 58 and Syntax 59) and 13.4.2 describes the *while-expression* variant.

13.4.1.1 DSL syntax

```
activity_repeat_stmt ::=
    repeat ( [ index_identifier : ] expression ) activity_stmt
  | . . .
```

Syntax 58—DSL: repeat-count statement

The following also apply:

a) *expression* shall be a non-negative numeric expression (**int** or **bit**).

b) Intuitively, the *activity_stmt* is iterated the number of times specified in the *expression*. An optional index-variable identifier can be specified that ranges between `0` and one less than the iteration count. If the expression evaluates to `0`, the *activity_stmt* is not evaluated at all.

c) Formally, the **repeat-**count statement specifies sequential scheduling between *N* sets of action executions per the evaluation of *activity_stmt* *N* times, where *N* is the number to which *expression* evaluates (see 6.3.2).

d) The choice of values to **rand** attributes figuring in the *expression* shall be such that it yields legal execution scheduling.

13.4.1.2 C++ syntax

The corresponding C++ syntax for Syntax 58 is shown in Syntax 59.

pss::repeat

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class repeat;
```

Declare a repeat statement.

Member functions

```
repeat (const detail::AlgebExpr& count,
    const detail::Stmt& activity) : declare a repeat (count) activity
repeat (const attr<int>& iter, const detail::AlgebExpr& count,
    const detail::Stmt& activity) : declare a repeat (count) activity with iterator
```

Syntax 59—C++: repeat-count statement

13.4.1.3 Examples

In Example 92 and Example 93, the resulting execution is six sequential action executions, alternating `A`'s and `B`'s, with five scheduling dependencies: `(`A_0`) -> (`B_0`), (`B_0`) -> (`A_1`), (`A_1`) -> (`B_1`), (`B_1`) -> (`A_2`), (`A_2`) -> (`B_2`).`

```
action  my_test {
   A a;
   B b;
   activity {
      repeat (3) {
         a;
         b;
      }
   }
};
```

Example 92—DSL: repeat statement

```
class my_test : public action { ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};

  activity act {
    repeat { 3,
      sequence { a, b }
    }
  };
};
...
```

Example 93—C++: repeat statement

Example 94 and Example 95 show an additional example of using **repeat**-count.

```
action my_test {
    my_action1 action1;
    my_action2 action2;
    activity {
      repeat (i : 10) {
         if ((i % 4) == 0) {
           action1;
         } else {
           action2;
         }
      }
   }
};
```

Example 94—DSL: Another repeat statement

```
class my_test : public action { ...
  action_handle<my_action1> action1{"action1"};
  action_handle<my_action2> action2{"action2"};
  attr<int> i {"i"};

  activity act {
    repeat { i, 10,
      if_then_else {
        cond((i % 4) == 0), action1, action2
      }
    }
  };
};
...
```

Example 95—C++: Another repeat statement

13.4.2 repeat-while

The **repeat** statement allows the specification of a loop consisting of one or more actions inside an activity. This section describes the *while-expression* variant (see Syntax 60 and Syntax 61).

13.4.2.1 DSL syntax

```
activity_repeat_stmt ::=
  . . .
  | repeat activity_stmt while ( expression ) ;
```

Syntax 60—DSL: repeat-while statement

The following also apply:

a) *expression* shall be of type **bool**.
b) Intuitively, the *activity_stmt* is iterated so long as the *expression* condition is *true*, as sampled after the *activity_stmt*.
c) Formally, the **repeat-while** statement specifies sequential scheduling between multiple sets of action executions per the iterative evaluation of *activity_stmt*. The evaluation of *activity_stmt* continues repeatedly so long as *expression* evaluates to *true*. *expression* is evaluated after the execution of each set in the **repeat-while** block.

13.4.2.2 C++ syntax

The corresponding C++ syntax for Syntax 60 is shown in Syntax 61.

pss::repeat_while

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class repeat_while;
```

Declare a repeat-while activity.

Member functions

```
repeat_while (const detail::Stmt& activity, const cond& a_cond)
```
: constructor

Syntax 61—C++: repeat-while statement

13.4.2.3 Examples

```
component top {

   function bit is_last_one();

   action do_something {
      bit last_one;

      exec post_solve {
         last_one = comp.is_last_one();
      }

      exec body C = """
         printf("Do Something\n");
      """;
   }

   action entry {
      do_something s1;

      activity {
         repeat {
            s1;
         } while (s1.last_one !=0);
      }
   }
}
```

Example 96—DSL: repeat-while statement

```
class top : public component { ...
  function<result<bit> ()> is_last_one {"is_last_one", result<bit>()};

  class do_something : public action { ...
    attr<bit> last_one {"last_one"};
    exec post_solve {exec::post_solve,
      last_one = comp<top>()->is_last_one()
    };

    exec body {exec::body, "C",
      "printf(\"Do Something\n\");"
    };
  };
  type_decl<do_something> do_something_t;

  class entry : public action { ...
    action_handle<do_something> s1{"s1"};
    activity act {
      repeat_while {s1, s1->last_one != 0}
    };
  };
  type_decl<entry> entry_t;
};
...
```

Example 97—C++: repeat-while statement

13.4.3 foreach

The **foreach** construct iterates over the elements of a collection (see Syntax 62 and Syntax 63). See also Example 98 and Example 99.

13.4.3.1 DSL syntax

activity_foreach_stmt ::=
foreach ([*iterator*_identifier **:**] expression [**[** *index*_identifier **]**] **)** activity_stmt

Syntax 62—DSL: foreach statement

The following also apply:

a) *expression* shall be of a collection type (i.e., **array**, **list**, **map** or **set**).
b) The body of the **foreach** statement is a sequential block in which *activity_stmt* is evaluated once for each element in the collection.
c) *iterator_identifier* specifies the name of an iterator variable of the collection element type. Within *activity_stmt*, the iterator variable, when specified, is an alias to the collection element of the current iteration.
d) *index_identifier* specifies the name of an index variable. Within *activity_stmt,* the index variable, when specified, corresponds to the element index of the current iteration.
   1) For **array**s and **list**s, the index variable shall be a variable of type **int**, ranging from `0` to one less than the size of the collection variable, in that order.
   2) For **map**s, the index variable shall be a variable of the same type as the **map** keys, and range over the values of the keys. The order of key traversal is undetermined.

    3) For **set**s, an index variable shall not be specified.

e) Both the index and iterator variables, if specified, are implicitly declared within the **foreach** scope and limited to that scope. Regular name resolution rules apply when the implicitly declared variables are used within the **foreach** body. For example, if there is a variable in an outer scope with the same name as the index variable, that variable is shadowed (masked) by the index variable within the **foreach** body. The index and iterator variables are not visible outside the **foreach** scope.

f) Either an index variable or an iterator variable or both shall be specified. For a **set**, an iterator variable shall be specified, but not an index variable.

13.4.3.2 C++ syntax

The corresponding C++ syntax for Syntax 62 is shown in Syntax 63.

pss::foreach

Defined in **`pss/foreach.h`** (C.29).

```
class foreach;
```

Iterate activity across array of non-rand and rand attributes.

Member functions

```
foreach (const attr& iter, const attr<vec<T>>& array,
    const detail::Stmt& activity)
```
: non-rand attributes (specializations provided for T = int and bit)

```
foreach (const attr& iter, const rand_attr<vec<T>>& array,
    const detail::Stmt& activity)
```
: rand attributes (specializations provided for T = int and bit)

Syntax 63—C++: foreach statement

NOTE—Only iteration over arrays is supported in PSS/C++. foreach iteration over other collection types is not supported.

NOTE—In PSS/C++, the index and iteration variables must be explicitly declared in the containing scope of the foreach loop.

13.4.3.3 Examples

```
action my_action1 {
    rand bit[4] val;
    // ...
}

action my_test {
    rand bit[4] in [0..7] a[16];
    my_action1 action1;

    activity {
      foreach (a[j]) {
        action1 with {val <= a[j];};
      }
    }
};
```

Example 98—DSL: foreach statement

```
class my_action1 : public action { ...
  rand_attr<bit> val {"val", width(4)};
};
...

class my_test : public action { ...
  rand_attr_vec<bit> a {"a", 16, width(4), range(0,7)};
  attr<int> j {"j"};

  action_handle<my_action1> action1 {"action1"};

  activity act {
    foreach {j, a,
      action1.with (action1->val <= a[j])
    }
  };
};
...
```

Example 99—C++: foreach statement

13.4.4 select

The **select** statement specifies a branch point in the traversal of the activity (see Syntax 64 and Syntax 65).

13.4.4.1 DSL syntax

```
activity_select_stmt ::= select { select_branch select_branch { select_branch } }
select_branch ::= [ [ ( expression ) ] [ [ expression ] ] : ] activity_stmt
```

Syntax 64—DSL: select statement

The following also apply:

a) Intuitively, a **select** statement executes one out of a number of possible activities.

b) One or more of the *activity_stmt*s may optionally have a guard condition specified in parentheses (()). Guard condition expressions shall be of Boolean type. When the **select** statement is evaluated, only those *activity_stmt*s whose guard condition evaluates to *true* or that do not have a guard condition are considered enabled.

c) Formally, each evaluation of a **select** statement corresponds to the evaluation of just one of the *select_branch* statements. All scheduling requirements shall hold for the selected **activity** statement.

d) Optionally, all *activity_stmt*s may include a *weight expression*, which is a numeric expression that evaluates to a positive integer. The probability of choosing an enabled *activity_stmt* is the weight of the given statement divided by the sum of the weights of all enabled statements. If the *activity_stmt* is an array of action handles, then the *weight expression* is assigned to each element of the array, from which one element is selected and traversed.

e) If any *activity_stmt* has a *weight expression*, then any statement without an explicit *weight expression* associated with it shall have a weight of `1`.

f) It shall be illegal if no **activity** statement is valid according to the active constraint and scheduling requirements and the evaluation of the guard conditions.

13.4.4.2 C++ syntax

The corresponding C++ syntax for Syntax 64 is shown in Syntax 65.

pss::action::select

Defined in **`pss/action.h`** (see C.2).

```
template <class... R> class select;
```

Declare a select statement.

Member functions

```
template <class... R> select (R&&... /*detail::Stmt*/ r)  : constructor
```

pss::action::branch

Defined in **`pss/action.h`** (see C.2).

```
class branch;
```

Specify a select branch.

Member functions

```
template <class R> branch (const R&... /*detail::Stmt*/ r)  : constructor
template <class R> branch (const guard &g,
    const R& /*detail::Stmt*/ r)  : constructor
template <class R> branch (const guard &g, const weight &w,
    const R& /*detail::Stmt*/ r)  : constructor
template <class R> branch (const weight &w,
    const R& /*detail::Stmt*/ r)  : constructor
```

Syntax 65—C++: select statement

13.4.4.3 Examples

In Example 100 and Example 101, the **select** statement causes the activity to select `action1` or `action2` during each execution of the activity.

```
action my_test {
  my_action1         action1;
  my_action2         action2;
  activity {
    select {
      action1;
      action2;
    }
  }
}
```

Example 100—DSL: Select statement

```
class my_test : public action { ...
  action_handle<my_action1> action1{"action1"};
  action_handle<my_action2> action2{"action2"};

  activity act {
    select {
      action1,
      action2
    }
  };
};
...
```

Example 101—C++: Select statement

In Example 102 and Example 103, the branch selected shall depend on the value of `a` when the **select** statement is evaluated.

a) `a==0` means that all three branches could be chosen, according to their weights.
   1) `action1` is chosen with a probability of 20%.
   2) `action2` is chosen with a probability of 30%.
   3) `action3` is chosen with a probability of 50%.

b) `a in [1..3]` means that `action2` or `action3` is traversed according to their weights.
   1) `action2` is chosen with a probability of 37.5%.
   2) `action3` is chosen with a probability of 62.5%.

c) `a==4` means that only `action3` is traversed.

```
action my_test {
  my_action1 action1;
  my_action2 action2;
  my_action3 action3;
  rand int in [0..4] a;
  activity {
    select {
      (a == 0      )[20]: action1;
      (a in [0..3])[30]: action2;
                   [50]: action3;
    }
  }
}
```

Example 102—DSL: Select statement with guard conditions and weights

```
class my_test : public action { ...
  action_handle<my_action1> action1 {"action1"};
  action_handle<my_action2> action2 {"action2"};
  action_handle<my_action3> action3 {"action3"};
  rand_attr<int> a {"a", range(0,4)};

  activity act {
    select {
      branch {guard(a == 0), weight(20), action1},
      branch {guard(in(a, range(0,3))), weight(30), action2},
      branch {weight(50), action3}
    }
  };
};
...
```

Example 103—C++: Select statement with guard conditions and weights

In Example 104, the **select** statement causes the activity to select `action1` or one element of `action2` during the execution of the **activity**. Since the weight expression of `2` is applied to each element of the `action2` array, there is a 40% chance that either element of that array is chosen, and a 20% (weight of `1`) chance of choosing `action1`.

```
action my_test {
  my_action1 action1;
  my_action2 action2[2];

  activity {
    select {
           action1;
      [2]: action2;
    }
  }
}
```

Example 104—DSL: Select statement with array of action handles

13.4.5 if-else

The **if-else** statement introduces a branch point in the traversal of the activity (see Syntax 66 and Syntax 67).

13.4.5.1 DSL syntax

activity_if_else_stmt ::= **if (** expression **)** activity_stmt [**else** activity_stmt]

Syntax 66—DSL: if-else statement

The following also apply:

a) *expression* shall be of type **bool**.
b) Intuitively, an **if-else** statement executes some activity if a condition holds, and, otherwise (if specified), the alternative activity.
c) Formally, the **if-else** statement specifies the scheduling of the set of action executions per the evaluation of the first *activity_stmt* if *expression* evaluates to *true* or the second *activity_stmt* (following **else**) if present and *expression* evaluates to *false*.
d) The scheduling relationships need only be met for one branch for each evaluation of the activity.
e) The choice of values to **rand** attributes figuring in the *expression* shall be such that it yields legal execution scheduling.

13.4.5.2 C++ syntax

The corresponding C++ syntax for Syntax 66 is shown in Syntax 67.

pss::if_then

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then;
```

Declare if-then activity statement.

Member functions

`if_then (const cond& a_cond, const detail::Stmt& true_expr)` : constructor

pss::if_then_else

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then_else;
```

Declare if-then-else activity statement.

Member functions

```
if_then_else (const cond& a_cond, const detail::Stmt& true_expr,
   const detail::Stmt& false_expr)
```
: constructor

Syntax 67—C++: if-else statement

13.4.5.3 Examples

If the scheduling requirements for Example 105 and Example 106 required selection of the `b` branch, then the value selected for `x` must be `<= 5`.

```
action my_test {
   rand int in [1..10] x;
   A a;
   B b;
   activity {
      if (x > 5)
         a;
      else
         b;
   }
};
```

Example 105—DSL: if-else statement

```
class my_test : public action { ...
  rand_attr<int> x { "x", range(1,10) };
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};

  activity act {
    if_then_else {
      cond(x > 5), a, b
    }
  };
};
...
```

Example 106—C++: if-else statement

13.4.6 match

The **match** statement specifies a multi-way decision point in the traversal of the activity that tests whether an expression matches any of a number of other expressions and traverses one of the matching branches accordingly (see Syntax 68 and Syntax 69).

13.4.6.1 DSL syntax

```
activity_match_stmt ::= match ( match_expression ) { match_choice { match_choice } }
match_expression ::= expression
match_choice ::=
    [ open_range_list ] : activity_stmt
  | default : activity_stmt
```

Syntax 68—DSL: match statement

The following also apply:

a) When the **match** statement is executed, the *match_expression* is evaluated.

b) After the *match_expression* is evaluated, the *open_range_list* of each *match_choice* shall be compared to the *match_expression*. *open_range_list*s are described in 9.5.9.1.

c) If there is exactly one match, then the corresponding branch shall be traversed.

d) If there is more than one match, then one of the matching *match_choice*s shall be randomly traversed.

e) If there are no matches, then the **default** branch, if provided, shall be traversed.

f) The **default** branch is optional. There may be at most one **default** branch in the **match** statement.

g) As with a **select** statement, it shall be an error if no *match_choice* is valid according to the active constraint and scheduling requirements and the evaluation of the *match_expression* against the *match_choice open_range_list*s.

13.4.6.2 C++ syntax

The corresponding C++ syntax for Syntax 68 is shown in Syntax 69.

pss::match

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
   class match;
```

Declare a match statement.

Member functions

```
   template <class... R> match (const cond &expr,
      R&&... /* choice|default_choice */ stmts)
```
: constructor

pss::choice

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
   class choice;
```

Declare a match choice statement.

Member functions

```
   template <class... R> choice (const range &range_expr,
      R&&... /* detail::Stmt */ choice_stmts)
```
: constructor

pss::default_choice

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
   class default_choice;
```

Declare a match default choice statement.

Member functions

```
   template <class... R> default_choice
       R&&... /* detail::Stmt */ choice_stmts)
```
: constructor

Syntax 69—C++: match statement

13.4.6.3 Examples

In Example 107 and Example 108, the **match** statement causes the **activity** to evaluate the data field in_security_data.val and select a branch according to its value at each execution of the activity. If the data field is equal to LEVEL2, action1 is traversed. If the data field is equal to LEVEL5, action2 is traversed. If the data field is equal to LEVEL3 or LEVEL4, then either action1 or action2 is traversed at random. For any other value of the data field, action3 is traversed.

```
action my_test {
  rand security_data in_security_data;
  my_action1 action1;
  my_action2 action2;
  my_action3 action3;
  activity {
    match (in_security_data.val) {
      [LEVEL2..LEVEL4]:
        action1;
      [LEVEL3..LEVEL5]:
        action2;
      default:
        action3;
    }
}
```

Example 107—DSL: match statement

```
class my_test : public action{...
  rand_attr<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 108—C++: match statement

13.5 Activity construction statements

13.5.1 replicate

The **replicate** statement is a generative activity statement interpreted as an in-place expansion of a specified statement multiple times. The **replicate** statement does not introduce an additional layer of scheduling or control flow. The execution semantics applied to the expanded statements depend on the context. In particular, replicating a statement N times under a **parallel** statement executes the same statement N times in parallel. Unlike a **repeat** statement, **replicate** provides a way to reference specific expansion instances from above using a label array.

13.5.1.1 DSL syntax

```
activity_replicate_stmt ::=
  replicate ( [ index_identifier : ] expression ) [ label_identifier [ ] : ] labeled_activity_stmt
```

Syntax 70—DSL: replicate statement

The following also apply:

a) *expression* shall be a positive numeric expression (**int** or **bit**).

b) The **replicate** statement expands in-place to *labeled_activity_stmt* replicated the number of times specified in the *expression*. An optional index variable *index_identifier* may be specified that ranges between 0 and one less than the iteration count.

c) The execution semantics of a **replicate** statement where *expression* evaluates to N are equivalent to the execution semantics of N occurrences of *labeled_activity_stmt* directly traversed in its enclosing activity scope.

d) The number of replications must be known as part of the solve process. In other words, *expression* may not contain an attribute that is assigned in the context of a runtime *exec block* (**body**/**run_start**/**run_end**).

e) A *label_identifier* may optionally be used to label the replicated statement in the form of a label array. If used, each expanded occurrence of *labeled_activity_stmt* becomes a named sub-activity with the label *label_identifier*[0] … *label_identifier*[N-1] respectively, where N is the number of expanded occurrences. Reference can be made to labels and action handles declared under the **replicate** and its nested scopes using array indexing on the label. (See more on hierarchical activity references in 13.8).

f) Labels may be used to name sub-activities inside the scope of a **replicate** statement only if the *label_identifier* is specified. A label under a **replicate** statement without a named label array leads to name conflict between the replicated sub-activities (see scoping rules for named sub-activities in 13.8.2).

g) Inside the scope of a **replicate** statement there shall not be an action traversal statement using an action handle declared outside the **replicate** scope. Both anonymous action traversal and action traversal of an action handle declared locally inside the **replicate** scope are allowed.

13.5.1.2 C++ syntax

The corresponding C++ syntax for Syntax 70 is shown in Syntax 71.

pss::action::replicate

Defined in **`pss/action.h`** (see C.2).

```
   class replicate;
```

Declare a replicate statement.

Member functions

```
   replicate (const detail::AlgebExpr& count,
      const detail::Stmt& activity)
```
: declare a replicate (count) activity

```
   replicate (const attr<int>& iter, const detail::AlgebExpr& count,
      const detail::Stmt& activity)
```
: declare a replicate (count) activity with iterator

Syntax 71—C++: replicate statement

13.5.1.3 Examples

In Example 109, the resulting execution is either two, three, or four parallel executions of the sequence `A -> B`.

```
   action my_test {
     rand int in [2..4] count;
     activity {
       parallel {
         replicate (count) {
           do A;
           do B;
         }
       }
     }
   };
```

Example 109—DSL: replicate statement

Example 110 is the C++ equivalent of the previous example.

```
class my_test : public action { ...
  rand_attr<int> count {"count", range(2,4)};
  activity act {
    replicate { count,
      sequence {
        action_handle<A>(),
        action_handle<B>()
      }
    }
  };
};
...
```

Example 110—C++: replicate statement

In Example 111, the execution of action `my_test` results in one execution of `A` as well as four executions of `B`, all in the scope of the **schedule** statement, that is, invoked in any order that satisfies the scheduling rules.

```
action my_test {
  activity {
    schedule {
      do A;
      replicate (i: 4) do B with { size == i*10; };
    }
  }
};
```

Example 111—DSL: replicate statement with index variable

Example 111 can be rewritten in the following equivalent way to eliminate the **replicate** statement:

```
action my_test {
  activity {
    schedule {
      do A;
      do B with { size == 0*10; };
      do B with { size == 1*10; };
      do B with { size == 2*10; };
      do B with { size == 3*10; };
    }
  }
};
```

Example 112—DSL: Rewriting previous example without replicate statement

Example 113 illustrates the use of a **replicate** label array for unique hierarchical paths to specific expansion instances. References are made to action handles declared and traversed under specific expansion instances of a **replicate** statement from outside its scope.

```
action my_compound {
  rand int in [2..4] count;
  activity {
    parallel {
      replicate (count) RL[]: {
        A a;
        B b;
        a;
        b;
      }
    }
    if (RL[count-1].a.x ==0) { // 'a' of the last replicate expansion
      do C;
    }
  }
};

action my_test {
  activity {
    do my_compound with {
      RL[0].a.x == 10; // 'a' of the first replicate expansion
    };
  }
};
```

Example 113—DSL: replicate statement with label array

In Example 114 a number of error situations are demonstrated. Note that label `L` in this example causes a name conflict between the named sub-activities in the expansion of the **replicate** statement (see also 13.8.2).

```
action my_test {
  A a;
  activity {
    schedule {
      replicate (4) {
        B b;
        a; // Error - traversal of action handle
           // declared outside the replicate scope
        b; // OK - action handle declared inside the replicate scope
        L: select { // Error - label causes name conflict in expansion
          do A;
          do B;
        }
      }
    }
  }
};
```

Example 114—DSL: replicate statement error situations

13.6 Activity evaluation with extension and inheritance

Compound actions support both type inheritance and type extension (see Clause 20). When type extension is used to contribute one or more activities to an action type, the execution semantics are the same as if all the contributed activities were scheduled along with all the activities from the initial definition.

In Example 115, action type `entry` traverses action type `A`. Extensions to action type `entry` include activities that traverse action types `B` and `C`.

```
component pss_top {
    action A { };
    action B { };
    action C { };

    action entry {
        activity {
            do A;
        }
    }

    extend action entry {
        activity {
            do B;
        }
    }

    extend action entry {
        activity {
            do C;
        }
    }
}
```

Example 115—DSL: Extended action traversal

The semantics of **activity** in the presence of type extension state that all three activity blocks will be traversed under an implied **schedule** block. In other words, Example 115 is equivalent to the hand-coded example shown in Example 116.

```
component pss_top {
    action A { };
    action B { };
    action C { };

    action entry {
        activity {
            schedule {
                do A;
                do B;
                do C;
            }
        }
    }
}
```

Example 116—DSL: Hand-coded action traversal

When a compound action inherits from another compound action, the activity declared in the inheriting action *shadows* (masks) the activity declared in the base action. The "**super;**" statement can be used to traverse the activity declared in the base action.

In Example 117, action `base` declares an activity that traverses action type `A`. Action `ext1` inherits from `base` and replaces the activity declared in `base` with an activity that traverses action type `B`. Action `ext2` inherits from `base` and replaces the activity declared in `base` with an activity that first traverses the activity declared in `base`, then traverses action type `C`.

```
component pss_top {
    action A { }
    action B { }
    action C { }

    action base {
        activity {
            do A;
        }
    }

    action ext1 : base {
        activity {
            do B;
        }
    }

    action ext2 : base {
        activity {
            super;
            do C;
        }
    }
}
```

Example 117—DSL: Inheritance and traversal

NOTE—PSS/C++ does not define a mechanism for referring to the set of activities declared in a super-type—in other words, an equivalent to *activity_super_stmt*.

13.7 Symbols

To assist in reuse and simplify the specification of repetitive behaviors in a single activity, a *symbol* may be declared to represent a subset of activity functionality (see Syntax 72 and Syntax 73). The **symbol** may be used as a node in the activity.

A **symbol** may activate another **symbol**, but **symbol**s are not recursive and may not activate themselves.

13.7.1 DSL syntax

symbol_declaration ::= **symbol** *symbol*_identifier [**(** symbol_paramlist **)**] **{** { activity_stmt } **}**
symbol_paramlist ::= [symbol_param { **,** symbol_param }]
symbol_param ::= data_type identifier

Syntax 72—DSL: symbol declaration

13.7.2 C++ syntax

In C++, a **`symbol`** is created using a function that returns the sub-activity expression.

The corresponding C++ syntax for Syntax 72 is shown in Syntax 73.

pss::symbol

Defined in **`pss/symbol.h`** (see C.48).

```
symbol symbolName(parameters...) { return (...); }
```

Function declaration to return sub-activity.

Syntax 73—C++: symbol declaration

13.7.3 Examples

Example 118 and Example 119 depict using a symbol. In this case, the desired activity is a sequence of choices between `a`*N* and `b`*N*, followed by a sequence of `c`*N* actions. This statement could be specified in-line, but for brevity of the top-level activity description, a symbol is declared for the sequence of `a`*N* and `b`*N* selections. The symbol is then referenced in the top-level activity, which has the same effect as specifying the `a`*N*/`b`*N* sequence of selects in-line.

```
component entity {
   action a { }
   action b { }
   action c { }

   action top {
      a a1, a2, a3;
      b b1, b2, b3;
      c c1, c2, c3;

      symbol a_or_b {
         select {a1; b1; }
         select {a2; b2; }
         select {a3; b3; }
      }

      activity {
         a_or_b;
         c1;
         c2;
         c3;
      }
   }
}
```

Example 118—DSL: Using a symbol

```
class A : public action { ... };
class B : public action { ... };
class C : public action { ... };

class top : public action { ...
  action_handle<A> a1{"a1"}, a2{"a2"}, a3{"a3"};
  action_handle<B> b1{"b1"}, b2{"b2"}, b3{"b3"};
  action_handle<C> c1{"c1"}, c2{"c2"}, c3{"c3"};
  symbol a_or_b () {
    return (
      sequence {
        select {a1, b1},
        select {a2, b2},
        select {a3, b3}
      }
    );
  }
  activity a { a_or_b(), c1, c2, c3 };
};
...
```

Example 119—C++: Using a symbol

Example 120 and Example 121 depict using a parameterized symbol.

```
component entity {
  action a { }
  action b { }
  action c { }
  action top {
    a a1, a2, a3;
    b b1, b2, b3;
    c c1, c2, c3;
    symbol ab_or_ba (a aa, b bb) {
      select {
        { aa; bb; }
        { bb; aa; }
      }
    }
    activity {
      ab_or_ba(a1,b1);
      ab_or_ba(a2,b2);
      ab_or_ba(a3,b3);
      c1;
      c2;
      c3;
    }
  }
}
```

Example 120—DSL: Using a parameterized symbol

```
class A : public action { ... };
class B : public action { ... };
class C : public action { ... };

class top : public action {...
  action_handle<A> a1{"a1"}, a2{"a2"}, a3{"a3"};
  action_handle<B> b1{"b1"}, b2{"b2"}, b3{"b3"};
  action_handle<C> c1{"c1"}, c2{"c2"}, c3{"c3"};

  symbol ab_or_ba (const action_handle<A> &aa,
                   const action_handle<B> &bb)
  {
    return (
      select {
        sequence {aa, bb},
        sequence {bb, aa},
      }
    );
  }

  activity a {
    ab_or_ba(a1, b1),
    ab_or_ba(a2, b2),
    ab_or_ba(a3, b3),
    c1, c2, c3
  };
};
...
```

Example 121—C++: Using a parameterized symbol

13.8 Named sub-activities

Sub-activities are structured elements of an activity. Naming sub-activities is a way to specify a logical tree structure of sub-activities within an activity. This tree serves for making hierarchical references, both to action-handle variables declared in-line, as well as to the **activity** statements themselves. The hierarchical paths thus exposed abstract from the concrete syntactic structure of the activity, since only explicitly labeled statements constitute a new hierarchy level.

NOTE—Labeled activity statements are not supported in PSS/C++.

13.8.1 DSL syntax

A named sub-activity is declared by labeling an **activity** statement, see Syntax 47.

13.8.2 Scoping rules for named sub-activities

Activity statement labels shall be unique in the context of the containing named sub-activity—the nearest lexically-containing statement which is labeled. Unlabeled activity statements do not constitute a separate naming scope for sub-activities.

Note that labeling activity statements inside the scope of a **replicate** statement leads to name conflicts between the expanded sub-activities, unless a label array is specified (see 13.5.1.1). With a **replicate** label array, each expanded named sub-activity has a unique hierarchical path.

In Example 122, some **activity** statements are labeled while others are not. The second occurrence of label `L2` is conflicting with the first because the **if** statement under which the first occurs is not labeled and hence is not a separate naming scope for sub-activities.

```
action A {};

action B {
  int x;
  activity {
    L1: parallel { // 'L1' is 1st level named sub-activity
      if (x > 10) {
        L2: {      // 'L2' is 2nd level named sub-activity
          A a;
          a;
        }
        {
          A a; // OK - this is a separate naming scope for variables
          a;
        }
      }
      L2: { // Error - this 'L2' conflicts with 'L2' above
        A a;
        a;
      }
    }
  }
};
```

Example 122—DSL: Scoping and named sub-activities

13.8.3 Hierarchical references using named sub-activity

Named sub-activities, introduced through labels, allow referencing action-handle variables using hierarchical paths. References can be made to a variable from within the same activity, from the compound action top-level scope, and from outside the action scope.

A hierarchical activity path uses labels in a way similar to variables of struct and array types. The dot operator (.) in the case of simple labels, or the indexing operator (**[]**) and other array operators in the case of label arrays (introduced by **replicate** statements), may be used to reference named sub-activity blocks.

Only action handles declared directly under a labeled activity statement can be accessed outside their direct lexical scope. Action handles declared in an unnamed activity scope cannot be accessed from outside that scope.

Note that the top activity scope is unnamed. For an action handle to be directly accessible in the top-level action scope, or from outside the current scope, it shall be declared at the top-level action scope.

In Example 123, **action** `B` declares action-handle variables in labeled activity statement scopes, thus making them accessible from outside by using hierarchical paths. **action** `C` uses hierarchical paths to constrain the sub-actions of its sub-actions `b1` and `b2`.

```
action A { rand int x; };

action B {
  A a;
  activity {
    a;
    my_seq: sequence {
      A a;
      a;
      parallel {
        my_rep: repeat (3) {
          A a;
          a;
        };
        sequence {
          A a; // this 'a' is declared in unnamed scope
          a;   // can't be accessed from outside
        };
      };
    };
  };
};

action C {
  B b1, b2;
  constraint b1.a.x == 1;
  constraint b1.my_seq.a.x == 2;
  constraint b1.my_seq.my_rep.a.x == 3; // applies to all three iterations
                                        // of the loop
  activity {
    b1;
    b2 with { my_seq.my_rep.a.x == 4; }; // likewise
  }
};
```

Example 123—DSL: Hierarchical references and named sub-activities

13.9 Explicitly binding flow objects

Input and output fields of **action**s may be explicitly connected to actions using the **bind** statement (see Syntax 74 and Syntax 75). It states that the fields of the respective **action**s reference the same object—the output of one action is the input of another.

13.9.1 DSL syntax

```
activity_bind_stmt ::= bind hierarchical_id activity_bind_item_or_list ;
activity_bind_item_or_list ::=
    hierarchical_id
  | { hierarchical_id_list }
```

Syntax 74—DSL: bind statement

The following also apply:

a) Reference fields that are bound shall be of the same object type.

b) Explicit binding shall conform to the scheduling and connectivity rules of the respective flow object kind defined in 14.4.

c) Explicit binding can only associate reference fields that are statically bound to the same pool instance (see 16.4).

d) The order in which the fields are listed does not matter.

13.9.2 C++ syntax

The corresponding C++ syntax for Syntax 74 is shown in Syntax 75.

pss::bind

Defined in **`pss/bind.h`** (see C.6).

```
class bind;
```

Explicit binding of action inputs and outputs.

Member functions

```
template <class... R> bind (const R& /* input|output|lock|share */ io_items)
```
: constructor

Syntax 75—C++: bind statement

13.9.3 Examples

Examples of binding are shown in Example 124 and Example 125.

```
    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 124—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<P1> p1 {"p1"};
  action_handle<P2> p2 {"p2"};
  action_handle<C>  c  {"c"};

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

Example 125—C++: bind statement

13.10 Hierarchical flow object binding

As discussed in 14.4, actions, including compound actions, may declare inputs and/or outputs of a given flow object type. When a compound action has inputs and/or outputs of the same type and direction as its sub-action and which are statically bound to the same pool (see 16.4), the **bind** statement may be used to associate the compound action's input/output with the desired sub-action input/output. The compound action's input/output shall be the first argument to the **bind** statement.

The outermost compound action that declares the input/output determines its scheduling implications, even if it binds the input/output to that of a sub-action. The binding to a corresponding input/output of a sub-action simply delegates the object reference to the sub-action.

In the case of a buffer object input to the compound action, the action that produces the buffer object 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 shall 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 that 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 126 and Example 127.

```
action sub_a {
  input data_buf din;
  output data_buf dout;
}

action compound_a {
  input data_buf data_in;
  output data_buf data_out;
  sub_a a1, a2;
  activity {
    a1;
    a2;
    bind a1.dout a2.din;
    bind data_in a1.din;   // hierarchical bind
    bind data_out a2.dout; // hierarchical bind
//  The following bind statements would be illegal
//  bind data_in a1.dout;  // sub-action output may not be bound to
//                         // compound action's input
//  bind data_out a2.din;  // sub-action input may not be bound to
//                         // compound action's output
  }
}
```

Example 126—DSL: Hierarchical flow binding for buffer objects

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

  activity act{
    a1,
    a2,
    bind b1 {a1->dout, a2->din};
    bind b2 {data_in , a1->din};  // hierarchical bind
    bind b3 {data_out, a2->dout}; // hierarchical bind
//  The following bind statements would be illegal
//  bind b4 {data_in , a1->dout}; // sub-action output may not be bound to
//                                // compound action's input
//  bind b5 {data_out, a2->din};  // sub-action input may not be bound to
//                                // compound action's output
  };
};
...
```

Example 127—C++: Hierarchical flow binding for buffer objects

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. A sub-action within the activity of a compound action that is bound to a stream input/output of the compound action shall be an initial action in the activity of the compound action. Consider Example 128 and Example 129.

```
    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 128—DSL: Hierarchical flow binding for stream objects

```
    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 129—C++: Hierarchical flow binding for stream objects

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.

13.11 Hierarchical resource object binding

As discussed in 15.2, actions, including compound actions, may claim a resource object of a given type. When a compound action claims a resource of the same type as its sub-action(s) and where the compound action and the sub-action are bound to the same pool, the **bind** statement may be used to associate the compound action's resource with the desired sub-action resource. The compound action's resource shall be the first argument to the **bind** statement.

The outermost compound action that claims the resource determines its scheduling implications. The binding to a corresponding resource of a sub-action simply delegates the resource reference to the sub-action.

The compound action's claim on the resource determines the scheduling of the compound action relative to other actions and that claim is valid for the duration of the activity. The sub-actions' resource claim determines the relative scheduling of the sub-actions in the context of the activity. In the absence of the explicit resource binding, the compound action and its sub-action(s) claim resources from the pool to which they are bound. Thus, it shall be illegal for a sub-action to lock the same resource instance that is locked by the compound action.

A resource locked by the compound action may be bound to any resource(s) in the sub-action(s). Thus, only one sub-action that locks the resource reference may execute in the activity at any given time and no sharing sub-actions may execute at the same time. If the resource that is locked by the compound action is bound to a shared resource(s) in the sub-action(s), there is no further scheduling dependency.

A resource shared by the compound action may only be bound to a shared resource(s) in the sub-action(s). Since the compound action's shared resource may also be claimed by another action, there is no way to guarantee exclusive access to the resource by any sub-action; so, it shall be illegal to bind a shared resource to a locking sub-action resource.

In Example 130 and Example 131, the compound action locks resources `crlkA` and `crlkB`, so no other actions outside of `compound_a` may lock either resource for the duration of the activity. In the context of the activity, the bound resource acts like a resource pool of the given type of `size=1`.

```
action sub_a {
  lock reslk_r rlkA, rlkB;
  share resshr_r rshA, rshB;
}

action compound_a {
  lock reslk_r crlkA, crlkB;
  share resshr_r crshA, crshB;
  sub_a a1, a2;
  activity {
    schedule {
      a1;
      a2;
    }
    bind crlkA {a1.rlkA, a2.rlkA};
    bind crshA {a1.rshA, a2.rshA};
    bind crlkB {a1.rlkB, a2.rshB};
    bind crshB {a1.rshB, a2.rlkB}; //illegal
  }
}
```

Example 130—DSL: Hierarchical resource binding

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

Example 131—C++: Hierarchical resource binding

14. Flow objects

A *flow object* represents incoming or outgoing data/control flow for actions, or their pre-condition and post-condition. A flow object can have two modes of reference by actions: **input** and **output**.

14.1 Buffer objects

Buffer objects represent data items in some persistent storage that can be written and read. Once their writing is completed, they can be read as needed. Typically, buffer objects represent data or control buffers in internal or external memories. See Syntax 76 and Syntax 77.

14.1.1 DSL syntax

```
buffer identifier [ template_param_decl_list ] [ struct_super_spec ] { { struct_body_item } }
```

Syntax 76—DSL: buffer declaration

The following also apply:

a) Note that the buffer type does not imply any specific layout in memory for the specific data being stored.

b) Buffer types can inherit from previously defined structs or buffers.

c) Buffer object reference fields can be declared under actions using the **input** or **output** modifier (see 14.4). Instance fields of buffer type (taken as a plain-data type) can only be declared under higher-level buffer types, as their data attribute.

d) A buffer object shall be the output of exactly one action. A buffer object may be the input of any number (zero or more) of actions.

e) Execution of a consuming action that inputs a buffer shall not begin until after the execution of the producing action completes (see Figure 2).

14.1.2 C++ syntax

The corresponding C++ syntax for Syntax 76 is shown in Syntax 77.

pss::buffer

Defined in **`pss/buffer.h`** (see C.8).

```
class buffer;
```

Base class for declaring a buffer flow object.

Member functions

```
buffer (const scope& name)  : constructor
virtual void pre_solve()  : in-line pre_solve exec block
virtual void post_solve()  : in-line post_solve exec block
```

Syntax 77—C++: buffer declaration

14.1.3 Examples

Examples of buffer objects are show in Example 132 and Example 133.

```
struct mem_segment_s {...};
buffer data_buff_s {
   rand mem_segment_s seg;
};
```

Example 132—DSL: buffer object

```
struct mem_segment_s : public structure { ... };
...
struct data_buff_s : public buffer {
  PSS_CTOR(data_buff_s, buffer);
  rand_attr<mem_segment_s> seg {"seg"};
};
type_decl<data_buff_s> data_buff_s_decl;
```

Example 133—C++: buffer object

14.2 Stream objects

Stream objects represent transient data or control exchanged between actions during concurrent activity, e.g., over a bus or network, or across interfaces. They represent data item flow or message/notification exchange. See Syntax 78 and Syntax 79.

14.2.1 DSL syntax

stream identifier [template_param_decl_list] [struct_super_spec] **{** { struct_body_item } **}**

Syntax 78—DSL: stream declaration

The following also apply:

a) Stream types can inherit from previously defined structs or streams.

b) Stream object reference fields can be declared under actions using the **input** or **output** modifier (see 14.4). Instance fields of stream type (taken as a plain-data type) can only be declared under higher-level stream types, as their data attribute.

c) A stream object shall be the output of exactly one action and the input of exactly one action.

d) The outputting and inputting actions shall begin their execution at the same time, after the same preceding action(s) completes. The outputting and inputting actions are said to run *in parallel*. The semantics of parallel execution are discussed further in 13.3.4.

14.2.2 C++ syntax

The corresponding C++ syntax for Syntax 78 is shown in Syntax 79.

pss::stream

Defined in **`pss/stream.h`** (see C.46).

```
class stream;
```

Base class for declaring a stream flow object.

Member functions

```
stream (const scope& name)  : constructor
virtual void pre_solve()  : in-line pre_solve exec block
virtual void post_solve()  : in-line post_solve exec block
```

Syntax 79—C++: stream declaration

14.2.3 Examples

Examples of stream objects are show in Example 134 and Example 135.

```
struct mem_segment_s {...};
stream data_stream_s {
   rand mem_segment_s seg;
};
```

Example 134—DSL: stream object

```
struct mem_segment_s : public structure {...};
...
struct data_stream_s : public stream { ...
  PSS_CTOR(data_stream_s, stream);
  rand_attr<mem_segment_s> seg {"seg"};
};
type_decl<data_stream_s> data_stream_s_decl;
```

Example 135—C++: stream object

14.3 State objects

State objects represent the state of some entity in the execution environment at a given time. See Syntax 80 and Syntax 81.

14.3.1 DSL syntax

```
state identifier [ template_param_decl_list ] [ struct_super_spec ] { { struct_body_item } }
```

Syntax 80—DSL: state declaration

The following also apply:

a) The writing and reading of states in a scenario is deterministic. With respect to a pool of state objects, writing shall not take place concurrently to either writing or reading.

b) The initial state of a given type is represented by the built-in Boolean **`initial`** attribute. See 16.6 for more on state pools (and **`initial`**).

c) State object reference fields can be declared under actions using the **input** or **output** modifier (see 14.4). Instance fields of state type (taken as a plain-data type) can only be declared under higher-level state types, as their data attribute. It shall be illegal to access the built-in attributes **`initial`** and **`prev`** on an instance field.

d) State types can inherit from previously defined structs or states.

e) An action that has an input or output of state object type operates on a pool of the corresponding state object type to which its field is bound. Static pool **bind** directives are used to associate the action with the appropriate state object pool (see 16.4).

f) At any given time, a pool of state object type contains a single state object. This object reflects the last state specified by the output of an action bound to the pool. Prior to execution of the first action that outputs to the pool, the object reflects the initial state specified by constraints involving the **`initial`** built-in field of state object types.

g) The built-in variable **`prev`** is a reference from this state object to the previous one in the pool. **`prev`** has the same type as this state object. The value of **`prev`** is unresolved in the context of the initial state object. In the context of an action, **`prev`** may only be referenced relative to a state object output. In all cases, only a single level of **`prev`** reference is supported, i.e., `out_s.prev.prev.prev` shall be illegal.

h) An action that inputs a state object reads the current state object from the state object pool to which it is bound.

i) An action that outputs a state object writes to the state object pool to which it is bound, updating the state object in the pool.

j) Execution of an action that outputs a state object shall complete at any time before the execution of any inputting action begins.

k) Execution of an action that outputs a state object to a pool shall not be concurrent with the execution of any other action that either outputs or inputs a state object from that pool.

l) Execution of an action that inputs a state object from a pool may be concurrent with the execution of any other action(s) that input a state object from the same pool, but shall not be concurrent with the execution of any other action that outputs a state object to the same pool.

14.3.2 C++ syntax

The corresponding C++ syntax for Syntax 80 is shown in Syntax 81.

pss::state

Defined in **`pss/state.h`** (see C.45).

```
class state;
```

Base class for declaring a stream flow object.

Member functions

`state ( const scope& name )` : constructor
`rand_attr<bool> initial` : true if in initial state
`virtual void pre_solve()` : in-line pre_solve exec block
`virtual void post_solve()` : in-line post_solve exec block

Syntax 81—C++: state declaration

14.3.3 Examples

Examples of state objects are shown in Example 136 and Example 137.

```
enum mode_e {...};
state config_s {
   rand mode_e mode;
   ...
};
```

Example 136—DSL: state object

```
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 137—C++: state object

14.4 Using flow objects

Flow object references are specified by actions as **inputs** or **outputs**. These references are used to specify rules for combining actions in legal scenarios. An action that outputs a flow object is said to *produce* that object and an action that inputs a flow object is said to *consume* the object. See Syntax 82, Syntax 83 and Syntax 84.

A consumer may consume flow objects that are produced by multiple producers, and vice versa.

An action can produce or consume a fixed-size array of flow objects. Declaring such an array is equivalent to declaring multiple distinct object reference fields of the same type.

14.4.1 DSL syntax

```
action_field_declaration ::=
    attr_field
  | activity_data_field
  | action_handle_declaration
  | object_ref_field_declaration
object_ref_field_declaration ::=
    flow_ref_field_declaration
  | resource_ref_field_declaration
flow_ref_field_declaration ::=
    ( input | output ) flow_object_type object_ref_field { , object_ref_field } ;
flow_object_type ::=
    buffer_type_identifier
  | state_type_identifier
  | stream_type_identifier
object_ref_field ::= identifier [ array_dim ]
array_dim ::= [ constant_expression ]
```

Syntax 82—DSL: Flow object reference

The following apply for arrays of flow object references:

a) Individual elements in the array may be referenced by using the array name and the element index in square brackets.

b) A flow object array is specified as entirely input or entirely output. The mode cannot be specified separately for an individual element of the array.

c) The different elements in an array may be bound to different pools. Explicit binding must be used for array elements associated with different pools. Default (type-based) pool binding applies to all elements of an object-reference array, and therefore cannot be used for this purpose (see 16.4 for more details).

d) For an array of state object references, each object reference must be bound to a different state pool, since a state pool can store only one state object at a time (see 14.3.1 and Example 154).

14.4.2 C++ syntax

Action input and outputs are defined using the **`input`** (see Syntax 83) and **`output`** (see Syntax 84) classes respectively.

The corresponding C++ syntax for Syntax 82 is shown in Syntax 83 and Syntax 84.

NOTE—PSS/C++ does not support arrays of object references.

pss::input

Defined in **`pss/input.h`** (see C.34).

```
template <class T> class input;
```

Declare an action input.

Member functions

```
input (const scope& name)
T* operator->()
T& operator*()
```

`input (const scope& name)`	: constructor
`T* operator->()`	: access underlying input type
`T& operator*()`	: access underlying input type

Syntax 83—C++: action input

pss::output

Defined in **`pss/output.h`** (see C.37).

```
template <class T> class output;
```

Declare an action input.

Member functions

`output (const scope& name)`	: constructor
`T* operator->()`	: access underlying output type
`T& operator*()`	: access underlying output type

Syntax 84—C++: action output

14.4.3 Examples

Examples of using buffer flow objects are shown in Example 138 and Example 139.

```
struct mem_segment_s {...};
buffer data_buff_s {
   rand mem_segment_s seg;
};
action cons_mem_a {
   input  data_buff_s in_data;
};
action prod_mem_a {
   output data_buff_s out_data;
};
```

Example 138—DSL: buffer flow object

For a timing diagram showing the relative execution of two actions sharing a buffer object, see Figure 2.

The corresponding C++ example for Example 138 is shown in Example 139.

```
struct mem_segment_s : public structure { ... };
...
struct data_buff_s : public buffer { ...
  rand_attr<mem_segment_s> seg {"seg"};
};
...
struct cons_mem_a : public action  { ...
  input <data_buff_s> in_data  {"in_data"};
};
...
struct prod_mem_a : public action  { ...
  output<data_buff_s> out_data {"out_data"};
};
...
```

Example 139—C++: buffer flow object

Examples of using stream flow objects are shown in Example 140 and Example 141.

```
struct mem_segment_s {...};
stream data_stream_s {
  rand mem_segment_s seg;
};
action cons_mem_a {
  input  data_stream_s in_data;
};
action prod_mem_a {
  output data_stream_s out_data;
};
```

Example 140—DSL: stream flow object

For a timing diagram showing the relative execution of two actions sharing a stream object, see Figure 3.

The corresponding C++ example for Example 140 is shown in Example 141.

```
struct mem_segment_s : public structure { ... };
...
struct data_stream_s : public stream { ...
  rand_attr<mem_segment_s> seg {"seg"};
};
...
struct cons_mem_a : public action  { ...
  input<data_stream_s>  in_data {"in_data"};
};
type_decl<cons_mem_a> cons_mem_a_decl;

struct prod_mem_a : public action  { ...
  output<data_stream_s> out_data {"out_data"};
};
...
```

Example 141—C++: stream flow object

In Example 142, four **buffer** objects are produced, one by **action** `prod_1b` and three by **action** `prod_3b`, and five **buffer** objects are consumed, one by `cons_1b`, two by `cons_2b_0`, and two by `cons_2b_1`. All the **buffer** objects are produced and consumed from the same **pool,** `buff_p`. All the **buffer** objects have a random integer attribute, `int_attr`. Consumer objects in `cons_2b_0` constrain their `int_attr` attribute to `3`, while in `cons_2b_1`, the first consumer object's `int_attr` attribute is constrained to be greater than or equal to `2`, and the second is constrained to be less than `3`. `prod_3b`'s producer objects `int_attr` attributes are all constrained to `3`.

There is an explicit **bind** to bind the second consumer object in `cons_2b_1` with the first producer object in `prod_3b`, The explicit **bind** constraint will fail because `int_attr` in the consumer object is constrained to be less than `3`, while `int_attr` in the producer object is constrained to `3`. If we remove the explicit **bind**, then that same consumer object will bind to the producer `prod_1b`'s output object because its `int_attr` is constrained to be less than `3`.

.

```
    buffer data_buff {
        rand int int_attr;
    };

    component flow_object_array_c {
        pool data_buff buff_p;
        bind buff_p *;

        action prod_buff_a {
            output data_buff out_1_buff;
        };

        action prod_3_buff_a {
            output data_buff out_3_buff [3];
        };

        action cons_buff_a {
            input data_buff in_1_buff;
        };

        action cons_2_buff_a {
            input data_buff in_2_buff [2];
        };

        action activity_a {
            prod_buff_a   prod_1b;
            prod_3_buff_a prod_3b;

            cons_buff_a   cons_1b;
            cons_2_buff_a cons_2b_0;
            cons_2_buff_a cons_2b_1;

            activity {
                prod_1b with {out_1_buff.int_attr == 1;};
                prod_3b with {
                    foreach (b:out_3_buff) { b.int_attr == 3;};
                };

                cons_1b with { in_1_buff.int_attr == 3;};

                cons_2b_0;
                constraint { foreach (b: cons_2b_0.in_2_buff) {
                    b.int_attr == 3;
                };};

                cons_2b_1 with {
                    in_2_buff[0].int_attr >= 2 && in_2_buff[1].int_attr < 3;};
                bind cons_2b_1.in_2_buff[1] prod_3b.out_3_buff[0]; // conflict
            };
        };
    };
```

Example 142—DSL: Multiple producers/consumers using the same buffer pool

An example of use of an array of state object references can be seen in .Example 154.

15. Resource objects

Resource objects represent computational resources available in the execution environment that may be assigned to actions for the duration of their execution.

15.1 Declaring resource objects

Resource types can inherit from previously defined structs or resources. See Syntax 85 and Syntax 86. Resources reside in *pools* (see Clause 16) and may be claimed by specific actions.

15.1.1 DSL syntax

```
resource identifier [ template_param_decl_list ] [ struct_super_spec ] { { struct_body_item } }
```

Syntax 85—DSL: resource declaration

The following also apply:

a) Resources have a built-in non-negative integer attribute called **`instance_id`**. This attribute represents the relative index of the resource instance in the pool. The value of **`instance_id`** ranges from **`0`** to *pool_size* **`- 1`**. See also 16.5.

b) There can only be one resource object per **`instance_id`** value for a given pool. Thus, actions referencing a resource object of some type with the same **`instance_id`** are necessarily referencing the very same object and agreeing on all its properties.

c) *Resource object reference* fields can be declared under actions using the **lock** or **share** modifier (see 15.2). Instance fields of resource type (taken as a plain-data type) can only be declared under higher-level resource types, as their data attribute.

15.1.2 C++ syntax

The corresponding C++ syntax for Syntax 85 is shown in Syntax 86.

pss::resource

Defined in **`pss/resource.h`** (see C.42).

```
class resource;
```

Base class for declaring a resource.

Member functions

```
resource (const scope& name) : constructor
virtual void pre_solve() : in-line pre_solve exec block
virtual void post_solve() : in-line post_solve exec block
rand_attr<int> instance_id : get resource instance id
```

Syntax 86—C++: resource declaration

15.1.3 Examples

For examples of how to declare a resource, see Example 143 and Example 144.

```
resource DMA_channel_s {
  rand bit[3:0] priority;
};
```

Example 143—DSL: Declaring a resource

The corresponding C++ example for Example 143 is shown in Example 144.

```
struct DMA_channel_s : public resource {
  PSS_CTOR(DMA_channel_s, resource);
  rand_attr<bit> priority {"priority", width (3,0)};
};
type_decl<DMA_channel_s> DMA_channel_s_decl;
```

Example 144—C++: Declaring a resource

15.2 Claiming resource objects

Resource objects may be *locked* or *shared* by actions. This is expressed by declaring the resource reference field of an action. See Syntax 87, Syntax 88 and Syntax 89.

An action can claim a fixed-size array of resource objects. Declaring such an array is equivalent to declaring multiple distinct object reference fields of the same type.

15.2.1 DSL syntax

```
action_field_declaration ::=
    attr_field
  | activity_data_field
  | action_handle_declaration
  | object_ref_field_declaration
object_ref_field_declaration ::=
    flow_ref_field_declaration
  | resource_ref_field_declaration
resource_ref_field_declaration ::=
    ( lock | share) resource_object_type object_ref_field { , object_ref_field } ;
resource_object_type ::= resource_type_identifier
object_ref_field ::= identifier [ array_dim ]
array_dim ::= [ constant_expression ]
```

Syntax 87—DSL: Resource object reference

lock and **share** are modes of resource use by an action. They serve to declare resource requirements of the action and restrict legal scheduling relative to other actions. *Locking* excludes the use of the resource instance by another action throughout the execution of the locking action and *sharing* guarantees that the resource is not locked by another action during its execution.

In a PSS-generated test scenario, no two actions may be assigned the same resource instance if they overlap in execution time and at least one is locking the resource. In other words, there is a strict scheduling dependency between an action referencing a resource object in **lock** mode and all other actions referencing the same resource object instance.

The following apply for arrays of resource object references:

a) Individual elements in the array may be referenced by using the array name and the element index in square brackets.
b) A resource object array is specified as entirely locked or entirely shared. The mode cannot be specified separately for an individual element of the array.
c) All elements of a resource object array must be bound to the same pool.
d) When claiming an array of resource objects, the pool size must be at least as large as the array, in order to accommodate all distinct resource claims.

15.2.2 C++ syntax

The corresponding C++ syntax for Syntax 87 is shown in Syntax 88 and Syntax 89.

NOTE—PSS/C++ does not support arrays of object references.

pss::lock

Defined in **`pss/lock.h`** (see C.36).

```
template <class T> class lock;
```

Claim a locked resource.

Member functions

```
lock (const scope& name) : constructor
T* operator->() : access underlying type
T& operator*() : access underlying type
```

Syntax 88—C++: Claim a locked resource

pss::share

Defined in **`pss/share.h`** (see C.44).

```
    template <class T> class share;
```

Claim a shared resource.

Member functions

```
    share (const scope& name) : constructor
    T* operator->() : access underlying type
    T& operator*() : access underlying type
```

Syntax 89—C++: Claim a shared resource

15.2.3 Examples

Example 145 and Example 146 demonstrate resource claims in lock and share mode. Action `two_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.

```
    resource DMA_channel_s {
      rand bit[3:0] priority;
    };
    resource CPU_core_s {...};
    action two_chan_transfer {
      lock DMA_channel_s chan_A;
      lock DMA_channel_s chan_B;
      share CPU_core_s ctrl_core;
    ...
    };
```

Example 145—DSL: Resource object

```
    struct DMA_channel_s : public resource { ...
      rand_attr<bit> priority {"priority", width (3,0)};
    };
    ...
    struct CPU_core_s : public resource { ... };
    ...
    class two_chan_transfer : public action { ...
      lock<DMA_channel_s> chan_A {"chan_A"};
      lock<DMA_channel_s> chan_B {"chan_B"};
      share<CPU_core_s> ctrl_core {"ctrl_core"};
    };
    ...
```

Example 146—C++: Resource object

In Example 147, there is a pool of 16 resource objects of type `config`. The **action** `baz_lock_a` claims a lock for 8 resource objects. The **action** `baz_share_a` claims to share 16 resource objects. The **action** `entry_a` can legally traverse two `baz_share_a` actions in parallel, as the same resource object can be shared between concurrent activities. It can also legally traverse two `baz_lock_a` actions in parallel because overall there are 16 resource objects and each action instance consumes only 8.

```
resource config {}

component foo_c {
  pool config[16] config_p;
  bind config_p *;

  action baz_lock_a {
    lock config  config_object[8];
  }

  action baz_share_a {
    share config  config_object[16];
  }

  action entry_a {
    activity {
      parallel {
        do baz_share_a;
        do baz_share_a;
      }
      parallel {
        do baz_lock_a;
        do baz_lock_a;
      }
    }
  }
}
```

Example 147—DSL: Locking and sharing arrays of resource objects

16. Pools

Pools are used to determine possible assignment of objects to actions, and thus shape the space of legal test scenarios. *Pools* represent collections of resources, state variables, and connectivity for data flow purposes. Flow object exchange is always mediated by a pool. One action outputs an object to a pool and another action inputs it from that same pool. Similarly, actions **lock** or **share** a resource object within some pool.

Pools are structural entities instantiated under components. They are used to determine the accessibility that **action**s (see Clause 11) have to flow and resource objects. This is done by binding object reference fields of action types to pools of the respective object types. Bind directives in the component scope associate resource references with a specific resource pool, state references with a specific state pool (or state variable), and buffer/stream object references with a specific data flow object pool (see 16.4).

16.1 DSL syntax

```
component_pool_declaration ::= pool [ [ expression ] ] type_identifier identifier ;
```

Syntax 90—DSL: Pool instantiation

In Syntax 90, *type_identifier* refers to a flow/resource object type, i.e., a **buffer**, **stream**, **state**, or **resource** struct type.

The *expression* applies only to pools of resource type; it specifies the number of resource instances in the pool. If omitted, the size of the resource pool defaults to `1`.

The following also apply:

a) The execution semantics of a pool are determined by its object type.

b) A pool of **state** type can hold one object at any given time, a pool of **resource** type can hold up to the given maximum number of unique resource objects throughout a scenario, and a pool of **buffer** or **stream** type is not restricted in the number of objects at a given time or throughout the scenario.

16.2 C++ syntax

The corresponding C++ syntax for Syntax 90 is shown in Syntax 91.

pss::pool

Defined in **`pss/pool.h`** (see C.39).

```
    template <class T> class pool;
```

Instantiation of a pool.

Member functions

```
    pool (const scope& name, std::size_t count = 1) : constructor
```

Syntax 91—C++: Pool instantiation

16.3 Examples

Example 148 and Example 149 demonstrate how to declare a pool.

```
buffer data_buff_s {
  rand mem_segment_s seg;
};
resource channel_s {...};
component dmac_c {
  pool data_buff_s buff_p;
  ...
  pool [4] channel_s chan_p;
}
```

Example 148—DSL: Pool declaration

The corresponding C++ example for Example 148 is shown in Example 149.

```
struct data_buff_s : public buffer { ...
  rand_attr<mem_segment_s> seg {"seg"};
};
...
struct channel_s : public resource {...};
...
class dmac_c : public component { ...
  pool<data_buff_s> buff_p{"buff_p"};
  ...
  pool<channel_s>   chan_p{"chan_p", 4};
};
...
```

Example 149—C++: Pool declaration

16.4 Static pool binding directive

Every action executes in the context of a single component instance, and every object resides in some pool. Multiple actions may execute concurrently, or over time, in the context of the same component instance, and multiple objects may reside concurrently, or over time, in the same pool. Actions of a specific component instance output objects to or input objects from a specific pool. Actions of a specific component instance can only be assigned a resource of a certain pool.

Static **bind** directives determine which pools are accessible to the actions' object references under which component instances (see Syntax 92 and Syntax 93). Binding is done relative to the component sub-tree of the component type in which the **bind** directive is applied. See also 20.1.

16.4.1 DSL syntax

```
object_bind_stmt ::= bind hierarchical_id object_bind_item_or_list ;
object_bind_item_or_list ::=
    object_bind_item_path
  | { object_bind_item_path { , object_bind_item_path } }
object_bind_item_path ::= { component_path_elem . } object_bind_item
component_path_elem ::= component_identifier [ [ constant_expression ] ]
object_bind_item ::=
    action_type_identifier . identifier [ [ constant_expression ] ]
  | *
```

Syntax 92—DSL: Static bind directives

Pool binding can take one of two forms:

— *Explicit binding:* associating a pool with a specific object reference field (input/output/resource-claim) of an action type under a component instance.

— *Default binding:* associating a pool generally with a component instance sub-tree, by object type.

The following also apply:

a) Components and pools are identified with a relative instance path expression. A specific object reference field is identified with the component instance path expression, followed by an action-type name and field name, separated by dots (**.**).

b) Default binding can be specified for an entire sub-tree by using a wildcard instead of specific paths.

c) Explicit binding always takes precedence over default bindings.

d) Conflicting explicit bindings for the same object reference field shall be illegal.

e) If multiple bindings apply to the same object reference field, the **bind** directive in the context of the top-most component instance takes precedence (i.e., the order of default binding resolution is top-down).

f) Applying multiple default bindings to the same object reference field(s) from the same component shall be illegal.

g) When binding object reference fields to a pool, the object and the pool must be of the exact same type. Thus, it shall be illegal to bind an object of a derived type to a pool of its base type, or vice versa.

16.4.2 C++ syntax

The corresponding C++ syntax for Syntax 92 is shown in Syntax 93.

pss::bind

Defined in **`pss/bind.h`** (see C.6).

```
   class bind;
```

Static bind of a type to multiple targets within the current scope.

Member functions

```
   template <class R /*type*/, typename... T /*targets*/>
      bind (const pool<R>& a_pool, const T&... targets)  : constructor
```

Syntax 93—C++: Static bind directives

16.4.3 Examples

Example 150 and Example 151 illustrate default binding pools.

In these examples, the `buff_p` pool of `data_buff_s` objects is bound using the wildcard specifier (`{*}`). Because the **bind** statement is applied in the context of component `dma_c`, the `buff_p` pool is bound to all component instances and actions defined in `dma_c` (i.e., component instances `dmas1` and `dmas2`, and action `mem2mem_a`). Thus, the `in_data` input and `out_data` output of the `mem2mem_a` action share the same `buff_p` pool. The `chan_p` pool of `channel_s` resources is bound to the two instances.

```
struct mem_segment_s {...};
buffer data_buff_s {
  rand mem_segment_s seg;
};
resource channel_s {...};
component dma_sub_c {
  ...
};
component dma_c {
  dma_sub_c dmas1, dmas2;
  pool data_buff_s buff_p;
  bind buff_p {*};
  pool [4] channel_s chan_p;
  bind chan_p {dmas1.*, dmas2.*};
  action mem2mem_a {
    input data_buff_s in_data;
    output data_buff_s out_data;
    ...
  };
};
```

Example 150—DSL: Static binding

The corresponding C++ example for Example 150 is shown in Example 151.

```
struct mem_segments_s : public structure {...};
...
struct data_buff_s : public buffer { ...
  rand_attr<mem_segment_s> seg {"seg"};
};
...
struct channel_s : public resource { ... };
...
class dma_sub_c : public component { ... };
...
class dma_c : public component { ...
  comp_inst <dma_sub_c> dmas1{"dmas1"}, dmas2{"dmas2"};
  pool <data_buff_s> buff_p { "buff_p" };
  bind b {buff_p};
  pool<channel_s> chan_p{"chan_p", 4};
  bind b2 { chan_p, dmas1, dmas2};
  class mem2mem_a : public action { ...
    input <data_buff_s> in_data {"in_data"};
    output <data_buff_s> out_data {"out_data"};
    ...
  };
  type_decl<mem2mem_a> mem2mem_a_decl;
};
...
```

Example 151—C++: Static binding

Example 152 and Example 153 illustrate the two forms of binding:, explicit and default. Action `power_transition_a`'s input and output are both associated with the context component's (`graphics_c`) state object pool. However, action `observe_same_power_state_a` has two inputs, each of which is explicitly associated with a different state object pool, the respective sub-component state variable. The `channel_s` resource pool is instantiated under the multimedia subsystem and is shared between the two engines.

```
state power_state_s { rand int in [0..4] level; }
resource channel_s {}
component graphics_c {
   pool power_state_s power_state_var;
   bind power_state_var *; // accessible to all actions under this
                           // component (specifically power_transition's
                           // input/output)
   action power_transition_a {
      input power_state_s curr;  //current state
      output power_state_s next; //next state
      lock channel_s chan;
   }
}
component my_multimedia_ss_c {
   graphics_c gfx0;
   graphics_c gfx1;
   pool [4] channel_s channels;
   bind channels {gfx0.*,gfx1.*};// accessible by default to all actions
                                 // under these component sub-trees
                                 // (specifically power_transition's chan)
   action observe_same_power_state_a {
      input power_state_s gfx0_state;
      input power_state_s gfx1_state;
      constraint gfx0_state.level == gfx1_state.level;
   }
   // explicit binding of the two power state variables to the
   // respective inputs of action observe_same_power_state_a
   bind gfx0.power_state_var observe_same_power_state_a.gfx0_state;
   bind gfx1.power_state_var observe_same_power_state_a.gfx1_state;
}
```

Example 152—DSL: Pool binding

```
struct power_state_s : public state { ...
  attr<int> level{"level", range(0,4) };
};
...
struct channel_s : public resource { ... };
...
class graphics_c : public component { ...
  pool<power_state_s> power_state_var {"power_state_var"};
 bind b1 {power_state_var}; // accessible to all actions under this component
                          // (specifically power_transition's input/output)
  class power_transition_a : public action { ...
    input <power_state_s> curr {"curr"};
    output <power_state_s> next {"next"};
    lock <channel_s> chan{"chan"};
  };
  type_decl<power_transtion_a> power_transition_a_decl;
};
...
class my_multimedia_ss_c : public component { ...
  comp_inst<graphics_c> gfx0 {"gfx0"};
  comp_inst<graphics_c> gfx1 {"gfx1"};
  pool <channel_s> channels {"channels", 4};
  bind b1 { channels, gfx0, gfx1}; // accessible by default to all actions
                                  // under these component sub-trees
                                 // (specifically power_transition's chan)
  class observe_same_power_state_a : public action { ...
    input <power_state_s> gfx0_state {"gfx0_state"};
    input <power_state_s> gfx1_state {"gfx1_state"};
    constraint c1 { gfx0_state->level == gfx1_state->level };
  };
  type_decl<observe_same_power_state_a> observe_same_power_state_a_decl;
  // explicit binding of the two power state variables to the
  // respective inputs of action observe_same_power_state
  bind b2 {gfx0->power_state_var,
           observe_same_power_state_a_decl->gfx0_state};
  bind b3 {gfx1->power_state_var,
           observe_same_power_state_a_decl->gfx1_state};
};
...
```

Example 153—C++: Pool binding

In Example 154, there is a `observe_same_power_state_a` **action** type with an array of 2 input **state** objects. Action `power_transition_a` will cause at least one inferred instance to bind with the respective `observe_same_power_state_a` action's object for each one of the `graphics_c` component instances. Using explicit pool bind statements, each element in the object array of `observe_same_power_state_a` is bound to a different pool.

```
state power_state_s {
   rand int in [0..4] level;
   constraint initial -> level == 0;
}

// graphics component with power state
component graphics_c {
   pool power_state_s power_state_var;
   bind power_state_var *; // accessible to all actions under this
                           // component (specifically power_transition's
                           // input/output)
   action power_transition_a {
      input  power_state_s curr; //current state
      output power_state_s next; //next state
   }
}

component my_multimedia_ss_c {
   graphics_c gfx[2];

   action observe_same_power_state_a {
      rand int in [1..4] observed_level;

      input power_state_s gfx_state[2];
      constraint { foreach (s: gfx_state) {
        s.level == observed_level;
      }}
   }
   // explicit binding of the two power state variables to the
   // respective inputs of action observe_same_power_state_a
   bind gfx[0].power_state_var observe_same_power_state_a.gfx_state[0];
   bind gfx[1].power_state_var observe_same_power_state_a.gfx_state[1];
}
```

Example 154—DSL: Multiple state pools of the same state type

16.5 Resource pools and the instance_id attribute

Each object in a resource pool has a unique **`instance_id`** value, ranging from `0` to the pool's size `- 1`. Two actions that reference a resource object with the same **`instance_id`** value in the same pool are referencing the same resource object. See also 17.1.

For example, in Example 155 and Example 156, action `transfer` is locking two kinds of resources: `channel_s` and `cpu_core_s`. Because `channel_s` is defined under component `dma_c`, each `dma_c` instance has its own pool of two channel objects. Within action `par_dma_xfers`, the two transfer actions can be assigned the same channel **`instance_id`** because they are associated with different `dma_c` instances. However, these same two actions must be assigned a different `cpu_core_s` object, with a different **`instance_id`**, because both `dma_c` instances are bound to the same resource pool of `cpu_core_s` objects defined under **`pss_top`** and they are scheduled in parallel. The **bind** directive designates the pool of `cpu_core_s` resources is to be utilized by both instances of the `dma_c` component.

```
resource cpu_core_s {}
component dma_c {
  resource channel_s {}
  pool[2] channel_s channels;
  bind channels {*}; // accessible to all actions
                     // under this component (and its sub-tree)
  action transfer {
    lock channel_s chan;
    lock cpu_core_s core;
    }
}
component pss_top {
  dma_c dma0,dma1;
  pool[4] cpu_core_s cpu;
  bind cpu {dma0.*, dma1.*};// accessible to all actions
                            // under the two sub-components
  action par_dma_xfers {
    dma_c::transfer xfer_a;
    dma_c::transfer xfer_b;

    constraint xfer_a.comp != xfer_b.comp;
    constraint xfer_a.chan.instance_id==xfer_b.chan.instance_id; //OK
   constraint xfer_a.core.instance_id==xfer_b.core.instance_id; //conflict!
   activity {
      parallel {
         xfer_a;
         xfer_b;
      }
   }
  }
}
```

Example 155—DSL: Resource object assignment

```
struct cpu_core_s : public resource { ... };
...
class dma_c : public component { ...
  struct channel_s : public resource { ... };
  ...
  pool <channel_s> channels {"channels", 2};
  bind b1 {channels}; // accessible to all actions
                      // under this component (and its sub-tree)
  class transfer : public action { ...
    lock <channel_s> chan {"chan"};
    lock <cpu_core_s> core {"core"};
  };
  type_decl<transfer> transfer_decl;
};
...
class pss_top : public component { ...
  comp_inst<dma_c> dma0{"dma0"}, dma1{"dma1"};
  pool <cpu_core_s> cpu {"cpu", 4};
  bind b2 {cpu, dma0, dma1}; // accessible to all actions
                             // under the two sub-components
  class par_dma_xfers : public action { ...
    action_handle<dma_c::transfer> xfer_a {"xfer_a"};
    action_handle<dma_c::transfer> xfer_b {"xfer_b"};

    constraint c1 {xfer_a->comp() != xfer_b->comp()};
   constraint c2 {xfer_a->chan->instance_id == xfer_b->chan->instance_id};
                                                                  // OK
   constraint c3 {xfer_a->core->instance_id == xfer_b->core->instance_id};
                                                                  // conflict!

    activity act {
      parallel {
        xfer_a,
        xfer_b
      }
    };
  };
  type_decl<par_dma_xfers> par_dma_xfers_decl;
};
...
```

Example 156—C++: Resource object assignment

16.6 Pool of states and the initial attribute

Each pool of a **state** type contains exactly one state object at any given point in time throughout the execution of the scenario. A state pool serves as a state variable instantiated on the context component. Actions outputting to a state pool can be viewed as transitions in a finite state machine. See also 17.1.

Prior to execution of an action that outputs a state object to the pool, the pool contains the initial object. The **`initial`** flag is *true* for the initial object and *false* for all other objects subsequently residing in the pool. The initial state object is overwritten by the first state object (if any) which is output to the pool. The initial object is only input by actions that are scheduled before any action that outputs a state object to the same pool.

Consider, for example, the code in Example 157 and Example 158. The action `codec_c::configure` has an `UNKNOWN` mode as its configuration state precondition, due to the constraint on its input `prev_conf`. Because it outputs a new state object with a different mode value, there can only be one such action per `codec` component instance (unless another action, not shown here, sets the mode back to `UNKNOWN`).

```
enum codec_config_mode_e {UNKNOWN, A, B}
component codec_c {
  state configuration_s {
    rand codec_config_mode_e mode;
    constraint initial -> mode == UNKNOWN;
  }
  pool configuration_s config_var;
  bind config_var *;
  action configure {
    input configuration_s prev_conf;
    output configuration_s next_conf;
    constraint prev_conf.mode == UNKNOWN && next_conf.mode in [A, B];
  }
}
```

Example 157—DSL: State object binding

```
PSS_ENUM(codec_config_mode_e, UNKNOWN, A, B);
...
class codec_c : public component { ...
  struct configuration_s : public state { ...
    rand_attr<codec_config_mode_e> mode {"mode"};
    constraint c1 {
      if_then {
        cond(initial),
        mode == codec_config_mode_e::UNKNOWN
      }
    };
  };
  ...
  pool <configuration_s> config_var {"config_var"} ;
  bind b1 { config_var };

  class configure_a : public action { ...
    input <configuration_s> prev_conf {"prev_conf"};
    output <configuration_s> next_conf {"next_conf"};

    constraint c1 { prev_conf->mode == codec_config_mode_e::UNKNOWN &&
        in ( next_conf->mode,
             range(codec_config_mode_e::A)
                  (codec_config_mode_e::B) )
    };
  };
  type_decl<configure_a> configure_a_decl;
};
...
```

Example 158—C++: State object binding

17. Randomization specification constructs

Scenario properties can be expressed in PSS declaratively, as algebraic constraints over attributes of scenario entities.

a) There are several categories of **struct** and **action** fields.
   1) *Random attribute field* - a field of a plain-data type (e.g., **bit**) that is qualified with the **rand** keyword.
   2) *Non-random attribute field* - a field of a plain-data type (e.g., **int**) that is not qualified with the **rand** keyword.
   3) *Sub-action field* - a field of an action type or a plain-data type that is qualified with the **action** keyword.
   4) *Input/output flow object reference field* - a field of a flow object type that is qualified with the **input** or **output** keyword.
   5) *Resource claim reference field* - a field of a resource object type that is qualified with the **lock** or **share** keyword.
b) Constraints may shape every aspect of the scenario space. In particular:
   1) Constraints are used to determine the legal value space for attribute fields of actions.
   2) Constraints affect the legal assignment of resources to actions and, consequently, the scheduling of actions.
   3) Constraints may restrict the possible binding of action inputs to action outputs, and, thus, possible action inferences from partially specified scenarios.
   4) Constraints determine the association of actions with context component instances.
   5) Constraints may be used to specify all of the above properties in a specific context of a higher level activity encapsulated via a compound action.
   6) Constraints may also be applied also to the operands of control flow statements—determining loop count and conditional branch selection.

Constraints are typically satisfied by more than just one specific assignment. There is often room for randomness or the application of other considerations in selecting values. The process of selecting values for scenario variables is called *constrained randomization* or simply *randomization*.

Randomized values of variables become available in the order in which they are used in the execution of a scenario, as specified in activities. This provides a natural way to express and reason about the randomization process. It also guarantees values sampled from the environment and fed back into the PSS domain during the generation and/or execution have clear implications on subsequent evaluation. However, this notion of ordering in variable randomization does not introduce ordering into the constraint system—the solver is required to look ahead and accommodate for subsequent constraints.

17.1 Algebraic constraints

17.1.1 Member constraints

PSS supports two types of constraint blocks (see Syntax 94 and Syntax 95) as **action**/**struct** members: static constraints that always hold and dynamic constraints that only hold when they are referenced by the user by traversing them in an activity (see 17.4.9) or referencing them inside a constraint. Dynamic constraints associate a name with a constraint that would typically be specified as an in-line constraint.

17.1.1.1 DSL syntax

```
constraint_declaration ::=
    constraint constraint_set
  | [ dynamic ] constraint identifier constraint_block
constraint_set ::=
    constraint_body_item
  | constraint_block
constraint_block ::= { { constraint_body_item } }
constraint_body_item ::=
    expression_constraint_item
  | foreach_constraint_item
  | forall_constraint_item
  | if_constraint_item
  | implication_constraint_item
  | unique_constraint_item
  | default hierarchical_id == constant_expression ;
  | default disable hierarchical_id ;
  | stmt_terminator
```

Syntax 94—DSL: Member constraint declaration

17.1.1.2 C++ syntax

The corresponding C++ syntax for Syntax 94 is shown in Syntax 95.

pss::constraint

Defined in **`pss/constraint.h`** (see C.13).

```
class constraint;
```

Declare a member constraint.

Member functions

```
template <class... R> constraint(
   const R&&... /*detail::AlgebExpr*/ expr)  : declare a constraint
template <class... R> constraint(const std::string& name,
   const R&&... /*detail::AlgebExpr*/ expr)  : declare a named constraint
```

pss::dynamic_constraint

Defined in **`pss/constraint.h`** (see C.13).

```
class dynamic_constraint;
```

Declare a dynamic member constraint.

Member functions

```
template <class... R> dynamic_constraint(const std::string& name,
   const R&&... /*detail::AlgebExpr*/ expr)  : declare a named dynamic constraint
```

Syntax 95—C++: Member constraint declaration

17.1.1.3 Examples

Example 159 and Example 160 declare a static constraint block, while Example 161 and Example 162 declare a dynamic constraint block. In the case of the static constraint, the name is optional.

```
action A {
  rand bit[31:0]    addr;

  constraint addr_c {
    addr == 0x1000;
  }
}
```

Example 159—DSL: Declaring a static constraint

```
class A : public action { ...
  rand_attr <bit> addr {"addr", width (31, 0) };

  constraint addr_c { "addr_c", addr == 0x1000 };
};
...
```

Example 160—C++: Declaring a static constraint

```
action B {
  action bit[31:0]    addr;

  dynamic constraint dyn_addr1_c {
    addr in [0x1000..0x1FFF];
  }

  dynamic constraint dyn_addr2_c {
    addr in [0x2000..0x2FFF];
  }
}
```

Example 161—DSL: Declaring a dynamic constraint

```
class B : public action { ...
  action_attr<bit> addr {"addr", width (31, 0) };

  dynamic_constraint dyn_addr1_c { "dyn_addr1_c",
    in (addr, range (0x1000, 0x1fff) )
  };

  dynamic_constraint dyn_addr2_c { "dyn_addr2_c",
    in (addr, range (0x2000, 0x2fff) )
  };
};
...
```

Example 162—C++: Declaring a dynamic constraint

Example 163 and Example 164 show a dynamic constraint inside a static constraint. In the examples, the `send_pkt` action sends a packet of a random size. The static constraint `pkt_sz_c` ensures the packet is of a legal size and the two dynamic constraints, `small_pkt_c` and `jumbo_pkt_c`, specialize the packet size to be small or large, respectively. The static constraint `interesting_sz_c` restricts the size to be either `<=100` for `small_pkt_c` or `>1500` for `jumbo_pkt_c`.

```
   action send_pkt {
     rand bit[16] pkt_sz;

     constraint pkt_sz_c {pkt_sz > 0;}

     constraint interesting_sz_c {small_pkt_c || jumbo_pkt_c;}

     dynamic constraint small_pkt_c {pkt_sz <= 100;}
     dynamic constraint jumbo_pkt_c {pkt_sz > 1500;}
   }

   action scenario {
     activity {
       // Send a packet with size in [1..100, 1501..65535]
       do send_pkt;
       // Send a small packet with a directly-specified in-line constraint
       do send_pkt with {pkt_sz <= 100;};
       // Send a small packet by referencing a dynamic constraint
       do send_pkt with {small_pkt_c;};
     }
   }
```

Example 163—DSL: Referencing a dynamic constraint inside a static constraint

```
class send_pkt : public action {...
  rand_attr<bit> pkt_sz {"pkt_sz", width(16)};

  constraint pkt_sz_c {"pkt_sz_c", pkt_sz > 0};
  constraint interesting_sz_c {"interesting_sz_c",
                                 small_pkt_c || jumbo_pkt_c};
  dynamic_constraint small_pkt_c {"small_pkt_c", pkt_sz <= 100};
  dynamic_constraint jumbo_pkt_c {"jumbo_pkt_c", pkt_sz > 1500};
};
...
class scenario : public action {...
  activity act {
    action_handle<send_pkt>(),
    action_handle<send_pkt>().with(action_handle<send_pkt>()->pkt_sz<=100),
    action_handle<send_pkt>().with(action_handle<send_pkt>()->small_pkt_c)
  };
};
...
```

Example 164—C++: Referencing a dynamic constraint inside a static constraint

17.1.2 Constraint inheritance

As discussed in 20.1, an **action**/**struct** subtype has all of the constraints that are declared in the context of its super-type or that are inherited by the super-type. Unnamed static constraints in a subtype are added to all other constraints. A named static or dynamic **constraint** in a subtype *shadows* (masks) a constraint of the same name from the super-type. Constraint inheritance applies in the same way to static constraints and dynamic constraints.

Example 165 and Example 166 illustrate a simple case of constraint inheritance and shadowing. Instances of **struct** `corrupt_data_buff` satisfy the unnamed constraint of `data_buff` based on which `size` is in the range `1` to `1024`. Additionally, `size` is greater than `256`, as specified in the subtype. Finally, per constraint `size_align` as specified in the subtype, `size` divided by `4` has a reminder of `1`.

```
buffer data_buff {
   rand int size;
   constraint size in [1..1024];
   constraint size_align { size%4 == 0; }       // 4-byte aligned
}

buffer corrupt_data_buff : data_buff {
   constraint size_align { size%4 == 1; }       // alignment 1 byte off
   constraint corrupt_data_size { size > 256; } // additional constraint
}
```

Example 165—DSL: Inheriting and shadowing constraints

```
struct data_buff : public buffer { ...
  rand_attr<int> size {"size"};
  constraint size_in { "size_in", in (size, range(1,1024)) };
  constraint size_align { "size_align", size % 4 == 0 }; // 4-byte aligned
};
...
struct corrupt_data_buff : public data_buff { ...
  constraint size_align { "size_align", size % 4 == 1 };
    // alignment 1 byte off
  constraint corrupt_data_size { "corrupt_data_size", size > 256 };
    // additional constraint
};
...
```

Example 166—C++: Inheriting and shadowing constraints

17.1.3 Action traversal in-line constraints

Constraints on sub-action data attributes can be in-lined directly in the context of an *action traversal statement* in the **activity** clause (for syntax and other details, see 13.3.1).

In the context of in-line constraints, attribute field paths of the traversed sub-action can be accessed without the sub-action field qualification. Fields of the traversed sub-action take precedence over fields of the containing action. Other attribute field paths are evaluated in the context of the containing action. In cases where the containing-action fields are shadowed (masked) by fields of the traversed sub-action, they can be explicitly accessed using the built-in variable **`this`**. In particular, fields of the context component of the containing action shall be accessed using the prefix path **`this.comp`** (see also Example 169 and Example 170).

If a sub-action field is traversed uniquely by a single traversal statement in the **activity** clause, in-lining a constraint has the same effect as declaring the same member constraint on the sub-action field of the containing action. In cases where the same sub-action field is traversed multiple times, in-line constraints apply only to the specific traversal in which they occur.

Unlike member constraints, in-line constraint are evaluated in the specific scheduling context of the *action traversal statement*. If attribute fields of sub-actions other than the one being traversed occur in the constraint, these sub-action fields have already been traversed in the activity. In cases where a sub-action field has been traversed multiple times, the most recently selected values are considered.

Example 167 and Example 168 illustrate the use of in-line constraints. The traversal of `a3` is illegal, because the path `a4.f` occurs in the in-line constraint, but `a4` has not yet been traversed at that point. Constraint `c2`, in contrast, equates `a1.f` with `a4.f` without having a specific scheduling context, and is, therefore, legal and enforced.

```
action A {
  rand bit[3:0]   f;
};

action B {
  A a1, a2, a3, a4;

  constraint c1 { a1.f in [8..15]; };
  constraint c2 { a1.f == a4.f; };

  activity {
    a1;
    a2 with {
      f in [8..15]; // same effect as constraint c1 has on a1
    };
    a3 with {
      f == a4.f;    // illegal: a4.f unresolved at this point
    };
    a4;
  }
};
```

Example 167—DSL: Action traversal in-line constraint

```
class A : public action { ...
  rand_attr< bit > f {"f", width(3, 0)};
};
...

class B : public action { ...
  action_handle<A> a1{"a1"}, a2{"a2"}, a3{"a3"}, a4{"a4"};
  constraint c1 { "c1", in (a1->f, range(8, 15)) };
  constraint c2 { "c2", a1->f == a4->f };
  activity a {
    a1,
    a2.with
      (in {a2->f, range(8,15)}), // same effect as constraint c1 has on a1
    a3.with
      (a3->f == a4->f),          // illegal: a4->f unresolved at this point
    a4
  };
};
...
```

Example 168—C++: Action traversal in-line constraint

Example 169 and Example 170 illustrate different name resolutions within an in-line **with** clause.

```
component subc {
   action A {
      rand int f;
      rand int g;
   }
}

component top {
   subc sub1, sub2;
   action B {
      rand int f;
      rand int h;
      subc::A a;

      activity {
         a with {
            f < h;        // sub-action's f and containing action's h
            g == this.f; // sub-action's g and containing action's f
            comp == this.comp.sub1;
                          // sub-action's component is sub-component
                          // 'sub1' of the parent action's component
         };
      }
   }
}
```

Example 169—DSL: Name resolution inside **with** *constraint block*

```
class subc : public component { ...
  class A : public action { ...
    rand_attr<int> f {"f"};
    rand_attr<int> g {"g"};
  };
  type_decl<A> A_decl;
};

class top : public component { ...
  comp_inst<subc> sub1 {"sub1"}, sub2 {"sub2"};
  class B : public action { ...
    rand_attr<int> f {"f"};
    rand_attr<int> h {"h"};
    action_handle<subc::A> a{"a"};

    activity act {
      a.with (
        ( a->f < h )
        && ( a->g == f )
        && ( a->comp() == comp<top>()->sub1 )
                           // sub-action's component is sub-component
                           // 'sub1' of the parent action's component
      )
    };
  };
  type_decl<B> B_decl;
};
...
```

Example 170—C++: Name resolution inside **with** *constraint block*

17.1.4 Logical expression constraints

A logical (Boolean) constraint can be used to specify a constraint. Syntax 96 shows the syntax for an expression constraint.

17.1.4.1 DSL syntax

```
expression_constraint_item ::= expression ;
```

Syntax 96—DSL: Expression constraint

expression may be any logical expression. The constraint is satisfied if the expression evaluates to *true*.

17.1.4.2 C++ syntax

Class **`detail::AlgebExpr`** is used to represent an expression constraint item.

17.1.5 Implication constraints

Conditional constraints can be specified using the *implication* operator (**->**). Syntax 97 shows the syntax for an implication constraint.

17.1.5.1 DSL syntax

```
implication_constraint_item ::= expression -> constraint_set
```

Syntax 97—DSL: Implication constraint

expression may be any logical expression. *constraint_set* represents any valid constraint or an unnamed constraint set.

The following also apply:

a) The Boolean equivalent of the implication operator `a -> b` is `(!a || b)`. This states that if the *expression* is *true*, all of the constraints in *constraint_set* shall be satisfied. In other words, if the *expression* is *true*, then the random values generated are constrained by the constraint set. Otherwise, the random values generated are unconstrained.

b) The implication constraint is bidirectional.

17.1.5.2 C++ syntax

C++ uses the **`if_then`** construct to represent implication constraints.

The Boolean equivalent of **`if_then`**`(a, b)` is `(!a || b)`.

17.1.5.3 Examples

Consider Example 171 and Example 172. Here, `b` is forced to have the value `1` whenever the value of the variable `a` is greater than `5`. However, since the constraint is bidirectional, if `b` has the value `1`, then the evaluation expression `(!(a>5) || (b==1))` is *true*, so the value of `a` is unconstrained. Similarly, if `b` has a value other than `1`, `a` is `<= 5`.

```
struct impl_s {
  rand bit[7:0]     a, b;

  constraint ab_c {
    (a > 5) -> b == 1;
  }
}
```

Example 171—DSL: Implication constraint

```
class impl_s : public structure { ...
  rand_attr<bit> a {"a", width(7,0)}, b {"b", width(7,0)};
  constraint ab_c {
    if_then {
      cond(a > 5),
      b == 1
    }
  };
};
...
```

Example 172—C++: Implication constraint

17.1.6 if-else constraints

Conditional constraints can be specified using the **if** and **if-else** constraint statements.

Syntax 98 and Syntax 99 shows the syntax for an **if-else** constraint.

17.1.6.1 DSL syntax

```
if_constraint_item ::= if ( expression ) constraint_set [ else constraint_set ]
```

Syntax 98—DSL: Conditional constraint

expression may be any logical expression. *constraint_set* represents any valid constraint or an unnamed constraint set.

The following also apply:

a) If the *expression* is *true*, all of the constraints in the first *constraint_set* shall be satisfied; otherwise, all the constraints in the optional **else** *constraint_set* shall be satisfied.
b) Constraint sets may be used to group multiple constraints.
c) Just like *implication* (see 17.1.5), *if-else* style constraints are bidirectional.

17.1.6.2 C++ syntax

The corresponding C++ syntax for Syntax 98 is shown in Syntax 99.

pss::if_then

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then;
```

Declare if-then constraint statement.

Member functions

```
if_then (const detail::AlgebExpr& a_cond, const detail::AlgebExpr&
   true_expr) : constructor
```

pss::if_then_else

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then_else;
```

Declare if-then-else constraint statement.

Member functions

```
if_then_else (const detail::AlgebExpr& a_cond, const detail::AlgebExpr&
   true_expr, const detail::AlgebExpr& false_expr) : constructor
```

Syntax 99—C++: Conditional constraint

17.1.6.3 Examples

In Example 173 and Example 174, the value of a constrains the value of b and the value of b constrains the value of a.

Attribute a cannot take the value 0 because both alternatives of the **if-else** constraint preclude it. The maximum value for attribute b is 4, since in the if alternative it is 1 and in the else alternative it is less than a, which itself is <= 5.

In evaluating the constraint, the if-clause evaluates to !(a>5) || (b==1). If a is in the range {1,2,3,4,5}, then the !(a>5) expression is *true*, so the (b==1) constraint is ignored. The else-clause evaluates to !(a<=5), which is *false*, so the constraint expression (b<a) is *true*. Thus, b is in the range {0..(a-1)}. If a is 2, then b is in the range {0,1}. If a > 5, then b is 1.

However, if b is 1, the (b==1) expression is *true*, so the !(a>5) expression is ignored. At this point, either !(a<=5) or a > 1, which means that a is in the range {2,3, … 255}.

```
struct if_else_s {
  rand bit[7:0]     a, b;

  constraint ab_c {
    if (a > 5) {
      b == 1;
    } else {
      b < a;
    }
  }
}
```

Example 173—DSL: if constraint

```
struct if_else_s : public structure { ...
  rand_attr<bit> a{"a", width(7,0)} , b{"b", width(7,0)};

  constraint ab_c {
    if_then_else {
      cond(a > 5),
      b == 1,
      b < a
    }
  };
};
...
```

Example 174—C++: if constraint

17.1.7 foreach constraints

Elements of collections can be iteratively constrained using the **foreach** constraint.

Syntax 100 and Syntax 101 show the syntax for a **foreach** constraint.

17.1.7.1 DSL syntax

```
foreach_constraint_item ::=
    foreach ( [ iterator_identifier : ] expression [ [ index_identifier ] ] ) constraint_set
```

Syntax 100—DSL: foreach constraint

constraint_set represents any valid constraint or an unnamed constraint set.

The following also apply:

a) *expression* shall be of a collection type (i.e., **array**, **list**, **map** or **set**).

b) All of the constraints in *constraint_set* shall be satisfied for each of the elements in the collection specified by *expression*.

c) *iterator_identifier* specifies the name of an iterator variable of the collection element type. Within *constraint_set*, the iterator variable, when specified, is an alias to the collection element of the current iteration.

d) *index_identifier* specifies the name of an index variable. Within *constraint_set*, the index variable, when specified, corresponds to the element index of the current iteration.

   1) For **array**s and **list**s, the index variable shall be a variable of type **int**, ranging from `0` to one less than the size of the collection variable.

   2) For **map**s, the index variable shall be a variable of the same type as the **map** keys, and range over the values of the keys.

   3) For **set**s, an index variable shall not be specified.

e) Both the index and iterator variables, if specified, are implicitly declared within the **foreach** scope and limited to that scope. Regular name resolution rules apply when the implicitly declared variables are used within the **foreach** body. For example, if there is a variable in an outer scope with the same name as the index variable, that variable is shadowed (masked) by the index variable within the **foreach** body. The index and iterator variables are not visible outside the **foreach** scope.

f) Either an index variable or an iterator variable or both shall be specified. For a **set**, an iterator variable shall be specified, but not an index variable.

17.1.7.2 C++ syntax

The corresponding C++ syntax for Syntax 100 is shown in Syntax 101.

pss::foreach

Defined in **`pss/foreach.h`** (see C.29).

```
class foreach;
```

Iterate constraint across array of non-rand and rand attributes.

Member functions

```
foreach (const attr& iter, const attr<vec<T>>& array, const
   detail::AlgebExpr& constraint)
```
: non-rand attributes (specializations provided for T = int and bit)

```
foreach (const attr& iter, const rand_attr<vec<T>>& array, const
   detail::AlgebExpr& constraint)
```
: rand attributes (specializations provided for T = int and bit)

Syntax 101—C++: foreach constraint

NOTE—Only iteration over arrays is supported in PSS/C++. foreach iteration over other collection types is not supported.

NOTE—In PSS/C++, the index and iteration variables must be explicitly declared in the containing scope of the foreach loop.

17.1.7.3 Examples

Example 175 and Example 176 show an iterative constraint that ensures that the values of the elements of a fixed-size array increment.

```
struct foreach_s {
  rand bit[9:0]   fixed_arr[10];

  constraint fill_arr_elem_c {
    foreach (fixed_arr[i]) {
      if (i > 0) {
        fixed_arr[i] > fixed_arr[i-1];
      }
    }
  }
}
```

Example 175—DSL: foreach iterative constraint

```
class foreach_s : public structure { ...
  rand_attr_vec<bit> fixed_arr {"fixed_arr", 10, width(9,0)};
  attr<int> i {"i"};
  constraint fill_arr_elem_c {"fill_arr_elem_c",
     foreach {i, fixed_arr,
       if_then {
         cond(i > 0),
         fixed_arr[i] > fixed_arr[i-1]
       }
     }
  };
};
...
```

Example 176—C++: foreach iterative constraint

17.1.8 forall constraints

The **forall** constraint is used to apply constraints to all instances of a specific type within the instance subtree in which the constraint is placed.

Syntax 102 and Syntax 103 show the syntax for a **forall** constraint.

17.1.8.1 DSL syntax

```
forall_constraint_item ::=
   forall ( iterator_identifier : type_identifier [ in ref_path ] ) constraint_set
```

Syntax 102—DSL: forall constraint

type_identifier specifies the type of the entity (**action**, **struct**, **stream**, **buffer**, **state**, **resource**) to which the constraint applies. *iterator_identifier* can be used inside *constraint_set* as an alias to each instance, much like the *iterator_identifier* in a **foreach** constraint is an alias to each element in the collection (see 17.1.7). *ref_path* is optionally used to restrict the constraint's scope of application to a certain instance subtree.

The following also apply:

a) All of the constraints in *constraint_set* shall be satisfied for every instance of the specified type in the **forall** constraint's application scope.

b) When *ref_path* is omitted, the application scope is the subtree of the constraint's enclosing scope:

   1) In the case of a member (type-level) non-dynamic constraint, its application scope includes all of the context type's fields (attributes, object references), and in the case of a compound action, also its entire activity.

   2) In the case of an in-line **with** constraint (see 17.1.3), its application scope is the traversed sub-action's fields and, if compound, also its entire activity.

   3) In the case of an activity constraint statement or the activation of a named dynamic constraint, the application scope is the activity scope immediately enclosing the activity statement.

c) When *ref_path* is specified, the application scope is the subtree under the entity (**action**, object, or **struct**) designated by *ref_path*.

d) The **forall** constraint applies to sub-actions within its application scope regardless of whether they are traversed using an action handle or anonymously.

17.1.8.2 C++ syntax

The corresponding C++ syntax for Syntax 102 is shown in Syntax 103.

pss::forall

Defined in **`pss/forall.h`** (see C.28).

```
   template <class T> class forall;
```

Iterate constraint across attributes of all instances of the specified type reachable from the enclosing scope.

Member functions

```
   forall (const iterator<T>& iter_var, const detail::AlgebExpr& constraint)
```
: constructor

Syntax 103—C++: forall constraint

17.1.8.3 Examples

Example 177 demonstrates the use of a **forall** constraint in a compound action, constraining sub-actions traversed directly and indirectly under its activity (case b.1 above). Action `entry` places a constraint on all instances of action `A`, relating attribute `x` to its own attribute `ax_limit`. The constraint does not apply to an attribute of sub-action `B` by the same name.

```
action A {
  rand int in [0..9] x;
};

action B {
  rand int in [0..9] x;
};

action C {
  A a;
  B b;
  activity {
    schedule {
      a; b;
    }
  }
};

action entry {
  rand int in [0..9] ax_limit;
  A a;
  C c;
  constraint {
    forall (a_it: A) {
      a_it.x <= ax_limit;
    }
  }
  activity {
    a; c;
  }
};
```

Example 177—DSL: forall constraint

The **forall** constraint in Example 177 is equivalent to the corresponding constraint on each path to an action handle of type `A`. Hence, action `entry` in Example 177 can be rewritten in the way shown in Example 178.

```
action entry {
  rand int in [0..9] ax_limit;
  A a;
  C c;
  constraint {
    a.x <= ax_limit;
    c.a.x <= ax_limit;
  }
  activity {
    a; c;
  }
};
```

Example 178—DSL: rewrite of forall constraint in terms of explicit paths

Example 179 below shows the same definitions of action `entry` from Example 177 above in C++.

```
class entry : public action {
  PSS_CTOR(entry, action);
  rand_attr<int> ax_limit {"ax_limit", range(0,9)};
  action_handle<A> a {"a"};
  action_handle<B> b {"b"};
  iterator<C1::A_a> a_it {"a_it"};
  constraint cnst {
    forall<A> {a_it,
      a_it->a <= ax_limit
    }
  };
  activity act {
    sequence {a, c}
  };
};
type_decl<entry> entry_type;
```

Example 179—C++: forall constraint

Example 180 demonstrates the use of **forall** constraints in two different contexts inside an activity. The first is an in-line **with** constraint item (case b.2 above), applying to all instances of type `A` under action `C` that is being traversed in this statement. The second is an activity constraint statement (case b.3 above). It applies to all instances of type `A` in the immediately enclosing activity scope – in this case the **parallel** statement. Hence this constraint applies to action `A` in the first **parallel** branch, and to all actions of type `A` under action `C` in the second **parallel** branch.

```
action entry {
  activity {
    do C with {
      forall (a_it: A) {
        a_it.x == 1;
      }
    }
    parallel {
      do A;
      do C;
      constraint forall (a_it: A) {
        a_it.x in [2, 4];
      }
    }
  }
};
```

Example 180—DSL: forall constraint in different activity scopes

Example 181 demonstrates the use of a **forall** constraint item in a dynamic constraint under an action. The dynamic constraint is activated from above for one traversal of that action, and not for the other. In this case, `A`'s attributes `s1.x` and `s2.x` may be randomized to the value `0xff` in the first execution of `B`, but not in the second.

```
struct S {
  rand bit[8] x;
};

action A {
  rand S s1, s2;
};

action B {
  dynamic constraint c1 {
    forall (it: S) { it.x != 0xff; }
  }
  activity { do A; }
};

action entry {
  activity {
    do B;
    do B with { c1; };
  }
};
```

Example 181—DSL: forall constraint item in a dynamic constraint

17.1.9 Unique constraints

The **unique** constraint causes unique values to be selected for each element in the specified set.

Syntax 104 and Syntax 105 show the syntax for a **unique** constraint.

17.1.9.1 DSL syntax

unique_constraint_item ::= **unique {** hierarchical_id_list **} ;**

hierarchical_id_list ::= hierarchical_id { **,** hierarchical_id }

Syntax 104—DSL: unique constraint

17.1.9.2 C++ syntax

The corresponding C++ syntax for Syntax 104 is shown in Syntax 105.

pss::unique

Defined in **`pss/unique.h`** (see C.50).

```
class unique;
```

Declare a unique constraint.

Member functions

```
template <class... R> unique(R&&... /*rand_attr<T>*/ r)
```
: constructor

Syntax 105—C++: unique constraint

17.1.9.3 Examples

Example 182 and Example 183 force the solver to select unique values for the random attribute fields `A`, `B`, and `C`. The **unique** constraint is equivalent to the following constraint statement: `((A != B) && (A != C) && (B != C))`.

```
struct my_struct {
   rand bit[4] in [0..12] A, B, C;
   constraint unique_abc_c {
      unique {A, B, C};
   }
}
```

Example 182—DSL: Unique constraint

```
class my_struct : public structure { ...
  rand_attr<bit> A {"A", width(4), range(0,12) },
                 B {"B", width(4), range(0,12) },
                 C {"C", width(4), range(0,12) };
  constraint unique_abc_c {"unique_abc_c",
    unique {A, B, C};
  };
};
...
```

Example 183—C++: Unique constraint

17.1.10 Default value constraints

A default value constraint determines the value of an attribute, unless explicitly disabled for that specific attribute from its direct or indirect containing type. Default value constraints may only take the form of equality of the attribute to a constant expression. Disabling a default value is done with the **default disable** constraint form.

17.1.10.1 DSL syntax

```
constraint_body_item ::=
   ...
   | default hierarchical_id == constant_expression ;
   | default disable hierarchical_id ;
   | ...
```

Syntax 106—DSL: Default constraints

The following also apply:

a) A **default** value constraint has the same semantics as the corresponding equality constraint, unless explicitly disabled. The equality must hold, and conflict with other constraints shall be flagged as a contradiction.

b) A **default disable** constraint is a directive to remove default constraints on the designated attribute, if any are specified.

c) *hierarchical_id* for both **default** and **default disable** constraints shall be a random attribute (a field with **rand** modifier). It shall be an error to apply a **default** constraint on a non-**rand** attribute.

d) Multiple **default** constraints and **default disable** constraints may be applied to the same attribute, with the following precedence rules:

   1) A constraint from a higher-level containing context overrides one from a lower-level containing context.

   2) A constraint from a derived type context overrides one from a base type context.

   3) A constraint overrides another in the same type context if it occurs later in the code.

e) **default** value constraints and **default disable** constraints may be applied to an attribute of an aggregate data type. The semantics in this case are equivalent to applying the corresponding constraints to all the **rand** scalar attributes it comprises. In particular, applying a **default disable** constraint to an attribute of an aggregate data type disables **default** value constraints on all attributes under it.

f) **default** and **default disable** constraints may not be conditioned on non-constant expressions.

g) **default** and **default disable** constraints may not be used under dynamic constraints (constraints prefixed with the **dynamic** modifier).

17.1.10.2 C++ syntax

The corresponding C++ syntax for Syntax 106 is shown in Syntax 107.

pss::default_value

Defined in **`pss/default_value.h`** (see C.23).

```
class default_value;
```

Declare a default value constraint.

Member functions

```
template <class T> default_value(const rand_attr<T>& attribute,
   const detail::AlgebExpr& default expr) : constructor
```

pss::default_disable

Defined in **`pss/default_disable.h`** (see C.22).

```
class default_disable;
```

Declare a default disable constraint.

Member functions

```
template <class T> default_disable(const rand_attr<T>& attribute) : constructor
```

Syntax 107—C++: Default constraints

17.1.10.3 Examples

In Example 184, `my_struct` has two attributes, and a **default** value constraint on one of them. This **struct** is instantiated three times under `my_action`.

```
struct my_struct {
  rand int in [0..3] attr1;
  constraint default attr1 == 0; // (1)

  rand int in [0..3] attr2;
  constraint attr1 < attr2; // (2)
};

action my_action {
  rand my_struct s1;

  rand my_struct s2;
  constraint default s2.attr1 == 2; // (3)

  rand my_struct s3;
  constraint default disable s3.attr1; // (4)
  constraint s3.attr1 > 0; // (5)
};
```

Example 184—DSL: Use of default value constraints

When randomizing `my_action`, `s1.attr1` is resolved to `0` because of constraint (1), and `s1.attr2` is randomized in the domain `1..3` because of constraint (2). `s2.attr1` is resolved to `2`, because constraint (3) overrides constraint (1), and `s2.attr2` is resolved to 3 because of constraint (2). Within `s3`, constraint (1) was disabled by (4), and has no effect. Due to constraints (2) and (5), `s3.attr1` is randomized in the domain `1..2` and `s3.attr2` in the domain `2..3` such that `s3.attr1` is less than `s3.attr2`.

Example 185 is the equivalent of Example 184 above in C++.

```
class my_struct : public structure {
    PSS_CTOR(my_struct, structure);
    rand_attr<int> attr1 {"attr1", range(0,3)};
    constraint default_c1 {default_value{ attr1, 0}};

    rand_attr<int> attr2 {"attr2", range(0,3)};
    constraint c1 {attr1 < attr2};
};
type_decl<my_struct> my_struct_type;

class my_action : public action {
    PSS_CTOR(my_action, action);
    rand_attr<my_struct> s1 {"s1"};

    rand_attr<my_struct> s2 {"s2"};
    constraint default_s2 {default_value {s2->attr1, 2} };

    rand_attr<my_struct> s3 {"s3"};
    constraint default_s3 {default_disable {s3->attr1} };
    constraint c1 {s3->attr1 > 0};
};
type_decl<my_action> my_action_type;
```

Example 185—C++: Use of default value constraints

In Example 186 below, two attributes of `my_action` have **default** value constraints. If `my_derived_action` is randomized, `attr1` is resolved to `0`, because **default** constraint (1) is disabled (3) and a different constraint is in effect (4). However, there is no consistent assignment to `attr2`, because both **default** constraint (2) and the regular constraint (5) are in effect and conflicting.

```
action my_action {
  rand int attr1;
  constraint default attr1 == -1; // (1)

  rand int attr2;
  constraint default attr2 == -1; // (2)
};

action my_derived_action : my_action {
  constraint {
    default disable attr1;        // (3)
    attr1 == 0;                   // (4) OK
  }

  constraint attr2 == 0;          // (5) contradiction!
};
```

Example 186—DSL: Contradiction with default value constraints

Example 187 below shows how **default** value constraints and **default disable** constraints apply to aggregate data types. A **default** value constraint is placed on an array as a whole (1). Under `my_action`, for instance `s1` of the struct, the default is replaced by another for a specific element (3), while the other elements retain their original default. Constraint (4) disables the default for all array elements under `s2`, and they are randomized over their full domain. Constraint (5) disables defaults of all attributes under the struct, including the 4 `arr` elements and `attr`. A subsequent constraint determines that `s3.attr` randomizes to `50`.

```
struct my_struct {
  rand array<int,4> arr;
  constraint default arr == {0, 10, 20, 30}; // (1)

  rand int attr;
  constraint default attr == 40; // (2)
};

action my_action {
  rand my_struct s1, s2, s3;

  constraint default s1.arr[3] == 100; // (3)

  constraint default disable s2.arr; // (4)

  constraint default disable s3; // (5)
  constraint s3.attr == 50;
};
```

Example 187—DSL: Default value constraints on compound data types

17.2 Scheduling constraints

Scheduling constraints relate two or more actions or sub-activities from a scheduling point of view. Scheduling constraints do not themselves introduce new action traversals. Rather, they affect actions explicitly traversed in contexts that do not already dictate specific relative scheduling. Such contexts necessarily involve actions directly or indirectly under a **schedule** statement (see 13.3.5). Similarly, scheduling constraints can be applied to named sub-activities, see Syntax 108.

17.2.1 DSL syntax

```
activity_scheduling_constraint ::= constraint ( parallel | sequence )
    { hierarchical_id , hierarchical_id { , hierarchical_id } } ;
```

Syntax 108—DSL: Scheduling constraint statement

The following also apply:

a) **constraint sequence** schedules the related actions so that each completes before the next one starts (equivalent to a sequential activity block, see 13.3.3).

b) **constraint parallel** schedules the related actions such that they are invoked in a synchronized way and then proceed without further synchronization until their completion (equivalent to a parallel activity statement, see 13.3.4).

c) Scheduling constraints may not be applied to action handles that are traversed multiple times. In particular, they may not be applied to actions traversed inside an iterative statement: **repeat**, **repeat-while**, and **foreach** (see 13.4). However, the iterative statement itself, as a named sub-activity, can be related in scheduling constraints.

d) Scheduling constraints involving action-handle variables that are not traversed at all, or are traversed under branches not actually chosen from **select** or **if** statements (see 13.4), hold vacuously.

e) Scheduling constraints shall not undo or conflict with any scheduling requirements of the related actions.

NOTE—PSS/C++ does not support scheduling constraints.

17.2.2 Example

Example 188 demonstrates the use of a scheduling constraint. In it, compound action `my_sub_flow` specifies an activity in which action a is executed, followed by the group `b`, `c`, and `d`, with an unspecified scheduling relation between them. Action `my_top_flow` schedules two executions of `my_sub_flow`, relating their sub-actions using scheduling constraints.

```
action my_sub_flow {
   A a; B b; C c; D d;

   activity {
      sequence {
         a;
         schedule {
            b; c; d;
         };
      };
   };
};

action my_top_flow {
   my_sub_flow sf1, sf2;

   activity {
      schedule {
         sf1;
         sf2;
      };
   };

   constraint sequence {sf1.a, sf2.b};
   constraint parallel {sf1.b, sf2.b, sf2.d};
};
```

Example 188—DSL: Scheduling constraints

17.3 Sequencing constraints on state objects

A pool of **state** type stores exactly one state object at any given time during the execution of a test scenario, thus serving as a state variable (see 16.4). Any **action** that outputs a state object to a pool is considered a state transition with respect to that state variable. Within the context of a state type, reference can be made to attributes of the previous state, relating them in Boolean expressions to attributes values of this state. This is done by using the built-in reference variable **`prev`** (see 14.3).

NOTE—Any constraint in which **`prev`** occurs is vacuously satisfied in the context of the initial state object.

In Example 189 and Example 190, the first constraint in `power_state_s` determines that the value of `domain_B` may only decrement by `1`, remain the same, or increment by `1` between consecutive states. The second constraint determines that if a `domain_C` in any given state is `0`, the subsequent state has a `domain_C` of `0` or `1` and `domain_B` is `1`. These rules apply equally to the output of the two actions declared under component `power_ctrl_c`.

```
state power_state_s {
  rand int in [0..3] domain_A, domain_B, domain_C;

  constraint domain_B in { prev.domain_B - 1,
                           prev.domain_B,
                           prev.domain_B + 1};

  constraint prev.domain_C==0 -> domain_C in [0,1] || domain_B==0;
};
...
component power_ctrl_c {
  pool power_state_s psvar;
  bind psvar *;

  action power_trans1 {
    output power_state_s next_state;
  };

  action power_trans2 {
    output power_state_s next_state;
    constraint next_state.domain_C == 0;
  };
};
...
```

Example 189—DSL: Sequencing constraints

```
struct power_state_s : public state { ...
  rand_attr<int> domain_A { "domain_A", range(0,3) };
  rand_attr<int> domain_B { "domain_B", range(0,3) };
  rand_attr<int> domain_C { "domain_C", range(0,3) };
  constraint c1 { in(domain_B,
                  range(prev(this)->domain_B-1)
                  (prev(this)->domain_B)
                  (prev(this)->domain_B+1) )
  };
  constraint c2 { if_then {
    cond (prev(this)->domain_C == 0),
    in(domain_C, range(0,1) ) || domain_B == 0 } };
};
...
class power_ctrl_c : public component { ...
  pool <power_state_s> psvar {"psvar"};
  bind psvar_bind {psvar};

  class power_trans : public action { ...
    output <power_state_s> next_state {"next_state"};
  };
  type_decl<power_trans> power_trans_decl;

  class power_trans2 : public action { ...
    output <power_state_s> next_state {"next_state"};
    constraint c { next_state->domain_C == 0 };
  };
  type_decl<power_trans2> power_trans2_decl;
};
...
```

Example 190—C++: Sequencing constraints

17.4 Randomization process

PSS supports randomization of plain-data type fields associated with scenario elements, as well as randomization of different relations between scenario elements, such as scheduling, resource allocation, and data flow. Moreover, the language supports specifying the order of random value selection, coupled with the flow of execution, in a compound action’s sub-activity, the **activity** clause. Activity-based random value selection is performed with specific rules to simplify activity composition and reuse and minimize complexity for the user.

Random attribute fields of **struct** type are randomized as a unit. Traversal of a sub-action field triggers randomization of random attribute fields of the **action** and the resolution of its flow/resource object references. This is followed by evaluation of the action’s activity if the action is compound.

17.4.1 Random attribute fields

This section describes the rules that govern whether an element is considered randomizable.

17.4.1.1 Semantics

a) Struct attribute fields qualified with the **rand** keyword are randomized if a field of that struct type is also qualified with the **rand** keyword.

b) Action attribute fields qualified with the **rand** keyword are randomized at the beginning of action execution. In the case of compound actions, **rand** attribute fields are randomized prior to the execution of the activity and, in all cases, prior to the execution of the action's *exec block*s (except **pre_solve**, see 17.4.10).

NOTE—It is often helpful to directly traverse attribute fields within an activity. This is equivalent to creating an intermediate action with a random attribute field of the plain-data type.

17.4.1.2 Examples

In Example 191 and Example 192, struct `S1` contains two attribute fields. Attribute field `a` is qualified with the **rand** keyword, while `b` is not. Struct `S2` creates two attribute fields of type `S1`. Attribute field `s1_1` is also qualified with the **rand** keyword. `s1_1.a` will be randomized, while `s1_1.b` will not. Attribute field `s1_2` is not qualified with the **rand** keyword, so neither `s1_2.a` nor `s1_2.b` will be randomized.

```
struct S1 {
  rand bit[3:0]   a;
  bit[3:0]        b;
}

struct S2 {
  rand S1         s1_1;
  S1              s1_2;
}
```

Example 191—DSL: Struct rand and non-rand fields

```
class S1 : public structure { ...
  rand_attr<bit> a { "a", width(3,0) };
  attr<bit> b { "b", width (3,0) };
};
...

class S2 : public structure { ...
  rand_attr<S1> s1_1 {"s1_1"};
  attr<S1> s1_2 {"s1_2"};
};
...
```

Example 192—C++: Struct rand and non-rand fields

Example 193 and Example 194 show two **action**s, each containing a **rand**-qualified data field (`A::a` and `B::b`). Action `B` also contains two fields of action type `A` (`a_1` and `a_2`). When action `B` is executed, a value is assigned to the random attribute field `b`. Next, the **activity** body is executed. This involves assigning a value to `a_1.a` and subsequently to `a_2.a`.

```
action A {
  rand bit[3:0]   a;
}

action B {
  A  a_1, a_2;
  rand bit[3:0]   b;

  activity {
    a_1;
    a_2;
  }
}
```

Example 193—DSL: Action rand-qualified fields

```
class A : public action { ...
  rand_attr<bit> a {"a", width(3,0) };
};
...

class B : public action { ...
  action_handle<A> a_1 { "a_1"}, a_2 {"a_2"};
  rand_attr<bit> b { "b", width (3, 0) };

  activity act {
    a_1,
    a_2
  };
};
...
```

Example 194—C++: Action rand-qualified fields

Example 195 and Example 196 show an action-qualified field in action `B` named `a_bit`. The PSS processing tool assigns a value to `a_bit` when it is traversed in the `activity` body. The semantics are identical to assigning a value to the **rand**-qualified action field `A::a`.

```
action A {
    rand bit[3:0]   a;
  }

  action B {
    action bit[3:0] a_bit;
    A               a_1;

    activity {
      a_bit;
      a_1;
    }
  }
```

Example 195—DSL: Action-qualified fields

```
class A : public action { ...
  rand_attr<bit> a {"a", width(3,0) };
};
...

class B : public action { ...
  action_attr<bit> a_bit { "a_bit", width (3, 0) };
  action_handle<A> a_1 { "a_1"};

  activity act {
    a_bit,
    a_1
  };
};
...
```

Example 196—C++: Action-qualified fields

17.4.2 Randomization of flow objects

When an **action** is randomized, its **input** and **output** fields are assigned a reference to a flow object of the respective type. On entry to any of the action's *exec block*s (except **pre_solve**, see 22.1.3), as well as its **activity** clause, values for all **rand** data attributes accessible through its inputs and outputs fields are resolved. The values accessible in these contexts satisfy all constraints. Constraints can be placed on attribute fields from the immediate type context, from a containing struct or action at any level or via the input/output fields of actions.

The same flow object may be referenced by an action outputting it and one or more actions inputting it. The binding of inputs to outputs may be explicitly specified in an **activity** clause or may be left unspecified. In cases where binding is left unspecified, the counterpart action of a flow object's input/output may already be one explicitly traversed in an activity or it may be introduced implicitly by the PSS processing tool to satisfy the binding rules (see Clause 18). In all of these cases, value selection for the data attributes of a flow object shall satisfy all constraints coming from the action that outputs it and actions that input it.

Consider the model in Example 197 and Example 198. Assume a scenario is generated starting from action `test`. The traversal of action `write1` is scheduled, followed by the traversal of action `read`. When `read` is randomized, its input `in_obj` must be resolved. Every buffer object shall be the output of some action. The activity does not explicitly specify the binding of `read`'s input to any action's output, but it must be resolved regardless. Action `write1` outputs a `mem_obj` whose `dat` is in the range `1` to `5`, due to a constraint in action `write1`. But, `dat` of the `mem_obj` instance `read` inputs must be in the range `8` to `12`. So `read.in_obj` cannot be bound to `write1.out_obj` without violating a constraint. The PSS processing tool shall schedule another action of type `write2` at some point prior to `read`, whose `mem_obj` is bound to `read`'s input. In selecting the value of `read.in_obj.dat`, the PSS processing tool shall consider the following:

- `dat` is an even integer, due to the constraint in `mem_obj`.
- `dat` is in the range `6` to `10`, due to a constraint in `write2`.
- `dat` is in the range `8` to `12`, due to a constraint in `read`.

This restricts the legal values of `read.in_obj.dat` to either `8` or `10`.

```
component top {
   buffer mem_obj {
   rand int dat;
   constraint dat%2 == 0; // dat must be even
   }

   action write1 {
      output mem_obj out_obj;
      constraint out_obj.dat in [1..5];
   }

   action write2 {
      output mem_obj out_obj;
      constraint out_obj.dat in [6..10];
   }

   action read {
      input mem_obj in_obj;
      constraint in_obj.dat in [8..12];
   }

   action test {
      activity {
         do write1;
         do read;
      }
   }
}
```

Example 197—DSL: Randomizing flow object attributes

```
class top : public component { ...
  class mem_obj : public buffer { ...
    rand_attr<int> dat {"dat"};
    constraint c { dat%2 == 0 }; // dat must be even
  };
...

  class write1 : public action { ...
    output<mem_obj> out_obj {"out_obj"};
    constraint c {in (out_obj->dat, range(1,5)}
  };
  type_decl<write1> write1_decl;

  class write2 : public action { ...
    output<mem_obj> out_obj {"out_obj"};
    constraint c {in (out_obj->dat, range(6,10)}
  };
  type_decl<write2> write2_decl;

  class read : public action { ...
    input<mem_obj> in_obj {"in_obj"};
    constraint c {in (in_obj->dat, range(8,12)}
  };
  type_decl<read> read_decl;

  class test : public action { ...
    activity _activity {
      action_handle<write1>(),
      action_handle<read>()
    };
  };
  type_decl<test> test_decl;
};
...
```

Example 198—C++: Randomizing flow object attributes

17.4.3 Randomization of resource objects

When an **action** is randomized, its resource claim fields (of **resource** type declared with **lock** / **share** modifiers, see 15.1) are assigned a reference to a resource object of the respective type. On entry to any of the action's *exec block*s (except **pre_solve**, see 22.1.3) or its **activity** clause, values for all random attribute fields accessible through its resource fields are resolved. The same resource object may be referenced by any number of actions, given that no two concurrent actions lock it (see 15.2). Value selection for random attribute fields of a resource object satisfy constraints coming from all actions to which it was assigned, either in **lock** or **share** mode.

Consider the model in Example 199 and Example 200. Assume a scenario is generated starting from action `test`. In this scenario, three actions are scheduled to execute in parallel: `a1`, `a2`, and `a3`, followed sequentially by a traversal of `a4`. In the **parallel** statement, action `a3` of type `do_something_else` shall be exclusively assigned one of the two instances of resource type `rsrc_obj`, since `do_something_else` claims it in **lock** mode. Therefore, the other two actions, of type `do_something`, necessarily share the other instance. When selecting the value of attribute `kind` for that instance, the PSS processing tool considers the following constraints:

— `kind` is an enumeration whose domain has the values `A`, `B`, `C`, and `D`.

— kind is not A, due to a constraint in do_something.
— a1.my_rsrc_inst is referencing the same rsrc_obj instance as a2.my_rsrc_inst, as there would be a resource conflict otherwise between one of these actions and a3.
— kind is not B, due to an in-line constraint on a1.
— kind is not C, due to an in-line constraint on a2.

D is the only legal value for a1.my_rsrc_inst.kind and a2.my_rsrc_inst.kind.

Since there are only two instances of rsrc_obj in rsrc_pool, and one of the instances is claimed via the **share** in a1 and a2, the other instance will be locked by a3. In order to determine the value of its kind field, we must consider the in-line constraint on the traversal of a4. Since a4.my_rsrc_inst.kind is constrained to the value A, this must be a different instance from the one shared by a1 and a2. Therefore, this is the same instance that is claimed by a3, and therefore a3.my_rsrc_inst.kind shall also have the value of A.

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

   resource rsrc_obj {
      rand rsrc_kind_e kind;
   }

   pool[2] rsrc_obj rsrc_pool;
   bind rsrc_pool *;

   action do_something {
      share rsrc_obj my_rsrc_inst;
      constraint my_rsrc_inst.kind != A;
   }

   action do_something_else {
      lock rsrc_obj my_rsrc_inst;
   }

   action test {
      do_something      a1, a2;
      do_something_else a3, a4;
      activity {
         parallel {
            a1 { my_rsrc_inst.kind != B; };
            a2 { my_rsrc_inst.kind != C; };
            a3;
         }
         a4 with { my_rsrc_inst.kind == A; };
      }
   }
}
```

Example 199—DSL: Randomizing resource object attributes

```
class top : public component { ...
  PSS_ENUM(rsrc_kind_e, A, B, C, D);
  ...
  class rsrc_obj : public resource { ...
    rand_attr<rsrc_kind_e> kind {"kind"};
  };
  ...
  pool<rsrc_obj> rsrc_pool {"rsrc_pool", 2};
  bind b1 {rsrc_pool};

  class do_something : public action { ...
    share<rsrc_obj> my_rsrc_inst {"my_rsrc_inst"};
    constraint c { my_rsrc_inst->kind != rsrc_kind_e::A };
  };
  type_decl<do_something> do_something_decl;

  class do_something_else : public action { ...
    lock<rsrc_obj> my_rsrc_inst {"my_rsrc_inst"};
  };
  type_decl<do_something_else> do_something_else_decl;

  class test : public action { ...
    action_handle<do_something>      a1 {"a1"}, a2 {"a2"};
    action_handle<do_something_else> a3 {"a3"}, a4 {"a4"};

    activity act {
      parallel {
        a1.with (a1->my_rsrc_inst->kind != rsrc_kind_e::B),
        a2.with (a2->my_rsrc_inst->kind != rsrc_kind_e::C),
        a3
      }
      a4.with (a4->my_rsrc_inst->kind == rsrc_kind_e::A)
    };
  };
  type_decl<test> test_decl;
};
...
```

Example 200—C++: Randomizing resource object attributes

17.4.4 Randomization of component assignment

When an **action** is randomized, its association with a component instance is determined. The built-in field **`comp`** is assigned a reference to the selected component instance. The assignment shall satisfy constraints where **`comp`** fields occur (see 10.6). Furthermore, the assignment of an action's **`comp`** field corresponds to the pools in which its inputs, outputs, and resources reside. If action `a` is assigned resource instance `r`, `r` is taken out the pool bound to `a`'s resource reference field in the context of the component instance assigned to `a`. If action `a` outputs a flow object which action `b` inputs, both output and input reference fields shall be bound to the same pool under `a`'s component and `b`'s component respectively. See Clause 16 for more on pool binding.

17.4.5 Random value selection order

A PSS processing tool conceptually assigns values to sub-action fields of the **action** in the order they are encountered in the **activity**. On entry into an activity, the value of plain-data fields qualified with **action** and **rand** sub-fields of action-type fields are considered to be undefined.

Example 201 and Example 202 show a simple activity with three action-type fields (`a`, `b`, `c`). A PSS processing tool might assign `a.val=2`, `b.val=4`, and `c.val=7` on a given execution.

```
action A {
  rand bit[3:0] val;
}

action my_action {
  A a, b, c;

  constraint abc_c {
    a.val < b.val;
    b.val < c.val;
  }
   activity {
    a;
    b;
    c;
  }
}
```

Example 201—DSL: Activity with random fields

```
class A : public action { ...
  rand_attr<bit> val {"val", width(3,0)};
};
...

class my_action : public action { ...
  action_handle<A> a {"a"}, b {"b"}, c {"c"};

  constraint abc_c { "abc_c",
    a->val < b->val,
    b->val < c->val
  };
  activity act {
    a,
    b,
    c
  };
};
...
```

Example 202—C++: Activity with random fields

17.4.6 Evaluation of expressions with action handles

Upon entry to an activity, all action handles (fields of action type) are considered uninitialized. Additionally, action handles previously traversed in an activity are reset to their uninitialized state upon entry to an activity block in which they are traversed again (an action handle may be traversed only once in any given activity scope and its nested scopes (see 13.3.1.1)). This applies equally to traversals of an action handle in a loop and to multiple occurrences of the same action handle in different activity blocks.

The value of all attributes reachable through uninitialized action handles, including direct attributes of the sub-actions and attributes of objects referenced by them, are unresolved. Only when all action handles in an expression are initialized, and all accessed attributes assume definite value, can the expression be evaluated.

Constraints accessing attributes through action handles are never violated. However, they are considered vacuously satisfied so long as these action handles are uninitialized. The Boolean expressions only need to evaluate to *true* at the point(s) in an activity when all action handles used in a constraint have been traversed.

Expressions in activity statements accessing attributes through action handles shall be illegal if they are evaluated at a point in which any of the action handles are uninitialized. Similarly, expressions in solve-exec (**pre_solve** and **post_solve**) statements of compound actions accessing attributes of sub-actions shall be illegal, since these are evaluated prior to the activity (see 17.4.10), and all action handles are uninitialized at that point. This applies equally to right-value and left-value expressions.

Example 203 shows a root action (`my_action`) with sub-action fields and an **activity** containing a loop. A value for `a.x` is selected, then two sets of values for `b.x` and `c.x` are selected.

```
action A {
  rand bit[3:0] x;
}

action my_action {
  A a, b, c;
  constraint abc_c {
    a.x < b.x;
    b.x < c.x;
  }
  activity {
    a;
    repeat (2) {
      b;
      c; // at this point constraint 'abc_c' must hold non-vacuously
    }
  }
}
```

Example 203—DSL: Value selection of multiple traversals

The following breakout shows valid values that could be selected here:

Repetition	**a.x**	**b.x**	**c.x**
1	3	5	6
2	3	9	13

Note that `b.x` of the second iteration does not have to be less than `c.x` of the first iteration since action handle `c` is uninitialized on entry to the second iteration. Note also that similar behavior would be observed if the **repeat** would be unrolled, i.e., if the activity contained instead two blocks of `b`, `c` in sequence.

Example 204 demonstrates two cases of illegal access of action-handle attributes. In these cases, accessing sub-action attributes through uninitialized action handles shall be flagged as errors.

```
action A {
  rand bit[3:0] x;
  int y;
}

action my_action {
  A a, b, c;

  exec post_solve {
    a.y = b.x; // ERROR - cannot access uninitialized action handle
   attributes
  }

  activity {
    a;
    if (a.x > 0) { // OK - 'a' is resolved
      b;
      c;
    }
    {
      if (c.y == a.x) { // ERROR - cannot access attributes of
                        // uninitialized action handle 'c.y'
        b;
      }
      c;
    }
  }
}
```

Example 204—DSL: Illegal accesses to sub-action attributes

17.4.7 Relationship lookahead

Values for random fields in an **activity** are selected and assigned as the fields are traversed. When selecting a value for a random field, a PSS processing tool shall take into account both the explicit constraints on the field and the implied constraints introduced by constraints on those fields traversed during the remainder of the activity traversal (including those introduced by inferred actions, binding, and scheduling). This rule is illustrated by Example 205 and Example 206.

17.4.7.1 Example 1

Example 205 and Example 206 show a simple **struct** with three random attribute fields and constraints between the fields. When an instance of this struct is randomized, values for all the random attribute fields are selected at the same time.

```
struct abc_s {
   rand bit[4] in [0..12] a_val, b_val, c_val;

   constraint {
      a_val < b_val;
      b_val < c_val;
   }
}
```

Example 205—DSL: Struct with random fields

```
class abc_s : public structure { ...
  rand_attr<bit> a_val{"a_val", width(4), range(0,12)},
                 b_val{"b_val", width(4), range(0,12)},
                 c_val{"c_val", width(4), range(0,12)};

  constraint c {
    a_val < b_val,
    b_val < c_val
  };
};
...
```

Example 206—C++: Struct with random fields

17.4.7.2 Example 2

Example 207 and Example 208 show a root action (`my_action`) with three sub-action fields and an activity that traverses these sub-action fields. It is important that the random-value selection behavior of this activity and the **struct** shown in Example 205 and Example 206 are the same. If a value for `a.val` is selected without knowing the relationship between `a.val` and `b.val`, the tool could select `a.val=15`. When `a.val=15`, there is no legal value for `b.val`, since `b.val` must be greater than `a.val`.

a) When selecting a value for `a.val`, a PSS processing tool shall consider the following:

   1) `a.val` is in the range `0` to `15`, due to its domain.
   2) `b.val` is in the range `0` to `15`, due to its domain.
   3) `c.val` is in the range `0` to `15`, due to its domain.
   4) `a.val < b.val`.
   5) `b.val < c.val`.

   This restricts the legal values of `a.val` to `0` to `13`.

b) When selecting a value for `b.val`, a PSS processing tool shall consider the following:

   1) The value selected for `a.val`.
   2) `b.val` is in the range `0` to `15`, due to its domain.
   3) `c.val` is in the range `0` to `15` due to its domain.
   4) `a.val < b.val`.
   5) `b.val < c.val`.

```
action A {
  rand bit[3:0] val;
}

action my_action {
  A a, b, c;

  constraint abc_c {
    a.val < b.val;
    b.val < c.val;
  }
   activity {
    a;
    b;
    c;
  }
}
```

Example 207—DSL: Activity with random fields

```
class A : public action { ...
  rand_attr<bit> val {"val", width(3,0)};
};
...

class my_action : public action { ...
  action_handle<A> a {"a"}, b {"b"}, c {"c"};

  constraint abc_c { "abc_c",
    a->val < b->val,
    b->val < c->val
  };

  activity act {
    a,
    b,
    c
  };
};
...
```

Example 208—C++: Activity with random fields

17.4.8 Lookahead and sub-actions

Lookahead shall be performed across traversal of sub-action fields and must comprehend the relationships between action attribute fields.

Example 209 and Example 210 show an action named `sub` that has three sub-action fields of type `A`, with constraint relationships between those field values. A top-level action has a sub-action field of type `A` and type `sub`, with a constraint between these two action-type fields. When selecting a value for the `top_action.v.val` random attribute field, a PSS processing tool shall consider the following:

- `top_action.s1.a.val == top_action.v.val`
- `top_action.s1.a.val < top_action.s1.b.val`

This implies that `top.v.val` shall be less than `14` to satisfy the `top_action.s1.a.val < top_action.s1.b.val` constraint.

```
component top {
   action A {
      rand bit[3:0] val;
   }

   action sub {
      A a, b, c;

      constraint abc_c {
         a.val < b.val;
         b.val < c.val;
      }

      activity {
         a;
         b;
         c;
      }
   }

   action top_action {
      A v;
      sub s1;

      constraint c {
         s1.a.val == v.val;
      }

      activity {
         v;
         s1;
      }
   }
}
```

Example 209—DSL: Sub-activity traversal

```
class top : public component { ...
  class A : public action { ...
    rand_attr<bit> val {"val", width(3,0)};
  };
  type_decl<A> A_decl;

  class sub : public action { ...
    action_handle<A> a {"a"}, b {"b"}, c {"c"};

    constraint abc_c { "abc_c",
      a->val < b->val,
      b->val < c->val
    };

    activity act {
      a,
      b,
      c
    };
  };
  type_decl<sub> sub_decl;

  class top_action : public action { ...
    action_handle<A> v {"v"};
    action_handle<sub> s1 {"s1"};

    constraint c { "c", s1->a->val == v->val };
    activity act {
      v,
      s1
    };
  };
  type_decl<top_action> top_action_decl;
};
...
```

Example 210—C++: Sub-activity traversal

17.4.9 Lookahead and dynamic constraints

Dynamic constraints introduce traversal-dependent constraints. A PSS processing tool must account for these additional constraints when making random attribute field value selections. A dynamic constraint shall hold for the entire activity branch on which it is referenced, as well to the remainder of the activity.

Example 211 and Example 212 show an activity with two dynamic constraints which are mutually exclusive. If the first branch is selected, `b.val <= 5` and `b.val < a.val`. If the second branch is selected, `b.val <= 7` and `b.val > a.val`. A PSS processing tool shall select a value for `a.val` such that a legal value for `b.val` also exists (presuming this is possible).

Given the dynamic constraints, legal value ranges for `a.val` are `1` to `15` for the first branch and `0` to `6` for the second branch.

```
action A {
  rand bit[3:0] val;
}

action dyn {
  A         a, b;

  dynamic constraint d1 {
    b.val < a.val;
    b.val <= 5;
  }

  dynamic constraint d2 {
    b.val > a.val;
    b.val <= 7;
  }

  activity {
    a;
    select {
      d1;
      d2;
    }
    b;
  }
}
```

Example 211—DSL: Activity with dynamic constraints

```
class A : public action { ...
  rand_attr<bit> val {"val", width(3,0)};
};
...

class dyn : public action { ...
  action_handle<A> a {"a"}, b {"b"};

  dynamic_constraint d1 { "d1",
    b->val < a->val,
    b->val <= 5
  };

  dynamic_constraint d2 { "d2",
    b->val > a->val,
    b->val <= 7
  };

  activity act {
    a,
    select {
      d1,
      d2
    },
    b
  };
};
...
```

Example 212—C++: Activity with dynamic constraints

17.4.10 pre_solve and post_solve exec blocks

The **pre_solve** and **post_solve** *exec block*s enable external code to participate in the solve process. **pre_solve** and **post_solve** *exec block*s may appear in **struct** and **action** type declarations. Statements in **pre_solve** blocks are used to set non-random attribute fields that are subsequently read by the solver during the solve process. Statements in **pre_solve** blocks can read the values of non-random attribute fields and their non-random children. Statements in **pre_solve** blocks cannot read values of random fields or their children, since their values have not yet been set. Statements in **post_solve** blocks are evaluated after the solver has resolved values for random attribute fields and are used to set the values for non-random attribute fields based on randomly-selected values.

The execution order of **pre_solve** and **post_solve** *exec block*s, respectively, corresponds to the order random attribute fields are assigned by the solver. The ordering rules are as follows:

a) Order within a compound action is top-down—both the **pre_solve** and **post_solve** *exec block*s, respectively, of a containing action are executed before any of its sub-actions are traversed, and, hence, before the **pre_solve** and **post_solve**, respectively, of its sub-actions.
b) Order between actions follows their relative scheduling in the scenario: if action a_1 is scheduled before a_2, a_1's **pre_solve** and **post_solve** blocks, if any, are called before the corresponding block of a_2.
c) Order for flow objects (instances of struct types declared with a **buffer**, **stream**, or **state** modifier) follows the order of their flow in the scenario: a flow object's **pre_solve** or **post_solve** *exec block* is called after the corresponding *exec block* of its outputting action and before that of its inputting action(s).

d) A resource object's **pre_solve** or **post_solve** *exec block* is called before the corresponding *exec block* of all actions referencing it, regardless of their use mode (**lock** or **shared**).

e) Order within an aggregate data type (nested struct and collection fields) is top-down—the *exec block* of the containing instance is executed before that of the contained.

PSS does not specify the execution order in other cases. In particular, any relative order of execution for sibling random **struct** attributes is legitimate and so is any order for actions scheduled in parallel where no flow objects are exchanged between them.

See 22.1 for more information on the *exec block* construct.

17.4.10.1 Example 1

Example 213 and Example 214 show a top-level struct `S2` that has rand and non-rand scalar fields, as well as two fields of struct type `S1`. When an instance of `S2` is randomized, the *exec block* of `S2` is evaluated first, but the execution for the two `S1` instances can be in any order. The following is one such possible order:

a) **pre_solve** in `S2`

b) **pre_solve** in `S2.s1_2`

c) **pre_solve** in `S2.s1_1`

d) assignment of attribute values

e) **post_solve** in `S2`

f) **post_solve** in `S2.s1_1`

g) **post_solve** in `S2.s1_2`

```
function bit[5:0] get_init_val();
function bit[5:0] get_exp_val(bit[5:0] stim_val);

struct S1 {
   bit[5:0] init_val;
   rand bit[5:0] rand_val;
   bit[5:0] exp_val;

   exec pre_solve {
      init_val = get_init_val();
   }

   constraint rand_val_c {
      rand_val <= init_val+10;
   }

   exec post_solve {
      exp_val = get_exp_val(rand_val);
   }
}

struct S2 {
   bit[5:0] init_val;
   rand bit[5:0] rand_val;
   bit[5:0] exp_val;

   rand S1 s1_1, s1_2;

   exec pre_solve {
      init_val = get_init_val();
   }

   constraint rand_val_c {
      rand_val > init_val;
   }

   exec post_solve {
      exp_val = get_exp_val(rand_val);
   }
}
```

Example 213—DSL: pre_solve/post_solve

```
function<result<bit> () > get_init_val {"get_init_val",
  result<bit>(width(5,0))
}

function<result<bit> ( in_arg<bit> ) > get_exp_val {"get_exp_val",
  result<bit>(width(5,0)),
  in_arg<bit>("stim_val", width(5,0))
};

class S1 : public structure { ...
  attr<bit> init_val {"init_val", width(5,0)};
  rand_attr<bit> rand_val {"rand_val", width(5,0)};
  attr<bit> exp_val {"exp_val", width(5,0)};

  exec pre_solve {
    exec::pre_solve,
    init_val = get_init_val()
  };

  constraint rand_val_c { rand_val <= init_val+10 };

  exec post_solve {
    exec::post_solve,
    exp_val = get_exp_val(rand_val)
  };
};
...

class S2 : public structure { ...
  attr<bit> init_val {"init_val", width(5,0)};
  rand_attr<bit> rand_val {"rand_val", width(5,0)};
  attr<bit> exp_val {"exp_val", width(5,0)};
  rand_attr<S1> s1_1 {"s1_1"}, s1_2 {"s1_2"};

  exec pre_solve {
    exec::pre_solve,
    init_val = get_init_val()
  };

  constraint rand_val_c { rand_val > init_val };

  exec post_solve {
    exec::post_solve,
    exp_val = get_exp_val(rand_val)
  };
};
...
```

Example 214—C++: pre_solve/post_solve

17.4.10.2 Example 2

Example 215 and Example 216 illustrate the relative order of execution for **post_solve** *exec block*s of a containing action `test`, two sub-actions: `read` and `write`, and a buffer object exchanged between them.

The calls therein are executed as follows:

a) **post_solve** in `test`
b) **post_solve** in `write`
c) **post_solve** in `mem_obj`
d) **post_solve** in `read`

```
buffer mem_obj {
  exec post_solve { ... }
};

action write {
  output mem_obj out_obj;
  exec post_solve { ... }
};

action read {
  input mem_obj in_obj;
  exec post_solve { ... }
};

action test {
  write wr;
  read  rd;

  activity {
    wr;
    rd;
    bind wr.out_obj rd.in_obj;
  }
  exec post_solve { ... }
};
```

Example 215—DSL: post_solve ordering between action and flow objects

```
class mem_obj : public buffer { ...
  exec post_solve { ... };
};

class write : public action { ...
  output<mem_obj> out_obj {"out_obj"};
  exec post_solve { ... };
};
...
class read : public action { ...
  input<mem_obj> in_obj {"in_obj"};
  exec post_solve { ... };
};
...
class test : public action { ...
  action_handle<write> wr{"wr"};
  action_handle<read> rd {"rd"};

  activity act {
    wr,
    rd
    bind b1 { wr->out_obj, rd->in_obj};
 };
  exec post_solve { ... };
};
...
```

Example 216—C++: post_solve ordering between action and flow objects

17.4.11 Body blocks and sampling external data

exec body blocks, or functions invoked by them, can assign values to attribute fields. **exec body** blocks are evaluated for atomic actions as part of the test execution on the target platform (see 22.1). The impact of any field values modified by an **exec body** block is evaluated after the entire **exec body** block has completed.

Example 217 and Example 218 show an **exec body** block that assigns two non-rand attribute fields. The impact of the new values applied to y1 and y2 are evaluated against the constraint system after the **exec body** block completes execution. It shall be illegal if the new values of y1 and y2 conflict with other attribute field values and constraints. Backtracking is not performed.

```
function bit[3:0] compute_val1(bit[3:0] v);
function bit[3:0] compute_val2(bit[3:0] v);
component pss_top {

  action A {
    rand bit[3:0] x;
    bit[3:0] y1, y2;

    constraint assume_y_c {
      y1 >= x && y1 <= x+2;
      y2 >= x && y2 <= x+3;

      y1 <= y2;
    }

    exec body {
      y1 = compute_val1(x);
      y2 = compute_val2(x);
    }
  }
}
```

Example 217—DSL: exec body block sampling external data

```
function<result<bit> (in_arg<bit>)> compute_val1 {"compute_val1",
  result<bit>(width(3,0)),
  in_arg<bit>("v", width(3,0))
};

function<result<bit> ( in_arg<bit> )> compute_val2 {"compute_val2",
  result<bit>(width(3,0)),
  in_arg<bit>("v", width(3,0))
};

class pss_top : public component { ...
  class A : public action { ...
    rand_attr<bit> x {"x", width(3,0)};
    attr<bit> y1{"y1", width(3,0)}, y2{"y2", width(3,0)};

    constraint assume_y_c {
      y1 >= x && y1 <= x+2,
      y2 >= x && y2 <= x+3,
      y1 <= y2
    };

  exec body {
    exec::body,
      sequence {
        y1 = compute_val1(x),
        y2 = compute_val2(x)
      }
    };
  };
  type_decl<A> A_decl;
};
...
```

Example 218—C++: exec body block sampling external data

18. Action inferencing

Perhaps the most powerful feature of PSS is the ability to focus purely on the user's verification intent, while delegating the means to achieve that intent. Previous clauses have introduced the semantic concepts to define such abstract specifications of intent. The modeling constructs and semantic rules thus defined for a portable stimulus model allow a tool to generate a number of scenarios from a single (partial) specification to implement the desired intent.

Beginning with a root action, which may contain an activity, a number of actions and their relative scheduling constraints is used to specify the verification intent for a given model. The other elements of the model, including flow objects, resources and their binding, as well as algebraic constraints throughout, define a set of rules that shall be followed to generate a valid scenario matching the specified intent. It is possible to fully specify a verification intent model, in which only a single valid scenario of actions may be generated. The randomization of data fields in the actions and their respective flow and resource objects would render this scenario as what is generally referred to as a "directed random" test, in which the actions are fully defined, but the data applied through the actions is randomized. The data values themselves may also be constrained so that there is only one scenario that may be generated, including fully-specified values for all data fields, in which case the scenario would be a "directed" test.

There are a number of ways to specify the scheduling relationship between actions in a portable stimulus model. The first, which allows explicit specification of verification intent, is via an activity. As discussed in Clause 13, an activity may define explicit scheduling dependencies between actions, which may include statements, such as **schedule**, **select**, **if-else** and others, to allow multiple scenarios to be generated even for a fully-specified intent model. Consider Example 219 and Example 220.

```
component pss_top {
  buffer data_buff_s {
    rand int val;
  };
  pool data_buff_s data_mem;
  bind data_mem *;

  action A_a {output data_buff_s dout;};
  action B_a {output data_buff_s dout;};
  action C_a {input  data_buff_s din;};
  action D_a {input  data_buff_s din;};

  action root_a {
    A_a a;
    B_a b;
    C_a c;
    D_a d;
    activity {
      select {a; b;}
      select {c; d;}
    }
  }
}
```

Example 219—DSL: Generating multiple scenarios

```
class pss_top : public component { ...
  struct data_buff_s : public buffer { ...
    rand_attr<int> val{"val"};
  };

  pool <data__buff_s> data_mem{"data_mem"};
  bind b1 {data_mem};

  class A_a : public action {...
    output <data_buff_s> dout{"dout"};
  }; type_decl<A_a> A_a_decl;

  class B_a : public action {...
    output <data_buff_s> dout{"dout"};
  }; type_decl<B_a> B_a_decl;

  class C_a : public action {...
    input <data_buff_s> din{"din"};
  }; type_decl<C_a> C_a_decl;

  class D_a : public action {...
    input <data_buff_s> din{"din"};
  }; type_decl<D_a> D_a_decl;

  class root_a : public action { ...
    action_handle<A_a> a{"a"};
    action_handle<B_a> b{"b"};
    action_handle<C_a> c{"c"};
    action_handle<D_a> d{"d"};
    activity act {
      select {a, b},
      select {c, d}
    };
  };
  type_decl<root_a> root_a_decl;
...
};
...
```

Example 220—C++: Generating multiple scenarios

While an activity may be used to fully express the intent of a given model, it is more often used to define the critical actions that must occur to meet the verification intent while leaving the details of how the actions may interact unspecified. In this case, the rules defined by the rest of the model, including flow object requirements, resource limitations and algebraic constraints, permit a tool to infer the instantiation of additional actions as defined by the model to ensure the generation of a valid scenario that meets the critical intent as defined by the activity.

The evaluation ordering rules for **pre_solve** and **post_solve** exec blocks of actions, objects, and structs, as specified in 17.4.10, apply regardless of whether the actions are explicitly traversed or inferred, and whether objects are explicitly or implicitly bound. In particular, the order conforms to the scheduling relations between **action**s, such that if an action is scheduled before another, its **pre_solve** and **post_solve** execs are evaluated before the other's. Backtracking is not performed across exec blocks. Assignments in **exec** blocks to attributes that figure in constraints may therefore lead to unsatisfied constraint errors. This applies to inferred parts of the scenarios in the same way as to parts that are explicitly specified in activities.

18.1 Implicit binding and action inferences

In a scenario description, the explicit binding of outputs to inputs may be left unspecified. In these cases, an implementation shall execute a scenario that reflects a valid completion of the given partial specification in a way that conforms to pool binding rules. If no valid scenario exists, the tool shall report an error. Completing a partial specification may involve decisions on output-to-input binding of flow objects in actions that are explicitly traversed. It may also involve introducing the traversal of additional actions, beyond those explicitly traversed, to serve as the counterpart of a flow object exchange. The introduction of an action in the execution of a scenario to complete a partially specified flow is called *action inferencing*.

Action inferences are necessary to make a scenario execution legal if the following conditions hold:

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.
b) There is no explicitly traversed action available to legally bind its output/input to the unbound input/output, i.e.,
    1) There is no action that is or may be scheduled before the inputting action in the case of buffer or state objects.
    2) There is no action that is or may be scheduled in parallel to the inputting/outputting action in the case of stream objects.

The inferencing of actions may be based on random or policy-driven (which may include specified coverage goals) decisions of a processing tool. Actions may only be inferred so as to complete a partially-specified flow. If all required input-to-output bindings are specified by explicit bindings to the traversed actions in the activity, an implementation may not introduce additional actions in the execution. See Annex F for more details on inference rules.

Consider the model in Example 221 and Example 222.

If action `send_data` is designated as the root action, this is clearly a case of partial scenario description, since action `send_data` has an input and an output, each of which is not explicitly bound. The buffer input `src_data` is bound to the `data_mem` object pool, so there must be a corresponding output object also bound to the same pool to provide the buffer object. The only action type outputting an object of the required type that is bound to the same object pool is `load_data`. Thus, an implementation shall infer the prior execution of `load_data` before executing `send_data`.

Similarly, `load_data` has a state input that is bound to the `config_var` pool. Since the output objects of action types `setup_A` and `setup_B` are also bound to the same pool, `load_data.curr_cfg` can be bound to the output of either `setup_A` or `setup_B`, but cannot be the initial state. In the absence of other constraints, the choice of whether to infer `setup_A` or `setup_B` may be randomized and the chosen action traversal shall occur before the traversal of `load_data`.

Moreover, `send_data` has a stream output `out_data`, which shall be bound to the corresponding input of another action that is also bound to the `data_bus` pool. So, an implementation shall infer the scheduling of an action of type `receive_data` in parallel to `send_data`.

```
component pss_top {
  state config_s {};
  pool config_s config_var;
  bind config_var *;

  buffer data_buff_s {};
  pool data_buff_s data_mem;
  bind data_mem *;

  stream data_stream_s {};
  pool data_stream_s data_bus;
  bind data_bus *;

  action setup_A {
    output config_s new_cfg;
  };

  action setup_B {
    output config_s new_cfg;
  };

  action load_data {
    input config_s curr_cfg;
    constraint !curr_cfg.initial;
    output data_buff_s out_data;
  };

  action send_data {
    input data_buff_s src_data;
    output data_stream_s out_data;
  };

  action receive_data {
    input data_stream_s in_data;
  };
};
```

Example 221—DSL: Action inferences for partially-specified flows

```
class pss_top : public component { ...
  struct config_s : public state {...};

  pool <config_s> config_var{"config_var"};
  bind b1 {config_var};

  struct data_buff_s : public buffer {...};

  pool <data__buff_s> data_mem{"data_mem"};
  bind b2 {config_var};

  struct data_stream_s : public stream {...};

  pool <data_stream_s> data_bus{"data_bus"};
  bind b3 {data_bus};

  class setup_A : public action {...
    output <config_s> new_cfg{"new_cfg");
  }; type_decl<setup_A> setup_A_decl;

  class setup_B : public action {...
    output <config_s> new_cfg{"new_cfg");
  }; type_decl<setup_B> setup_B_decl;

  class load_data : public action {...
    input <config_s> curr_cfg{"curr_cfg"};
    constraint c1 {!curr_cfg->initial};
    output <data_buff_s> out_data{"out_data"};
  }; type_decl<load_data> load_data_decl;

  class send_data : public action {...
    input <data_buff_s> src_data{"src_data"};
    output <data_stream_s> out_data{"out_data"};
  }; type_decl<send_data> send_data_decl;

  class receive_data : public action {...
    input <data_stream_s> in_data{"in_data"};
  }; type_decl<receive_data> receive_data_decl;
};
...
```

Example 222—C++: Action inferences for partially-specified flows

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. Consider adding another action within the **pss_top** component in Example 221 and Example 222, e.g.,

```
// DSL
action xfer_data {
  input data_buff_s src_data;
  output data_buff_s out_data;
};
// C++
class xfer_data : public action {...
  input <data_buff_s> src_data{"src_data"};
  output <data_buff_s> out_data{"out_data"};
};
```

In this case, the xfer_data action could also be inferred, along with setup_A or setup_B to provide the data_buff_s input to send_data.src_data. If xfer_data were inferred, then its src_data input would require the additional inference of another instance of setup_A, setup_B, or xfer_data to provide the data_buff_s. This "inference chain" would continue until either an instance of setup_A or setup_B is inferred, which would require no further inferencing, or the inference limit of the tool is reached, in which case an error would be reported.

Since the type of the inferred action is randomly selected from all available compatible action types, a tool may ensure that either setup_A or setup_B gets inferred before the inferencing limit is reached.

18.2 Object pools and action inferences

Action traversals may be inferred to support the flow object requirements of actions that are explicitly traversed or have been previously inferred. The set of actions from which a traversal may be inferred is determined by object pool bindings.

In Example 223 and Example 224, there are two object pools of type data_buff_s, each of which is bound to a different set of object field references. The **select** statement in the activity of root_a will randomly choose either c or d, each of which has a data_buff_s buffer input type that requires a corresponding action be inferred to supply the buffer object. Since C_a is bound to the same pool as A_a, if the generated scenario chooses c, then an instance of A_a shall be inferred to supply the c.din buffer input. Similarly, if d is chosen, then an instance of B_a shall be inferred to supply the d.din buffer input.

```
component pss_top {
  buffer data_buff_s {...};
  pool data_buff_s data_mem1, data_mem2;
  bind data_mem1 {A_a.dout, C_a.din};
  bind data_mem2 {B_a.dout, D_a.din};

  action A_a {output data_buff_s dout;};
  action B_a {output data_buff_s dout;};
  action C_a {input  data_buff_s din;};
  action D_a {input  data_buff_s din;};

  action root_a {
    C_a c;
    D_a d;
    activity {
      select {c; d;}
    }
  }
}
```

Example 223—DSL: Object pools affect inferencing

```
class pss_top : public component { ...
  struct data_buff_s : public buffer {... };
  pool <data__buff_s> data_mem1{"data_mem1"}, data_mem2{"data_mem2"};
  bind b1 {data_mem1, A_a.dout, C_a.din};
  bind b2 {data_mem2, B_a.dout, D_a.din};

  class A_a : public action {...
    output <data_buff_s> dout{"dout");
  }; type_decl<A_a> A_a_decl;

  class B_a : public action {...
    output <data_buff_s> dout{"dout");
  }; type_decl<B_a> B_a_decl;

  class C_a : public action {...
    input <data_buff_s> din{"din");
  }; type_decl<C_a> C_a_decl;

  class D_a : public action {...
    input <data_buff_s> din{"din");
  }; type_decl<D_a> D_a_decl;

  action root_a {
    action_handle<C_a> c{"c"};
    action_handle<D_a> d{"d"};

    activity act {
      select {c, d}
    };
  };
  type_decl<root_a> root_a_decl;
  ...
};
...
```

Example 224—C++: Object pools affect inferencing

18.3 Data constraints and action inferences

As mentioned in Clause 17, introducing data constraints on flow objects or other elements of the design may affect the inferencing of actions. Consider a slightly modified version of Example 219 and Example 220, as shown in Example 225 and Example 226.

Since the explicit traversal of c does not constrain the `val` field of its input, it may be bound to the output of either explicitly traversed action `a` or `b`; thus, there are two legal scenarios to be generated with the second **select** statement evaluated to traverse action `c`. However, since the data constraint on the traversal of action `d` is incompatible with the in-line data constraints on the explicitly-traversed actions `a` or `b`, another instance of either `A_a` or `B_a` shall be inferred whose output shall be bound to `d.din`. Since there is no requirement for the buffer output of either `a` or `b` to be bound, one of these actions shall be traversed from the first **select** statement, but no other action shall be inferred.

```
component pss_top {
  buffer data_buff_s {
    rand int val;
  };
  pool data_buff_s data_mem;
  bind data_mem *;

  action A_a {output data_buff_s dout;};
  action B_a {output data_buff_s dout;};
  action C_a {input  data_buff_s din;};
  action D_a {input  data_buff_s din;};

  action root_a {
    A_a a;
    B_a b;
    C_a c;
    D_a d;
    activity {
      select {a with{dout.val<5;}; b with {dout.val<5;};}
      select {c; d with {din.val>5;};}
    }
  }
}
```

Example 225—DSL: In-line data constraints affect action inferencing

```
class pss_top : public component {
  struct data_buff_s : public buffer {...
    rand_attr<int> val{"val"};
  };
...

  pool <data_buff_s> data_mem{"data_mem"};
  bind b1 {data_mem};

  class A_a : public action {...
    output <data_buff_s> dout{"dout");
  }; type_decl<A_a> A_a_decl;

  class B_a : public action {...
    output <data_buff_s> dout{"dout");
  }; type_decl<B_a> B_a_decl;

  class C_a : public action {...
    input <data_buff_s> din{"din");
  }; type_decl<C_a> C_a_decl;

  class D_a : public action {...
    input <data_buff_s> din{"din");
  }; type_decl<D_a> D_a_decl;

  class root_a : public action {...
    action_handle<A_a> a{"a"};
    action_handle<B_a> b{"b"};
    action_handle<C_a> c{"c"};
    action_handle<D_a> d{"d"};

    activity act {
      select {a.with(a->dout->val()<5), b.with(b->dout->val()<5)},
      select {c, d.with(d->din->val()>5)}
    };
  }; type_decl<root_a> root_a_decl;
  ...
};
...
```

Example 226—C++: In-line data constraints affect action inferencing

Consider, instead, if the in-line data constraints were declared in the action types, as shown in Example 227 and Example 228.

In this case, there is no valid action type available to provide the `d.din` input that satisfies its constraint as defined in the `D_a` action declaration, since the only actions that may provide the `data_buff_s` type, actions `A_a` and `B_a`, have constraints that contradict the input constraint in `D_a`. Therefore, the only legal action to traverse in the second select statement is `c`. In fact, it would be illegal to traverse action `D_a` under any circumstances for this model, given the contradictory data constraints on the flow objects.

```
component pss_top {
  buffer data_buff_s {
    rand int val;
  };
  pool data_buff_s data_mem;
  bind data_mem *;

  action A_a {
    output data_buff_s dout;
    constraint {dout.val<5;}
  };
  action B_a {
    output data_buff_s dout;
    constraint {dout.val<5;}
  };
  action C_a {
    input data_buff_s din;
  };
  action D_a {
    input data_buff_s din;
    constraint {din.val > 5;}
  };

  action root_a {
    A_a a;
    B_a b;
    C_a c;
    D_a d;
    activity {
      select {a; b;}
      select {c; d;}
    }
  }
}
```

Example 227—DSL: Data constraints affect action inferencing

```
class pss_top : public component {...
  struct data_buff_s : public buffer {...
    rand_attr<int> val{"val"};
  };
...

  pool <data_buff_s> data_mem{"data_mem"};
  bind b1 {data_mem};

  class A_a : public action {...
    output <data_buff_s> dout{"dout");
    constraint c {dout->val < 5};
  }; type_decl<A_a> A_a_decl;

  class B_a : public action {...
    output <data_buff_s> dout{"dout");
    constraint c {dout->val < 5};
  }; type_decl<B_a> B_a_decl;

  class C_a : public action {...
    input <data_buff_s> din{"din");
  }; type_decl<C_a> C_a_decl;

  class D_a : public action {...
    input <data_buff_s> din{"din");
    constraint c {din->val > 5};
  }; type_decl<D_a> D_a_decl;

  class root_a : public action {...
    action_handle<A_a> a{"a"};
    action_handle<B_a> b{"b"};
    action_handle<C_a> c{"c"};
    action_handle<D_a> d{"d"};

    activity act {
      select {a, b},
      select {c, d}
    };
  }; type_decl<root_a> root_a_decl;
};
...
```

Example 228—C++: Data constraints affect action inferencing

19. Coverage specification constructs

The legal state space for all non-trivial verification problems is very large. Coverage goals identify key value ranges and value combinations that need to occur in order to exercise key functionality. The **covergroup** construct is used to specify these targets.

The coverage targets specified by the **covergroup** construct are more directly related to the test scenario being created. As a consequence, in many cases the coverage targets would be considered coverage targets on the "generation" side of stimulus. PSS also allows data to be sampled by calling external functions. Coverage targets specified on data fields set by external functions can be related to the system state.

19.1 Defining the coverage model: covergroup

The **covergroup** construct encapsulates the specification of a coverage model. Each **covergroup** specification can include the following elements:

- A set of coverage points
- Cross coverage between coverage points
- Optional formal arguments
- Coverage options

The **covergroup** construct is a user-defined type. There are two forms of the **covergroup** construct. The first form allows an explicit type definition to be written once and instantiated multiple times in different contexts. The second form allows an in-line specification of an anonymous **covergroup** type and a single instance.

a) An *explicit* **covergroup** type can be defined in a **package**, **component**, **action**, or **struct**. In order to be reusable, an explicit **covergroup** type shall specify a list of formal parameters and shall not reference fields in the scope in which it is declared. An instance of an explicit **covergroup** type can be created in an **action** or **struct**. Syntax 109 and Syntax 110 define an explicit **covergroup** type.

b) An *in-line* **covergroup** can be defined in an **action** or **struct** scope. An in-line covergroup can reference fields in the scope in which it is defined. 19.2 contains more information on in-line covergroups.

19.1.1 DSL syntax

The syntax for **covergroup**s is shown in Syntax 109.

```
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
  | stmt_terminator
covergroup_option ::=
    option . identifier = constant_expression ;
  | type_option . identifier = constant_expression ;
```

Syntax 109—DSL: covergroup declaration

The following also apply:

a) The identifier associated with the **covergroup** declaration defines the name of the coverage model type.

b) A **covergroup** can contain one or more coverage points. A *coverage point* can cover a variable or an expression.

c) Each coverage point includes a set of bins associated with its sampled value. The bins can be user-defined or automatically created by a tool. Coverage points are detailed in 19.3.

d) A **covergroup** can specify cross coverage between two or more coverage points or variables. Any combination of more than two variables or previously declared coverage points is allowed. See also Example 231 and Example 232.

e) A **covergroup** can also specify one or more options to control and regulate how coverage data are structured and collected. Coverage options can be specified for the **covergroup** as a whole or for specific items within the **covergroup**, i.e., any of its coverage points or crosses. In general, a coverage option specified at the **covergroup** level applies to all of its items unless overridden by them. Coverage options are described in 19.6.

19.1.2 C++ syntax

The corresponding C++ syntax for Syntax 109 is shown in Syntax 110.

pss::covergroup

Defined in **`pss/covergroup.h`** (see C.14).

```
class covergroup;
```

Base class for declaring a covergroup.

Member functions

`covergroup (const scope & name)` : constructor

Syntax 110—C++: covergroup declaration

19.1.3 Examples

Example 229 and Example 230 define an in-line covergroup `cs1` with a single coverage point associated with struct field `color`. The value of the variable `color` is sampled at the default sampling point: the end of the action's traversal in which it is randomized. Sampling is discussed in more detail in 19.7.

Because the coverage point does not explicitly define any bins, the tool automatically creates three bins, one for each possible value of the enumeration type. Automatic bins are described in 19.3.6.

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

struct s {
   rand color_e color;

   covergroup {
      c: coverpoint color;
   } cs1;
}
```

Example 229—DSL: Single coverage point

```
PSS_ENUM(color_e, red, green, blue);

class s: public structure {...

   rand_attr<color_e> color {"color"};

   covergroup_inst<> cs1 {"cs1", [&]() {
          coverpoint c {"c", color};
   }
   };

};
type_decl<s> s_t;
...
```

Example 230—C++: Single coverage point

Example 231 and Example 232 creates an in-line covergroup `cs2` that includes two coverage points and two cross coverage items. Explicit coverage points labeled `Offset` and `Hue` are defined for variables `pixel_offset` and `pixel_hue`. PSS implicitly declares coverage points for variables `color` and `pixel_adr` to track their cross coverage. Implicitly declared coverage points are described in 19.4.

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

struct s {
    rand color_e        color;
    rand bit[3:0]       pixel_adr, pixel_offset, pixel_hue;

    covergroup {
        Hue : coverpoint pixel_hue;
        Offset : coverpoint pixel_offset;
        AxC: cross color, pixel_adr;
        all : cross color, Hue, Offset;
    } cs2;
}
```

Example 231—DSL: Two coverage points and cross coverage items

```
PSS_ENUM(color_e, red, green, blue);

class s : public structure { ...

   rand_attr<color_e> color        {"color"};
   rand_attr<bit>     pixel_adr    {"pixel_adr"   , width(4)};
   rand_attr<bit>     pixel_offset {"pixel_offset", width(4)};
   rand_attr<bit>     pixel_hue    {"pixel_hue"   , width(4)};

   covergroup_inst<> cs2 { "cs2", [&]() {
      coverpoint Hue    {"Hue"   , pixel_hue   };
      coverpoint Offset {"Offset", pixel_offset};
      cross AxC {"AxC", color, pixel_adr};
      cross all {"all", color, Hue, Offset};
   }
   };
};
...
```

Example 232—C++: Two coverage points and cross coverage items

19.2 covergroup instantiation

A **covergroup** type can be instantiated in **struct** and **action** contexts. If the **covergroup** declared formal parameters, these shall be bound to variables visible in the instantiation context. Instance-specific coverage options (see 19.6) may be specified as part of instantiation. In many cases, a **covergroup** is specific to the containing type and will not be instantiated independently multiple times. In these cases, it is possible to declare a covergroup instance in-line. In this case, the **covergroup** type is *anonymous*.

19.2.1 DSL syntax

Syntax 111 specifies how a **covergroup** is instantiated and how an in-line covergroup instance is declared.

```
covergroup_instantiation ::=
    covergroup_type_instantiation
  | inline_covergroup
inline_covergroup ::= covergroup { { covergroup_body_item } } identifier ;
covergroup_type_instantiation ::= covergroup_type_identifier covergroup_identifier
    ( covergroup_portmap_list ) covergroup_options_or_empty
covergroup_type_identifier ::= type_identifier
covergroup_portmap_list ::=
    covergroup_portmap { , covergroup_portmap }
  | hierarchical_id_list
covergroup_portmap ::= . identifier ( hierarchical_id )
covergroup_options_or_empty ::=
    with { { covergroup_option } }
  | ;
```

Syntax 111—DSL: covergroup instantiation

19.2.2 C++ syntax

The corresponding C++ syntax for Syntax 111 is shown in Syntax 112 and Syntax 113.

pss::covergroup_inst

Defined in **`pss/covergroup_inst.h`** (see C.19).

```
template <class T> class covergroup_inst;
```

Class for instantiating a user-defined covergroup type.

Member functions

```
covergroup_inst (const std::string &name, const options &opts) : constructor
template <class... R> covergroup_inst (const std::string &name,
    const options &opts, const R&... ports ) : constructor
template <class... R> covergroup_inst (const std::string &name,
    const R&... ports ) : constructor
```

Syntax 112—C++: User-defined covergroup instantiation

pss::covergroup_inst<covergroup>

Defined in **`pss/covergroup_inst.h`** (see C.19).

```
template <> class covergroup_inst<covergroup>;
```

Class for instantiating an in-line covergroup instance.

Member functions

```
covergroup_inst (const std::string &name, const options &opts) : constructor
template <class... R> covergroup_inst (const std::string &name,
    std::function<void(void)> body) : constructor
```

Syntax 113—C++: In-line covergroup instantiation

19.2.3 Examples

Example 233 and Example 234 create a covergroup instance with a formal parameter list.

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

struct s {
   rand color_e color;

   covergroup cs1(color_e c) {
      c : coverpoint c;
   }

   cs1 cs1_inst(color);
}
```

Example 233—DSL: Creating a covergroup instance with a formal parameter list

```
PSS_ENUM(color_e, red, green, blue);

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

   class cs1 : public covergroup {...
      attr<color_e> c {"c"};

      coverpoint cp_c {"c", c};
   };
   type_decl<cs1> cs1_t;

   covergroup_inst<cs1> cs1_inst {"cs1_inst", color};
};
...
```

Example 234—C++: Creating a covergroup instance with a formal parameter list

Example 235 and Example 236 create a covergroup instance and specifying instance options.

```
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 235—DSL: Creating a covergroup instance with instance options

```
PSS_ENUM(color_e, red, green, blue);

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

   class cs1 : public covergroup { ...
      attr<color_e> c {"c"};
      coverpoint c_cp {"c", c};
   };
   type_decl<cs1> cs1_t;

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

Example 236—C++: Creating a covergroup instance with instance options

Example 237 and Example 238 create an in-line covergroup instance.

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

struct s {
   rand color_e color;

   covergroup {
      option.at_least = 2;
      c: coverpoint color;
   } cs1_inst;
}
```

Example 237—DSL: Creating an in-line covergroup instance

```
PSS_ENUM(color_e, red, green, blue);

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

   covergroup_inst<> cs1_inst { "cs1_inst", [&](){
      options {at_least(2)};
      coverpoint{"c", color};
   }
   };
};
...
```

Example 238—C++: Creating an in-line covergroup instance

19.3 Defining coverage points

A **covergroup** can contain one or more coverage points. A coverage point specifies a numeric expression or **enum** that is to be covered. Each coverage point includes a set of bins associated with the sampled values of the covered expression. The bins can be explicitly defined by the user or automatically created by the PSS processing tool. The syntax for specifying coverage points is shown in Syntax 114, Syntax 115, Syntax 116, and Syntax 117.

Evaluation of the coverage point expression (and of its enabling **iff** condition, if any) takes place when the **covergroup** is sampled (see 19.7).

19.3.1 DSL syntax

The syntax for **coverpoint**s is shown in Syntax 114.

```
covergroup_coverpoint ::= [ [ data_type ] coverpoint_identifier : ] coverpoint
   expression [ iff ( expression ) ] bins_or_empty
bins_or_empty ::=
     { { covergroup_coverpoint_body_item } }
   | ;
covergroup_coverpoint_body_item ::=
     covergroup_option
   | covergroup_coverpoint_binspec
```

Syntax 114—DSL: coverpoint declaration

The following also apply:

a) A **coverpoint** coverage point creates a hierarchical scope and can be optionally labeled. The label (*coverpoint_identifier*) designates the name of the coverage point. This name can be used to add this coverage point to a cross coverage specification. If the coverage point is associated with a single variable and the label is omitted, the variable name becomes the name of the coverage point. A coverage point on an expression is required to specify a label.

b) A data type for the coverpoint may be specified. The data type shall be a numeric or **enum** type. If a data type is specified, then a label shall also be specified.

c) If a data type is specified, the **coverpoint** expression shall be assignment compatible with the data type. Values for the **coverpoint** shall be of the specified data type and shall be determined as though the **coverpoint** expression were assigned to a variable of the specified type.

d) If no data type is specified, the inferred type for the **coverpoint** shall be the self-determined type of the **coverpoint** expression.

e) The expression within the **iff** construct specifies an optional condition that disables coverage sampling for that **coverpoint**. If the **iff** *expression* evaluates to *false* at a sampling point, the coverage point is not sampled.

f) A coverage point bin associates a name and a count with a set of values. The count is incremented every time the coverage point matches one of the values in the set. The bins for a coverage point can automatically be created by the PSS processing tool or explicitly defined using the **bins** construct to name each bin. If the **bins** are not explicitly defined, they are automatically created by the PSS processing tool. The number of automatically created bins can be controlled using the **`auto_bin_max`** coverage option. Coverage options are described in Table 20.

g) The **default** specification defines a bin that is associated with none of the defined value bins. The default bin catches the values of the coverage point that do not lie within any of the defined bins. However, the coverage calculation for a coverage point shall not take into account the coverage captured by the default bin. The default bin is also excluded from cross coverage. The default is useful for catching unplanned or invalid values. A **default** bin specification cannot be explicitly ignored. It shall be an error for bins designated as **ignore_bins** to also specify **default**.

19.3.2 C++ syntax

The corresponding C++ syntax for Syntax 114 is shown in Syntax 115, Syntax 116, and Syntax 117.

pss::coverpoint

Defined in **`pss/covergroup_coverpoint.h`** (see C.16).

```
class coverpoint;
```

Class for declaring a coverpoint.

Member functions

```
template <class... T> coverpoint(const std::string &name,
    const detail::AlgebExpr &target,
    const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
template <class... T> coverpoint(const std::string &name,
    const detail::AlgebExpr &target, const iff &cp_iff,
    const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
template <class... T> coverpoint(const std::string &name,
    const detail::AlgebExpr &target, const options &cp_options,
    const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
template <class... T> coverpoint(const std::string &name,
    const detail::AlgebExpr &target, const iff &cp_iff,
    const options &cp_options,
    const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
```

Syntax 115—C++: coverpoint declaration

pss::coverpoint

Defined in **`pss/covergroup_coverpoint.h`** (see C.16).

```
    class coverpoint;
```

Class for declaring a coverpoint.

Constructors for unnamed coverpoints.

Member functions

```
    template <class... T> coverpoint(const detail::AlgebExpr &target,
        const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
    template <class... T> coverpoint(const detail::AlgebExpr &target,
        const iff &cp_iff,
        const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
    template <class... T> coverpoint(const detail::AlgebExpr &target,
        const options &cp_options,
        const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
    template <class... T> coverpoint(const detail::AlgebExpr &target,
        const iff &cp_iff, const options &cp_options,
        const T&... /* bins|ignore_bins|illegal_bins */ bin_items) : constructor
```

Syntax 116—C++: constructors for unnamed coverpoint declaration

pss::iff

Defined in **`pss/covergroup_iff.h`** (see C.18).

```
    class iff;
```

Class for specifying an iff condition on a coverpoint.

Member functions

```
    iff(const detail::AlgebExpr &expr)   : constructor
```

Syntax 117—C++: Specifying an iff condition on a coverpoint

19.3.3 Examples

In Example 239 and Example 240, coverage point `s0` is covered only if `is_s0_enabled` is *true*.

```
struct s {
    rand bit[4] s0;
    rand bool   is_s0_enabled;

    covergroup {
        coverpoint s0 iff (is_s0_enabled);
    } cs4;
}
```

Example 239—DSL: Specifying an iff condition

```
class s : public structure {...
    rand_attr<bit>  s0 {"s0", width(4)};
    rand_attr<bool> is_s0_enabled {"is_s0_enabled"};

    covergroup_inst<> cs4 { "cs4", [&]() {
        coverpoint s0 {s0, iff(is_s0_enabled)};
    }
    };
};
...
```

Example 240—C++: Specifying an iff condition

19.3.4 Specifying bins

The **bins** construct creates a separate bin for each value in the given range list or a single bin for the entire range of values. The syntax for defining bins is shown in Syntax 118, Syntax 119 and Syntax 120.

19.3.4.1 DSL syntax

The syntax for **bin**s is shown in Syntax 118.

covergroup_coverpoint_binspec ::= bins_keyword identifier
  [**[** [constant_expression] **]**] **=** coverpoint_bins
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**
covergroup_expression ::= expression

Syntax 118—DSL: bins declaration

The following also apply:

a) To create a separate bin for each value (an array of bins), add square brackets (**[ ]**) after the bin name.
   1) To create a fixed number of bins for a set of values, a single positive integral expression can be specified inside the square brackets.
   2) The bin name and optional square brackets are followed by a *covergroup_range_list* that specifies the set of values associated with the bin.
   3) It shall be legal to use the range value form *expression..* and *..expression* to denote a range that extends to the upper or lower value (respectively) of the coverpoint data type.

b) If a fixed number of bins is specified and that number is smaller than the specified number of values, the possible bin values are uniformly distributed among the specified bins.
   1) The first *N* specified values (where *N* = int(number of values / number of bins)) are assigned to the first bin, the next *N* specified values are assigned to the next bin, etc.
   2) Duplicate values are retained; thus, the same value can be assigned to multiple bins.
   3) If the number of values is not evenly divisible by the number of bins, then the last bin will include the remaining items, e.g., for

      ```
      bins fixed [4] = [1..10, 1, 4, 7];
      ```

      The 13 possible values are distributed as follows: `<1,2,3>`, `<4,5,6>`, `<7,8,9>`, `<10,1,4,7>`.

c) A *covergroup_expression* is an *expression*. In the case of a **with** *covergroup_expression*, the expression can involve constant terms and the **coverpoint** variable (see 19.3.5).

19.3.4.2 C++ syntax

The corresponding C++ syntax for Syntax 118 is shown in Syntax 119 and Syntax 120. Classes with the same C++ API are also defined for **illegal_bins** and **ignore_bins**. See also C.15.

pss::bins

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

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

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

Member functions

`bins(const std::string &name)` : constructor for default bins
`bins(const std::string &name, const range &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 119—C++: coverpoint bins with template parameter of bit or int

pss::bins

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

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

Syntax 120—C++: coverpoint bins with template parameter of vec<bit> or vec<int>

19.3.4.3 Examples

In Example 241 and Example 242, 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 bins `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 241—DSL: Specifying bins

```
class s : public structure { ...
   rand_attr<bit> v_a {"v_a", width(10)};

   covergroup_inst<> cs { "cs", [&]() {
      coverpoint v_a { v_a,
         bins<bit>      {"a", range(0,63)(65)},
         bins<vec<bit>> {"b", range(127,150)(148,191)},
         bins<vec<bit>> {"c", range(200)(201)(202)},
         bins<bit>      {"d", range(1000, upper)},
         bins<vec<bit>> {"others"}
      };
   }
   };
};
...
```

Example 242—C++: Specifying bins

19.3.5 Coverpoint bin with covergroup expressions

The **with** clause specifies that only those values in the *covergroup_range_list* (see Syntax 118) that satisfy the given expression (i.e., for which the expression evaluates to *true*) are included in the bin. In the expression, the name of the **coverpoint** shall be used to represent the candidate value. The candidate value is of the same type as the **coverpoint**.

The **with** clause behaves as if the expression were evaluated for every value in the *covergroup_range_list* at the time the covergroup instance is created. The **with** *covergroup_expression* is applied to the set of values in the *covergroup_range_list* prior to distribution of values to the bins. The result of applying a **with** *covergroup_expression* shall preserve multiple, equivalent bin items as well as the bin order. The intent of these rules is to allow the use of non-simulation analysis techniques to calculate the bin (e.g., formal symbolic analysis) or for caching of previously calculated results.

Consider Example 243 and Example 244, where the bin definition selects all values from `0` to `255` that are evenly divisible by `3`.

```
struct s {
   rand bit[8] x;

   covergroup {
      a: coverpoint x {
            bins mod3[] = [0..255] with ((a % 3) == 0);
      }
   } cs;
}
```

Example 243—DSL: Select constrained values between 0 and 255

```
class s : public structure { ...
   rand_attr<bit> x {"x", width(8)};

   covergroup_inst<> cs { "cs", [&]() {
      coverpoint a { "a", x,
         bins<vec<bit>> {"mod3", range(0,255)}.with ((a % 3) == 0)
      };
   }
   };
};
...
```

Example 244—C++: Select constrained values between 0 and 255

The name of the **coverpoint** itself may be used in place of the *covergroup_range_list*, preceding the **with** keyword, to denote all values of the **coverpoint**. Only the name of the **coverpoint** containing the bin being defined shall be allowed.

In Example 245 and Example 246, **coverpoint** name `a` is used in place of the *covergroup_range_list* to denote that the **with** *covergroup_expression* will be applied to all values of the **coverpoint**.

```
struct s {
   rand bit[8] x;

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

Example 245—DSL: Using with in a coverpoint

```
class s : public structure {...
   rand_attr<bit> x {"x", width(8)};

   covergroup_inst<> cs { "cs", [&]() {
      coverpoint a { "a", x,
          bins<vec<bit>> {"mod3", a}.with((a % 3) == 0)
      };
   }
   };
};
...
```

Example 246—C++: Using with in a coverpoint

19.3.6 Automatic bin creation for coverage points

If a coverage point does not define any bins, PSS automatically creates bins. This provides an easy-to-use mechanism for binning different values of a coverage point. Users can either let the tool automatically create bins for coverage points or explicitly define named bins for each coverage point.

When the automatic bin creation mechanism is used, PSS creates *N* bins to collect the sampled values of a coverage point. The value *N* is determined as follows:

— For an **enum** coverage point, *N* is the cardinality of the enumeration.

— For a numeric coverage point, *N* is the minimum of 2^M and the value of the **`auto_bin_max`** option (see Table 20), where *M* is the number of bits needed to represent the coverage point.

If the number of automatic bins is smaller than the number of possible values ($N < 2^M$), the 2^M values are uniformly distributed in the *N* bins. If the number of values, 2^M, is not divisible by *N*, then the last bin will include the additional remaining items. For example, if *M* is `3` and *N* is `3`, the eight possible values are distributed as follows: `<0..1>`, `<2..3>`, `<4..7>`.

PSS implementations can impose a limit on the number of automatic bins. See Table 20 for the default value of **`auto_bin_max`**.

Each automatically created bin will have a name of the form **`auto[`**`value`**`]`**, where *`value`* is either a single coverage point value or the range of coverage point values included in the bin (in the form *`low`*`..`*`high`*). For enumeration types, *`value`* is the named constant (enum item) associated with the particular enumeration value.

19.3.7 Excluding coverage point values

A set of values associated with a coverage point can be explicitly excluded from coverage by specifying them as **ignore_bins**. See Example 247 and Example 248.

All values associated with ignored bins are excluded from coverage. Each ignored value is removed from the set of values associated with any coverage bin. The removal of ignored values shall occur after distribution of values to the specified bins.

Examples

Example 247 and Example 248 may result in a bin that is associated with no values or sequences. Such empty bins are excluded from coverage.

```
struct s {
   rand bit[4] a;

   covergroup {
      coverpoint a {
         ignore_bins ignore_vals = [7, 8];
      }
   } cs23;
}
```

Example 247—DSL: Excluding coverage point values

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

   covergroup_inst<> cs23 { "cs23", [&]() {
      coverpoint a_cp { a,
         ignore_bins<bit> {"ignore_vals", range(7)(8)}
      };
   }
   };
};
...
```

Example 248—C++: Excluding coverage point values

19.3.8 Specifying illegal coverage point values

A set of values associated with a coverage point can be marked as illegal by specifying them as **illegal_bins**. See Example 249 and Example 250.

All values associated with illegal bins are excluded from coverage. Each illegal value is removed from the set of values associated with any coverage bin. The removal of illegal values shall occur after the distribution of values to the specified bins. If an illegal value occurs, a runtime error shall be issued. Illegal bins take precedence over any other bins, i.e., they result in a runtime error even if they are also included in another bin.

Examples

Example 249 and Example 250 may result in a bin that is associated with no values or sequences. Such empty bins are excluded from coverage.

```
struct s {
   rand bit[4] a;

   covergroup {
      coverpoint a {
         illegal_bins illegal_vals = [7, 8];
      }
   } cs23;
}
```

Example 249—DSL: Specifying illegal coverage point values

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

   covergroup_inst<> cs23 { "cs23", [&]() {
      coverpoint a_cp { a,
         illegal_bins<bit> {"illegal_vals", range(7)(8)}
      };
   }
   };
};
...
```

Example 250—C++: Specifying illegal coverage point values

19.3.9 Value resolution

A *coverpoint expression*, the expressions in a **bins** construct, and the **coverpoint** type, if present, are all involved in comparison operations in order to determine into which bins a particular value falls. Let *e* be the coverpoint expression and *b* be an expression in a **bins** *covergroup_range_list*. The following rules shall apply when evaluating *e* and *b*:

a) If there is no coverpoint type, the effective type of *e* shall be self-determined. In the presence of a coverpoint type, the effective type of *e* shall be the coverpoint type.

b) *b* shall be statically cast to the effective type of *e*. An implementation shall issue a warning under the following conditions:

   1) If the effective type of *e* is unsigned and *b* is signed with a negative value.

   2) If assigning *b* to a variable of the effective type of *e* would yield a value that is not equal to *b* under normal comparison rules for ==.

If a warning is issued for a **bins** element, the following rules shall apply:

c) If an element of a bins *covergroup_range_list* is a singleton value *b*, that element shall not appear in the bins values.

d) If an element of a bins *covergroup_range_list* is a range `b1..b2` and there exists at least one value in the range for which a warning would not be issued, the range shall be treated as containing the intersection of the values in the range and the values expressible by the effective type of *e*.

Examples

Example 251 leads to the following:

— For `b1`, a warning is issued for the range `6..10`. `b1` is treated as though it had the specification [`1, 2..5, 6..7`].

— For `b2`, a warning is issued for the range `1..10` and for the values `-1` and `15`. `b2` is treated as though it had the specification [`1..7`].

— For `b3`, a warning is issued for the ranges `2..5` and `6..10`. `b3` is treated as though it had the specification [`1, 2..3`].

— For `b4`, a warning is issued for the range `1..10` and for the value `15`. `b4` is treated as though it had the specification [`-1, 1..3`].

```
struct s {
   rand bit[3] p1;        // type expresses values in the range  0 to 7
   int [3]     p2;        // type expresses values in the range -4 to 3

   covergroup {
      coverpoint p1 {
         bins b1 = [1, 2..5, 6..10]; // warning issued for range 6..10
         bins b2 = [-1, 1..10, 15];  // warning issued for range 1..10
      }                              //             and values -1 and 15
      coverpoint p2 {
         bins b3 = [1, 2..5, 6..10]; // warning issued for ranges 2..5
                                     //                       and 6..10
         bins b4 = [-1, 1..10, 15];  // warning issued for range  1..10
      }                              //                     and value 15
   } c1;
}
```

Example 251—DSL: Value resolution

19.4 Defining cross coverage

A **covergroup** can specify cross coverage between two or more coverage points or variables. Cross coverage is specified using the **cross** construct (see Syntax 121 and Syntax 122). When a variable `V` is part of a cross coverage, the PSS processing tool implicitly creates a coverage point for the variable, as if it had been created by the statement `coverpoint V;`. Thus, a *cross* involves only coverage points. Expressions cannot be used directly in a **cross**; a coverage point must be explicitly defined first.

19.4.1 DSL syntax

Syntax 121 declares a **cross**.

covergroup_cross ::= *covercross*_identifier **: cross**
  *coverpoint*_identifier { **,** *coverpoint*_identifier }
  [**iff (** expression **)**] cross_item_or_null
*covercross*_identifier ::= identifier
cross_item_or_null ::=
    { { covergroup_cross_body_item } **}**
  | **;**
covergroup_cross_body_item ::=
    covergroup_option
  | covergroup_cross_binspec
covergroup_cross_binspec ::=
  bins_keyword identifier **=** *covercross*_identifier **with (** covergroup_expression **) ;**
covergroup_expression ::= expression

Syntax 121—DSL: cross declaration

The following also apply:

a) The label is required for a **cross**. The expression within the optional **iff** provides a conditional sampling guard for the cross coverage. If the condition evaluates to *false* at any sampling point, the cross coverage is not sampled.

b) Cross coverage of a set of *N* coverage points is defined as the coverage of all combinations of all bins associated with the N coverage points, i.e., the Cartesian product of the *N* sets of coverage point bins. See also Example 252 and Example 253.

19.4.2 C++ syntax

The corresponding C++ syntax for Syntax 121 is shown in Syntax 122.

pss::cross

Defined in **`pss/covergroup_cross.h`** (see C.17).

```
class cross;
```

Class for capturing a coverpoint cross. In all variadic-template constructors, fields of coverpoint, attr, rand_attr, bins, ignore_bins, and illegal_bins may be specified.

Member functions

```
template <class... T> cross( const std::string &name, const T&... items) :
    constructor
template <class... T> cross( const std::string &name, const iff &cp_iff,
     const T&... items) : constructor
template <class... T> cross( const std::string &name,
    const options &cp_options, const T&... items) : constructor
template <class... T> cross( const std::string &name, const iff &cp_iff,
    const options &cp_options, const T&... items) : constructor
```

Syntax 122—C++: cross declaration

19.4.3 Examples

The covergroup `cov` in Example 252 and Example 253 specifies the cross coverage of two 4-bit variables, `a` and `b`. The PSS processing tool implicitly creates a coverage point for each variable. Each coverage point has 16 bins, specifically `auto[0]`..`auto[15]`. The cross of `a` and `b` (labeled `aXb`), therefore, has 256 cross products and each cross product is a bin of `aXb`.

```
struct s {
    rand bit[4] a, b;

    covergroup {
        aXb : cross a, b;
    } cov;
}
```

Example 252—DSL: Specifying a cross

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

   covergroup_inst<> cov { "cov", [&]() {
      cross aXb { "aXb", a, b};
   }
   };
};
...
```

Example 253—C++: Specifying a cross

19.5 Defining cross bins

In addition to specifying the coverage points that are crossed, PSS allows the definition of cross coverage bins. Cross coverage bins are specified to group together a set of cross products. A *cross coverage bin* associates a name and a count with a set of cross products. The count of the bin is incremented any time any of the cross products match; i.e., every coverage point in the **cross** matches its corresponding bin in the cross product.

User-defined bins for cross coverage are defined using **bins with** expressions. The names of the **coverpoint**s used as elements of the cross coverage are used in the **with** expressions. User-defined cross bins and automatically generated bins can coexist in the same **cross**. Automatically generated bins are retained for those cross products that do not intersect cross products specified by any user-defined cross bin.

Examples

Consider Example 254 and Example 255, where two coverpoints are declared on fields a and b. A cross coverage is specified between these two coverpoints. The small_a_b bin collects those bins where both a and b <= 10.

```
struct s {
   rand bit[8] a, b;

   covergroup {
      coverpoint a {
         bins low[] = [0..127];
         bins high = [128..255];
      }
      coverpoint b {
         bins two[] = b with (b%2 == 0);
      }

      X : cross a, b {
         bins small_a_b = X with (a<=10 && b<=10);
      }
   } cov;
}
```

Example 254—DSL: Specifying cross bins

```
class s : public structure {...
   rand_attr<bit> a {"a", width(8)};
   rand_attr<bit> b {"b", width(8)};

   covergroup_inst<> cov { "cov", [&]() {
      coverpoint cp_a { "a", a,
         bins<vec<bit>> {"low", range(0,127)},
         bins<bit> {"high", range(128,255)}
      };
      coverpoint cp_b { "b", b,
         bins<vec<bit>> {"two", b}.with((b%2) == 0)
      };

      cross X { "X", cp_a, cp_b,
         bins<bit>{"small_a_b", X}.with(a<=10 && b<=10)
      };
   }
   };
};
...
```

Example 255—C++: Specifying cross bins

19.6 Specifying coverage options

Options control the behavior of the **covergroup**, **coverpoint**, and **cross** elements. There are two types of options: those that are specific to an instance of a **covergroup** and those that specify an option for the **covergroup** type as a whole. Instance-specific options can be specified when creating an instance of a reusable **covergroup**. Both type and instance-specific options can be specified when defining an in-line **covergroup** instance.

Specifying a value for the same option more than once within the same **covergroup** definition shall be an error. Specifying a value for the option more than once when creating a **covergroup** instance shall be an error.

Table 20 lists the instance-specific **covergroup** options and their description. Each instance of a reusable **covergroup** type can initialize an instance-specific option to a different value.

Table 20—Instance-specific covergroup options

Option name	Default	Description
weight=number	1	If set at the **covergroup** syntactic level, it specifies the weight of this **covergroup** instance for computing the overall instance coverage. If set at the **coverpoint** (or **cross**) syntactic level, it specifies the weight of a **coverpoint** (or **cross**) for computing the instance coverage of the enclosing **covergroup**. The specified weight shall be a non-negative integral value.
goal=number	100	Specifies the target goal for a **covergroup** instance or for a **coverpoint** or **cross**. The specified value shall be a non-negative integral value.

Table 20—Instance-specific covergroup options (Continued)

Option name	Default	Description
name=string	unique name	Specifies a name for the **covergroup** instance. If unspecified, a unique name for each instance is automatically generated by the tool.
comment=string	""	A comment that appears with the **covergroup** instance or with a **coverpoint** or **cross** of a **covergroup** instance. The comment is saved in the coverage database and included in the coverage report.
at_least=number	1	Minimum number of hits for each bin. A bit with a hit count that is less than *number* is not considered covered. The specified value shall be a non-negative integral value.
detect_overlap=bool	false	When *true*, a warning is issued if there is an overlap between the range list of two bins of a **coverpoint**.
auto_bin_max=number	64	Maximum number of automatically created bins when no bins are explicitly defined for a **coverpoint**. The specified value shall be a positive integral value.
per_instance=bool	false	Each instance contributes to the overall coverage information for the **covergroup** type. When *true*, coverage information for this **covergroup** instance shall be saved in the coverage database and included in the coverage report. When *false*, implementations are not required to save instance-specific information.

Instance options can only be specified at the **covergroup** level. Except for the **`weight`**, **`goal`**, **`comment`**, and **`per_instance`** options (see Table 20), all other options set at the covergroup syntactic level act as a default value for the corresponding option of all **coverpoint**s and **cross**es in the **covergroup**. Individual **coverpoint**s and **cross**es can overwrite these defaults. When set at the **covergroup** level, the **`weight`**, **`goal`**, **`comment`**, and **`per_instance`** options do not act as default values to the lower syntactic levels.

The identifier *type_option* is used to specify type options when declaring a **covergroup**:

```
type_option.member_name = constant_expression ;
```

19.6.1 C++ syntax

Syntax 123, Syntax 124, Syntax 125, Syntax 126, Syntax 127, Syntax 128, Syntax 129, Syntax 130, and Syntax 131 show how to define the C++ options and option values.

pss::options

Defined in **`pss/covergroup_options.h`** (see C.20).

```
class options;
```

Class for capturing coverpoint, cross, and covergroup options.

Member functions

```
template <class... O> options(const O&... /*
        weight
      | goal
      | name
      | comment
      | detect_overlap
      | at_least
      | auto_bin_max
      | per_instance */ options)  : constructor
```

Syntax 123—C++: options declaration

pss::weight

Defined in **`pss/covergroup_options.h`** (see C.20).

```
class weight;
```

Class for capturing the weight coverage option.

Member functions

```
weight(uint32_t w)  : constructor
```

Syntax 124—C++: weight option

pss::goal

Defined in **`pss/covergroup_options.h`** (see C.20).

```
class goal;
```

Class for capturing the goal coverage option.

Member functions

```
goal(uint32_t w)  : constructor
```

Syntax 125—C++: goal option

pss::name

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class name;
```

Class for capturing the name coverage option.

Member functions

```
    name(const std::string &name)
```
: constructor

Syntax 126—C++: name option

pss::comment

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class comment;
```

Class for capturing the comment coverage option.

Member functions

```
    comment(const std::string &c)
```
: constructor

Syntax 127—C++: comment option

pss::detect_overlap

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class detect_overlap;
```

Class for capturing the detect_overlap coverage option.

Member functions

```
    detect_overlap(bool detect)
```
: constructor

Syntax 128—C++: detect_overlap option

pss::at_least

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class at_least;
```

Class for capturing the at_least coverage option.

Member functions

`at_least(uint32_t l)` : constructor

Syntax 129—C++: at_least option

pss::auto_bin_max

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class auto_bin_max;
```

Class for capturing the auto_bin_max coverage option.

Member functions

`auto_bin_max(uint32_t l)` : constructor

Syntax 130—C++: auto_bin_max option

pss::per_instance

Defined in **`pss/covergroup_options.h`** (see C.20).

```
    class per_instance;
```

Class for capturing the per_instance coverage option.

Member functions

`per_instance(bool v)` : constructor

Syntax 131—C++: per_instance option

19.6.2 Examples

The instance-specific options mentioned in Table 20 can be set in the **covergroup** definition. Example 256 and Example 257 show this, and how coverage options can be set on a specific **coverpoint**.

```
covergroup cs1 (bit[64] a_var, bit[64] b_var) {
   option.per_instance = true;
   option.comment = "This is CS1";

   a : coverpoint a_var {
      option.auto_bin_max = 128;
   }

   b : coverpoint b_var {
      option.weight = 10;
   }
}
```

Example 256—DSL: Setting options

```
class cs1 : public covergroup {...
   attr<bit> a_var {"a_var", width(64)};
   attr<bit> b_var {"b_var", width(64)};

   options opts {
      per_instance(true),
      comment("This is CS1")
   };

   coverpoint a { "a", a_var,
      options {
         auto_bin_max(64)
      }
   };

   coverpoint b { "b", b_var,
      options {
         weight(10)
      }
   };
};
...
```

Example 257—C++: Setting options

19.7 covergroup sampling

Coverage credit can be taken once execution of the **action** containing **covergroup** instance(s) is complete. Thus, by default, all **covergroup** instances that are created as a result of a given **action**'s traversal are sampled when that **action**'s execution completes. Table 21 summarizes when **covergroup**s are sampled, based on the context in which they are instantiated.

Table 21—covergroup sampling

Instantiation context	**Sampling point**
Flow objects	Sampled when the outputting **action** completes traversal.
Resource objects	Sampled before the first **action** referencing them begins traversal.
Action	Sampled when the instantiating **action** completes traversal.
Data structures	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.

19.8 Per-type and per-instance coverage collection

By default, **covergroup**s collect coverage on a *per-type* basis. This means that all coverage values sampled by instances of a given **covergroup** type, where **`per_instance`** is *false*, are merged into a single collection.

Per-instance coverage is collected when **`per_instance`** is *true* for a given **covergroup** instance and when a contiguous path of named handles exists from the root component or root action to where new instances of the containing type are created. If one of these conditions is not satisfied, *per-type* coverage is collected for the **covergroup** instance.

19.8.1 Per-instance coverage of flow and resource objects

Per-instance coverage of flow objects (**buffer** (see 14.1), **stream** (see 14.2), **state** (see 14.3)) and **resource** objects (see 15.1)) is collected for each pool of that type.

In Example 258, there is one pool (`pss_top.b1_p`) of buffer type `b1`. When the PSS model runs, coverage from all 10 executions of `P_a` and `C_a` is placed in the same coverage collection that is associated with the pool through which `P_a` and `C_a` exchange the buffer object `b1`.

```
enum mode_e { M0, M1, M2 }

buffer b1 {
   rand mode_e mode;

   covergroup {
      option.per_instance = true;

      coverpoint mode;
   } cs;
}

component pss_top {
   pool b1 b1_p;
   bind b1_p *;

   action P_a {
      output b1 b1_out;
   }

   action C_a {
      input b1 b1_in;
   }

   action entry {
      activity {
         repeat (10) {
            do C_a;
         }
      }
   }
}
```

Example 258—DSL: Per-instance coverage of flow objects

19.8.2 Per-instance coverage in actions

Per-instance coverage for **action**s is enabled when **`per_instance`** is *true* for a **covergroup** instance and when a contiguous path of named handles exists from the root action to the location where the **covergroup** is instantiated.

In Example 259, a contiguous path of named handles exists from the root action to the covergroup instance inside `a1` (`entry.a1.cg`). Coverage data collected during traversals of action `A` are placed in a coverage collection unique to this named path. Plus, four samples are placed in the coverage collection associated with the instance path `entry.a1.cg` because the named action handle `a1` is traversed four times.

Also in Example 259, a contiguous path of named handles does not exist from the root action to the covergroup instance inside the action traversal by type (`do A`). In this case, coverage data collected during the 10 traversals of action `A` by type (`do A`) are placed in the per-type coverage collection associated with covergroup type `A::cg`.

```
enum mode_e { M0, M1, M2 }

component pss_top {

    action A {
        rand mode_e mode;

        covergroup {
            option.per_instance = true;

            coverpoint mode;
        } cg;
    }

    action entry {
        A      a1;
        activity {
            repeat (4) {
                a1;
            }
            repeat (10) {
                do A;
            }
        }
    }
}
```

Example 259—DSL: Per-instance coverage in actions

20. Type inheritance, extension, and overrides

PSS supports the concepts of *object-oriented inheritance* and *type extension* to maximize reuse and portability of the model. *Type inheritance* allows the declaration of model entities such as actions, objects, components and struct types to be derived from a *base type* (or *super-type*), where the new *derived type* (or *subtype*) includes all attributes and other members of the base type, and allows the declaration of the derived type to add new members or mask the definition of existing members. *Type extension* allows the declaration of additional fields in an *existing* type using a separate declaration. Type inheritance is described in 20.1, and type extension is described in 20.2. *Type overrides* allow type-specific and instance-specific replacement of the declared type of a field with a specified subtype, and are described in 20.5.

20.1 Type inheritance

For actions, components, structs, data flow and resource objects, the declaration may include an optional *super-spec* qualifier to declare a base type of the same type category (**action**, **component**, **struct**, **buffer**, **stream**, **state**, **resource**), from which the element is to be derived. The only exception is that data flow and resource objects may inherit from an element of the same type category or from a **struct**.

A derived type includes all elements from the base type, and may declare new elements that may or may not have the same name as a corresponding element in the base type. For fields declared in a derived type with the same name as a field in the base type, the derived type's field shadows (masks) the base type's field, and the base type's field may be referenced as "**`super.`**`<name>`". Certain unnamed elements, such as *activities* and procedural *exec* blocks, may invoke the corresponding element(s) from the base type by the "**`super;`**" statement.

The behavior of specific elements when declared in a derived type is shown in Table 22.

Table 22—Derived type element behaviors

Element kind	In a component	In an action	In a struct, data flow or resource object
activity	n/a	shadow, may call **super;**	n/a
dynamic constraint	n/a	shadow (may access base constraint as **super.**name)	shadow (may access base constraint as **super.**name)
named static constraint	n/a	shadow	shadow
unnamed static constraint	n/a	added	added
field	shadow (may access base field as **super.**name)	shadow (may access base field as **super.**name)	shadow (may access base field as **super.**name)
function	shadow (may call base function as **super.**name(args))	n/a	n/a
override declaration	added	added	n/a
object pool bind	added	n/a	n/a
procedural exec block	shadow, may call **super;**	shadow, may call **super;**	shadow, may call **super;**
target-template exec block	n/a	shadow	shadow

Activities in derived actions shadow the activities from the base action type. However, the "**`super;`**" statement may be used to traverse the base activity (or activities). See Example 117 in 13.6.

Procedural *exec* blocks defined in a derived type shadow same-kind *exec* block(s) defined in the base type. The *exec* block in the derived type may include the "**`super;`**" statement, which will execute the contents of the corresponding base-type *exec* block(s) at that point. See 22.1.5.1 and 22.1.5.2.

Target-template *exec* blocks defined in a derived type shadow same-kind *exec* blocks with the same target language identifier in the base type. The "**`super;`**" statement shall not be allowed in a target-template *exec* block.

Example 260 and Example 261 show a simple case of declaring a component `base_c`, which contains an action declaration, `base_a`. Derived component `der_c` inherits from `base_c`, so it is treated as having action `base_a` already declared within it. Note that `base_c` and `der_c` are different component types. Action `der_a` inherits from `base_a`, so it already includes random integer `i` and bit-vector `b`, as well as the unnamed constraint limiting `i` to be less than `10` and constraint `c` forcing `b > 7`. Derived action `der_a` adds an additional random integer, `j`, a new unnamed constraint that relates the values of `i` and `j`, and a new constraint `c` that relates the values of `b` and `j`, shadowing constraint `c` from action `base_a`.

```
component base_c {
  action base_a {
    rand int i;
    rand bit[31:0] b;
    constraint {i < 10;}
    constraint c {b > 7;}
  }
}

component der_c : base_c {
  action der_a : base_a {
    rand int j;
    constraint {j > 5 -> i < 5;}
    constraint c {j < 10 -> b < 128;}
  }
}
```

Example 260—DSL: Declaring derived components and actions

The corresponding C++ example for Example 260 is shown in Example 261.

```
class base_c : public component { ...
  class base_a : public action { ...
    rand_attr<int> i {"i"};
    rand_attr<bit> b {"b", width(31,0)};
    constraint c0 {"c0", i < 10};
    constraint c  {"c" , b > 7 };
  }
  ...
}

class der_c : base_c { ...
  class der_a : base_c::base_a { ...
    rand_attr<int> j {"j"};
    constraint c1 {"c1", if_then{cond(j > 5) , i < 5  }};
    constraint c  {"c" , if_then{cond(j < 10), b < 128}};
  }
  ...
}
```

Example 261—C++: Declaring derived components and actions

When a pool **bind** statement (see 16.4) is used in a base component type, it may also apply to a derived type, provided that any new component instances and actions in the derived type also match the path specification in the **bind** statement and that the types of the object references match the pool type exactly.

In Example 262 and Example 263, the default **bind** statement in `base_c` binds the `cpu_p` pool to the actions `act1_a` and `act2_a` defined therein. Since `der_c` is derived from `base_c`, it also inherits the **bind** statement, which applies to all action definitions in `der_c` that match the path specification. In the context of `der_c`, the default bind statement binds all three actions `act1_a`, `act2_a` and `act3_a` to the `cpu_p` pool.

```
resource cpu_core_s {...
}

component base_c {
  pool[4] cpu_core_s cpu_p;
  bind cpu_p *;
  action act1_a {
    share cpu_core_s cpu_share;
  }
  action act2_a {
    lock cpu_core_s cpu_lock;
  }
}

component der_c : base_c {
  action act3_a {
    share cpu_core_s cpu_share;
  }
  ...
}
```

Example 262—DSL: Default pool with inheritance

The corresponding C++ example for Example 262 is shown in Example 263.

```
struct cpu_core_s : public resource {...};
...

class base_c : public component {
  pool<cpu_core_s> cpu_p{"cpu_p", 4};
  bind b {cpu_p};
  class act1_a : public action {...
    share <cpu_core_s> cpu_share{"cpu_share"};
  };
  class act2_a : public action {...
    lock <cpu_core_s> cpu_lock{"cpu_lock"};
  };
}

class der_c : base_c {
  class act3_a : public action {...
    share <cpu_core_s> cpu_share{"cpu_share"};
  };
  ...
}
```

Example 263—C++: Default pool with inheritance

As mentioned above, a derived type inherits all members from the base type and may declare additional elements specific to the derived type. When a named element (other than a function) is declared in the derived type with the same name as an element in the base type, the derived type's declaration shadows (masks) the base type's declaration (as with constraint `c` in Example 260 and Example 261).

When the shadowed element is a function, the function call is *polymorphic*, that is, the actual function called depends on its context component. In Example 264 and Example 265, component `der_c` shadows the definition of function `foo()` in component `base_c`. Action `call_foo` invokes the appropriate definition of `foo()` depending on the type of its context component. Action `test` schedules `call_foo` in the context of a component of type `base_c`, followed by `call_foo` in the context of `der_c`. Assuming that target function `print()` prints out a line, executing `test` will print the following:

```
base_c::foo
der_c::foo
```

```
component base_c {
   function void foo() {
      print("base_c::foo");
   }

   action call_foo {
      exec body {
         comp.foo();
      }
   }
}

component der_c : base_c {
   function void foo() {
      print("der_c::foo");
   }
};

component pss_top {
   base_c b;
   der_c d;
   action test {
      base_c::call_foo b_foo, d_foo;
      constraint {b_foo.comp == this.comp.b;
                  d_foo.comp == this.comp.d;}
      activity {
         b_foo;
         d_foo;
      }
   }
}
```

Example 264—DSL: Polymorphic function calls

The corresponding C++ example for Example 264 is shown in Example 265.

```
class base_c : public component {
   public:
      function<result<void>()> foo {"foo", [&]() {
         print("base_c::foo");
      } };

      class call_foo : public action {
         exec e { exec::body, [&]() {
         comp<base_c>()->foo();
         }
      };
};

class der_c : public base_c {
   public:
      function<result<void>()> foo {"foo", [&]() {
         print("der_c::foo");
      } };
};

class pss_top : public component {
   public:
      comp_inst<base_c> b{"b"};
      comp_inst<der_c> d{"d"};
      class test : public action {
         public:
            action_handle<base_c::call_foo> b_foo{"b_foo"},d_foo{"d_foo"};
            constraint c1 {"bfoo", b_foo->comp() == comp<pss_top>()->b};
            constraint c2 {"dfoo", d_foo->comp() == comp<pss_top>()->d};
            activity a {
               sequence {
                  b_foo,
                  d_foo
               }
            };
      };
};
```

Example 265—C++: Polymorphic function calls

As discussed in 10.4, the qualified name of an action declared in a component is of the form *component-type*::*action-type*'. In Example 266, the base component dma_base_c declares action xfer_a. The derived component dma_der_c declares the compound action mult_xfer_a, which traverses the xfer_a action. Since dma_der_c inherits the xfer_a action, the anonymous (by type) traversal in mult_xfer_a correctly resolves to the xfer_a action declared in the base component. It is thus not necessary to further qualify the type name xfer_a in the anonymous traversal in mult_xfer_a.

The component dma_test_c instantiates the derived component dma_der_c. The first traversal statement in the activity is an anonymous traversal of the dma_der_c::mult_xfer_a action. The next statement anonymously traverses the dma_base_c::xfer_a action. We can use the dma_base_c path qualifier because the instantiated subcomponent of type dma_der_c *is also* considered a dma_base_c component. It would be illegal to refer to dma_base_c::mult_xfer_a because mult_xfer_a is not declared in dma_base_c. To promote reuse, the third anonymous traversal statement is preferred, referring to dma_der_c::xfer_a, since xfer_a can be used without knowing whether it was declared in the base component or the derived component. Note that, since there is only a single instance of the dma_der_c component, the instance context of these traversals is the same.

```
component dma_base_c {
  action xfer_a {
    ...
  }
}

component dma_der_c : dma_base_c {
  action mult_xfer_a {
    activity {
      repeat(3) {
        do xfer_a; // dma_base_c::xfer_a
      }
    }
  }
}

component dma_test_c {
  dma_der_c dma;

  action test_a {
    activity {
      do dma_der_c::mult_xfer_a;
      do dma_base_c::xfer_a;
      do dma_der_c::xfer_a; // dma_base_c::xfer_a
    }
  }
}
```

Example 266—DSL: Derived type is also a base type

In Example 267, there are two instances of the `dma_der_c` component instantiated in `dma_test_c`. For the first anonymous traversal of `dma_base_c::xfer_a`, either instance may be chosen as context for the `xfer_a` action. In the second anonymous traversal, the **`comp`** attribute is constrained to specify that the context component must be `dma_test_c.dma1`. As stated in 10.6, the static type of the **`comp`** attribute of `dma_der_c::xfer_a` is actually `dma_base_c`, since that is its containing component type (See also 17.1.3).

Because **`comp`** is of type `dma_base_c` and not `dma_der_c`, it would be illegal to refer to fields of `dma_der_c` as relative to **`comp`**, since these fields are not in `dma_base_c`. Rather, fields of `dma_der_c` may be referred to relative to `this.comp.dma1`, which is the actual instance of `dma_der_c` (which is also a `dma_base_c`) in which `xfer_a` will execute. Thus, based on the actual instance of a context component, we can constrain the fields of `xfer_a` even though `xfer_a` may not have visibility otherwise to the `dma_der_c` fields that control the constraints.

```
component dma_base_c {
  action xfer_a {
    rand int i;
    ...
  }
}

component dma_der_c : dma_base_c {
  int j;
  action mult_xfer_a {
    activity {
      repeat(3) {
        do xfer_a; // dma_base_c::xfer_a
      }
    }
  }
}

component dma_test_c {
  dma_der_c dma1, dma2;

  action test_a {
    activity {
      do dma_base_c::xfer_a;
      do dma_der_c::xfer_a with {comp == this.comp.dma1;
                           (this.comp.dma1.j < 8) -> i>4;};
    }
  }
}
```

Example 267—DSL: Use of comp and this.comp with inheritance

20.2 Type extension

*Type extension*s in PSS enable the decomposition of model code so as to maximize reuse and portability. Model entities, actions, objects, components, and data types, may have a number of properties that are logically independent. Moreover, distinct concerns with respect to the same entities often need to be developed independently. Later, the relevant definitions need to be integrated, or woven into one model, for the purpose of generating tests.

Some typical examples of concerns that cut across multiple model entities are:

- Implementation of actions and objects for, or in the context of, some specific target platform/language.
- Model configuration of generic definitions for a specific device under test (DUT) / environment configuration, affecting components and data types that are declared and instantiated elsewhere.
- Definition of functional elements of a system that introduce new properties to common objects, which define their inputs and outputs.

Such crosscutting concerns can be decoupled from one another by using type extensions and then encapsulated as **packages** (see 21.1).

Composite and enumeration types in PSS are extensible. They are declared once, along with their initial definition, and may later be extended any number of times, with new body items being introduced into their scope. Items introduced in extensions may be of the same kinds as those introduced in the initial definition. Extension statements may appear in **package** and **component** definitions.

An extension statement explicitly specifies the kind of type being extended, which must agree with the specific type named (see Syntax 132 and Syntax 133).

The overall definition of any given type in a model is the sum total of its definition statements—the initial one along with extensions in active packages (see 21.1). The semantics of extensions are those of weaving all those statements into a single definition.

Every type extension, regardless of whether it extends a package-level type or a component-level inner type, is associated with the nearest **package** that lexically encloses its definition (an explicit **package** if enclosed in a *package_declaration* statement or otherwise the unnamed global package (see 21.1)).

Members introduced in an extension of a type can be referenced throughout the package in which they were introduced. As a corollary, members introduced in extensions associated with the global package can be referenced everywhere. Members introduced in extensions cannot be referenced *outside* the scope of the package in which the extension is defined unless the reference occurs in a lexical scope that wildcard-imports that package.

These rules concern reference of static members as well as non-static members, and apply regardless of whether fully-qualified static paths are used (for static members).

20.2.1 DSL syntax

```
extend_stmt ::=
   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 132—DSL: type extension

20.2.2 C++ syntax

In C++, extension classes derive from a base class as normal, and then the extension is registered via the appropriate **`extend_xxx<>`** template class:

The corresponding C++ syntax for Syntax 132 is shown in Syntax 133.

pss::extend_action

Defined in **`pss/extend.h`** (see C.27).

```
template <class Foundation, class Extension> class extend_action;
```

Extend an action.

pss::extend_component

Defined in **`pss/extend.h`** (see C.27).

```
template <class Foundation, class Extension> class extend_component;
```

Extend a component.

pss::extend_structure

Defined in **`pss/extend.h`** (see C.27).

```
template <class Foundation, class Extension> class extend_structure;
```

Extend a structure.

pss::extend_enum

Defined in **`pss/extend.h`** (see C.27).

```
template <class Foundation, class Extension> class extend_enum;
```

Extend an enum.

Syntax 133—C++: type extension

20.2.3 Examples

Examples of type extension are shown in Example 268 and Example 269.

```
enum config_modes_e {UNKNOWN, MODE_A=10, MODE_B=20};

component uart_c {
    action configure {
        rand config_modes_e mode;
        constraint {mode != UNKNOWN;}
    }
}

package additional_config_pkg {
    extend enum config_modes_e {MODE_C=30, MODE_D=50}

    extend action uart_c::configure {
        constraint {mode != MODE_D;}
    }
}
```

Example 268—DSL: Type extension

```
PSS_ENUM(config_modes_e, UNKNOWN, MODE_A=10, MODE_B=20);
...
class uart_c : public component { ...
  class configure : public action { ...
    rand_attr<config_modes_e> mode{"mode"};
    constraint mode_c {mode != config_modes_e::UNKNOWN};
  };
  type_decl<configure> configure_decl;
};
type_decl<uart_c> uart_c_decl;

namespace additional_config_pkg {
  PSS_EXTEND_ENUM(config_modes_ext_e, config_modes_e, MODE_C=30, MODE_D=50);
  ...
  // declare action extension for base type configure
  class configure_ext : public uart_c::configure { ...
   constraint mode_c_ext {"mode_c_ext", mode != config_modes_ext_e::MODE_D};
  };
  // register action extension
  extend_action<uart_c::configure, configure_ext>
    extend_action_configure_ext;
};
```

Example 269—C++: Type extension

20.2.4 Composite type extensions

Any kind of member declared in the context of the initial definition of a composite type can be declared in the context of an extension, as per its entity category (**action**, **component**, **buffer**, **stream**, **state**, **resource**, **struct**, or **enum**).

Named type members of any kind, fields in particular, may be introduced in the context of a type extension. Names of fields introduced in an extension shall not conflict with those declared in the initial definition of the type. They shall also be unique in the scope of their type within the **package** in which they are declared. However, field names do not have to be unique across extensions of the same type in different packages.

Fields are always accessible within the scope of the package in which they are declared, shadowing (masking) fields with the same name declared in other packages. Members declared in a different package are accessible if the declaring package is wildcard-imported into the scope of the accessing **package** or **component**, given that the reference is unique. If the same field name or type name is wildcard-imported from two or more separate packages, it shall be an error to reference it.

In Example 270 and Example 271, an **action** type is initially defined in the context of a **component** and later extended in a separate **package**. Ultimately the **action** type is used in a compound action of a parent **component**. The **component** explicitly wildcard-imports the **package** with the extension and can therefore constrain the attribute introduced in the extension.

```
component mem_ops_c {
   enum mem_block_tag_e {SYS_MEM, A_MEM, B_MEM, DDR};

   buffer mem_buff_s {
      rand mem_block_tag_e mem_block;
   }

   pool mem_buff_s mem;
   bind mem *;

   action memcpy {
      input mem_buff_s src_buff;
      output mem_buff_s dst_buff;
   }
}

package soc_config_pkg {
   extend action mem_ops_c::memcpy {
      rand int in [1, 2, 4, 8] ta_width; // introducing new attribute

      constraint { // layering additional constraint
         src_buff.mem_block in [SYS_MEM, A_MEM, DDR];
         dst_buff.mem_block in [SYS_MEM, A_MEM, DDR];
         ta_width < 4 -> dst_buff.mem_block != A_MEM;
      }
   }
}

component pss_top {
   import soc_config_pkg::*;// explicitly importing the package grants
                            // access to types and type members
   mem_ops_c mem_ops;

   action test {
      mem_ops_c::memcpy cpy1, cpy2;
      constraint cpy1.ta_width == cpy2.ta_width;// constraining an
                              // attribute introduced in an extension
      activity {
         repeat (3) {
            parallel { cpy1; cpy2; };
         }
      }
   }
}
```

Example 270—DSL: Action type extension

```
class mem_ops_c : public component { ...
  PSS_ENUM(mem_block_tag_e, SYS_MEM, A_MEM, B_MEM, DDR);
  ...
  struct mem_buff_s : public buffer { ...
    rand_attr<mem_block_tag_e> mem_block {"mem_block"};
  };
  pool <mem_buff_s> mem{"mem"};
  bind b1 {mem};

  class memcpy : public action { ...
    input<mem_buff_s> src_buff {"src_buff"};
    output<mem_buff_s> dst_buff {"dst_buff"};
  };
  type_decl<memcpy> memcpy_decl;
};
...

namespace soc_config_pkg {
  class memcpy_ext : public mem_ops_c::memcpy { ...
    using mem_block_tag_e = mem_ops_c::mem_block_tag_e;
    // introducing new attribute
    rand_attr<int> ta_width {"ta_width", range(1)(2)(4)(8)};
    constraint c { // layering additional constraint
      in { src_buff->mem_block, range(mem_block_tag_e::SYS_MEM)
                                      (mem_block_tag_e::A_MEM)
                                      (mem_block_tag_e::DDR) },
      in { dst_buff->mem_block, range(mem_block_tag_e::SYS_MEM)
                                      (mem_block_tag_e::A_MEM)
                                      (mem_block_tag_e::DDR) },
      if_then { cond(ta_width < 4),
        dst_buff->mem_block != mem_block_tag_e::A_MEM
      }
    };
  };
  extend_action<memcpy_ext, mem_ops_c::memcpy> memcpy_ext_decl;
};

class pss_top : public component { ...
  comp_inst<mem_ops_c> mem_ops {"mem_ops"};
  class test : public action { ...
    action_handle<soc_config_pkg::memcpy_ext> cpy1 {"cpy1"},
                                              cpy2 {"cpy2"};
    // note - handles are declared with action extension class
    // in order to access attributes introduced in the extension
    constraint c { cpy1->ta_width == cpy2->ta_width };
    activity a {
      repeat { 3,
        parallel { cpy1, cpy2 }
      };
    };
  };
  type_decl<test> test_decl;
};
...
```

Example 271—C++: Action type extension

20.2.5 Enumeration type extensions

Enumeration types can be extended in one or more package contexts, introducing new enum items to the domain of all variables of that type. Each enum item in an **enum** type shall be associated with an integer value that is unique across the initial definition and all the extensions of the type. Enum item values are assigned according to the same rules they would be if all the enum items appeared in the initial definition, according to the order of package evaluations. An explicit conflicting value assignment shall be illegal.

An enum item introduced in an extension can be referenced within the **package** in which the extension is defined. Outside that **package**, enum items can be referenced inside a lexical scope that wildcard-imports the respective package.

In Example 272 and Example 273, an **enum** type is initially declared empty and later extended in two independent **package**s. Ultimately items are referenced from a **component** that wildcard-imports both **package**s.

```
package mem_defs_pkg { // reusable definitions
   enum mem_block_tag_e {}; // initially empty

   buffer mem_buff_s {
      rand mem_block_tag_e mem_block;
   }
}
package AB_subsystem_pkg {
   import mem_defs_pkg ::*;

   extend enum mem_block_tag_e {A_MEM, B_MEM};
}
package soc_config_pkg {
   import mem_defs_pkg ::*;

   extend enum mem_block_tag_e {SYS_MEM, DDR};
}
component dma_c {
  import mem_defs_pkg::*;
  action mem2mem_xfer {
    input mem_buff_s src_buff;
    output mem_buff_s dst_buff;
 }
}
extend component dma_c {
   import AB_subsystem_pkg::*; // wildcard-importing the package
   import soc_config_pkg::*;   // grants access to enum items

   action dma_test {

      activity {
         do mem2mem_xfer with {
            src_buff.mem_block == A_MEM;
            dst_buff.mem_block == DDR;
         };
      }
   }
}
```

Example 272—DSL: Enum type extensions

```
namespace mem_defs_pkg {  // reusable definitions
  PSS_ENUM(mem_block_tag_e); // initially empty
  ...
  class mem_buff_s : public buffer { ...
    rand_attr<mem_block_tag_e> mem_block {"mem_block"};
  };
  ...
};

class dma_c : public component { ...
  class mem2mem_xfer : public action { ...
    input <mem_defs_pkg::mem_buff_s> src_buff {"src_buff"};
    output<mem_defs_pkg::mem_buff_s> dst_buff {"dst_buff"};
  };
  type_decl<mem2mem_xfer> mem2mem_xfer_decl;
};
...

namespace AB_subsystem_pkg {
  PSS_EXTEND_ENUM(mem_block_tag_e_ext,
                mem_defs_pkg::mem_block_tag_e, A_MEM, B_MEM);
  };

namespace soc_config_pkg {
  PSS_EXTEND_ENUM(mem_block_tag_e_ext,
                mem_defs_pkg::mem_block_tag_e, SYS_MEM, DDR);
};

class dma_c_ext : public dma_c { ...
  class dma_test : public action { ...
    action_handle<mem2mem_xfer> xfer {"xfer"};

    activity a {
      xfer.with (
           xfer->src_buff->mem_block==AB_subsystem_pkg::
                           mem_block_tag_e_ext::A_MEM
        && xfer->dst_buff->mem_block==soc_config_pkg::
                           mem_block_tag_e_ext::DDR )
    };
  };
  type_decl<dma_test> dma_test_decl;
};
extend_component<dma_c, dma_c_ext> dma_c_ext_decl;
```

Example 273—C++: Enum type extensions

20.2.6 Ordering of type extensions

Multiple type extensions of the same type can be coded independently, and be integrated and woven into a single stimulus model, without interfering with or affecting the operation of one another. Methodology should encourage making no assumptions on their relative order.

From a semantics point of view, order would be visible in the following cases:

— Invocation order of *exec block*s of the same kind

— Multiple **default** value constraints, **default disable** constraints, and type override declarations occurring in a scope of the same type

— Integer values associated with enum items that do not explicitly have a value assignment

The initial definition always comes first in ordering of members. The order of extensions conforms to the order in which packages are processed by a PSS implementation.

NOTE—This standard does not define specific ways in which a user can control the package processing order.

20.2.7 Template type extensions

Template types, as all other user-defined types, may be extended using the **extend** statement.

Template types may be extended in two ways:

a) Extending the generic template type. The extension will apply to all instances of the template type.

b) Extending the template type instance. The extension will apply to all instances of the template type that are instantiated with the same set of parameter values.

NOTE—Partial template specialization is not supported.

NOTE—Since PSS/C++ does not support template types (see 12.1), type extension of template types is obviously not supported.

20.2.7.1 Examples

Examples of extending the generic template type and the template type instance are shown in Example 274.

```
struct domain_s <int LB = 4, int UB = 7> {
  rand int attr;
  constraint attr >= LB && attr <= UB;
}

struct container_s {
  domain_s<2, 7> domA;          // specialized with LB = 2, UB = 7
  domain_s<2, 8> domB;          // specialized with LB = 2, UB = 8
}

extend struct domain_s {
  rand int attr_all;            // container_s::domA and container_s::domB
                                // will have attr_all
  constraint attr_all > LB && attr_all < UB;
}

extend struct domain_s<2> {     // extend instance specialized with
                                // LB = 2, UB = 7 (default)
  rand int attr_2_7;            // container_t::domA will have attr_2_7
  constraint attr_2_7 > LB && attr_2_7 < UB; // parameters accessible in
                                             // template instance extension
}

struct sub_domain_s<int MIN, int MAX> : domain_s<MIN, MAX> {
  rand int domain_size;
  constraint domain_size == MAX - MIN + 1;

  dynamic constraint half_max_domain {
   attr >= LB && attr <= UB/2;   // Error - LB and UB parameters not accessible
                                // in inherited struct
  }
}
```

Example 274—DSL: Template type extension

In the example above, the generic template type extension is used to add `attr_all` to all instances of `domain_s`. The template type instance extension is used to add `attr_2_7` to the specific `<2,7>` instance of `domain_s`.

20.3 Combining inheritance and extension

It is important to understand that *inheritance* creates a *new* type derived from the base type, while *extension modifies* the definition of an *existing* type. Once a derived type is created by inheriting from a base type, the derived type may be extended just as any other type. In this case, the extensions to the derived type do not affect the base type. However, since a derived type inherits from its base type, any extensions to the base type will also affect the derived type. If multiple types are derived from the same base type, extensions to the base type will affect all derivations thereof.

Extending types in a component scope is only allowed for types that are defined in that scope. It shall be illegal to extend a type defined in a base component type from a derived or unrelated component type.

In Example 275, by extending action `der_a` in component `der_c`, we add a new constraint on the `j` field. This constraint is added to the existing constraints in the initial definition of `der_a`. By extending action `base_a` in the `base_c` extension, we add a new constraint, `i > 2`, which is then inherited by the derived action, `der_a`. The result is that `j` is constrained to be greater than `7`, implying that `i` must be less than `5`, and the additional constraint requires that `i` must also be greater than `2`.

The attempt to extend action `base_a` in component `der_c` is illegal, since `base_a` was originally declared in `base_c`, which is a different type from `der_c`.

```
component base_c {
  action base_a {
    rand int i;
    rand bit[31:0] b;
    constraint { i < 10; }
    constraint c { b > 7; }
  }
}

component der_c : base_c {
  action der_a : base_a {
    rand int j;
    constraint { j > 5 -> i < 5; }
    constraint c { j < 10 -> b < 128; }
  }

  extend action der_a {
    constraint { j > 7; }
  }

  extend action base_a {...} // ILLEGAL
}

extend component base_c {
  extend action base_a {
    constraint { i > 2; }
  }
}
```

Example 275—DSL: Combining inheritance and extension

In Example 277, in the **`pss_top`** root action, the anonymous traversal of `der_c::base_a` will use the `base_a` action as extended in `base_c` in the global scope. Thus, the constraints `i > 2` and `i < 10` will apply. Its execution context will be either instance `c1` or `c2` of `der_c`.

The anonymous traversal of `der_c::der_a` similarly will use the extended definition of `der_a`, but the **`with`** constraint forces the execution context to be instance `c1`. Note that the constraint `c` in `der_c::der_a` masks the original constraint `c` in `base_c::base_a`, so the resolved set of applicable constraints will be:

- `j > 7`
- `i < 5` (due to constraint `j > 5 -> i < 5`)
- `j < 10 -> b < 128`

```
component base_c {
  action base_a {
    rand int i;
    rand bit[31:0] b;
    constraint { i < 10; }
    constraint c { b > 7; }
  }
}

component der_c : base_c {
  action der_a : base_a {
    rand int j;
    constraint { j > 5 -> i < 5; }
    constraint c { j < 10 -> b < 128; }
  }
}

extend component der_c {
  extend action der_a {
    constraint { j > 7; }
  }
}

extend component base_c {
  extend action base_a {
    constraint { i > 2; }
  }
}

component pss_top {
  der_c c1, c2;

  action root {
    activity {
      do der_c::base_a;
      do der_c::der_a with {comp == this.comp.c1; };
    }
  }
}
```

Example 276—DSL: Inheritance and extension of constraints

20.4 Access protection

By default, all data attributes of **component**s, **action**s, and **struct**s have public accessibility. The default accessibility can be modified for a single data attribute by prefixing the attribute declaration with the desired accessibility. The default accessibility can be modified for all attributes going forward by specifying a block-access modifier.

The following also apply:

a) A **public** attribute is accessible from any element in the model.

b) A **private** attribute is accessible only from the element in which the attribute is declared.

c) A **protected** attribute is accessible only from the element in which the attribute is declared, from sub-elements that inherit from it, and from their extensions.

Example 277 shows using a per-attribute access modifier to change the accessibility of the random attribute b. Fields a and c are publicly accessible.

```
struct S1 {
    rand int a;         // public accessibility (default)
    private rand int b; // private accessibility
    rand int c;         // public accessibility (default)
}
```

Example 277—DSL: Per-attribute access modifier

Example 278 shows using block access modifiers to set the accessibility of a group of attributes. Fields w and x are private due to the **private:** directive. Field y is public because its access modifier is explicitly specified. Field z is private, since the **private:** block access modifier is in effect. Field s is public, since the preceding **public:** directive has changed the default accessibility back to public.

```
struct S2 {
    private:
        rand int w;         // private accessibility
        rand int x;         // private accessibility
        public rand int y;  // public accessibility
        rand int z;         // private accessibility

    public:
        rand int s;         // public accessibility
}
```

Example 278—DSL: Block access modifier

20.5 Overriding types

The **override** block (see Syntax 134 and Syntax 135) allows type- and instance-specific replacement of the declared type of a field with some specified subtype.

Overrides apply to action fields, struct attribute fields, and component instance fields. In the presence of **override** blocks in the model, the actual type that is instantiated under a field is determined according to the following rules:

a) Walking from the field up the hierarchy from the contained entity to the containing entity, the applicable **override** directive is the one highest up in the containment tree.
b) Within the same container, **instance** override takes precedence over **type** override.
c) For the same container and kind, an override introduced later in the code takes precedence.

Overrides do not apply to reference fields, namely fields with the modifiers **input**, **output**, **lock**, and **share**. Component-type overrides under actions as well as action-type overrides under components are not applicable to any fields; this shall be an error.

20.5.1 DSL syntax

```
override_declaration ::= override { { override_stmt } }
override_stmt ::=
    type_override
  | instance_override
  | stmt_terminator
type_override ::= type type_identifier with type_identifier ;
instance_override ::= instance hierarchical_id with type_identifier ;
```

Syntax 134—DSL: override declaration

20.5.2 C++ syntax

The corresponding C++ syntax for Syntax 134 is shown in Syntax 135.

pss::override_type

Defined in **`pss/override.h`** (see C.38).

```
template <class Foundation, class Override> class override_type;
```

Type override declaration.

pss::override_instance

Defined in **`pss/override.h`** (see C.38).

```
template <class Override> class override_instance;
```

Instance override declaration.

Syntax 135—C++: override declaration

20.5.3 Examples

Example 279 and Example 280 combine type- and instance-specific overrides with type inheritance. Action `reg2axi_top` specifies that all `axi_write_action` instances shall be instances of `axi_write_action_x`. The specific instance `xlator.axi_action` shall be an instance of `axi_write_action_x2`. Action `reg2axi_top_x` specifies that all instances of `axi_write_action` shall be instances of `axi_write_action_x4`, which supersedes the override in `reg2axi_top`. In addition, action `reg2axi_top_x` specifies that the specific instance `xlator.axi_action` shall be an instance of `axi_write_action_x3`.

```
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 { ... };

action axi_write_action_x2 : axi_write_action_x { ... };

action axi_write_action_x3 : axi_write_action_x { ... };

action axi_write_action_x4 : 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 {
    type axi_write_action with axi_write_action_x4;
    instance xlator.axi_action with axi_write_action_x3;
  }
};
```

Example 279—DSL: Type inheritance and overrides

```
class axi_write_action : public action { ... };
...
class xlator_action : public action { ...
  action_handle<axi_write_action> axi_action {"axi_action"};
  action_handle<axi_write_action> other_axi_action
                                            {"other_axi_action"};

  activity a {
    axi_action,      // overridden by instance
    other_axi_action // overridden by type
  };
};
...
class axi_write_action_x : public axi_write_action { ... };
class axi_write_action_x2 : public axi_write_action_x { ... };
class axi_write_action_x3 : public axi_write_action_x { ... };
class axi_write_action_x4 : public axi_write_action_x { ... };

class reg2axi_top : public action { ...
  override_type<axi_write_action,
    axi_write_action_x> override_type_decl;
  override_instance<axi_write_action_x2>
    _override_inst_1{xlator->axi_action};

  action_handle<xlator_action> xlator {"xlator"};

  activity a {
    repeat { 10,
      xlator // override applies equally to all 10 traversals
    }
  };
};
...

class reg2axi_top_x : public reg2axi_top { ...
  override_type<axi_write_action,
    axi_write_action_x4> override_type_decl;
  override_instance<axi_write_action_x3>
    _override_inst_2{xlator->axi_action};
};
```

Example 280—C++: Type inheritance and overrides

21. Source organization and processing

A PSS model is captured in one or more *source units*. Source units contain declarations of PSS elements. Name resolution rules for types are specified with respect to source units. The bounds of a source unit are specified either by a single file or by a collection of files identified to the PSS processing tool as being part of a single source unit. The files comprising a multi-file source unit could be identified to the PSS processing tool in several different ways. For example, the PSS processing tool could be instructed to consider all PSS source files in a given directory to be a single source unit. The PSS processing tool could be instructed to consider all PSS source files listed in a filelist to be a single source unit. Tool implementations shall support both single-file and multi-file source unit processing modes, but this standard does not dictate the mechanism by which source units shall be specified to the PSS processing tool.

A lexical scope must be fully contained within a single source file, independent of whether source files are processed as single- or multi-file source units.

The processing order of a set of source units is user-specified to the PSS processing tool. This standard does not dictate a specific processing order for files *within* a multi-file source unit, but tools may provide users with means to control it.

21.1 Packages

Packages are a way to group, encapsulate, and identify sets of related definitions, namely type declarations and type extensions. In a verification project, some definitions may be required for the purpose of generating certain tests, while others need to be used for different tests. Moreover, extensions to the same types may be inconsistent with one another, e.g., by introducing contradicting constraints or specifying different mappings to the target platform. By enclosing these definitions in packages, they may coexist and be managed more easily.

Packages also constitute namespaces for the types, functions, and constants declared in their scope. From a namespace point of view, **package**s and **component**s have the same meaning and use (see also 10.4). However, in contrast to **component**s, **package**s cannot be instantiated, and cannot contain attributes, sub-component instances, or concrete **action** definitions.

Type declarations, functions, and constants declared under the scope of a **package** declaration statement are members of that package. Package members may be referenced from outside the package using a qualified reference or made visible by *importing* them into the referencing scope (see 21.1.3).

Definition statements that do not occur inside the lexical scope of a **package** declaration are implicitly associated with the *unnamed global package*. Elements in the unnamed global package are visible to all user-defined namespaces without the need for an **import** statement.

Tools may provide means to control and query which packages are active in the generation of a given test. Tools may also provide ways to locate source files of a given package in the file system. However, these means are not covered herein.

21.1.1 Package declarations

21.1.1.1 DSL syntax

```
package_declaration ::= package package_id_path { { package_body_item } }
package_id_path ::= package_identifier { :: package_identifier }
package_identifier ::= identifier
package_body_item ::=
    abstract_action_declaration
  | struct_declaration
  | enum_declaration
  | covergroup_declaration
  | function_decl
  | import_class_decl
  | procedural_function
  | import_function
  | target_template_function
  | export_action
  | typedef_declaration
  | import_stmt
  | extend_stmt
  | const_field_declaration
  | component_declaration
  | package_declaration
  | compile_assert_stmt
  | package_body_compile_if
  | stmt_terminator
const_field_declaration ::= [ static ] const data_declaration
```

Syntax 136—DSL: package declaration

The following also apply:

a) Multiple **package** statements can apply to the same package name. The **package** contains the members and type extensions declared in all package scopes with the same name.

b) In a *const_field_declaration*, the **static** keyword is optional, but the field is a static constant even if the **static** keyword is not used.

21.1.1.2 C++ Syntax

PSS/C++ does not provide features directly corresponding to the PSS/DSL **package** and **import** constructs. The C++ **namespace** and **using** constructs provide similar functionality.

21.1.1.3 Examples

For an example of package usage, see 22.2.6.

21.1.2 Nested packages

A package may be *nested* inside another package. There are two way to declare a *nested package*.

One way is to include a package declaration inside the outer package declaration, as shown in the following example:

```
package my_lib {
        package impl {
            struct internal_impl_s {}
    }
}
```

Example 281—DSL: Hierarchical declaration of nested package

In the example above, the fully-qualified type name of the **struct** `internal_impl_s` is `my_lib::impl::internal_impl_s`.

Nested packages can also be specified with double-colon-separated package identifier paths. In the example below, the fully-qualified type name of the **struct** `internal_impl_s` is also `my_lib::impl::internal_impl_s`.

```
package my_lib::impl {
            struct internal_impl_s {}
    }
```

Example 282—DSL: Direct declaration of nested package

Declaring a package inside another is equivalent to directly specifying a hierarchical name for a package namespace

The declaration order of package namespaces is not significant. So, for example, it is not necessary to declare an outer namespace prior to declaring an inner namespace. In the example below, two **structs** are declared. `my_lib::impl::internal_impl_s` is declared first, while `my_lib::public_s` is declared second.

```
package my_lib::impl {
   struct internal_impl_s {}
}

package my_lib {
   struct public_s {}
}
```

Example 283—DSL: Declaration of nested package before outer package

21.1.3 Referencing package members

There are three ways to reference package members from outside the scope of their declaring package: *qualified reference*, *explicit import*, and *wildcard import*.

One way to use a declaration from a package is to reference it explicitly using the scope resolution operator **::**. This is called a *qualified reference*. Example:

```
my_lib::public_s my_struct;
```

An alternate method for referencing package declarations is via the **import** statement. Importing an identifier into a **package** or **component** makes that identifier visible within that lexical scope without requiring the scope resolution operator. An **import** statement is a name resolution directive, and does not introduce symbol declarations or symbol aliases into the namespace in which it appears.

Two forms of the **import** statement are provided: *explicit import* and *wildcard import*. An *explicit import* only imports the symbols specifically referenced by the **import**. Example:

```
import my_lib::public_s;
public_s my_struct;
```

It shall be illegal to explicitly import an identifier from a package if the same name is already declared in the importing namespace or to explicitly import the same identifier from two different **package**s.

A *wildcard import* allows all identifiers declared within a package to be imported into a lexical scope, provided the identifier is not otherwise defined anywhere in the importing **component** or **package**. A wildcard import also allows access from the lexical scope to members declared in type extensions found in the imported package. Note that type extensions are unnamed and therefore cannot be explicitly imported.

A wildcard import is of the following form:

```
import my_lib::*;
public_s my_struct;
```

A local declaration of an identifier takes precedence over a wildcard import of the same identifier. An explicit import of an identifier takes precedence over a wildcard import of the same identifier from a different package. If the same name is declared in two wildcard-imported packages, neither is imported, a qualified reference must be used.

import specifications may appear in **package** and **component** declaration statements and in **component** extension statements, but shall come first in those statements. The scope of an **import** statement is limited to the declaration statement or extension statement in which it appears.

Elements in the unnamed global package are visible to all user-defined namespaces without the need for an explicit **import** statement. To explicitly refer to a type declared in the unnamed global package, prefix the type name with "`::`".

import statements are not transitive. If package `B` imports package `A`, package `B` does not have unqualified access to contents declared in packages that `A` may have imported. Package `B` must import those packages directly in order to have unqualified access to contents declared within them.

21.1.3.1 DSL syntax

```
import_stmt ::= import package_import_pattern ;
package_import_pattern ::= type_identifier [ package_import_qualifier ]
package_import_qualifier ::= package_import_wildcard | package_import_alias
package_import_wildcard ::= :: *
package_import_alias ::= as package_identifier
```

Syntax 137—DSL: import statement

Note: *Package aliases* are described in 21.1.4.

Importing content from a **package** namespace using a wildcard only imports content from that exact namespace, and does not import content from *nested* namespaces.

Note that using a wildcard import on an outer package namespace, as shown with `p1::*` in the example below, allows inner package namespaces to be located without specifying the fully-qualified name of the namespace. In this example, **struct** `p1::p2::u` can be referenced as `p2::u` because the elements of `p1` are imported with a wildcard import.

```
package p1 {
    struct s { }
    package p2 {
        struct u { }
    }
}

struct t { }
struct s { }

package top {
    import p1::*;
    struct my_s {
        s     v1; // Resolves to p1::s
        t     v2; // Resolves to ::t
        p2::u v3; // Resolves to p1::p2::u
    }
}
```

Example 284—DSL: Importing the name of a nested package

21.1.4 Package aliases

The use of nested namespaces benefits from the ability to define a named *alias* for a given namespace. This is used when it is necessary to disambiguate between content declared in different namespaces and it is undesirable to use the fully-qualified name of the namespace. The syntax for declaring a package alias is shown in Syntax 137.

A namespace alias is only visible in the lexical scope (e.g., a package declaration statement) in which it appears. It is a name resolution shortcut, and does not introduce a new entity into the scope in which it is specified.

In the example below, this means that p1 and p2 are not visible in the scope of any other declaration statement of consumer_pkg. p1 and p2 may not be referenced from outside the package (e.g., as consumer_pkg::p1). Wildcard-importing consumer_pkg into another package namespace does not make symbols p1 and p2 visible in that namespace.

```
package pkg1::a::b::c {
   struct my_s {}
}

package pkg2::d::e::f {
   struct my_s {}
}

package consumer_pkg {
   import pkg1::a::b::c as p1;
   import pkg2::d::e::f as p2;
   struct s {
      p1::my_s            v1_1; // Refers to pkg1::a::b::c::my_s
      pkg1::a::b::c::my_s v1_2; // v1_1 and v1_2 have the same type
      p2::my_s            v2;   // Refers to pkg2::d::e::f::my_s
   }
}
```

Example 285—DSL: Package alias

21.2 Declaration and reference ordering

Elements may be referenced after their declaration, within the same source unit or in a subsequent source unit. PSS also enables referencing most elements prior to their declaration within the same source unit, but places stronger ordering requirements on some elements. The following apply:

a) A variable declared and referenced within a procedural block or an **activity** block may only be referenced after its declaration.

b) A constant or enum item may be referenced in the initialization assignment expression of another constant only after its declaration.

c) A constant declared within a type may reference type-level and package-level constants in its initialization assignment expression. A package-level constant may only reference other package-level constants in its initialization assignment expression.

21.2.1 Examples

In the example below, file1.pss (the first source unit) declares a **component** named lib_base_c. file2.pss (the second source unit) declares a type my_base_c that inherits from lib_base_c, so file1.pss must be processed before file2.pss. However, within file2.pss, the declaration of my_a_c that refers to my_base_c as a super-type may be placed either before or after the declaration of my_base_c.

```
// Source Unit 1 (file1.pss)
component lib_base_c { /* ... */ }

// Source Unit 2 (file2.pss)
component my_a_c : my_base_c { /* ... */ }

component my_base_c : lib_base_c { /* ... */ }
```

Example 286—DSL: Reference to a previous source unit

In the example below, **action** `pss_top::entry` declares a field named `val` that is referenced in the **constraint** `val_c`. Field `val` may be declared before or after the **constraint** that references it.

```
component pss_top {
    action entry {
        constraint val_c {
            val < 10;
        }

        rand bit[4]    val;
    }
}
```

Example 287—DSL: Reference to a later-declared action field

In the example below, a local variable is declared within an **exec** block. As per requirement a) above, the variable `val` may only be referenced after it is declared.

```
function int get_val();

component pss_top {
    exec init_up {
        int val;
        val = get_val();
    }
}
```

Example 288—DSL: Reference to local variable after declaration

In the example below, constants are declared and referenced in initialization expressions of other constants. As per requirement b) above, a constant must be declared prior to its reference in an initialization expression of a constant or in a type-width expression. Consequently, it is an error to reference the yet undeclared constant `C` in the initialization expression for `A`. It is legal to reference the previously declared constant `A` in the initialization expression for `B`.

```
package my {
  const int A = C /* Error: C is not yet declared */;
  const int B = A + 2;
  const int C = 3;
}
```

Example 289—DSL: Initialization of constants

21.3 Name resolution

For the purpose of the following description, the term *namespace* refers to either a **package** or a type (e.g., **component**, **struct**) under which static members (types, static constants, static functions, and enum items) may be declared.

The members of a package namespace include the members declared in the union of all the package definition statements of that package (see 21.1.1.1). The visible members of a type namespace include the members declared in the union of the type's initial definition and all visible extensions of the type (see 20.2),

Members of PSS namespaces shall have unique names in the context of their namespace, but members may have the same name if declared under different namespaces.

Types can be referenced in different contexts, such as declaring a variable, extending a type, or inheriting from a type. In all cases, a qualified name of the type can be used, using the scope operator **::**.

Constants, static functions, and enum items can be referenced in expression contexts. In these cases too, a qualified name can be used, using the scope operator.

Informally, unqualified entity names can be used in the following cases:

— when referencing an entity that was declared in the same namespace or in an enclosing namespace.

— when referencing an entity that was declared in a **package** imported into a logical scope enclosing the reference.

Precedence is given to the current namespace scope; explicit qualification can be used to override the precedence.

Formally, unqualified names are resolved using the following process, starting with step a, continuing with step b, and then step c, in the absence of resolution in previous steps:

a) If the reference occurs within an expression whose *expected type* is an enumeration type (see 9.4.3 for definition of expected type):
    1) Search enum items declared in the expected type's initial definition.
    2) Search enum items declared in the expected type's extensions that are defined under the current package or one of its containing packages (see 20.2), or in the expected type's extensions that are within a package wildcard-imported into a lexical scope enclosing the reference.

b) If the reference occurs within the definition of a type:
    1) Search members of the type declared in its initial definition.
    2) Search members of the type declared in its extensions that are defined under the current package or one of its containing packages (see 20.2), or in its extensions that are within a package wildcard-imported into a lexical scope enclosing the reference.
    3) If the type inherits from a super-type, search members declared in the super-type using the process described in steps 1 and 2. Repeat for all super-types in the inheritance hierarchy.
    4) If the scope is a component initial definition or extension:
        i) Search package members explicitly imported into the lexical scope of the initial definition or extension, respectively.
        ii) Search members of packages wildcard-imported into the lexical scope of the initial definition or extension, respectively.
    5) If the type is an inner type (e.g., an **action** declared inside a **component**), search members declared in the outer type using the process described in steps 1 through 4 above.

c) Search **package** namespaces, starting with the package namespace of the immediate lexical scope and working outward along the package hierarchy. At each level, do the following:
   1) Search package members declared under all *package_declaration*s of the same package.
   2) If the reference is enclosed in a lexical package scope corresponding to the namespace being searched:
      i) Search package members explicitly imported into the lexical scope.of the corresponding *package_declaration* statement.
      ii) Search members of packages wildcard-imported into the lexical scope.of the corresponding *package_declaration* statement.

A qualified name is composed of double-colon-separated elements. Qualified name elements are resolved by first applying the same process for unqualified names described above on the first element of the static path. Having resolved the first element to a certain **package**/type, the rest of the static path is used to access down from it.

21.3.1 Name resolution examples

In Example 290, `s` is declared in three places: imported package `P1`, encapsulating package `P2`, and nested component `C1`. The `s` referenced in nested component `C1` is resolved to the `s` locally defined in nested component `C1`. Using qualifiers, `P1::s` would be used to resolve to `s` in imported package `P1`, and `P2::s` would be used to resolve to `s` in encapsulating package `P2`.

```
package P1 {
  struct s {};
};

package P2 {
  struct s {};

  component C1 {
    import P1::*;
    struct s {};
    s f;
  };
};
```

Example 290—DSL: Name resolution to declaration in nested namespace

In Example 291, `s` is declared in two places: imported package `P1` and encapsulating package `P2`. The `s` referenced in nested component `C1` is resolved to the `s` defined in imported package `P1`. Using qualifiers, `P2::s` would be used to resolve to `s` in encapsulating package `P2`.

```
package P1 {
  struct s {};
};

package P2 {
  struct s {};

  component C1 {
    import P1::*;
    s f;
  };
};
```

Example 291—DSL: Name resolution to declaration in imported package in nested namespace

In Example 292, `s` is declared in two places: imported package `P1` and encapsulating package `P2`. The `s` referenced in nested component `C1` is resolved to the `s` defined in encapsulating package `P2`. Using qualifiers, `P1::s` would be used to resolve to `s` in package `P1` imported in encapsulating package `P2`.

```
package P1 {
  struct s {};
};

package P2 {
  import P1::*;
  struct s {};

  component C1 {
    s f;
  };
};
```

Example 292—DSL: Name resolution to declaration in encapsulating package

In Example 293, `s` is declared in one place: imported package `P1`. The `s` referenced in nested component `C1` is resolved to the `s` defined in package `P1` imported inside encapsulating package `P2`.

```
package P1 {
  struct s {};
};

package P2 {
  import P1::*;

  component C1 {
    s f;
  };
}
```

Example 293—DSL: Name resolution to declaration in imported package in encapsulating package

Example 294 shows a case where importing the encapsulating package has no effect on the resolution rules. `s` will resolve to the same `s` in `P2`.

```
package P1 {
  struct s {};
};

package P2 {
  import P1::*;
  struct s {};

  component C1 {
    import P2::*;
    s f;
  };
}
```

Example 294—DSL: Package import has no effect on name resolution

In Example 295 below, `a_pkg` declares a **struct** `S1`, `b_pkg` imports content from `a_pkg`, and `b_pkg` declares a **struct** `S2` that inherits from `S1`. `pss_top` imports content from `b_pkg`.

— Line (1): `S2` is resolved via the import of `b_pkg`.

— Line (2): Imports are not transitive. Therefore, the import of `b_pkg` does not make content from `a_pkg` visible in **component** `pss_top`.

— Line (3): `S1` can be referenced with a fully-qualified type name, `a_pkg::S1`.

— Line (4): Importing a package does not introduce symbols into the importing namespace.

```
package a_pkg {
  struct S1 { }
}

package b_pkg {
  import a_pkg::*;
  struct S2 : S1 { }
}

component pss_top {
  import b_pkg::*;
  S2         s2_i0; // (1) OK
  S1         s1_i1; // (2) Error: S1 is not made visible
                    //            by importing b_pkg
  a_pkg::S1  s1_i2; // (3) OK: S1 is declared in a_pkg
  b_pkg::S1  s1_i3; // (4) Error: import of a_pkg in b_pkg
                    //            does not make S1 a b_pkg member
};
```

Example 295—DSL: Package import is not a declaration

Example 296 demonstrates the use of qualified and unqualified enum item references. The unqualified references are resolved based on the expected type in context, namely the type of the expression on the other side of the equality operator and on the left-hand side of the **in** operator.

```
component my_ip_c {
  enum mode_e {A, B, C, D};
  action my_op {
    rand mode_e mode;
  }
}

component pss_top {
  my_ip_c my_ip;
  action test {
    my_ip_c::my_op op;
    constraint op.mode == my_ip_c::mode_e::A;
    constraint op.mode == A;
    constraint op.mode in [A, C, D];

    activity {
      op;
    }
  }
}
```

Example 296—DSL: Resolution of enum item references

22. Test realization

A PSS model interacts with foreign languages in order to drive, or bring about, the behaviors that leaf-level actions represent in a test scenario. This is done by calling application programming interfaces (APIs) available in the execution environment, or generating foreign language code that executes as part of the test. In addition, external code, such as reference models and checkers, may be used to help compute stimulus values or expected results during stimulus generation.

The platform on which test generation takes place is generally referred to as the *solve platform*, while the platform on which test execution takes place is called the *target platform*.

Logic used to help compute stimulus values is coded using *procedural constructs* (see 22.7), possibly invoking a foreign procedural interface on the solve platform (see 22.4). The implementation of runtime behavior of leaf-level actions can similarly be specified with procedural constructs, possibly invoking a foreign procedural interface on the target platform or invoking *target template functions* (see 22.6). Alternatively, implementation of actions and other scenario entities can be specified as *target code template blocks* (see 22.5). In all cases, the constructs for specifying implementation of PSS entities are called *exec block*s.

Functions can be defined in PSS as a means to factor out and reuse portable procedural logic required for the implementation of scenario entities in *exec block*s (see 22.3). Functions may take parameters and optionally return a result value. Like *exec block*s, functions are defined in terms of procedural constructs or as target code templates.

22.1 exec blocks

*exec block*s provide a mechanism for associating specific functionality with a **component**, an **action**, a flow/resource object, or a **struct** (see Syntax 138 and Syntax 139). A number of *exec block* kinds are used to implement scenario entities.

— **init_down** and **init_up** *exec block*s allow component data fields to be assigned a value as the component tree is being elaborated (see 10.5).

— **body** *exec block*s specify the actual runtime implementation of atomic actions.

— **pre_solve** and **post_solve** *exec block*s of **action**s, flow/resource objects, and **struct**s are a way to involve arbitrary computation as part of the scenario solving.

— Other **exec** kinds serve more specific purposes in the context of pre-generated test code and auxiliary files.

22.1.1 DSL syntax

```
exec_block_stmt ::=
    exec_block
  | target_code_exec_block
  | target_file_exec_block
  | stmt_terminator
exec_block ::= exec exec_kind { { exec_stmt } }
exec_kind ::=
    pre_solve
  | post_solve
  | body
  | header
  | declaration
  | run_start
  | run_end
  | init_down
  | init_up
  | init
exec_stmt ::=
    procedural_stmt
  | exec_super_stmt
exec_super_stmt ::= super ;
target_code_exec_block ::= exec exec_kind language_identifier = string_literal ;
target_file_exec_block ::= exec file filename_string = string_literal ;
```

Syntax 138—DSL: exec block declaration

The following also apply:

a) *exec block* content is given in one of two forms: as a sequence of procedural constructs (possibly involving foreign function calls) or as a text segment of target code parameterized with PSS attributes.

b) In either case, a single *exec block* is always mapped to implementation in no more than one foreign language.

c) In the case of a target-template block, the target language shall be explicitly declared; however, when using procedural constructs, the corresponding language may vary.

d) **“exec init**” is an alias for “**exec init_up**,” and is considered deprecated as of PSS 2.0. The keyword “**init**” may be removed in a future version of this standard. Users should use “**init_up**” instead.

22.1.2 C++ syntax

The corresponding C++ syntax for Syntax 138 is shown in Syntax 139.

pss::exec

Defined in **`pss/exec.h`** (see C.25).

```
class exec;
/// Kinds of exec blocks
enum ExecKind {
  run_start,
  header,
  declaration,
  init_down,
  init_up,
  init,
  pre_solve,
  post_solve,
  body,
  run_end,
  file
};
```

Declare an exec block.

Member functions

```
exec(ExecKind kind, std::initializer_list<detail::AttrCommon>&& write_vars)
```
: declare inline exec
```
exec(ExecKind kind, const char* language_or_file,
    const char* target_template)
```
: declare target template exec
```
exec(ExecKind kind, std::string&& language_or_file,
    std::string&& target_template)
```
: declare target template exec
```
exec(ExecKind kind, const detail::ExecStmt& r)
```
: declare native exec - with single exec statement
```
exec(ExecKind kind, const detail::AlgebExpr& r)
```
: declare native exec - with single AlgebExpr statement
```
exec(ExecKind kind, const detail::Stmt& /* sequence& */ r)
```
: declare native exec - with sequence statement
```
exec(ExecKind kind, std::function<void()> genfunc)
```
: declare generative procedural exec
```
exec(ExecKind kind, std::string&& language_or_file,
    std::function<void(std::ostream&)> genfunc)
```
: declare generative target-template exec

Syntax 139—C++: exec block declaration

*exec block*s can be specified in the following ways in C++:

- A native procedural *exec block* (similar to DSL) given as a single procedural statement, typically a **sequence** statement enclosing a sequential block
- A target-template block (similar to DSL) for target execs (described in 22.5)
- An inline exec for solve execs (described in 22.8) – available only when using PSS/C++
- A generative exec with full support for procedural constructs (described in 22.9 and 22.7) – available only when using PSS/C++

A native exec is perhaps the simplest form for an *exec block* that is not specified directly in a target language. This form can be used when the *exec block* contents have only variable assignments or function calls. For example:

```
exec e {exec::init_up, a = 1};
```

If multiple statements need to be evaluated, these should be enclosed in a **sequence** construct like this:

```
exec e {exec::init_up, sequence {a = 1, b = 2, func()}};
```

22.1.3 exec block kinds

The following list describes the different *exec block* kinds:

— **pre_solve**—valid in **action**, flow/resource object, and **struct** types. The **pre_solve** block is processed prior to solving of random-variable relationships in the PSS model. **pre_solve** *exec block*s are used to initialize non-random variables that the solve process uses. See also 17.4.10.

— **post_solve**—valid in **action**, flow/resource object, and **struct** types. The **post_solve** block is processed after random-variable relationships have been solved. The **post_solve** *exec block* is used to compute values of non-random fields based on the solved values of random fields. See also 17.4.10.

— **body**—valid in **action** types. The **body** block constitutes the implementation of an atomic **action**. The **body** block of each **action** is invoked in its respective order during the execution of a scenario—after the **body** blocks of all predecessor **actions** complete. Execution of an **action**'s **body** may be logically time-consuming and concurrent with that of other actions. In particular, the invocation of **exec** blocks of **action**s with the same set of scheduling dependencies logically takes place at the same time. Implementation of the standard should guarantee that executions of **exec** blocks of same-time **action**s take place as close as possible.

— **run_start**—valid in **action**, flow/resource object, and **struct** types. The **run_start** block is a procedural non-time-consuming code block to be executed before any **body** block of the scenario is invoked. It is used typically for one-time test bring-up and configuration required by the context action or object. **exec run_start** is restricted to pre-generation flow (see Table 24).

— **run_end**—valid in **action**, flow/resource object, and **struct** types. The **run_end** block is a procedural non-time-consuming code block to be executed after all **body** blocks of the scenario are completed. It is used typically for test bring-down and post-run checks associated with the context action or object. **exec run_end** is restricted to pre-generation flow (see Table 24).

— **init_down/init_up**(**init**)—valid in **component** types. The **init_down** and **init_up** blocks are used to assign values to component attributes and to initialize foreign language objects. Component **init_down** and **init_up** blocks are called before the scenario root action's **pre_solve** block is invoked. **init_down** and **init_up** blocks may not call target template functions.

  1) **init_down**—Starting with the root component, **init_down** blocks are evaluated top-down for each component in the hierarchy. The relative order of evaluating **init_down** blocks for components at the same level of hierarchy is undefined. For any component, the **init_down** block shall be evaluated before its **init_up** block is evaluated.

  2) **init_up**—For a leaf-level component (i.e., one that does not instantiate any subcomponents), the **init_up** block shall be evaluated after its **init_down** block (if any). A parent component's **init_up** block shall be evaluated only after all subcomponent **init_up** blocks have been evaluated.

— **header**—valid in **action**, flow/resource object, and **struct** types. The **header** block specifies top-level statements for header declarations presupposed by subsequent code blocks of the context action or object. Examples are `'#include'` directives in C, or forward function or class declarations.

— **declaration**—valid in **action**, flow/resource object, and **struct** types. The **declaration** block specifies declarative statements used to define entities that are used by subsequent code blocks. Examples are the definition of global variables or functions.

exec header and **declaration** blocks shall only be specified in terms of target code templates. All other **exec** kinds may be specified in terms of procedural constructs or target code templates.

22.1.4 Examples

In Example 297 and Example 298, the **init_up** *exec block*s are evaluated in the following order:

a) `init_up in pss_top.s1`
b) `init_up in pss_top.s2`
c) `init_up in pss_top`

This results in the **component** fields having the following values:

a) `s1.base_addr=0x2000 (init_up in pss_top overwrote the value set by init_up in sub_c)`
b) `s2.base_addr=0x1000 (value set by init_up in sub_c)`

```
component sub_c {
  int base_addr;

  exec init_up {
    base_addr = 0x1000;
  }
};

component pss_top {
  sub_c s1, s2;

  exec init_up {
    s1.base_addr = 0x2000;
  }
};
```

Example 297—DSL: Data initialization in a component

```
class sub_c : public component { ...
  attr<int> base_addr {"base_addr"};
  exec e {exec::init_up,
    base_addr = 0x1000
  };
};
...

class pss_top : public component { ...
  comp_inst<sub_c> s1{"s1"}, s2{"s2"};
  exec e {exec::init_up,
    s1->base_addr = 0x2000
  };
};
...
```

Example 298—C++: Data initialization in a component

In Example 299, the **init_down** and **init_up** blocks will be evaluated in the following order:

— `init_down in T`
— `init_down in T.c1`
— `init_down in T.c2`

— `init_up in T.c1`
— `init_up in T.c2`
— `init_up in T`

```
component C {
    exec init_down {
    }
    exec init_up {
    }
}

component T {
    C  c1, c2;
    exec init_down {
    }
    exec init_up {
    }
}
```

Example 299—DSL: init_down and init_up exec blocks

A diagram of the example is shown below:

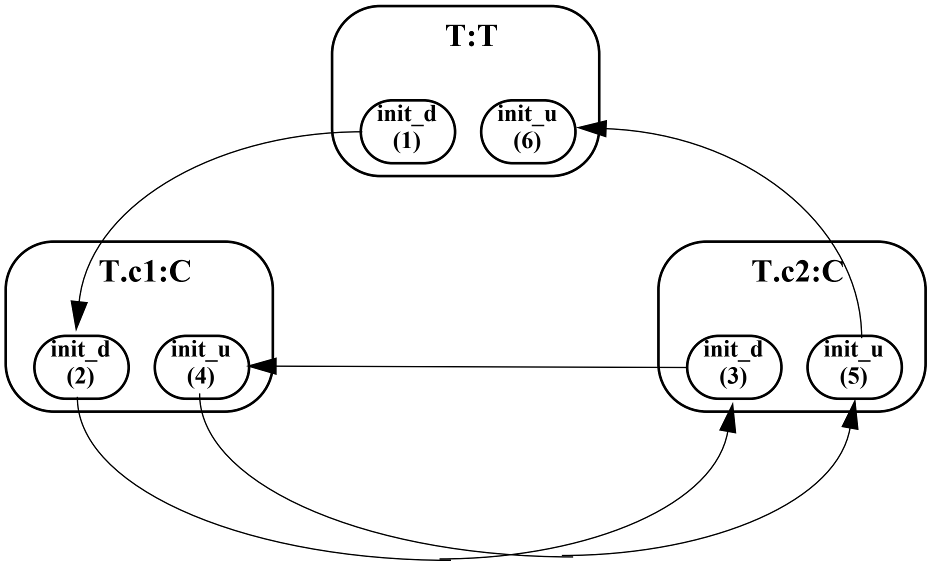

Figure 18—Order of invocation of init_down and init_up exec blocks

The order of initialization calls is annotated on each of the **init_d(own)** and **init_u(p)** blocks. Note that **init_down** in `T` is called first, followed by **init_down** in `T.c1`, etc.

Note that a tool is free to execute the **exec init_down** and **init_up** blocks of sibling instances in arbitrary order. For example, while the diagram above shows **init_down** in `T.c1` executing before **init_down** in `T.c2`, the opposite order is also correct. The key requirements are that the **exec init_down** block of a parent component instance (e.g., `T`) execute before the **exec init_down** block of any child **component** instances, and that the **exec init_up** block of a parent component instance (e.g., `T`) execute after all **exec init_up** blocks of child component instances have executed. This implies that the following ordering of execution is also legal:

— `init_down in T`
— `init_down in T.c1`
— `init_up in T.c1`
— `init_down in T.c2`
— `init_up in T.c2`
— `init_up in T`

In Example 300 and Example 301, component `pss_top` contains two instances of component `sub_c`, named `s1` and `s2`. Component `sub_c` contains a data field named `base_addr` that controls the value to function `activate()` when action `A` is traversed.

During construction of the component tree, component `pss_top` sets `s1.base_addr=0x1000` and `s2.base_addr=0x2000`.

Action `pss_top::entry` traverses action `sub_c::A` twice. Depending on which component instance `sub_c::A` is associated with during traversal, it will cause `sub_c::A` to be associated with a different `base_addr`.

— If `sub_c::A` executes in the context of `pss_top.s1`, `sub_c::A` uses `0x1000`.
— If `sub_c::A` executes in the context of `pss_top.s2`, `sub_c::A` uses `0x2000`.

```
component sub_c {
    bit[32] base_addr = 0x1000;
    action A {
        exec body {
            // reference base_addr in context component
            activate(comp.base_addr + 0x10);
                              // activate() is an imported function
        }
    }
}

component pss_top {
    sub_c s1, s2;
    exec init_up {
        s1.base_addr = 0x1000;
        s2.base_addr = 0x2000;
    }
    action entry {
        sub_c::A a;
        activity {
            repeat (2) {
                a; // Runs sub_c::A with 0x1000 as base_addr when
                    // associated with s1
                    // Runs sub_c::A with 0x2000 as base_addr when
                    // associated with s2
            }
        }
    }
}
```

Example 300—DSL: Accessing component data field from an action

```
class sub_c : public component { ...
  attr<bit> base_addr {"base_addr", width (32), 0x1000};

  class A : public action { ...
    exec e {exec::body,
      activate(comp<sub_c>()->base_addr + 0x10)
    };
  };
  type_decl<A> A_decl;
};
...

class pss_top : public component { ...
  comp_inst<sub_c> s1{"s1"}, s2{"s2"};

  exec e {exec::init_up,
    sequence {
       s1->base_addr = 0x1000,
       s2->base_addr = 0x2000
     }
  };

  class entry : public action { ...
    action_handle<sub_c::A> a {"a"};

    activity g {
      repeat {2,
        a // Runs sub_c::A with 0x1000 as base_addr when associated with s1
          // Runs sub_c::A with 0x2000 as base_addr when associated with s2
      }
    };
  };
  type_decl<entry> entry_decl;
};
...
```

Example 301—C++: Accessing component data field from an action

For additional examples of *exec block* usage, see 22.2.6.

22.1.5 exec block evaluation with inheritance and extension

Both inheritance and type extension can impact the behavior of *exec blocks*. See also 20.1 and 20.2.

22.1.5.1 Inheritance and shadowing

exec blocks are considered to be *virtual*, in that a derived type that defines an *exec block* masks the behavior of a same-kind *exec block* (e.g., **body**) specified by its base type. Procedural *exec blocks* may include the "**`super;`**" statement, which will execute the contents of the corresponding base-type *exec block*(s) at that point (see 22.1.5.2).

In the following examples, `print`() is a target function that prints out a formatted line. In Example 302, action `B` inherits from action `A` and shadows the **pre_solve** and **body** *exec blocks* defined by action `A`.

```
action A {
  int a;

  exec pre_solve {
    a=1;
  }
  exec body {
    print("Hello from A %d", a);
  }
}

action B : A {
  exec pre_solve {
    a=2;
  }
  exec body {
    print("Hello from B %d", a);
  }
}
```

Example 302—DSL: Inheritance and shadowing

When an instance of action `B` is evaluated, the following is printed:

```
Hello from B 2
```

22.1.5.2 Using super

Specifying "**`super;`**" as a statement in a subtype executes the behavior of the same-kind procedural *exec block*(s) from the base type, allowing a type to prepend or append behavior. The "**`super;`**" statement shall not be allowed in a target-template *exec* block.

In Example 303, both `A1` and `A2` inherit from action `A`. Both execute the **pre_solve** *exec block* inherited from `A`. `A1` invokes the **body** behavior of `A`, then displays an additional statement. `A2` displays an additional statement, then invokes the **body** behavior of `A`.

```
action A {
  int a;

  exec pre_solve {
    a=1;
  }
  exec body {
    print("Hello from A %d", a);
  }
}

action A1 : A {
  exec body {
    super;
    print("Hello from A1 %d", a);
  }
}
action A2 : A {
  exec body {
    print("Hello from A2 %d", a);
    super;
  }
}
```

Example 303—DSL: Using super

When an instance of `A1` is evaluated, the following is printed:

```
Hello from A 1
Hello from A1 1
```

When an instance of `A2` is evaluated, the following is printed:

```
Hello from A2 1
Hello from A 1
```

NOTE—PSS/C++ does not define a mechanism to invoke procedural blocks in a super-type—in other words, an equivalent to *exec_super_stmt*.

22.1.5.3 Type extension

Type extension enables additional features to be contributed to **action**, **component**, and **struct** types. Type extension is additive and all *exec block*s contributed via type extension are evaluated, along with *exec block*s specified within the initial definition. First, the initial definition's *exec block*s (if any) are evaluated. Next, the *exec block*s (if any) contributed via type extension are evaluated, in the order that they are processed by the PSS processing tool.

In Example 304, a type extension contributes an *exec block* to action `A1`.

```
action A {
  int a;

  exec pre_solve {
    a=1;
  }
  exec body {
    print("Hello from A %d", a);
  }
}

action A1 : A {
  exec body {
    super;
    print("Hello from A1 %d", a);
  }
}

extend action A1 {
  exec body {
   print("Hello from A1 extension %d", a);
  }
}
```

Example 304—DSL: Type extension contributes an exec block

When an instance of `A1` is evaluated, the following is printed:

```
Hello from A 1
Hello from A1 1
Hello from A1 extension 1
```

In Example 305, two *exec block*s are added to action `A1` via extension.

```
action A {
  int a;

  exec pre_solve {
    a=1;
  }
  exec body {
    print("Hello from A %d", a);
  }
}

action A1 : A {
  exec body {
    super;
    print("Hello from A1 %d", a);
  }
}

extend action A1 {
  exec body {
   print("Hello from A1(1) extension %d", a);
  }
}

extend action A1 {
  exec body {
   print("Hello from A1(2) extension %d", a);
  }
}
```

Example 305—DSL: exec blocks added via extension

If the PSS processing tool processes the first extension followed by the second extension, then the following is produced:

```
Hello from A 1
Hello from A1 1
Hello from A1(1) extension 1
Hello from A1(2) extension 1
```

If the PSS processing tool processes the second extension followed by the first extension, then the following is produced:

```
Hello from A 1
Hello from A1 1
Hello from A1(2) extension 1
Hello from A1(1) extension 1
```

22.2 Functions

Functions are a means to encapsulate behaviors used by **action**s and other entities to implement test scenarios. Functions are called in procedural description contexts, and are akin to procedures in conventional programming languages.

Functions can be declared in global or **package** scopes. Functions can also be declared in **component** scopes, in which case each call is associated with a specific instance of that **component** type.

A function may be defined in one of three ways:

— Using native PSS procedural statements, possibly calling other functions (see 22.3)

— As bound to a procedural interface in a foreign programming language, such as a function in C/C++, or a function/task in SystemVerilog (see 22.4)

— As a target code template block (see 22.6)

The definition of a functions in one of these three ways may be coupled with the function's initial declaration. The definition may also be provided separately, in a different lexical scope. The intent and semantics of a function are fixed by its declaration, but its implementation could vary between different environments and contexts.

Functions may be called from procedural *exec* blocks, namely **exec init_down**, **init_up**, **pre_solve**, **post_solve**, **body**, **run_start**, and **run_end**. Functions called from **exec init_down**, **init_up**, **pre_solve**, and **post_solve** are evaluated on the *solve platform*, whereas functions called from **exec body**, **run_start** and **run_end** are evaluated on the *target platform*.

A function declared in a component scope may be shadowed by a function declaration with the same name in a derived component. The function declaration in the derived component must have the same return type and arguments as that in the base component. The function in the base type may be called from within the function in the derived type by calling "`super.<function name>(...)`".

When the shadowed element is a function, the function call is *polymorphic*, that is, the actual function called depends on its context component. See 20.1 for details.

22.2.1 Function declarations

A function prototype is declared in a **package** or **component** scope within a PSS description. The function prototype specifies the function name, return type, and function parameters. See Syntax 140 and Syntax 141. Note that the syntax shown here is for the declaration of a function prototype only, where the definition is provided separately. A function can also be declared and defined at once using a procedural statement block or a target code template (see 22.3 and 22.6, respectively). The same syntax is used for specifying the prototype in these cases also.

22.2.1.1 DSL syntax

```
function_decl ::= [ pure ] function function_prototype ;
function_prototype ::= function_return_type function_identifier function_parameter_list_prototype
function_return_type ::=
    void
  | data_type
function_parameter_list_prototype ::=
    ( [ function_parameter { , function_parameter } ] )
  | ( { function_parameter , } varargs_parameter )
function_parameter ::=
    [ function_parameter_dir ] data_type identifier [ = constant_expression ]
  | ( type | ref type_category | struct ) identifier
function_parameter_dir ::=
    input
  | output
  | inout
varargs_parameter ::= ( data_type | type | ref type_category | struct ) ... identifier
type_category ::=
   action
  | component
  | struct_kind
```

Syntax 140—DSL: Function declaration

The following also apply:

a) Functions declared in global or package scopes are considered static, and are called optionally using package qualification with the scope operator (::).

b) Functions declared in **component** scopes are considered instance (non-static) functions, and are called optionally using the dot operator (.) on a component instance expression.

22.2.1.2 C++ syntax

The corresponding C++ syntax for Syntax 140 is shown in Syntax 141.

pss::function

Defined in **`pss/function.h`** (see C.30).

```
template <class T> class arg;
template <class T> class in_arg;
template <class T> class out_arg;
template <class T> class inout_arg;
template <class T> class result;
enum kind {solve, target};
template <typename T> class function;
template <typename R, typename... Args> class function<R(Args...)>; // 1
template <typename... Args>  class function<result<void>(Args...)>; // 2
```

1) Declare a function object with result R
2) Declare a function object with no result (void)

Member functions

`function (const scope &name, R result, Args... args)` : constructor with result
`function (const scope &name, Args... args)` : constructor with void result
`function (const scope &name, bool is_pure, R result, Args... args)` : constructor with pure modifier and result
`function (const scope &name, bool is_pure, Args... args)` : constructor with pure modifier and void result
`operator()(const T&... /*detail::AlgebExpr*/ params)` : operator

Syntax 141—C++: Function declaration

NOTE—PSS/C++ functions declared outside a component are considered to be in the global PSS namespace even if they are declared within a C++ **`namespace`**.

22.2.1.3 Examples

For an example of declaring a function, see 22.2.2, below.

22.2.2 Parameters and return types

A function shall explicitly specify a data type as its return type or use the keyword **void** to indicate that the function does not return a value. Function return values shall be either plain-data types (scalars and aggregates thereof) or reference types. Functions shall not return **action** types, **component** types, or flow/resource object types without the **ref** modifier.

A function may specify any number of formal parameters, stating their types and names. Function parameters shall be either plain-data types or reference types. Functions shall not have parameters of **action** types, **component** types, or flow/resource object types without the **ref** modifier. Functions may also declare generic parameters without stating their specific type, and may declare a variable number of parameters—see 22.2.4. Note that the set of types allowed for imported foreign functions is restricted (see 22.4).

Parameter direction modifiers (**input**, **output**, or **inout** in DSL, and template classes **`in_arg`**, **`out_arg`**, or **`inout_arg`** in C++) are optional in the function declaration. However, if they are specified in the function declaration, such a function may only be imported (see 22.4). In the declaration of native functions and target-template functions, direction modifiers shall not be used. In C++, parameter directions shall be left unspecified by using the template class **`arg`**.

NOTE—PSS/C++ function parameters and return types support **`int`** and **`bit`** types with width and range information, but do not support associating range information with **`string`** or **`enum`** types. PSS/C++ function parameters of class **`arg`** support array types with width or range specifications, while the **`in_arg`**/**`out_arg`**/**`inout_arg`**/**`result`** classes do not.

Example 306 and Example 307 declare a function in a **package** scope. In this case, the function `compute_value` returns an **int**, accepts an input value (`val`), and returns an output value via the `out_val` parameter.

```
package generic_functions {
  function int compute_value(
   int        val,
   output int out_val);
}
```

Example 306—DSL: Function declaration

```
namespace generic_functions {
  function<result<int>(in_arg<int>, out_arg<int>)> compute_value {
    "compute_value", result<int>(), in_arg <int>("val"),
                                    out_arg<int>("out_val")
  };
};
```

Example 307—C++: Function declaration

22.2.3 Default parameter values

Default parameter values serve as the actual values for the respective parameters if explicit actual parameters are missing in the function call.

The following also apply:

a) A default parameter value shall be specified as a constant expression, and therefore can only be specified for a parameter of a plain-data type.

b) In a function declaration, following a parameter with a specified default value, all subsequent parameters must also have default values specified.

c) A default parameter value is in effect for redeclarations (and overrides) of a function. A default parameter value shall not be specified in the redeclaration of a function if already declared for the same parameter in a previous declaration, even if the value is the same.

Example 308 demonstrates the declaration and use of a default parameter value.

```
function void foo(int x, int y = 100);
function void bar() {
   foo(3,200); // the value 200 is used for parameter y
   foo(3);     // the value 100 is used for parameter y
}
```

Example 308—DSL: Default parameter value

NOTE—PSS/C++ function parameter default values of type **`int`** and **`bit`** may be specified using the **`in_arg`** class. Parameters of other types, and parameters specified using **`arg`** and **`inout_arg`** may not be given default values.

22.2.4 Generic and varargs parameters

Generic parameters and *varargs parameters* are means to declare functions that are generic or variadic with respect to their parameters. Examples are functions that apply to all actions or objects as such, and functions that involve string formatting.

Generic and varargs parameters are used for the declaration of functions whose definition is built into implementations. In particular, they are used to declare functions included in the PSS core library (see Clause 24). PSS does not provide a native mechanism to operate on an unspecified number of parameters or on parameters with no declared type, nor does PSS define mapping of functions with generic/varargs parameters to foreign languages.

The following also apply:

a) A generic parameter is declared either with the keyword **type** or with a *type category*, rather than with a specific type. A value of any type (if **type** was specified), or any type that belongs to the specified category (if a type category was specified), is accepted in the function call. In the case of the **struct** category, the **ref** modifier shall not be used, but for the other categories (**component**, **action**, one of the object kinds), the **ref** modifier shall be used. See more on the use of type categories in 12.3.2.

b) Default values may not be specified for generic parameters.

c) The varargs parameter (ellipsis notation – "`...`") signifies that zero or more trailing values may be passed as actual parameters in the function call. Note that a varargs parameter may only occur as the last parameter in the parameter list.

d) In a function call, the expressions corresponding to a varargs parameter must all be of the declared type if a type is specified, or belong to the same type category if one is specified. Note that in the case of a type category, the types of the actual parameter expressions may vary, so long as they all belong to the specified category. When a varags parameter is declared with the keyword **type**, actual parameters types may vary with no restriction.

Example 309 demonstrates the declaration and use of a generic parameter.

```
function void foo(struct x);
struct my_struct {};
struct your_struct {};
function void bar() {
   my_struct s1;
   your_struct s2;
   foo(s1);
   foo(s2);
}
```

Example 309—DSL: Generic parameter

Example 310 demonstrates the declaration and use of a varargs parameter.

```
function string format_string(string format, type ... args);
function void bar() {
   string name = "John";
   int age = 55;
   string result;
   result = format_string("name %s: age %d", name, age);
}
```

Example 310—DSL: Varargs parameter

22.2.5 Pure functions

Pure functions are functions for which the return value depends only on the values of their parameters, and their evaluation has no side-effects. Declaring a function as **pure** may provide the PSS implementation with opportunities for optimization. Note that a function declared as **pure** may lead to unexpected behavior if it fails to obey these rules.

The following rules apply to **pure** functions, that is, functions declared with the **pure** modifier:

a) Only non-void functions with no **output** or **inout** parameters may be declared **pure**.
b) A **pure** function will be considered **pure** in derived types even if the **pure** modifier is not explicitly specified in the derived type function declaration.

A non-**pure** function shall not be declared as **pure** in derived types.

22.2.5.1 Examples

Example 311 and Example 312 demonstrate declaration and use of **pure** functions.

```
pure function int factorial(int n);
action A {
   rand int vals[10];
   int factorial_vals[10];

   exec post_solve {
      foreach (vals[i]) {
         factorial_vals[i] = factorial(vals[i]);
      }
   }
}
```

Example 311—DSL: Pure function

```
function<result<int>(arg<int>)>
   factorial {"factorial", true, result<int>(), arg<int>("n")};

class A : public action {
   rand_attr_vec<int> vals{"vals", 10};
   attr_vec<int> factorial_vals{"factorial_vals", 10};

   exec e {exec::post_solve, [&]() {
      attr<int> i{"i"};
      foreach (i, vals,
         [&]() { factorial_vals[i] = factorial(vals[i]); }
      );
   }
   };
};
```

Example 312—C++: Pure function

In the example above, the function `factorial()` is pure and therefore will not necessarily be re-evaluated for each element in the array. If some elements in the array are equal, the PSS implementation may choose to use the result of a previous evaluation, and not evaluate the function again.

22.2.6 Calling functions

Functions may be called directly from *exec block*s or from other functions using *procedural constructs* (see 22.7). Recursive function calls are allowed.

Functions not returning a value (declared with **void** return type) may only be called as standalone procedural statements. Functions returning a value may be used as operands in expressions; the value of that operand is the value returned by the function. The function can be used as a standalone statement and the return value discarded by casting the function call to **void**:

```
(void)function_call();
```

Calling a nonvoid function as if has no return value shall be legal, but it is recommended to explicitly discard the return value by casting the function call to **void**, as shown above.

Example 313 and Example 314 demonstrate calling various functions. In this example, the `mem_segment_s` **buffer** object captures information about a memory buffer with a random size. The specific address in an instance of the `mem_segment_s` object is computed using the `alloc_addr` function. `alloc_addr` is called after the solver has selected random values for the **rand** fields (specifically, `size` in this case) to select a specific address for the `addr` field.

```
package external_functions_pkg {
  function bit[31:0] alloc_addr(bit[31:0] size);

  function void transfer_mem(
    bit[31:0] src, bit[31:0] dst, bit[31:0] size
  );

  buffer mem_segment_s {
    rand bit[31:0]        size;
    bit[31:0]             addr;

    constraint size in [8..4096];

    exec post_solve {
      addr = alloc_addr(size);
    }
  }
}

component mem_xfer {
  import external_functions_pkg::*;

  action xfer_a {
    input  mem_segment_s     in_buff;
    output mem_segment_s    out_buff;

    constraint in_buff.size == out_buff.size;

    exec body {
      transfer_mem(in_buff.addr, out_buff.addr, in_buff.size);
    }
  }
}
```

Example 313—DSL: Calling functions

```
namespace external_functions_pkg {
  function<result<bit>( in_arg<bit> )> alloc_addr {
    "alloc_addr",
    result<bit>(width(31,0)), in_arg<bit>("size", width(31,0))
  };

  function<result<void>( in_arg<bit>, in_arg<bit>, in_arg<bit> )>
  transfer_mem {
    "transfer_mem",
    in_arg<bit>("src", width(31,0)),
    in_arg<bit>("dst", width(31,0)),
    in_arg<bit>("size",width(31,0))
  };

  class mem_segment_s : public buffer { ...
    rand_attr<bit> size { "size", width(31,0) };
    attr<bit> addr { "addr", width(31,0) };

    constraint c { in (size, range(8, 4096) ) };
  };
  type_decl<mem_segment_s> mem_segment_s_decl;
};

class mem_xfer : public component { ...
  using mem_segment_s = external_functions_pkg::mem_segment_s;

  class xfer_a : public action { ...
    input  <mem_segment_s> in_buff  {"in_buff"};
    output <mem_segment_s> out_buff {"out_buff"};

    constraint c { in_buff->size == out_buff->size };

    exec body { exec::body, external_functions_pkg::transfer_mem
                ( in_buff->addr, out_buff->addr, in_buff->size )
    };
  };
  type_decl<xfer_a> xfer_a_decl;
};
```

Example 314—C++: Calling functions

22.3 Native PSS functions

It is possible to specify the definition for native PSS functions using the procedural constructs described in 22.7.

If the function declaration is in a **component**, then the definition (if provided) shall be in the same **component** type (either in its initial definition or in an extension**)** or in a derived **component**. If the declaration is in a **package** (outside of any **component**), then the definition shall be in the same **package**.

22.3.1 DSL syntax

```
procedural_function ::= [ platform_qualifier ] [ pure ] function function_prototype
    { { procedural_stmt } }
platform_qualifier ::=
    target
  | solve
function_prototype ::= function_return_type function_identifier function_parameter_list_prototype
function_return_type ::=
    void
  | data_type
function_parameter_list_prototype ::=
    ( [ function_parameter { , function_parameter } ] )
  | ( { function_parameter , } varargs_parameter )
function_parameter ::=
    [ function_parameter_dir ] data_type identifier [ = constant_expression ]
  | ( type | ref type_category | struct ) identifier
function_parameter_dir ::=
    input
  | output
  | inout
varargs_parameter ::= ( data_type | type | ref type_category | struct ) ... identifier
type_category ::=
  action
 | component
 | struct_kind
```

Syntax 142—DSL: Function definition

The optional *platform_qualifier* (either **solve** or **target**) specifies function availability. An unqualified function is assumed to be available during all phases of test generation and execution.

For native PSS functions, *function_parameter_dir* shall be left unspecified for all parameters of the function, both in the original function declaration (if provided) and in the native PSS function definition.

22.3.2 C++ syntax

The corresponding C++ syntax for Syntax 142 is shown in Syntax 143. A constructor of `pss::function` accepts an instance of `std::function` as an argument. Typically, the user provides an in-place lambda as the argument.

NOTE—PSS/C++ version only allows definition of native PSS target functions, not of solve functions. PSS/C++ does allow declaration of function prototypes for both target and solve functions.

pss::function

Defined in **`pss/function.h`** (see C.30).

```
template<class T> class arg;

template<typename T> class function;
template<typename R, typename... Args> class function<R(Args...)>;// 1
template<typename... Args> class function<result<void>(Args...)>; // 2
```

1) Declare a function object with result R
2) Declare a function object with no result (void)

Member functions

`function (const scope &name, R result, Args... args, std::function <R(Args...)> ast_builder)` : declare function specified procedurally (with result)
`function (const scope &name, Args... args, std::function <void(Args...)> ast_builder)` : declare function specified procedurally (with no result)
`function (const scope &name, bool is_pure, R result, Args... args, std::function<R(Args...)> ast_builder)` : declare function specified procedurally (with pure modifier and result)
`function (const scope &name, bool is_pure, Args... args, std::function<void(Args...)> ast_builder)` : declare function specified procedurally (with pure modifier and no result)
`operator()(const T&... /*detail::AlgebExpr*/ params)` : operator ()

Syntax 143—C++: Function definition

As the **`ast_builder`** function executes, the PSS constructs inside the function object progressively create the AST (abstract syntax tree). Once the execution is complete, the AST is fully formed. The function **`ast_builder`** may be invoked zero, one, or multiple times. If it is invoked multiple times, it shall create the same AST each time.

22.3.3 Parameter passing semantics

Parameter direction shall be unspecified in the function prototype for native PSS functions. In the case of DSL, this implies that the parameter direction (**input**, **output**, or **inout**) shall not be used. In the case of PSS/C++, the function parameters shall be specified using template class **`arg<T>`**.

In the implementation of these functions, the following apply:

— Parameters of scalar data types are passed by value. Any changes to these parameters in the callee do not update the values in the caller.

— Parameters of aggregate data types are passed as a handle to the instance in the caller. Updates to these parameters in the callee will modify the instances in the caller. Note that as variables, parameters of aggregate data types have value semantics in assignment and equality expressions (see 9.3 and 9.5.3).

— Parameters of reference data types are passed as reference assignments. The parameter points to (is an alias to) the entity referred to in the actual parameter expression. Note that as variables, parameters of reference types have reference semantics in assignment and equality expressions (see 9.3 and 9.5.3), and may evaluate to **null**.

Example 315 shows the parameter passing semantics.

```
package generic_functions {
   struct params_s {
      int x;
   };

   // Prototypes
   function void set_val0(params_s p, int a);
   function void set_val1(params_s p_dst, params_s p_src);
   function params_s zero_attributes();

   // Definitions
   function void set_val0(params_s p, int a)
   {
      p.x = a;
      a = 0;
   }
   function void set_val1(params_s p_dst, params_s p_src)
   {
      p_dst.x = p_src.x;
   }
   function params_s zero_attributes()
   {
      params_s s;
      s.x = 0;
      return s;
   }

   component A {
      params_s p;
      int a;

      exec init_up {
         a = 10;
         p.x = 20;
         set_val0(p, a);
         // p.x is set to 10 at this point and a is unchanged

         set_val1(p, zero_attributes());
         // p.x is set to 0 at this point
      }
   };
}
```

Example 315—DSL: Parameter passing semantics

22.4 Foreign procedural interface

Function declarations in PSS may expose, and ultimately be bound to, foreign language APIs (functions, tasks, procedures, etc.) available on the target platform and/or on the solve platform. A function that was previously declared in the PSS description can be designated as *imported*. Calling an imported function from a PSS procedural context invokes the respective API in the foreign language. Parameters and result passing are subject to the type mapping defined for that language.

22.4.1 Definition using imported functions

Additional qualifiers are added to imported functions to provide more information to the tool about the way the function is implemented and/or in what phases of the test-creation process the function is available.

Imported function qualifiers are specified separately from the function declaration for modularity (see Syntax 144 and Syntax 145). In typical use, qualifiers are specified in an environment-specific package (e.g., a UVM environment-specific package or C-test-specific package).

22.4.1.1 DSL syntax

```
import_function ::=
    import [ platform_qualifier ] [ language_identifier ] function type_identifier ;
  | import [ platform_qualifier ] [ language_identifier ] function function_prototype ;
platform_qualifier ::=
    target
  | solve
```

Syntax 144—DSL: Imported function qualifiers

The following also apply:

a) Return values and parameter values of imported functions are restricted to the following types:
   1) **bit** or **int**, provided width is no more than 64 bits
   2) bool
   3) enum
   4) string
   5) chandle
   6) struct
   7) array whose element type is one of these listed types, including a sub-array

   See Annex E for type-mapping rules to C, C++, and SystemVerilog.
b) Parameter direction modifiers may be used in the **function** declaration or in the **import** declaration to specify the passing semantics between PSS and the foreign language:
   1) If the value of an **input** parameter is modified by the foreign language implementation, the updated value is not reflected back to the PSS model.
   2) An **output** parameter sets the value of a PSS model variable. The foreign language implementation shall consider the value of an **output** parameter to be unknown on entry; it shall specify a value for an **output** parameter.
   3) An **inout** parameter takes an initial value from a variable in the PSS model and reflects the value specified by the foreign language implementation back to the PSS model.
c) In the absence of an explicit direction modifier, parameters default to **input**.

In addition, the following apply when the second form of *import_function* is used (with the function prototype specified):

a) If the direction for a parameter is left unspecified in the **import** declaration, it defaults to **input**.
b) The prototype specified in the **import** declaration must match the prototype specified in the **function** declaration in the following way:
   1) The number of parameters must be identical.
   2) The parameter names and types must be identical.
   3) The return types must be identical.
c) If the **function** declaration specifies a parameter direction explicitly, the direction specified in the **import** declaration (either explicitly or by default) must match the **function** declaration.

d) If in the **function** declaration, the direction was unspecified for any parameter, the prototype specified in the **import** declaration can provide the direction of the parameter as **input**, **output** or **inout**.

22.4.1.2 C++ syntax

The corresponding C++ syntax for Syntax 144 is shown in Syntax 145.

pss::import_func

Defined in **`pss/function.h`** (see C.30).

```
enum kind {solve, target};
template <typename T> class import_func;
template <typename R, typename... Args>
        class import_func<function<R(Args...)>>; // 1
template <typename R, typename... Args>
        class import_func<function<result<void>(Args...)>>; // 2
```

1) Import function availability with result R

2) Import function availability with no result (void)

Member functions

`import_func (const scope &name, const kind a_kind)` : constructor
`import_func (const scope &name, const std::string &language)` : declare import function language
`import_func (const scope &name, const kind a_kind, const std::string &language)` : import function language and availability
`operator() (const T&... /*detail::AlgebExpr*/ params)` : operator

Syntax 145—C++: Imported function qualifiers

22.4.1.3 Specifying function availability

In some environments, test generation and execution are separate activities. In those environments, some functions may only be available during test generation, on the *solve platform*, while others are only available during test execution, on the *target platform*. For example, reference model functions may only be available during test generation while the utility functions that program hardware devices may only be available during test execution.

An unqualified imported function is assumed to be available during all phases of test generation and execution. Qualifiers are specified to restrict a function's availability. Functions restricted to the *solve platform* shall not be called directly or indirectly from *target exec*s, namely **body**, **run_start**, and **run_end**. Similarly, functions restricted to the *target platform* shall not be called from *solve exec*s, namely **init_down**, **init_up**, **pre_solve**, and **post_solve**.

Example 316 and Example 317 specify function availability. Two imported functions are declared in the `external_functions_pkg` package. The `alloc_addr` function allocates a block of memory, while the `transfer_mem` function causes data to be transferred. Both of these functions are present in all phases of test execution in a system where solving is done on-the-fly as the test executes.

In a system where a pre-generated test is to be compiled and run on an embedded processor, memory allocation may be pre-computed. Data transfer shall be performed when the test executes. The `pregen_tests_pkg` package specifies these restrictions: `alloc_addr` is only available during the solving phase of stimulus generation, while `transfer_mem` is only available during the execution phase of stimulus generation. PSS processing uses this specification to ensure that the way imported functions are used aligns with the restrictions of the target environment. Notice the use of the **`decltype`** specifier in Example 317 in the **`import_func`** declarations of `alloc_addr` and `transfer_mem` in the `pregen_tests_pkg` package.

```
package external_functions_pkg {

  function bit[31:0] alloc_addr(bit[31:0] size);

  function void transfer_mem(
    bit[31:0] src, bit[31:0] dst, bit[31:0] size
  );
}

package pregen_tests_pkg {

  import solve function external_functions_pkg::alloc_addr;

  import target function external_functions_pkg::transfer_mem;

}
```

Example 316—DSL: Function availability

```
namespace external_functions_pkg {
   function<result<bit>(in_arg<bit>)>
      alloc_addr { "alloc_addr",
         result<bit>(width(31,0)),
         in_arg<bit>("size", width(31,0))
   };
   function<result<void>(in_arg<bit>, in_arg<bit>, in_arg<bit>)>
      transfer_mem {"transfer_mem",
         in_arg<bit>("src",  width(31,0)),
         in_arg<bit>("dst",  width(31,0)),
         in_arg<bit>("size", width(31,0))
   };
};

namespace pregen_tests_pkg {
  import_func<decltype(external_functions_pkg::alloc_addr)>
    alloc_addr   {"external_functions_pkg::alloc_addr"  , solve};
  import_func<decltype(external_functions_pkg::transfer_mem)>
    transfer_mem {"external_functions_pkg::transfer_mem", target};
};
```

Example 317—C++: Function availability

When C++ based PSS input is used, if the solve-time function is also implemented in C++, it is not necessary to explicitly import the function before it can be used in **pre_solve** and **post_solve**. For an example of calling C++ functions natively, see Example 344.

Example 318 and Example 319 demonstrate an **activity** with reactive control flow based on values returned from a target function called in an **exec body** block.

```
component my_ip_c {
  function int sample_DUT_state();
  import target C function sample_DUT_state;
     // specify mapping to target C function by that same name

  action check_state {
    int curr_val;
    exec body {
      curr_val = comp.sample_DUT_state();
         // value only known during execution on target platform
    }
  };

  action A { };
  action B { };

  action my_test {
    check_state cs;
    activity {
      repeat {
        cs;
        if (cs.curr_val % 2 == 0) {
          do A;
        } else {
          do B;
        }
      } while (cs.curr_val < 10);
    }
  };
};
```

Example 318—DSL: Reactive control flow

```
class my_ip_c : public component { ...
  function<result<int>()> sample_DUT_state
    {"sample_DUT_state",result<int>()};
  import_func<function<result<int>()>> impl_decl
    {"sample_DUT_state", target, "C"};

  class check_state : public action { ...
    attr<int> curr_val {"curr_val"};

    exec body { exec::body,
      curr_val = comp<my_ip_c>()->sample_DUT_state()
    };
  };
  type_decl<check_state> check_state_decl;

  class A : public action {...};
  class B : public action {...};

  class my_test : public action { ...
    action_handle<check_state> cs {"cs"};

    activity actv {
      repeat_while {
        sequence {
          cs,
          if_then_else { cond (cs->curr_val % 2 == 0),
                         action_handle<A>(),
                         action_handle<B>()
          }
        }
        , cs->curr_val < 10
       }
    };
  };
  type_decl<my_test> my_test_decl;
};
...
```

Example 319—C++: Reactive control flow

22.4.1.4 Specifying an implementation language

The implementation language for an imported function can be specified implicitly or explicitly. In many cases, the implementation language need not be explicitly specified because the PSS processing tool can use sensible defaults (e.g., all imported functions are implemented in C++). Explicitly specifying the implementation language using a separate statement allows different imported functions to be implemented in different languages, however (e.g., reference model functions are implemented in C++, while functions to drive stimulus are implemented in SystemVerilog).

Example 320 and Example 321 show explicit specification of the foreign language in which the imported function is implemented. In this case, the function is implemented in C. Notice that only the name of the imported function is specified and not the full function prototype.

```
package known_c_functions {
    import C function generic_functions::compute_expected_value;
}
```

Example 320—DSL: Explicit specification of the implementation language

```
namespace known_c_functions {
  import_func<function<result<void>()>> compute_expected_value {
   "generic_functions::compute_expected_value", "C"
  };
};
```

Example 321—C++: Explicit specification of the implementation language

22.4.2 Imported classes

In addition to interfacing with external foreign language functions, the PSS description can interface with foreign language classes. See also Syntax 146 and Syntax 147.

22.4.2.1 DSL syntax

import_class_decl ::= **import class** *import_class*_identifier [import_class_extends]
  { { import_class_function_decl } **}**

import_class_extends ::= **:** type_identifier { **,** type_identifier }

import_class_function_decl ::= function_prototype **;**

Syntax 146—DSL: Import class declaration

The following also apply:

a) Imported class functions support the same return and parameter types as imported functions. **import class** declarations also support capturing the class hierarchy of the foreign language classes.

b) Fields of **import class** type can be instantiated in **package** and **component** scopes. An **import class** field in a **package** scope is a global instance. A unique instance of an **import class** field in a **component** exists for each component instance.

c) Imported class functions are called from an *exec block* just as imported functions are.

22.4.2.2 C++ syntax

The corresponding C++ syntax for Syntax 146 is shown in Syntax 147.

pss::import_class

Defined in **`pss/import_class.h`** (see C.32).

```
class import_class;
```

Declare an import class.

Member functions

`import_class (const scope &name)` : constructor

Syntax 147—C++: Import class declaration

22.4.2.3 Examples

Example 322 and Example 323 declare two imported classes. **import class** `base` declares a function `base_function`, while **import class** `ext` extends from **import class** `base` and adds a function named `ext_function`.

```
import class base {
    void base_function();
}

import class ext : base {
    void ext_function();
}
```

Example 322—DSL: Import class

```
class base : public import_class { ...
  function<result<void>()> base_function {"base_function"};
};
type_decl<base> base_decl;

class ext : public base { ...
  function<result<void>()> ext_function {"ext_function"};
};
type_decl<ext> ext_decl;
```

Example 323—C++: Import class

22.5 Target-template implementation of exec blocks

Implementation of **exec**s may be specified using a *target template*—a string literal containing code in a specific foreign language, optionally embedding references to fields in the PSS description. Target-template implementation is restricted to *target exec* kinds (**body**, **run_start**, **run_end**, **header**, and **declaration)**. In addition, target templates can be used to generate other text files using **exec file**. Target-template implementations may not be used for *solve exec*s **(init_down**, **init_up**, **pre_solve**, and **post_solve**).

Target-template **exec**s are inserted by the PSS tool verbatim into the generated test code, with embedded expressions substituted with their actual values. Multiple target-template *exec block*s of the same kind are allowed for a given action, flow/resource object, or **struct**. They are (logically) concatenated in the target file, as if they were all concatenated in the PSS source.

22.5.1 Target language

A *language_identifier* serves to specify the intended target programming language of the code block. Clearly, a tool supporting PSS must be aware of the target language to implement the runtime semantics. PSS does not enforce any specific target language support, but recommends implementations reserve the identifiers **`C`**, **`CPP`**, and **`SV`** to denote the languages C, C++, and SystemVerilog respectively. Other target languages may be supported by tools, given that the abstract runtime semantics are kept. PSS does not define any specific behavior if an unrecognized *language_identifier* is encountered.

Each target-template **exec** block is restricted to one target language in the context of a specific generated test. However, the same **action** may have target-template **exec** blocks in different languages under different **package**s, given that these **package**s are not used for the same test.

22.5.2 exec file

Not all the artifacts needed for the implementation of tests are coded in a programming language that tools are expected to support as such. Tests may require scripts, command files, make files, data files, and files in other formats. The **exec file** construct (see 22.1) specifies text to be generated out to a given file. **exec file** constructs of different actions/objects with the same target are concatenated in the target file in their respective scenario flow order.

22.5.3 Referencing PSS fields in target-template exec blocks

Implementing test intent requires using data from the PSS model in the code created from target-template exec blocks. PSS variables are referenced using *mustache* notation: **`{{expression}}`**. A reference is to an expression involving variables declared in the scope in which the exec block is declared. Only scalar variables (except **chandle**) can be referenced in a target-template exec block.

22.5.3.1 Examples

Example 324 shows referencing PSS variables inside a target-template exec block using mustache notation.

```
component top {
   struct S {
      rand int b;
   }
   action A {
      rand int a;
      rand S   s1;
      exec body C = """
        printf("a={{a}} s1.b={{s1.b}} a+b={{a+s1.b}}\n");
      """;
   }
}
```

Example 324—DSL: Referencing PSS variables using mustache notation

A variable reference can be used in any position in the generated code. Example 325 shows a variable reference used to select the function being called.

```
component top {
    action A {
        rand bit[1:0] func_id;
        rand bit[3:0] a;
        exec body C = """
          func_{{func_id}}({{a}});
        """;
    }
}
```

Example 325—DSL: Variable reference used to select the function

One implication of this is that a mustache reference cannot be used to assign a value to a PSS variable.

Example 325 also declares a random `func_id` variable that identifies a C function to call. When a PSS tool processes this description, the following output shall result, assuming `func_id==1` and `a==4`:

```
func_1(4);
```

Example 326 shows how a procedural **pre_solve** exec block is used along with a target-template declaration exec block to allow programmatic declaration of a target variable declaration.

```
enum obj_type_e {my_int8,my_int16,my_int32,my_int64};
function string get_unique_obj_name();
import solve function get_unique_obj_name;

buffer mem_buff_s {
  rand obj_type_e obj_type;
  string obj_name;

  exec post_solve {
    obj_name = get_unique_obj_name();
  }

  // declare an object in global space
  exec declaration C = """
    static {{obj_type}} {{obj_name}};
  """;
};
```

Example 326—DSL: Allowing programmatic declaration of a target variable declaration

Assume that the solver selects `my_int16` as the value of the `obj_type` field and that the `get_unique_obj_name()` function returns `field__0`. In this case, the PSS processing tool shall generate the following content in the declaration section:

```
static my_int16 field__0;
```

22.5.3.2 Formatting

When a variable reference is converted to a string, the result is formatted as follows:

—	**int**	signed decimal (`%d`)
—	**bit**	unsigned decimal (`%ud`)
—	**bool**	`"true"` | `"false"`
—	**string**	string (`%s`)
—	**chandle**	pointer (`%p`)

22.6 Target-template implementation for functions

When integrating with languages that do not have the concept of a "function," such as assembly language, the implementation for functions can be provided by target-template code strings.

The target-template form of functions (see Syntax 148 and Syntax 149) allows interactions with a foreign language that do not involve a procedural interface. Examples are injecting assembly code or global variables into generated tests. The target-template forms of 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 22.2.1.1). PSS expressions embedded in the target code (using mustache notation) can make reference to the instance attributes, optionally using **`this`**.

See also 22.5.3.

22.6.1 DSL syntax

```
target_template_function ::= target language_identifier
    function function_prototype =  string_literal ;
```

Syntax 148—DSL: Target-template function implementation

The following also apply:

a) If the direction for a parameter is left unspecified in the target template declaration, it defaults to **input**.

b) The prototype specified in the target template declaration must match the prototype specified in the **function** declaration in the following way:

   1) The number of parameters must be identical.

   2) The parameter names and types must be identical.

   3) The return types must be identical.

c) If the **function** declaration specifies a parameter direction explicitly, the direction specified in the target template declaration (either explicitly or by default) must match the **function** declaration.

d) If in the **function** declaration, the direction was unspecified for any parameter, the prototype specified in the target template declaration can provide the direction of the parameter as **input**, **output** or **inout**.

22.6.2 C++ syntax

The corresponding C++ syntax for Syntax 148 is shown in Syntax 149.

pss::function

Defined in **`pss/function.h`** (see C.30).

```
template <typename T> class function;
template <typename R, typename... Args> class function<R(Args...)>;// 1
template <typename... Args> class function<result<void>(Args...)>; // 2
```

1) Declare a target template with result R
2) Declare a target template with no result (void)

Member functions

```
function (const scope &name, const std::string &language, R result, Args...
   args, const std::string &target_template ) 
```
: declare target-template function with result
```
function (const scope &name, const std::string &language, Args... args,
   const std::string &target_template ) 
```
: declare target-template function without result
```
operator()(const T&... /*detail::AlgebExpr*/ params) 
```
: operator

Syntax 149—C++: Target-template function implementation

22.6.3 Examples

Example 327 and Example 328 provide an assembly-language target-template code block implementation for the `do_stw` function. Function parameters are referenced using mustache notation (**`{{variable}}`**).

```
package thread_ops_pkg {
  function void do_stw(bit[31:0] val, bit[31:0] vaddr);
}

package thread_ops_asm_pkg {
   target ASM function void do_stw(bit[31:0] val, bit[31:0] vaddr) = """
      loadi RA {{val}}
      store RA {{vaddr}}
    """;
}
```

Example 327—DSL: Target-template function implementation

```
namespace thread_ops_pkg {
  function<result<void>(in_arg<bit>, in_arg<bit>)> do_stw { "do_stw",
      in_arg<bit>( "val", width(31,0)),
      in_arg<bit>("vaddr", width(31,0)) };
};

namespace thread_ops_asm_pkg {
  function<result<void>(in_arg<bit>, in_arg<bit>)> do_stw { "do_stw",
    "ASM",
    in_arg<bit>( "val", width(31,0)),
    in_arg<bit>("vaddr", width(31,0)),
    R"(
       loadi RA {{val}}
       store RA {{vaddr}}
      )"
  };
};
```

Example 328—C++: Target-template function implementation

22.7 Procedural constructs

This section specifies the procedural control flow constructs. When relevant, these constructs have the same syntax and execution semantics as the corresponding activity control flow statements (see 13.4).

The PSS/C++ control flow constructs are based on the same classes as used for activity control flow constructs; the constructors described below are overloads on the constructors specified under 13.4. PSS function definitions and exec definitions shall only use the constructor forms specified in this section; it shall be an error to use activity control flow constructs (except in cases explicitly specified in this chapter).

22.7.1 Scoped blocks

A scoped block creates a new unnamed nested scope, similar to C-style blocks.

22.7.1.1 DSL syntax

```
procedural_stmt ::=
   procedural_sequence_block_stmt
  | ...
procedural_sequence_block_stmt ::= [ sequence ] { { procedural_stmt } }
```

Syntax 150—DSL: Procedural block statement

The **sequence** keyword before the block statement is optional, and is provided to let users state explicitly that the statements are executed in sequence.

Typically, blocks are used to group multiple statements that are part of a control flow statement (such as **repeat**, **if-else**, etc.). It is also valid to have a stand-alone block that is not part of a control flow statement, in which case the following equivalencies apply:

— A stand-alone block that does not create new variables (and hence does not destroy any variables when the scope ends) is equivalent (in so far as to the AST constructed) to the case where the contents of the code block are merged with the enclosing parent block. For example:

```
{
    int a;
    int b;
    {
        b = a;
    }
}
```

is equivalent to

```
{
    int a;
    int b;
    b = a;
}
```

— If the start of an enclosing block coincides with the start of the stand-alone nested block (i.e., with no statements in between) and similarly the end of that enclosing block coincides with the end of the stand-alone nested block, it is then equivalent to the case where there is just a single code-block with the contents of the nested block. For example:

```
{
    {
        int a;
        int b;
        //
    }
}
```

is equivalent to

```
{
    int a;
    int b;
    //
}
```

22.7.1.2 C++ syntax

There is no special syntax element for stand-alone blocks in C++; the native C++ blocks (in braces) are used. The PSS/C++ implementation must correctly infer the start/end of blocks. An implementation may use the order in which variables are constructed/destructed to infer nested scope blocks.

```
{
  attr<int> a("a");
  attr<int> b("b");

  a = b;
  {
     attr<int> c("c");
     c = a;
  }
}
```

Example 329—C++: Procedural block statement

For control flow statements that accept a block as an argument, the following options are possible:

— An in-place lambda

— A **sequence** construct

For example, the conditional **`if_then`** statement may be specified as

```
if_then( cond(a > b),
   [&]() { c = a; d = b; }
);
```

or

```
if_then( cond(a > b),
   sequence { c = a, d = b }
);
```

The in-place lambda form is more general and also allows the user to declare variables (22.7.2). C++ user code can also be invoked by the PSS implementation during the solve phase.

Example:

```
if_then( cond(a > b),
   [&]() { attr<int> x; x = a; a = b; b = x; }
);
```

22.7.2 Variable declarations

Variables may be declared with the same notation used in other declarative constructs (e.g., **action**). The declaration may be placed at any point in a scope (i.e., C++ style) and does not necessarily have to be declared at the beginning of a scope. However, the declaration shall precede any reference to the variable.

All data types listed in Clause 8 may be used for variable types. It shall be an error to instantiate **rand** variables in a procedural context.

22.7.2.1 DSL syntax

```
procedural_stmt ::=
   procedural_sequence_block_stmt
  | procedural_data_declaration
  | ...
procedural_data_declaration ::= data_type procedural_data_instantiation
   { , procedural_data_instantiation } ;
procedural_data_instantiation ::= identifier [ array_dim ] [ = expression ]
```

Syntax 151—DSL: Procedural variable declaration

22.7.2.2 C++ syntax

C++ uses the constructs described in Clause 8 for declaring variables (**`attr<...>`**, etc.). Additionally, in procedural contexts, the constructors in Syntax 152 are also allowed, for initialization of scalar variables to a non-constant value.

pss::attr

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

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

Declare a scalar variable.

Member functions

`attr(const scope& s, const detail::AlgebExpr& init_val)` : constructor, with non-constant initial value (T = int, bit, string, or bool)
`attr(const scope& s, const width& a_width, const detail::AlgebExpr& init_val)` : constructor, with width and non-constant initial value (T = int or bit only)
`attr(const scope& s, const range& a_range, const detail::AlgebExpr& init_val)` : constructor, with range and non-constant initial value (T = int, bit, or string)
`attr(const scope& s, const width& a_width, const range& a_range, const detail::AlgebExpr& init_val)` : constructor, with width, range and non-constant initial value (T = int or bit only)

Syntax 152—C++: Initialization of variables with non-constant values

NOTE—The variables need to be destructed in the reverse order of construction. This is automatically achieved if all variables are on the stack. Otherwise, if they are allocated on the heap, the user must ensure correct order of destruction.

22.7.3 Assignments

Assignments to variables in the scope may be made.

22.7.3.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | ...
procedural_assignment_stmt ::= ref_path assign_op expression ;
```

Syntax 153—DSL: Procedural assignment statement

The following rules apply to assignments in native PSS functions and execs:

a) A plain-data variable declared within a function/exec scope may be assigned in the scope where it is visible with no restriction.
b) A native PSS function definition may set data attributes of **component** instances through **component** references passed as parameters. Instance functions may similarly set data attributes of their context **component** directly. Since **component** attributes can only be set during the initialization phase, a function that sets such data attributes shall be called only from within **exec init_down** or **init_up**.
c) An **exec init_down** or **init_up** block may set the data attributes of the **component** instance directly in the body of the **exec**.

d) Data attributes of a **struct** instance may be set using the handle passed as a parameter. Similarly, data attributes of **actions** and flow/resource objects may be set using the reference passed as a parameter. A function that sets such data attributes may be invoked in **init**, **solve** or **body** execs.

e) A **struct** instance may be assigned to another **struct** instance of the same type, which results in a deep-copy operation of the data attributes. That is, this single assignment is equivalent to individually setting data attributes of the left-side instance to the corresponding right-side instance, for all the data attributes directly present in that type or in a contained **struct** type.

22.7.4 Void function calls

Functions not returning a value (declared with **void** return type) may only be called as standalone procedural statements. Functions returning a value may be used as a standalone statement and the return value discarded by casting the function call to **void**:

```
(void)function_call();
```

Calling a nonvoid function as if has no return value shall be legal, but it is recommended to explicitly discard the return value by casting the function call to **void**, as shown above.

22.7.4.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | ...
procedural_void_function_call_stmt ::= [ ( void ) ] function_call ;
```

Syntax 154—DSL: Void function call

22.7.5 return statement

PSS functions shall return a value to the caller using the **return** statement. In PSS functions that do not return a value, the **return** statement without an argument shall be used.

The **return** statement without an argument can also be used in **exec**s. The **return** signifies end of execution—no further statements in the **exec** are executed.

22.7.5.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | ...
procedural_return_stmt ::= return [ expression ] ;
```

Syntax 155—DSL: Procedural return statement

22.7.5.2 C++ syntax

The corresponding C++ syntax for Syntax 155 is shown in Syntax 156.

pss::pss_return

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class pss_return;
```

Declare return procedural statement.

Member functions

`pss_return (void)`: constructor - with no parameters
`pss_return (const detail::AlgebExpr& expr)`: constructor - with return argument

Syntax 156—C++: Procedural return statement

22.7.5.3 Examples

```
target function int add(int a, int b) {
  return (a+b);
}
```

Example 330—DSL: Procedural return statement

```
function<result<int>(arg<int>, arg<int>)> add { "add",
  result<int>(), arg<int>("a"), arg<int>("b"),[&](arg<int> a, arg<int> b) {
     pss_return {a+b};
  }
};
```

Example 331—C++: Procedural return statement

22.7.6 repeat (count) statement

The procedural **repeat** statement allows the specification of a loop consisting of one or more procedural statements. This section describes the *count-expression* variant (see Syntax 157 and Syntax 158) and 22.7.7 describes the *while-expression* variants.

22.7.6.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | ...
procedural_repeat_stmt ::=
    repeat ( [ index_identifier : ] expression ) procedural_stmt
  | ...
```

Syntax 157—DSL: Procedural repeat-count statement

The following also apply:

a) *expression* shall be a non-negative numeric expression (**int** or **bit**).

b) Intuitively, the *procedural_stmt* is iterated the number of times specified in the *expression*. An optional index-variable identifier can be specified that ranges between `0` and one less than the iteration count. If the expression evaluates to `0`, the *procedural_stmt* is not evaluated at all.

22.7.6.2 C++ syntax

The corresponding C++ syntax for Syntax 157 is shown in Syntax 158.

pss::repeat

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class repeat;
```

Declare a repeat statement.

Member functions

`repeat (const detail::AlgebExpr& count, std::function<void(void)> loop_stmts)` : declare a procedural repeat (with count) statement using lambda
`repeat (const detail::AlgebExpr& count, detail::Stmt& /* sequence& */ loop_stmts)` : declare a procedural repeat (with count) statement using sequence construct
`repeat (const attr<int>& iter, const detail::AlgebExpr& count, std::function <void(void)> loop_stmts)` : declare a procedural repeat (with count) statement with iterator using lambda
`repeat (const attr<int>& iter, const detail::AlgebExpr& count, detail::Stmt& /* sequence& */ loop_stmts)` : declare a procedural repeat (with count) statement with iterator using sequence construct

Syntax 158—C++: Procedural repeat-count statement

22.7.6.3 Examples

```
target function int sum(int a, int b) {
  int res;

  res = 0;

  repeat(b) {
     res = res + a;
  }

  return res;
}
```

Example 332—DSL: Procedural repeat-count statement

```
function<result<int>(arg<int>, arg<int>)> sum {"sum",
  result<int>(), arg<int>("a"), arg<int>("b"), [&](arg<int> a, arg<int> b) {
     attr<int> res("res");

     res = 0;

     repeat (b,
        [&]() { res = res + a; }
     );

     pss_return {res};
  }
};
```

Example 333—C++: Procedural repeat-count statement

22.7.7 repeat-while statement

The procedural **repeat** statement allows the specification of a loop consisting of one or more procedural statements. This section describes the *while-expression* variants (see Syntax 159 and Syntax 160).

22.7.7.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | ...
procedural_repeat_stmt ::=
  . . .
  | repeat procedural_stmt while ( expression ) ;
  | while ( expression ) procedural_stmt
```

Syntax 159—DSL: Procedural repeat-while statement

The following also apply:

a) *expression* shall be of type **bool**.

b) Intuitively, the *procedural_stmt* is iterated so long as the *expression* condition is *true*, as sampled before the *procedural_stmt* (in the **while** variant) or after (in the **repeat-while** variant).

22.7.7.2 C++ syntax

The corresponding C++ syntax for Syntax 159 is shown in Syntax 160.

pss::repeat_while

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class repeat_while;
```

Declare a procedural repeat-while statement.

Member functions

```
repeat_while (std::function<void(void)> loop_stmts, const cond&
    a_cond)
```
: declare a procedural repeat-while statement using lambda

```
repeat_while (const detail::Stmt& /*sequence&*/ loop_stmts, const cond&
    a_cond)
```
: declare a procedural repeat-while statement using sequence construct

pss::while_do

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class while_do;
```

Declare a procedural while-do statement.

Member functions

```
while_do(const cond& a_cond, std::function<void(void)> loop_stmts) :
```
declare a procedural while-do statement using lambda

```
while_do(const cond& a_cond, const detail::Stmt& /*sequence&*/ loop_stmts) :
```
declare a procedural while-do statement using sequence construct

Syntax 160—C++: Procedural repeat-while statement

22.7.7.3 Examples

```
target function bool get_parity(int n) {
  bool parity;

  parity = false;
  while (n != 0) {
     parity = !parity;
     n = n & (n-1);
  }

  return parity;
}
```

Example 334—DSL: Procedural while statement

```
function<result<bool>(arg<int>)> get_parity {"get_parity",
  result<bool>(), arg<int>("n"), [&](arg<int> n) {

     attr<bool> parity("parity");

     while_do( (n != 0),
        [&]() {
           parity = !parity;
           n = n & (n-1);
        }
     );

     pss_return {parity};
  }
};
```

Example 335—C++: Procedural while statement

22.7.8 foreach statement

The procedural **foreach** statement allows the specification of a loop that iterates over the elements of a collection (see Syntax 161 and Syntax 162).

22.7.8.1 DSL syntax

```
procedural_stmt ::=
   procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | procedural_foreach_stmt
  | ...
procedural_foreach_stmt ::=
  foreach ( [ iterator_identifier : ] expression  [ [ index_identifier ] ] ) procedural_stmt
```

Syntax 161—DSL: Procedural foreach statement

The following also apply:

a) *expression* shall be of a collection type (i.e., **array**, **list**, **map** or **set**).

b) The body of the **foreach** statement is a sequential block in which *procedural_stmt* is evaluated once for each element in the collection.

c) *iterator_identifier* specifies the name of an iterator variable of the collection element type. Within *procedural_stmt*, the iterator variable, when specified, is an alias to the collection element of the current iteration.

d) *index_identifier* specifies the name of an index variable. Within *procedural_stmt,* the index variable, when specified, corresponds to the element index of the current iteration.

1) For **array**s and **list**s, the index variable shall be a variable of type **int**, ranging from `0` to one less than the size of the collection variable, in that order.
2) For **map**s, the index variable shall be a variable of the same type as the **map** keys, and range over the values of the keys. The order of key traversal is undetermined.
3) For **set**s, an index variable shall not be specified.

e) Both the index and iterator variables, if specified, are implicitly declared within the **foreach** scope and limited to that scope. Regular name resolution rules apply when the implicitly declared variables are used within the **foreach** body. For example, if there is a variable in an outer scope with the same name as the index variable, that variable is shadowed (masked) by the index variable within the **foreach** body. The index and iterator variables are not visible outside the **foreach** scope.

f) Either an index variable or an iterator variable or both shall be specified. For a **set**, an iterator variable shall be specified, but not an index variable.

g) The index and iterator variables are read-only. Their values shall not be changed within the **foreach** body. It shall be an error to change the contents of the iterated collection variable with the **foreach** body.

22.7.8.2 C++ syntax

The corresponding C++ syntax for Syntax 161 is shown in Syntax 162.

pss::foreach

Defined in **`pss/foreach.h`** (C.29).

```
class foreach;
```

Declare a foreach statement.

Member functions

```
template<T>
foreach (const attr<T>& iter, const attr<vec<T>>& array,
    std::function<void(void)> loop_stmts)
```
: declare procedural foreach statement on array of non-rand attributes using lambda (specializations provided for T = int and bit)

```
foreach (const attr<T>& iter, const rand_attr<vec<T>>& array,
    std::function<void(void)> loop_stmts)
```
: declare procedural foreach statement on array of rand attributes using lambda (specializations provided for T = int and bit)

```
template<T>
foreach (const attr<T>& iter, const attr<vec<T>>& array, const detail::Stmt&
    /* sequence& */ loop_stmts)
```
: declare procedural foreach statement on array of non-rand attributes using sequence construct (specializations provided for T = int and bit)

```
foreach (const attr<T>& iter, const rand_attr<vec<T>>& array, const
    detail::Stmt& /* sequence& */ loop_stmts)
```
: declare procedural foreach statement on array of rand attributes using sequence construct (specializations provided for T = int and bit)

Syntax 162—C++: Procedural foreach statement

NOTE—Only iteration over arrays is supported in PSS/C++. foreach iteration over other collection types is not supported.

NOTE—In PSS/C++, the index and iteration variables must be explicitly declared in the containing scope of the foreach loop.

22.7.9 if-else statement

The procedural **if-else** statement introduces a branch point (see Syntax 163 and Syntax 164).

22.7.9.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | procedural_foreach_stmt
  | procedural_if_else_stmt
  | ...
procedural_if_else_stmt ::= if ( expression ) procedural_stmt [ else procedural_stmt ]
```

Syntax 163—DSL: Procedural if-else statement

expression shall be of type **bool**.

22.7.9.2 C++ syntax

The corresponding C++ syntax for Syntax 163 is shown in Syntax 164.

pss::if_then

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then;
```

Declare if-then procedural statement.

Member functions

`if_then (const cond& a_cond, std::function<void(void)> true_stmts)`
: Declare procedural if-then statement using lamda

`if_then (const cond& a_cond, const detail::Stmt& /* sequence& */ true_stmts)`
: Declare procedural if-then statement using sequence construct

pss::if_then_else

Defined in **`pss/if_then.h`** (see C.31).

```
class if_then_else;
```

Declare if-then-else procedural statement.

Member functions

`if_then_else (const cond& a_cond, std::function<void(void)> true_stmts, std::function<void(void)> false_stmts)` : Declare procedural if-then-else statement using only lamda

`if_then_else (const cond& a_cond, std::function<void(void)> true_stmts, const detail::Stmt& /* sequence& */ false_stmts)` : Declare procedural if-then-else statement using lambda (for true statements) and sequence construct (for false statements)

`if_then_else (const cond& a_cond, const detail::Stmt& /* sequence& */ true_stmts, std::function<void(void)> false_stmts)` : Declare procedural if-then-else statement using sequence construct (for true statements) and lamda (for false statements)

`if_then_else (const cond& a_cond, const detail::Stmt& /* sequence& */ true_stmts, const detail::Stmt& /* sequence& */ false_stmts)` : Declare procedural if-then-else statement using only sequence constructs

Syntax 164—C++: Procedural if-else statement

22.7.9.3 Examples

```
target function int max(int a, int b) {
  int c;

  if (a > b) {
     c = a;
  } else {
     c = b;
  }

  return c;
}
```

Example 336—DSL: Procedural if-else statement

```
function<result<int>(arg<int>, arg<int>)> max {"max",
  result<int>(), arg<int>("a"), arg<int>("b"),[&](arg<int> a, arg<int> b) {
     attr<int> c("c");

     if_then_else(cond(a > b),
        [&]() { c = a; },
        [&]() { c = b; }
     );

     pss_return {c};
  }
};
```

Example 337—C++: Procedural if-else statement

22.7.10 match statement

The procedural **match** statement specifies a multi-way decision point that tests whether an expression matches one of a number of other expressions and executes the matching branch accordingly (see Syntax 165 and Syntax 166).

22.7.10.1 DSL syntax

```
procedural_stmt ::=
   procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | procedural_foreach_stmt
  | procedural_if_else_stmt
  | procedural_match_stmt
  | ...
procedural_match_stmt ::=
   match ( match_expression ) { procedural_match_choice { procedural_match_choice } }
match_expression ::= expression
procedural_match_choice ::=
   [ open_range_list ] : procedural_stmt
  | default : procedural_stmt
```

Syntax 165—DSL: Procedural match statement

The following also apply:

a) When the **match** statement is evaluated, the *match_expression* is evaluated.

b) After the *match_expression* is evaluated, the *open_range_list* of each *procedural_match_choice* shall be compared to the *match_expression*. *open_range_list*s are described in 9.5.9.1.

c) If there is exactly one match, then the corresponding branch shall be evaluated.

d) It shall be an error if more than one match is found for the *match_expression*.

e) If there are no matches, then the **default** branch, if provided, shall be evaluated.

f) The **default** branch is optional. There may be at most one **default** branch in the **match** statement.

g) If a **default** branch is not provided and there are no matches, it shall be an error.

22.7.10.2 C++ syntax

The corresponding C++ syntax for Syntax 165 is shown in Syntax 166.

pss::match

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class match;
```

Declare a match statement.

Member functions

```
template<class... R> match (const cond& expr,
       R&&... /* choice|default_choice */ stmts) : constructor
```

pss::choice

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class choice;
```

Declare a match choice statement.

Member functions

```
template<class... R> choice (const range& range_expr,
       R&&... /* std::function|sequence& */ choice_stmts) : constructor
```

pss::default_choice

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class default_choice;
```

Declare a match default choice statement.

Member functions

```
template<class... R> default_choice (
       R&&... /* std::function|sequence& */ choice_stmts) : constructor
```

Syntax 166—C++: Procedural match statement

22.7.10.3 Examples

```
target function int bucketize(int a) {
  int res;

  match (a) {
     [0..3]:  res = 1;
     [4..7]:  res = 2;
     [8..15]: res = 3;
     default: res = 4;
  }

  return res;
}
```

Example 338—DSL: Procedural match statement

```
function<result<int>(arg<int>)> bucketize { "bucketize",
  result<int>(), arg<int>("a"), [&](arg<int> a) {
     attr<int> res("res");

     match (cond(a),
        choice {range(0, 3), [&]() { res = 1; }},
        choice {range(4, 7), [&]() { res = 2; }},
        choice {range(8, 15),[&]() { res = 3; }},
        default_choice {     [&]() { res = 4; }}
     );

     pss_return {res};
  }
};
```

Example 339—C++: Procedural match statement

22.7.11 break/continue statement

The procedural **break** and **continue** statements allow for additional control in loop termination (see Syntax 167 and Syntax 168).

22.7.11.1 DSL syntax

```
procedural_stmt ::=
    procedural_sequence_block_stmt
  | procedural_data_declaration
  | procedural_assignment_stmt
  | procedural_void_function_call_stmt
  | procedural_return_stmt
  | procedural_repeat_stmt
  | procedural_foreach_stmt
  | procedural_if_else_stmt
  | procedural_match_stmt
  | procedural_break_stmt
  | procedural_continue_stmt
  | stmt_terminator
procedural_break_stmt ::= break ;
procedural_continue_stmt ::= continue ;
```

Syntax 167—DSL: Procedural break/continue statement

The following also apply:

a) The semantics are similar to **break** and **continue** in C++.

b) **break** and **continue** may only appear within loop statements (**repeat**-count, **repeat-while** or **foreach**). Within a loop, **break** and **continue** may be nested in conditional branch or **match** statements.

c) **break** and **continue** affect the innermost loop statement they are nested within.

d) **break** signifies that execution should continue from the statement after the enclosing loop construct. **continue** signifies that execution should proceed to the next loop iteration.

22.7.11.2 C++ syntax

The corresponding C++ syntax for Syntax 167 is shown in Syntax 168.

pss::pss_break

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class pss_break;
```

Declare a 'break' statement.

Member functions

`pss_break (void)`: constructor

pss::pss_continue

Defined in **`pss/ctrl_flow.h`** (see C.21).

```
class pss_continue;
```

Declare a 'continue' statement.

Member functions

`pss_continue (void)`: constructor

Syntax 168—C++: Procedural break/continue statement

22.7.11.3 Examples

```
// Sum all elements of 'a' that are even, starting from a[0], except those
// that are equal to 42. Stop summation if the value of an element is 0.

function int sum(int a[100]) {
   int res;

   res = 0;

   foreach (el : a) {
      if (el == 0)
         break;
      if (el == 42)
         continue;
      if ((el % 2) == 0) {
         res = res + el;
      }
   }

   return res;
}
```

Example 340—DSL: Procedural foreach statement with break/continue

```
  function<result<int>(arg<attr_vec<int>>)> sum { "sum",
    result<int>(), arg<attr_vec<int>>("a", 100) [&](arg<attr_vec<int>> a) {
       attr<int> res("res");
       attr<int> el("el");

       res = 0;

       foreach(el, a
          [&]() {
           if_then( cond(el == 0),
              [&]() { pss_break(); } );
           if_then( cond(el == 42),
              [&]() { pss_continue(); } );
           if_then( cond((el % 2) == 0),
              [&]() { res = res + el; } );
          }
       );

       pss_return {res};
    }
  };
```

Example 341—C++: Procedural foreach statement with break/continue

22.7.12 exec block

Example 342 shows how an **exec body** can be specified using procedural constructs in DSL. Example 343 shows the equivalent in PSS/C++.

```
  action A {
     rand bool flag;

     exec body {
        int var;

        if(flag) {
           var = 10;
        } else {
           var = 20;
        }
        // send_cmd is an imported function
        send_cmd(var);
     }
  }
```

Example 342—DSL: exec block using procedural control flow statements

```
class A : public action {
   // ...

   rand_attr<bool> flag;

   exec e { exec::body, [&]() {
         attr<int> var{"var"};

         if_then_else( flag,
            [&]() { var = 10; },
            [&]() { var = 20; }
         );
         send_cmd(var);
      }
   };
};
```

Example 343—C++: Generative exec block using procedural control flow statements

22.8 C++ in-line implementation for solve exec blocks

When C++-based PSS input is used, the overhead in user code (and possibly performance) of solve-time interaction with non-PSS behavior can be reduced. This is applicable in cases where the PSS/C++ user code can be invoked by the PSS implementation during the solve phase and computations can be performed natively in C++, not with PSS procedural code.

In-line *exec block*s (see Syntax 139) are simply predefined virtual member functions of the library classes (**action** and **structure**), the different flow/resource object classes (**pre_solve** and **post_solve**), and **component** (**init_down** and **init_up**). In these functions, arbitrary procedural C++ code can be used: statements, variables, and function calls, which are compiled, linked, and executed as regular C++. Using an in-line exec is similar in execution semantics to calling a foreign C/C++ function from the corresponding PSS-native **exec**.

In-line execs shall be declared in the context in which they are used with a class `exec`; if any PSS attribute is assigned in the **exec**'s context, it shall be declared through an exec constructor parameter.

NOTE—In-line solve execs are not supported in PSS/DSL.

Example 344 depicts an in-line **post_solve** exec. In it, a reference model for a decoder is used to compute attribute values. Notice that the functions that are called here are not PSS imported functions but rather natively declared in C++.

```
// C++ reference model functions
int predict_mode(int mode, int size){ return 0;}
int predict_size(int mode, int size){ return 0;}

class mem_buf : public buffer { ...
  attr<int> mode {"mode"};
  attr<int> size {"size"};
};

class decode_mem : public action { ...
  input<mem_buf> in {"in"};
  output<mem_buf> out {"out"};

  exec e { exec::post_solve, { out->mode, out->size } };
  void post_solve() {
    out->mode.val() = predict_mode(in->mode.val(), in->size.val());
    out->size.val() = predict_size(in->mode.val(), in->size.val());
  }
};
```

Example 344—C++: in-line exec

22.9 C++ generative implementation for target exec blocks

When C++-based PSS input is used, the generative mode for target exec blocks can be used. Computation can be performed in native C++ for purpose of constructing the description of procedural execs or target-template-code execs. This is applicable in cases where the C++ user code can be invoked by the PSS implementation during the solve or execution phase. Specifying an `exec` in generative mode has the same semantics as the corresponding `exec` in declarative code. However, the behavior exercised by the PSS implementation is the result of the computation performed in the context of the user PSS/C++ executable.

Specifying execs in generative mode is done by passing a function object as a lambda expression to the exec constructor—a generative function. The function gets called by the PSS implementation after solving the context entity, either before or during test execution, which may vary between deployment flows. For example, in pre-generation flow generative functions are called as part of the solving phase. However, in on-line-generation flow, the generative function for **exec body** may be called at runtime, as the actual invocation of the **action**'s **exec body**, and, in turn, invoke the corresponding functions directly as it executes. Native C++ functions can be called from generative functions, but should not have side-effects since the time of their call may vary.

A lambda capture list can be used to make scope variables available to the generative function. Typically simple by-reference capture (**'[&]'**) should be used to access PSS fields of the context entity. However, other forms of capture can also occur.

NOTE—Generative target execs are not supported in PSS/DSL.

22.9.1 Generative procedural execs

Target procedural execs (**body**, **run_start**, and **run_end**) can be specified in generative mode (see Syntax 169). However, **run_start** and **run_end** are restricted to pre-generation flow (see Table 24).

22.9.1.1 C++ syntax

pss::exec

Defined in **`pss/exec.h`** (see C.25).

```
   class exec;
```

Declare a generative procedural exec.

Member functions

```
   exec(ExecKind kind, std::function<void()> genfunc)
```

Syntax 169—C++: generative procedural exec definitions

The behavioral description of procedural execs is a sequence of function calls and assignment statements. In generative specification mode, the same C++ syntax is used as in the declarative mode, through variable references, **`operator=`**, and **`function::operator()`**. PSS implementation may define these operators differently for different deployment flows.

a) *Pre-generation flow*—The generative function call is earlier than the runtime invocation of the respective exec block. As the generative function runs, the PSS implementation must record function calls and assignments to attributes, along with the right-value and left-value expressions, to be evaluated at the right time on the target platform.

b) *Online-generation flow*—The generative function call may coincide with the runtime invocation of the respective exec block. In this case, the PSS implementation shall directly evaluate the right-value and left-value expressions, and perform any PSS function calls and PSS attribute assignments.

22.9.1.2 Examples

Example 345 depicts a generative procedural exec defining an **action**'s body. In this *exec block*, action attributes appear in the right-value and left-value expressions. Also, a function call occurs in the context of a native C++ loop, thereby generating a sequence of the respective calls as the loop unrolls.

```
namespace mem_ops_pkg {
   function<result<bit>(in_arg<int>)> alloc_mem {"alloc_mem",
      result<bit>(width(63,0)),
      in_arg<int>("size")
   };
  function<result<void>(in_arg<bit>, in_arg<bit>)> write_word {"write_word",
      in_arg<bit>("addr", width(63,0)),
      in_arg<bit>("data", width(31,0))
   };
};

class my_comp : public component { ...
  class write_multi_words : public action { ...
    rand_attr<int> num_of_words {"num_of_words", range(2,8)};
    attr<bit> base_addr {"base_addr", width(63,0)};

    // exec specification in generative mode
    exec body {exec::body, [&](){ // capturing action variables
        base_addr = mem_ops_pkg::alloc_mem(num_of_words*4);
        // in pre-gen unroll the loop,
        // evaluating num_of_words on solve platfrom
        for (int i=0; i < num_of_words.val(); i++) {
          mem_ops_pkg::write_word(base_addr + i*4, 0xA);
        }
      }
    };
  };
  type_decl<write_multi_words> write_multi_words_decl;
};
```

Example 345—C++: generative procedural exec

Example 346 illustrates the possible code generated for `write_multi_words()`.

```
void main(void) {
  ...
  uint64_t pstool_addr;
  pstool_addr = target_alloc_mem(16);
  *((uint32_t*)pstool_addr + 0) = 0xA;
  *((uint32_t*)pstool_addr + 4) = 0xA;
  *((uint32_t*)pstool_addr + 8) = 0xA;
  *((uint32_t*)pstool_addr + 12) = 0xA;
  ...
}
```

Example 346—C++: Possible code generated for write_multi_words()

22.9.2 Generative target-template execs

Target-template-code execs (**body**, **run_start**, **run_end**, **header**, **declaration**, and **file**) can be specified in generative mode (see Syntax 170); however, their use is restricted to pre-generation flow (see Table 24).

22.9.2.1 C++ syntax

pss::exec

Defined in **`pss/exec.h`** (see C.25).

```
class exec;
```

Declare a generative target-template exec.

Member functions

```
exec(ExecKind kind, std::string&& language_or_file,
   std::function<void(std::ostream&)> genfunc)
```
: generative target-template

Syntax 170—C++: generative target-template exec definitions

The behavioral description with target-template-code execs is given as a string literal to be inserted verbatim in the generated target language, with expression value substitution (see 22.5). In generative specification mode, a string representation with the same semantics is computed using a generative function. The generative function takes **`std::ostream`** as a parameter and should insert the string representation to it. As with the declarative mode, the target *language*_identifier must be provided.

22.9.2.2 Examples

Example 347 depicts a generative target-template-code exec defining an **`action`**'s body. In this function, strings inserted to the C++ **`ostream`** object are treated as C code-templates. Notice that a code line is inserted inside a native C++ loop here, thereby generating a sequence of the respective target code lines.

```
class my_comp : public component { ...
  class write_multi_words : public action { ...
    rand_attr<int> num_of_words { "num_of_words", range(2,8) };
    attr<int> num_of_bytes {"num_of_bytes"};

    void post_solve () {
      num_of_bytes.val() = num_of_words.val()*4;
    }
    // exec specification in target code generative mode
    exec body { exec::body, "C",
      [&](std::ostream& code){
        code<< " uint64_t pstool_addr;\n";
        code<< " pstool_addr = target_alloc_mem({{num_of_bytes}});\n";

        // unroll the loop,
        for (int i=0; i < num_of_words.val(); i++) {
          code<< "*((uint32_t*)pstool_addr + " << i*4 << ") = 0xA;\n";
        }
      }
    };
  };
  type_decl<write_multi_words> write_multi_words_decl;
};
```

Example 347—C++: generative target-template exec

The possible code generated for `write_multi_words()` is shown in Example 346.

22.10 Comparison between mapping mechanisms

Previous sections describe three mechanisms for mapping PSS entities to external (non-PSS) definitions: functions that directly map to foreign API (see 22.4), functions that map to foreign language procedural code using target code templates (see 22.6), and *exec block*s where arbitrary target code templates are in-lined (see 22.5). These mechanisms differ in certain respects and are applicable in different flows and situations. This section summarizes their differences.

PSS tests may need to be realized in different ways in different flows:

— by directly exercising separately-existing environment APIs via procedural linking/binding;

— by generating code once for a given model, corresponding to entity types, and using it to execute scenarios; or

— by generating dedicated target code for a given scenario instance.

Table 23 shows how these relate to the mapping constructs.

Table 23—Flows supported for mapping mechanisms

	No target code generation	**Per-model target code generation**	**Per-test target code generation**	**Non-procedural binding**
Direct-mapped functions	X	X	X	
Target-template functions		X	X	
Target-template exec-blocks			X	X

Not all mapping forms can be used for every **exec** kind. Solving/generation-related code must have direct procedural binding since it is executed prior to possible code generation. *exec block*s that expand declarations and auxiliary files shall be specified as target-templates since they expand non-procedural code. The **run_start** *exec block* is procedural in nature, but involves up-front commitment to the behavior that is expected to run.

Table 24 summarizes these rules.

The possible use of **action** and **struct** attributes differs between mapping constructs. Explicitly declared prototypes of **function**s enable the type-aware exchange of values of all data types. On the other hand, free parameterization of uninterpreted target code provides a way to use attribute values as target-language meta-level parameters, such as types, variables, functions, and even preprocessor constants.

Table 25 summarizes the parameter passing rules for the different constructs.

Table 24—exec block kinds supported for mapping mechanisms

	Action runtime behavior exec blocks: body	**Non-procedural exec blocks: header, declaration, file**	**Global test exec blocks: run_start, run_end**	**Solve exec blocks: init_down, init_up, pre_solve, post_solve**
Direct-mapped functions	X		X (only in pre-generation)	X
Target-template functions	X		X (only in pre-generation)	
Target-template exec-blocks	X	X	X	

Table 25—Data passing supported for mapping mechanisms

	Back assignment to PSS attributes	**Passing user-defined and aggregate data types**	**Using PSS attributes in non-expression positions**
Direct-mapped functions	X	X	
Target-template functions		X	
Target-template exec-blocks			X

22.11 Exported actions

Imported functions and classes specify functions and classes external to the PSS description that can be called from the PSS description. Exported actions specify actions that can be called from a foreign language. See also Syntax 171 and Syntax 172.

22.11.1 DSL syntax

```
export_action ::= export [ platform_qualifier ] action_type_identifier
        function_parameter_list_prototype ;
```

Syntax 171—DSL: Export action declaration

The **export** statement for an **action** specifies the action to export and the parameters of the action to make available to the foreign language, where the parameters of the exported action are associated by name with the action being exported. The **export** statement also optionally specifies in which phases of test generation and execution the exported action will be available.

The following also apply:

a) As with imported functions (see 22.2.1), the exported action is assumed to always be available if the function availability is not specified.

b) Each call into an **export** action infers an independent tree of actions, components, and resources.

c) Constraints and resource allocation are considered within the inferred action tree and are not considered across imported function / exported action call chains.

22.11.2 C++ syntax

The corresponding C++ syntax for Syntax 171 is shown in Syntax 172.

pss::export_action

Defined in **`pss/export_action.h`** (see C.26).

```
enum kind {solve, target};
template <class T=int> class export_action;
```

Declare an export action.

Member functions

```
export_action (const std::vector<detail::ExportActionParam> &params) :
```
constructor
```
export_action (kind, const std::vector<detail::ExportActionParam> &params) :
```
constructor

Syntax 172—C++: Export action declaration

22.11.3 Examples

Example 348 and Example 349 show an exported action. In this case, the action `comp::A1` is exported. The foreign language invocation of the exported action supplies the value for the `mode` field of action `A1`. The PSS processing tool is responsible for selecting a value for the `val` field. Note that `comp::A1` is exported to the target, indicating the target code can invoke it.

```
component comp {

  action A1 {
    rand bit          mode;
    rand bit[31:0]    val;

    constraint {
      if (mode!=0) {
        val in [0..10];
      } else {
        val in [10..100];
      }
    }
  }

}

package pkg {
  // Export A1, providing a mapping to field 'mode'
  export target comp::A1(bit mode);
}
```

Example 348—DSL: Export action

```
class comp : public component { ...
  class A1 : public action { ...
    rand_attr<bit> mode {"mode"};
    rand_attr<bit> val { "val", width(32) };

    constraint c {
      if_then_else { cond(mode!=0),
        in (val, range(0,10)),
        in (val, range(10,100))
      }
    };
  };
  type_decl<A1> A1_decl;
};

namespace pkg {
  // Export A1, providing a mapping to field ’mode’
  export_action<comp::A1> comp_A1 {export_action<>::target,
    { export_action<>::in<bit>("mode") }
  };
};
```

Example 349—C++: Export action

22.11.4 Export action foreign language binding

An exported action is exposed as a function in the target foreign language (see Example 350). The component namespace is reflected using a language-specific mechanism: C++ namespaces, SystemVerilog packages. Parameters to the exported action are implemented as parameters to the foreign language function.

```
namespace comp {
   void A1(unsigned char mode);
}
```

Example 350—Export action foreign language implementation

NOTE—Foreign language binding is the same for DSL and C++.

23. Conditional code processing

It is often useful to conditionally process portions of a PSS model based on some configuration parameters. This clause details a **compile if** construct that can be evaluated as part of the elaboration process.

NOTE—PSS/C++ does not provide features directly corresponding to PSS/DSL conditional code processing. Native C++ features (e.g., the pre-processor) provide similar features.

23.1 Overview

This section covers general considerations for using compile statements.

23.1.1 Statically-evaluated statements

A *statically-evaluated statement* marks content that may or may not be elaborated. The description within a statically-evaluated statement shall be syntactically correct, but need not be semantically correct when the static scope is disabled for evaluation.

A statically-evaluated statement may specify a block of statements. However, this does not introduce a new scope in the resulting description.

23.1.2 Elaboration procedure

Compile statements are processed top-to-bottom within a given source unit. The following steps are performed in processing source code in the presence of conditional compilation directives:

a) Syntactic code analysis is performed.
b) Compile-time expressions are evaluated in order within the following contexts:
   1) **static const** initializers
   2) **compile if** conditions (see 23.2)

   These expressions are evaluated based on types and static constants declared:
   1) Unconditionally, or in an enabled **compile if** branch, within a previously-processed source unit
   2) Unconditionally, or in an enabled **compile if** branch, previously processed within the current source unit
c) Globally-visible content and the content within enabled **compile if** branches is elaborated.

23.1.3 Compile-time expressions

The value of any **compile if** expressions must be determinable at compile time. Because **compile if** statements are evaluated early in PSS source processing, only types and constants declared in **package** scopes may be referenced. Types and constants declared in type scopes (e.g., an **action** type declared within a **component** type) may not be referenced.

The example below highlights the reference rules for conditional compilation directives:

a) Conditional compilation directives are evaluated based on previously defined elements.
   1) Consequently, the first directive (`compile has(s)`) evaluates true because `p1::s` is visible at this point in the evaluation.
   2) The second directive (`compile has(t)`) also evaluates true because `p2::t` has been previously declared in the source unit.
b) Conditional compilation directives may not reference inner members of types. Consequently, attempting to reference `t::A` is an error, since `t` is a type and `A` is an inner member of type `t`.

```
package p1 {
   struct s {
      static const int A = 3;
   };
};

package p2 {
   import p1::*;

   // derived from p2::s defined later in this file
   struct t : s { };

   // evaluates to true because such a type has been previously defined,
   // namely p1::s
   compile if (compile has (s)) { … }

   // evaluates to true because such a type has been previously defined,
   // namely p2::t (even though its super-type is not yet known)
   compile if (compile has (t)) { … }

   // Illegal! Cannot reference a member of a struct in compile-if context
   compile if (t::A == 2) { … }

   struct s {};
}
```

Example 351—DSL: Conditional compilation evaluation

23.2 compile if

23.2.1 Scope

compile if statements may appear in the following scopes:

- Global/package
- Action
- Component
- Struct

23.2.2 DSL syntax

Syntax 173 shows the grammar for a **compile if** statement.

```
package_body_compile_if ::= compile if ( constant_expression )
    package_body_compile_if_item [ else package_body_compile_if_item ]
package_body_compile_if_item ::=
    package_body_item
  | { { package_body_item } }
action_body_compile_if ::= compile if ( constant_expression )
    action_body_compile_if_item [ else action_body_compile_if_item ]
action_body_compile_if_item ::=
    action_body_item
  | { { action_body_item } }
component_body_compile_if ::= compile if ( constant_expression )
    component_body_compile_if_item [ else component_body_compile_if_item ]
component_body_compile_if_item ::=
    component_body_item
  | { { component_body_item } }
struct_body_compile_if ::= compile if ( constant_expression )
    struct_body_compile_if_item [ else struct_body_compile_if_item ]
struct_body_compile_if_item ::=
    struct_body_item
  | { { struct_body_item } }
```

Syntax 173—DSL: compile if declaration

23.2.3 Examples

Example 352 shows an example of conditional processing if PSS were to use C pre-processor directives. If the `PROTOCOL_VER_1_2` directive is defined, then action `new_flow` is evaluated. Otherwise, action `old_flow` is processed.

NOTE—Example 352 is only shown here to illustrate the functionality of C pre-processor directives in a familiar format. It is not part of PSS.

```
#ifdef PROTOCOL_VER_1_2
action new_flow {
  activity { ... }
}
#else
action old_flow {
  activity { ... }
}
#endif
```

Example 352—Conditional processing (C pre-processor)

Example 353 shows a DSL version of Example 352 using a **compile if** statement instead.

```
package config_pkg {
  const bool PROTOCOL_VER_1_2 = false;
}
compile if (config_pkg::PROTOCOL_VER_1_2) {
  action new_flow {
    activity { ... }
  }
} else {
  action old_flow {
    activity { ... }
  }
}
```

Example 353—DSL: Conditional processing (compile if)

When the *true* case is triggered, the code in Example 353 is equivalent to:

```
action new_flow {
    activity { ... }
  }
```

When the *false* case is triggered, the code in Example 353 is equivalent to:

```
action old_flow {
    activity { ... }
  }
```

23.3 compile has

compile has allows conditional elaboration to reason about the existence of types and constants. The **compile has** expression evaluates to *true* if a type or constant has been previously declared unconditionally or within an enabled conditional block (see 23.1.2); otherwise, it evaluates to *false*.

23.3.1 DSL syntax

Syntax 174 shows the grammar for a **compile has** expression.

compile_has_expr ::= **compile has (** static_ref_path **)**

static_ref_path ::= [**::**] { type_identifier_elem **::** } member_path_elem

Syntax 174—DSL: compile has expression

23.3.2 Examples

Example 354 checks whether the `config_pkg::PROTOCOL_VER_1_2` field exists and tests its value if it does. In this example, `old_flow` will be used because `config_pkg::PROTOCOL_VER_1_2` does not exist.

```
package config_pkg {
}

compile if (compile has(config_pkg::PROTOCOL_VER_1_2) &&
            config_pkg::PROTOCOL_VER_1_2) {
  action new_flow {
    activity { ... }
  }
} else {
  action old_flow {
    activity { ... }
  }
}
```

Example 354—DSL: compile has

Example 355 is composed of a single source unit.

— The first top-level **compile if** block checks for the existence of `X`. This evaluates to *false*, since `X` is only subsequently declared within the source unit.

— The second top-level **compile if** block checks for the non-existence of `Y`. This evaluates to *true*, since `Y` was not previously declared (the first **compile if** block was not expanded). As a consequence, `Y` is declared with a value of `0`.

```
compile if (compile has(X)) {
    const int Y = 2;
    compile if (compile has(Y)) {
        const int Z;
    }
}

const int X = 1;

compile if (!(compile has(Y))) {
    const int Y=0;
} else {
    compile if (compile has(Z)) {
        const int A;
    }
}
```

Example 355—DSL: Nested conditions

23.4 compile assert

compile assert assists in flagging errors when the source is incorrectly configured. This construct is evaluated during elaboration. A tool shall report a failure if *constant_expression* does not evaluate to *true*, and report the user-provided message, if specified.

23.4.1 DSL syntax

Syntax 175 shows the grammar for a **compile assert** statement.

```
compile_assert_stmt ::= compile assert ( constant_expression [ , string_literal ] ) ;
```

Syntax 175—DSL: compile assert statement

23.4.2 Examples

Example 356 shows a **compile assert** example.

```
compile if (compile has(FIELD2)) {
  static const FIELD1 = 1;
}

compile if (compile has(FIELD1)) {
  static const FIELD2 = 2;
}
compile assert(compile has(FIELD1), "FIELD1 not found");
```

Example 356—DSL: compile assert

24. PSS core library

The PSS *core library* provides standard portable functionality and utilities for common PSS applications. It defines a set of **component** types, data types, **functions**, and attributes. The interface of the core library is specified in PSS-language terms, and its use conforms to the rules of the language. However, the full semantics of its entities involve reference to type information, solving, scheduling, and runtime services. Hence, the implementation of the core library depends on inner workings of PSS processing tools and is expected to be coupled with them.

The core library currently covers functionality in the following areas:

- Representation of execution contexts in the target environment
- Assignments of actions and flow/resource objects to execution contexts
- Representation of target address spaces
- Allocation from and management of target address spaces
- Access to target address spaces
- Representation of and access to registers

This section covers the interface, semantics, and intended use of core library entities in the areas listed above. Note that it defines a library interface, not new language constructs. The code for the built-in library package contents appears in Annex D.

NOTE—PSS/C++ does not provide an interface to the core library features—storage management, register models, and executors.

24.1 Executors

A PSS generated test calls foreign functions available in the target environment, executes target-language code blocks, and performs target operations provided in the core-library. It does so in accordance with the user-defined realization of actions and of flow/resource objects specified in the form of target **exec** blocks—**body**, **run_start**, and **run_end**—and functions called from them. Foreign function calls, target-language code blocks, and built-in target operations, all need to be performed under a certain agent of execution available to the test in the runtime environment, or in short, an *executor*.

An *executor* is an abstract notion that may correspond to different kinds of entities in different environments. For example:

- An embedded processor core or HW thread in a bare-metal environment that executes code generated by the PSS tool
- A BFM instantiated as a master on an interconnect of the DUT that exposes transactional APIs to the PSS tool
- A transactor, or testbench agent, connected to an I/O interface of the system that exposes transactional APIs, or higher-level stimulus sequences, to the PSS tool

The PSS core library provides means to represent executors in the PSS description and to assign scenario entities to them. Executors are characterized by user-defined properties called *traits*, which serve to control the assignment of actions/objects to them. For example, the cluster of a CPU core could be represented as a trait attribute. Related executors are grouped together so that scenario entities can be assigned to a random instance out of a group. The selection of executors satisfies constraints on their trait attributes, if any are specified.

In addition, executors can be used to customize the implementation of target functions for specific environments. Actions assigned to different executors can thereby employ different mappings of portable operations.

The PSS built-in package **executor_pkg** defines types and functions related to the management of executors. In subsequent sections, except Syntax 176, the enclosing **executor_pkg** is omitted for brevity. Examples may also omit import of **executor_pkg**.

24.1.1 Executor representation

An executor is an execution agent or context available to the test in the runtime environment. Executors are represented using a core-library component type instantiated in the PSS description. Actions and flow/ resource objects may subsequently be assigned to these executors. This assignment is controlled through an *executor claim struct* (see 24.1.2).

Representing executors in a PSS description is optional. In the absence of executor instances, PSS tools are free to determine the execution context of entities based on other considerations, such as global defaults or policies.

24.1.1.1 Executor component type

An executor is represented using the template component **executor_c**, or a subtype of it. The template parameter is used to tag the executor and possibly to provide additional selection attributes. Template **executor_c** is derived from **executor_base_c**.

```
package executor_pkg {

   struct executor_trait_s {};

   struct empty_executor_trait_s : executor_trait_s {};

   component executor_base_c {};

   component executor_c
                 <struct TRAIT : executor_trait_s = empty_executor_trait_s>
                 : executor_base_c {
      TRAIT trait;
   };
...
```

Syntax 176—DSL: Executor component

An executor component is strictly a test-realization artifact. It shall be an error to declare in its scope scenario model elements, namely: **action** types, **pool** instances, and pool binding directives.

24.1.1.2 Executor group component type

Component **executor_group_c** is used to group one or more executors that serve similar purposes. Actions and flow/resource objects that *claim* an executor are assigned to an executor selected out of one specific group (see more on matching rules in 24.1.2.2).

```
component executor_group_c
             <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
   function void add_executor(ref executor_c<TRAIT> exe);
};
```

Syntax 177—DSL: Executor group component

An executor group component is strictly a test-realization artifact. It shall be an error to declare in its scope scenario model elements, namely: **action** types, **pool** instances, and pool binding directives.

24.1.1.2.1 add_executor function

Instance function **`add_executor`** (see Syntax 177) of **`executor_group_c`** is used to populate the group with executor instances. Executors added to a group must all match with the group's trait struct type. The **`add_executor`** function may only be called in **exec init_down** and **init_up** blocks.

The following also apply:

a) Any executor can be added to a given group, regardless of where it is instantiated in the component instance tree. This includes executors instantiated above the group, below it, or in a different sub-tree.

b) An executor instance may not be added more than once to the same group.

c) An executor instance may be added to more than one group.

d) An executor does not have to be added to any group. An executor that is not part of any group would be inactive—no **exec** blocks would ever be assigned to it.

Example 357 demonstrates how executors are defined, instantiated, and added to an executor group. The executor group `my_hybrid_group_c` is populated with two different executor types. These two types may vary in properties, but are both derived from the instantiation of template **`executor_c`** with the struct type `master_trait_s`. The executors in this group are treated symmetrically when assigning actions to them.

```
struct master_trait_s : executor_trait_s {};

component my_core_executor_c : executor_c<master_trait_s> { ... };

component my_bus_vip_executor_c : executor_c<master_trait_s> { ... };

component my_hybrid_group_c : executor_group_c<master_trait_s> {
   my_core_executor_c cores[4];
   my_bus_vip_executor_c bfms[2];

   exec init_down {
      foreach (c: cores) {
         add_executor(c);
      }
      foreach (b: bfms) {
         add_executor(b);
      }
   }
};
```

Example 357—DSL: Defining an executor group

24.1.2 Executor assignment

An action or a flow/resource object can declare its claim for an executor by instantiating a *claim struct*. Each claim instance is statically matched to an executor group that is nearest in the component instance tree and parameterized by the same trait struct type. The entity is assigned to an executor out of the matching group, which satisfies the trait constraints.

It is not required that scenario entities be explicitly assigned to an executor even if they contain target **exec** blocks. In the absence of explicit assignments, PSS tools are free to determine the execution context of entities based on other considerations, such as global defaults or policies.

Executors do not generally limit concurrency of PSS behaviors in a test scenario. In cases where concurrently scheduled actions are assigned to the same underlying executor, the PSS tool is responsible for employing the means to enable concurrent execution, such as preemptive or cooperative multitasking.

24.1.2.1 Executor claim struct type

An action or a flow/resource object can control its assignment to an executor by declaring an *executor claim* —an attribute of template struct type **`executor_claim_s`**. An executor claim can be a direct field of the entity, a field of any of its nested structs, or in the case of flow/resource objects, the super-type from which the object is derived. In all these cases, the assignment to an executor applies in the same way.

An action or a flow/resource object may be assigned to no more than one executor. Therefore, there can only be one executor claim struct anywhere under a given action or object. Multiple executor claim structs within the same action or object shall be flagged as an error. Note that the assignment of executors per an executor claim is not exclusive, and is generally unrelated to the relative scheduling of actions.

```
struct executor_claim_s
             <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
   rand TRAIT trait;
};
```

Syntax 178—DSL: Executor claim struct

Example 358 demonstrates the use of the **`executor_claim_s`** struct. In this case, **action** `A` declares an executor claim. `A`'s executor claim is matched with executor group `eg` that is instantiated directly under its context **component** `C`, as both are parameterized with the same (default) trait type. Consequently, **action** `A` is necessarily assigned to the executor `e` instantiated under its context component. **Component** `C` is instantiated twice under `pss_top`. Under the entry **action** `test`, **action** `A` is invoked three times. The generated test will call the function `do_something()` twice under the execution context associated with executor `c1.e`, and subsequently once under the execution context associated with executor `c2.e`.

```
component C {
   executor_c<> e;
   executor_group_c<> eg;
   exec init_down {
      eg.add_executor(e);
   }

   action A {
      rand executor_claim_s<> ec;
      exec body C = """
         do_something();
      """;
   };
};

component pss_top {
   C c1,c2;

   action test {
      C::A a1, a2, a3;
      activity {
         parallel {
            a1 with { comp == this.comp.c1; };
            a2 with { comp == this.comp.c1; };
            a3 with { comp == this.comp.c2; };
         }
      }
   };
};
```

Example 358—DSL: Simple executor assignment

24.1.2.2 Rules for matching an executor claim with an executor group

An executor claim is matched with an executor group for the purpose of selecting an executor. The matching is based on the static structure of the model. A claim is resolved to an executorgroup that:

a) is parameterized by the same trait type as the claim;

b) is instantiated in a containing component of the declaring scenario entity (the context component hierarchy of an action or the container component of a flow/resource object pool);

c) and is nearest in the component hierarchy going up from the context component to the root component.

It shall be an error if no executor group matches a claim per the above rules. Similarly, it shall be an error if more than one executor group matches a claim.

Note that given the above rules, instantiating a group within a group would be pointless, as no executor claim could match the inner group.

24.1.2.3 Claim trait semantics

The trait type of an executor claim must be the same as that of the executor selected for the declaring entity. In addition, the trait attribute values of the executor claim instance must be equal to the values of the corresponding attributes of the executor trait. Hence, the selected executor shall satisfy the claim trait constraints.

Example 359 demonstrates the use of the executor trait struct for the selection of executors. In this example, executors in group `my_embedded_cores_group_c`, representing eight CPU cores, are classified into two clusters, each consisting of four cores. Action `my_ip_c::op` claims an executor. It constrains the selection of the executor, relating the executor cluster ID to other attributes. Action `ops_on_two_clusters` executes two `op` actions, one on each cluster. Note that the one assigned to cluster `0` will have its input buffer `mem_kind` not equal to `DDR`, due to the constraint in action `op`.

```
struct my_core_trait_s : executor_trait_s {
   rand int in [0..1] cluster_id;
};

component my_embedded_cores_group_c : executor_group_c<my_core_trait_s> {
   executor_c<my_core_trait_s> cores[8];
   exec init_down {
      foreach (c: cores[i]) {
         c.trait.cluster_id = i/4;
         add_executor(c);
      }
   }
};

component my_ip_c {
   action op {
      input data_buff in_buff;
      rand executor_claim_s<my_core_trait_s> core;
      constraint in_buff.mem_kind == DDR -> core.trait.cluster_id != 0;
   };
};

component pss_top {
   my_embedded_cores_group_c embedded_core_group;
   my_ip_c my_ip;

   action ops_on_two_clusters {
      activity {
         do my_ip_c::op with { core.trait.cluster_id == 0; };
         do my_ip_c::op with { core.trait.cluster_id == 1; };
      }
   };
};
```

Example 359—DSL: Definition and use of executor trait

24.1.2.4 Executor resources

In some cases, the assignment of certain actions to executors needs to be exclusive, ruling out the handling of concurrent actions by the same execution agent. Resource claims and resource pools express such rules at the scenario model level, guaranteeing that random schedules satisfy the resource consistency of executors. In these cases, the executor assigned to actions needs to be in strict correspondence with the resource instance claimed by them.

A resource object that is derived from template struct **`executor_claim_s`** is considered a claim not just for the purpose of its own executor assignment, but also for that of the actions that claim it as a resource in either **lock** or **share** mode. In other words, from the executor assignment point of view, a reference to a resource object derived from struct **`executor_claim_s`** functions like an executor claim of the action itself.

In Example 360, resource object `my_core_r` represents a processor core at the scenario model level. Action `my_ip_c::op1` needs to be assigned a core exclusively for its duration, and therefore **lock**s a resource instance. Action `my_ip_c::op2` does not require exclusive use of a core, and therefore claims a resource instance in **share** mode. Action **`test`** executes a random selection of `op1` and `op2`, which need to be scheduled consistently across the different cores.

```
struct my_core_trait_s : executor_trait_s {
   rand int in [0..7] core_id;
};

resource my_core_r : executor_claim_s<my_core_trait_s> {
   constraint trait.core_id == instance_id;
};

component my_cores_group_c : executor_group_c<my_core_trait_s> {
   executor_c<my_core_trait_s> cores[8];
   exec init_down {
      foreach (c: cores[i]) {
         c.trait.core_id = i;
         add_executor(c);
      }
   }
};

component my_ip_c {
   action op1 {
      lock my_core_r core;
      exec body {
         my_ip_blocking_op();
      }
   };

   action op2 {
      share my_core_r core;
      exec body {
         while (!my_ip_op2_done()) { yield(); }
      }
   };
};

component pss_top {
   my_cores_group_c core_group;
   pool [8] my_core_r core_pool;
   bind core_pool *;

   my_ip_c my_ip;

   action test {
      activity {
         schedule {
            replicate (10) {
               select {
                  do my_ip_c::op1;
                  do my_ip_c::op2;
               }
            }
         }
      }
   };
};
```

Example 360—DSL: Use of resource objects as executor claims

24.1.2.5 Executor query function

The function **`executor()`** returns a reference to the executor instance currently operative. When called during the evaluation of **exec** blocks of an **action** or flow/resource object or of any function invoked by them, it returns the executor instance assigned to that entity. The function **`executor()`** can be used, among other purposes, to delegate generic target functions to an executor-specific implementation.

```
    function ref executor_base_c executor();
```

Syntax 179—DSL: Executor query function

Note that the reference returned from **`executor()`** for actions assigned to different executors would be different, even if these actions are executing concurrently. The returned value shall be **null** if the evaluating entity is not assigned to any executor. Since assignment to executors is only resolved as part of the solve process, calling **`executor()`** in **pre_solve exec** blocks shall always return **null**.

In Example 361, a call to the global function `my_target_op()` is delegated to the instance function `my_target_op_impl()` of the currently operative executor, through a call to **`executor()`**. Function `my_target_op_impl()` is declared in **component `executor_base_c`** and implemented differently in two executor subtypes. Consequently, the call to `my_target_op()` in the **exec body** of **action** `call_op` will be implemented differently based on the executor assignment of `call_op`.

```
function void my_target_op(int param) {
   if (executor() != null ) {
      executor().my_target_op_impl(param);
   } else {
      // default implementation
   }
}

extend component executor_base_c {
   function void my_target_op_impl(int param);
};

component A_executor_c : executor_c<> {
   function void my_target_op_impl(int param) {
      // implementation for execution agent of type A
   }
};

component B_executor_c : executor_c<> {
   function void my_target_op_impl(int param) {
      // implementation for execution agent of type B
   }
};

component pss_top {
   executor_group_c<> exe_g;
   A_executor_c a_exe;
   B_executor_c b_exe;

   exec init_down {
      exe_g.add_executor(a_exe);
      exe_g.add_executor(b_exe);
   }

   action call_op {
      rand executor_claim_s<> my_exe;
      exec body {
         my_target_op(10);
      }
   };
};
```

Example 361—DSL: Function delegation to executor

24.2 Address spaces

The *address space* concept is introduced to model memory and other types of storage in a system. An address space is a space of storage atoms accessible using unique addresses. System memory, external storage, internal SRAM, routing tables, memory mapped I/O, etc., are entities that can be modeled with address spaces in PSS.

An address space is composed of *regions*. Regions are characterized by user-defined properties called *traits*. For example, a trait could be the type of system memory of an SoC, which could be DRAM or SRAM. *Address claims* can be made by scenario entities (actions/objects) on an address space with optional constraints on user-defined properties. An *address space handle* is an opaque representation of an address within an address space.

Standard operations are provided to read data from and write data to a byte-addressable address space. *Registers* and *register groups* are allocated within an address space and use address space regions and handles to read and write register values. Data layout for packed PSS structs is defined for byte-addressable address spaces.

The PSS built-in package **`addr_reg_pkg`** defines types and functions for registers, address spaces, address allocation and operations on address spaces. In subsequent sections, except Syntax 180, the enclosing **`addr_reg_pkg`** is omitted for brevity. Examples may also omit import of **`addr_reg_pkg`**.

24.2.1 Address space categories

24.2.1.1 Base address space type

An *address space* is a set of storage atoms accessible using unique addresses. Actions/objects may allocate one or more atoms for their exclusive use.

Address spaces are declared as **components**. **`addr_space_base_c`** is the base type for all other address space types. This component cannot be instantiated directly. The definition of **`addr_space_base_c`** is shown in Syntax 180.

```
package addr_reg_pkg {
   component addr_space_base_c {};
...
}
```

Syntax 180—DSL: Generic address space component

24.2.1.2 Contiguous address spaces

A *contiguous address space* is an address space whose addresses are non-negative integer values. and whose atoms are contiguously addressed. Multiple atoms can be allocated in one contiguous chunk.

Byte-addressable system memory and blocks of data on disk drive are examples of contiguous address spaces.

A contiguous address space is defined by the built-in library component **`contiguous_addr_space_c`** shown in Syntax 181 below. The meanings of the struct type **`addr_trait_s`** and the template parameter **`TRAIT`** are defined in 24.2.2. Address space regions are described in 24.2.3.

```
struct addr_trait_s {};

struct empty_addr_trait_s : addr_trait_s {};

typedef chandle addr_handle_t;

component contiguous_addr_space_c <struct TRAIT : addr_trait_s =
   empty_addr_trait_s> : addr_space_base_c
{
   function addr_handle_t add_region(addr_region_s <TRAIT> r);
   function addr_handle_t add_nonallocatable_region(addr_region_s <> r);

   bool byte_addressable = true;
};
```

Syntax 181—DSL: Contiguous address space component

A contiguous address space is created in a PSS model by creating an instance of **component** **`contiguous_addr_space_c`** in a top-level **component** or any other **component** instantiated under the top-level **component**.

24.2.1.2.1 add_region function

The **`add_region`** function of contiguous address space components is used to add allocatable address space regions to a contiguous address space. The function returns an address handle corresponding to the start of the region in the address space. Actions and objects can allocate space only from allocatable regions of an address space.

Address space regions are defined in 24.2.3. Address space regions are part of the static component hierarchy. The **`add_region`** function may only be called in **exec init_down** and **init_up** blocks. Address handles are defined in 24.4.3.

24.2.1.2.2 add_nonallocatable_region function

The **`add_nonallocatable_region`** function of contiguous address space components is used to add non-allocatable address space regions to a contiguous address space. The function returns an address handle corresponding to the start of the region in the address space.

The address space allocation algorithm shall not use non-allocatable regions for allocation.

Address space regions are defined in 24.2.3. Address space regions are part of the static component hierarchy. The **`add_nonallocatable_region`** function may only be called in **exec init_down** and **init_up** blocks. Address handles are defined in 24.4.3.

24.2.1.2.3 Example

Example 362 demonstrates instantiating an address space and adding regions to it (for the definition of **struct `addr_region_s`**, see 24.2.3.2).

```
component pss_top {

   import addr_reg_pkg::*;

   my_ip_c ip;

   contiguous_addr_space_c<> sys_mem;

   exec init_up {
          // Add regions to space here
          addr_region_s<> r1;
          r1.size = 0x40000000; // 1 GB
          (void)sys_mem.add_region(r1);

          addr_region_s<> mmio;
          mmio.size = 4096;
          (void)sys_mem.add_nonallocatable_region(mmio);
   }
}
```

Example 362—DSL: Contiguous address space in pss_top

24.2.1.3 Byte-addressable address spaces

A *byte-addressable* space is a contiguous address space whose storage atom is a byte and to/from which PSS data can be written/read using standard generic operations. The PSS core library standardizes generic APIs to write data to or read data from any address value as bytes. The read/write API and data layout of PSS data into a byte-addressable space are defined in 24.4.

By default, **component `contiguous_addr_space_c`** is a byte-addressable space unless the **`byte_addressable`** Boolean field is set to *false*.

24.2.1.4 Transparent address spaces

Transparent address spaces are used to enable transparent claims—constraining and otherwise operating on concrete address values on the solve platform. For more information on transparent address claims, see 24.3.3.

All regions of a transparent space provide a concrete start address and the size of the region. Only transparent regions (see 24.2.3.3) may be added to a transparent address space using function **`add_region()`**. Note however that transparent regions may be added to a non-transparent space.

Component `transparent_addr_space_c` is used to create a transparent address space (see Syntax 182). See Example 364.

```
component transparent_addr_space_c
                    <struct TRAIT: addr_trait_s = empty_addr_trait_s>
                    : contiguous_addr_space_c<TRAIT> {};
```

Syntax 182—DSL: Transparent address space component

24.2.1.5 Other address spaces

Other kinds of address spaces, with different assumptions on allocations and generic operations, are possible. These may be represented as derived types of the corresponding base space/region/claim types. An example could be a space representing a routing table in a network router. PSS does not attempt to standardize these.

24.2.2 Address space traits

An address space *trait* is a PSS **struct**. A trait **struct** describes properties of a contiguous address space and its regions. **`empty_addr_trait_s`** is defined as an empty trait struct that is used as the default trait type for address spaces, regions and claims.

All regions of an address space share a trait *type*. Every region has its specific trait *value*.

```
package ip_pkg {

  struct mem_trait_s : addr_trait_s {
    rand mem_kind_e         kind;
    rand cache_attr_e       ctype;
    rand int in [0..3]      sec_level;
    rand bool               mmio;
  };

};
```

Example 363—DSL: Example address trait type

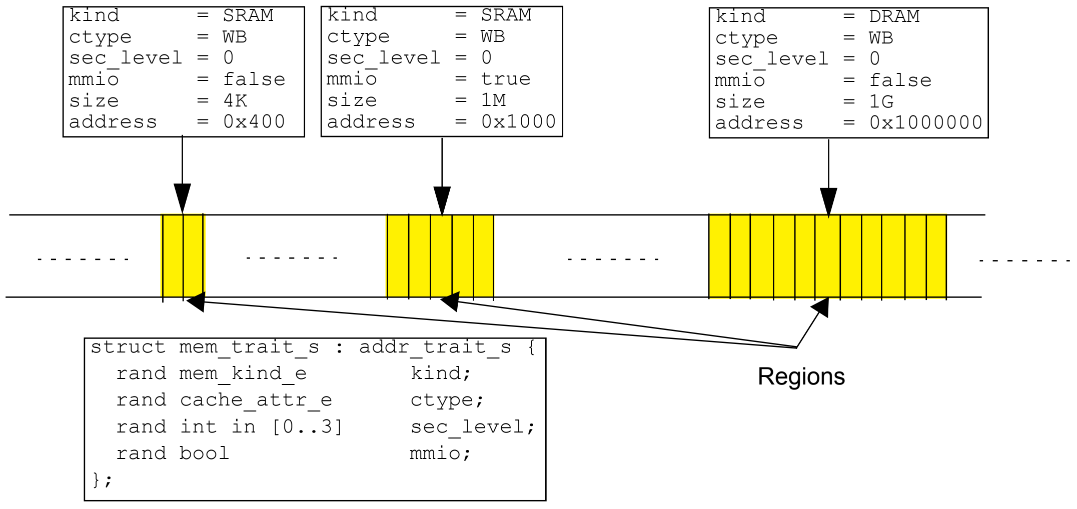

Figure 19—Address space regions with trait values

```
component pss_top {

   import addr_reg_pkg::*;
   import ip_pkg::*;

   // IP component
   my_ip_c ip;

   // mem_trait_s trait struct is used for sys_mem address space
   transparent_addr_space_c<mem_trait_s> sys_mem;

   exec init_up {
      // Add regions to space here. All regions added to sys_mem space
      // must have trait type mem_trait_s

      transparent_addr_region_s<mem_trait_s> sram_region;

      sram_region.trait.kind      = SRAM;
      sram_region.trait.ctype     = WB;
      sram_region.trait.sec_level = 0;
      sram_region.trait.mmio      = false;
      sram_region.size            = 4096;
      sram_region.addr            = 0x400;

      (void)sys_mem.add_region(sram_region);

      // add other regions
      // ...
    }
}
```

Example 364—DSL: Address space with trait

24.2.3 Address space regions

An address space may be composed of *regions*. Regions map to parts of an address space. A region may be characterized by values assigned to address space traits. Traits define properties of a region. Specific constraints are placed on *address claim* traits to allocate addresses from regions with desired characteristics. Regions with trait values that satisfy the claim's trait constraints are the candidate matching regions. An address claim may span more than one region that satisfies claim trait constraints.

Address space regions are part of the static component hierarchy. The **`add_region`** and **`add_nonallocatable_region`** functions (see 24.2.1.2.1 and 24.2.1.2.2) may only be called in **exec init_down** and **init_up** blocks.

24.2.3.1 Base region type

`addr_region_base_s` is the base type for all address space regions (see Syntax 183).

```
struct addr_region_base_s {
   bit[64] size;
};
```

Syntax 183—DSL: Base address region type

24.2.3.2 Contiguous address regions

The *addr_region_s* type represents a region in contiguous address space (see Syntax 184). The region type is fully characterized by the template **`TRAIT`** parameter value and the **`size`** attribute of the base region type.

```
struct addr_region_s <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                       : addr_region_base_s {
   TRAIT trait;
};
```

Syntax 184—DSL: Contiguous address space region type

The values of the trait struct attributes describes the contiguous address region. The PSS tool will match the trait attributes of regions to satisfy an address claim as described in 24.3. See an example of trait attribute setting in 24.3.7.

24.2.3.3 Transparent address regions

The **`transparent_addr_region_s`** type defines a *transparent* region over a contiguous address space. *Transparent* means that the region's start (lower) address is known to the PSS tool for solve-time resolution of a claim address within the address space.

The **`addr`** field of this region is assigned the start address of the region. The end address of the region is the calculated value of the expression: **`addr`** + **`size`** - 1.

See Example 364 where a transparent region is added to a transparent address space.

```
struct transparent_addr_region_s
                         <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                         : addr_region_s<TRAIT> {
   bit[64] addr;
};
```

Syntax 185—DSL: Transparent region type

24.3 Allocation within address spaces

The PSS input model can *allocate* storage atoms from an address space for the exclusive use of certain behaviors. For example, a DMA controller **action** might allocate a buffer in system memory for output data.

All address space allocations are done in the declarative domain of a PSS input model. An *address claim struct*, defined in the following sections, is used for allocation.

An instance of an address claim struct describes an address claim on an address space. A claim is *matched* to the address space nearest in the **component** instance tree, whose trait type matches the claim trait type (see 24.3.6). A claim is satisfied by allocation from a region (or regions) whose trait value satisfies the constraints on the claim trait (see 24.3.4).

A claim struct can be instantiated under an **action**, a flow object or resource object, or any of their nested structs. The declaration of a claim struct instance causes allocation to occur when the declaring object is instantiated or the **action** is traversed.

24.3.1 Base claim type

The **`addr_claim_base_s`** struct (see Syntax 186) is the base type for all address space claims.

```
struct addr_claim_base_s {
   rand bit[64] size;
   rand bool permanent;
   constraint default permanent == false;
};
```

Syntax 186—DSL: Base address space claim type

24.3.2 Contiguous claims

An address claim can be made on a contiguous address space by declaring a **struct** of type **`addr_claim_s`**. This claim is also known as an *opaque* claim. The absolute address of the claim is not assumed to be known at solve time.

This standard does not define any method by which the PSS tool might resolve address claims at solve time or might generate code for runtime allocation. One possible method could be PSS tool-specific APIs for solve-time and runtime allocation. The *address space handle* obtained from a claim shall fall within a region or regions whose traits satisfy the claim constraints.

An address claim in contiguous address space is always a contiguous chunk of addresses, potentially spanning multiple regions that are adjacent.

An address claim can be made on transparent (described below, in 24.3.3) or non-transparent address spaces.

```
struct addr_claim_s <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                    : addr_claim_base_s {
   rand TRAIT trait;
   rand bit[64] in [64'd2**0, 64'd2**1, 64'd2**2, 64'd2**3 , 64'd2**4 ,
      64'd2**5 , 64'd2**6 , 64'd2**7 , 64'd2**8 , 64'd2**9 , 64'd2**10,
      64'd2**11, 64'd2**12, 64'd2**13, 64'd2**14, 64'd2**15, 64'd2**16,
      64'd2**17, 64'd2**18, 64'd2**19, 64'd2**20, 64'd2**21, 64'd2**22,
      64'd2**23, 64'd2**24, 64'd2**25, 64'd2**26, 64'd2**27, 64'd2**28,
      64'd2**29, 64'd2**30, 64'd2**31, 64'd2**32, 64'd2**33, 64'd2**34,
      64'd2**35, 64'd2**36, 64'd2**37, 64'd2**38, 64'd2**39, 64'd2**40,
      64'd2**41, 64'd2**42, 64'd2**43, 64'd2**44, 64'd2**45, 64'd2**46,
      64'd2**47, 64'd2**48, 64'd2**49, 64'd2**50, 64'd2**51, 64'd2**52,
      64'd2**53, 64'd2**54, 64'd2**55, 64'd2**56, 64'd2**57, 64'd2**58,
      64'd2**59, 64'd2**60, 64'd2**61, 64'd2**62, 64'd2**63] alignment;
};
```

Syntax 187—DSL: Contiguous address space claim type

The **`alignment`** attribute specifies the address alignment of the resolved claim address.

24.3.3 Transparent claims

A claim of type **`transparent_addr_claim_s`** (see Syntax 188) is required to make a transparent claim on a transparent contiguous address space. A transparent claim is characterized by the absolute allocation address attribute (**`addr`**) of the claim. A transparent claim is associated with the nearest address space with the same trait type, in the same way that a non-transparent claim is. However, a transparent claim that is thereby associated with a non-transparent space shall be flagged as an error. The PSS tool has all the information at solve time about the transparent address space necessary to perform allocation within the limits of the address space. More details about allocation and claim lifetime can be found in the following section.

The **`addr`** field of this claim type can be used to put a constraint on an absolute address of a claim.

```
struct transparent_addr_claim_s
                        <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                        : addr_claim_s<TRAIT> {
       rand bit[64] addr;
};
```

Syntax 188—DSL: Transparent contiguous address space claim type

Example 365 illustrates how a transparent claim is used. A transparent address claim is used in **action** `my_op`. A constraint is placed on the absolute resolved address of the claim. This is possible only because of the transparent address space that contain transparent regions where the base address of the region is known at solve time.

```
component pss_top {

  transparent_addr_space_c<> mem;

  action my_op {
    rand transparent_addr_claim_s<> claim;
    constraint claim.size == 20;

    // Constraint on absolute address
    constraint claim.addr & 0x100 == 0x100;
  };

  exec init_up {
    transparent_addr_region_s<> region1, region2;
    region1.size = 50;
    region1.addr = 0x10000;
    (void)mem.add_region(region1);

    region2.size = 10;
    region2.addr = 0x20000;
    (void)mem.add_region(region2);
  }
};
```

Example 365—DSL: Transparent address claim

24.3.4 Claim trait semantics

Constraints placed on the trait attribute of a claim instance must be satisfied by the allocated addresses. Allocated addresses shall be in regions whose trait values satisfy claim trait constraints.

See an example in 24.3.7.

24.3.5 Allocation consistency

An address claim struct is resolved to represent the allocation of a set of storage atoms from the nearest storage space, for the exclusive use of actions that can access the claim attribute. In the case of a contiguous address space, the set is a contiguous segment, from the start address to the start address + **`size`** - 1. All addresses in the set are uniquely assigned to that specific instance of the address claim struct for the duration of its lifetime, as determined by the actions that can access it (see details below). Two instances of an address claim struct shall resolve to mutually exclusive sets of addresses if

- Both are taken from the same address space, and
- An action that has access to one may overlap in execution time with an action that has access to the other.

The number of storage atoms in an allocation is represented by the attribute **`size`**.

The start address is represented directly by the attribute **`addr`** in **`transparent_addr_claim_s<>`**, or otherwise obtained by calling the function **`addr_value()`** on the address space handle returned by **`make_handle_from_claim()`**.

Following is the definition of the lifetime of scenario entities:

Table 26—Scenario entity lifetimes

Entity	Lifetime
Atomic action	From the time of **exec body** entry (immediately before executing the first statement) to the time of the **exec body** exit (immediately after executing the last statement).
Compound action	From the start time of the first sub-action(s) to the end time of the last sub-action(s).
Flow object	From the start time of the action outputting it (for the initial state, the start time of the first action in the scenario) to the end time of the last action(s) inputting it (if any) or the end-time of the last action outputting it (if no action inputs it).
Resource object	From the start time of the first action(s) locking/sharing it to the end time of the last action(s) locking/sharing it.
Struct	Identical with the entity that instantiates it.

The lifetime of the allocation to which a claim struct resolves, and hence the exclusive use of the set of addresses, may be extended beyond the scenario entity in which the claim is instantiated in one of two ways:

- A handle that originates in a claim is assigned to entities that have no direct access to the claim in solve execs (for definition of address space handles, see 24.4.3). For example, if an action assigns a handle field (of type **`addr_handle_t`**) of its output **buffer** object with a handle it obtained from its own claim, the allocation lifetime is extended to the end of the last action that inputs that **buffer** object.
- The attribute **`permanent`** is constrained to *true*, in which case the lifetime of the claim is extended to the end of the test.

24.3.5.1 Example

The example below demonstrates how the scheduling of actions affects possible resolutions of address claims. In this model, **action** `my_op` claims 20 bytes from an address space, in which there is one region of size 50 bytes and another of size 10. In **action** `test1,` the three **action**s of type `my_op` are scheduled sequentially, as the iterations of a **repeat** statement. No execution of `my_op` overlaps in time with another, and therefore each one can be allocated any set of consecutive 20 bytes, irrespective of previous allocations. Note that all three allocations must come from the 50-byte region, as the 10-byte region cannot fit any of them. In `test2`, by contrast, the three **action**s of type `my_op` expanded from the **replicate** statement are scheduled in parallel. This means that they would overlap in execution time, and therefore need to be assigned mutually exclusive sets of addresses. However, such allocation is not possible out of the 50 bytes available in the bigger region. Here too, the smaller region cannot fit any of the three allocations. Nor can it fit part of an allocation, because it is not known to be strictly contiguous with the other region.

```
component pss_top {
  action my_op {
    rand addr_claim_s<> claim;
    constraint claim.size == 20;
  };

  contiguous_addr_space_c<> mem;

  exec init_up {
    addr_region_s<> region1, region2;
    region1.size = 50;
    (void)mem.add_region(region1);
    region2.size = 10;
    (void)mem.add_region(region2);
  }

  action test1 {
    activity {
      repeat (3) {
        do my_op; // OK - allocations can be recycled
      }
    }
  };

  action test2 {
    activity {
      parallel {
        replicate (3) {
          do my_op; // error - cannot satisfy concurrent claims
        }
      }
    }
  };
};
```

Example 366—DSL: Address space allocation example

24.3.6 Rules for matching a claim to an address space

a) A claim is associated with a unique address space based on the static structure of the model.

b) A claim is resolved to an address space that:

   1) matches the trait type of the claim

   2) is instantiated in a containing **component** of the current scenario entity (the context **component** hierarchy of an action or the container **component** of a flow/resource object pool)

   3) is nearest in the **component** hierarchy going up from the context **component** to the root **component**

c) It shall be an error if more than one address space in the same **component** hierarchy matches a claim.

24.3.7 Allocation example

In following example, `pss_top` has instances of the `sub_ip` and `great_ip` components. `sub_ip` is composed of the `good_ip` and `great_ip` components. `good_ip` and `great_ip` allocate space with trait `mem_trait_s`. Memory allocation in the `top_gr_ip` instance of `pss_top` will be matched to the `sys_mem` address space that is instantiated in `pss_top`. Memory claims in `gr_ip` and `go_ip` from `pss_top.sub_system` will be matched to the address space in `sub_ip`, as the `sub_ip` address_space will be the nearest space with a matching trait in the component tree.

Note how within the two address spaces, there are regions with the same base address. Claims from actions of the two instances of `great_ip` may be satisfied with overlapping addresses even if they are concurrent, since they are taken out of different address spaces.

```
import addr_reg_pkg::*;
import mem_pkg::*;

package mem_pkg {
   enum cache_attr_e {UC, WB, WT, WC, WP};

   struct mem_trait_s : addr_trait_s {
      rand cache_attr_e  ctype;
      rand int in [0..3] sec_level;
   }
};

component good_ip {
   action write_mem {
     // Allocate from nearest address space that matches TRAIT type and value
      rand transparent_addr_claim_s<mem_trait_s> mem_claim;

      constraint mem_claim.size == 128;
      constraint mem_claim.trait.ctype == UC;
   }
};

component great_ip {
   action write_mem {

     // Allocate from nearest address space that matches TRAIT type and value
      rand transparent_addr_claim_s<mem_trait_s> mem_claim;

      constraint mem_claim.size == 256;
      constraint mem_claim.trait.ctype == UC;
   }
};

component sub_ip {
   // Subsystem has its own address space
   transparent_addr_space_c<mem_trait_s> mem;

   good_ip  go_ip;
   great_ip gr_ip;
};
```

Example 367—DSL: Address space allocation example

```
component pss_top {
   sub_ip sub_system;
   great_ip top_gr_ip;

   transparent_addr_space_c<mem_trait_s> sys_mem;

   exec init_up {
      transparent_addr_region_s<mem_trait_s> region;

      region.size            = 1024;
      region.addr            = 0x8000;
      region.trait.ctype     = UC;
      region.trait.sec_level = 0;

      transparent_addr_region_s<mem_trait_s> great_region;

      great_region.size            = 1024;
      great_region.addr            = 0x8000;
      great_region.trait.ctype     = UC;
      great_region.trait.sec_level = 2;

      (void)sys_mem.add_region(region);

      (void)sub_system.mem.add_region(great_region);
   };
};
```

Example 367—DSL: Address space allocation example (cont.)

24.4 Data layout and access operations

24.4.1 Data layout

Many PSS use cases require writing structured data from the PSS model to byte-addressable space in a well-defined layout. In PSS, structured data is represented with a **struct**. For example, a DMA engine might expect DMA descriptors that encapsulate DMA operation to be in memory in a known layout. *Packed structs* may be beneficial to represent bit fields of hardware registers.

The built-in PSS library **struct `packed_s`** is used as a base **struct** to denote that a PSS **struct** is *packed*.

Any struct derived from built-in struct **`packed_s`** directly or indirectly is considered packed by the PSS tool. Packed structs are only allowed to have fields of numeric types, packed struct types, or arrays thereof. Following are the declarations of the endianness **enum** and packed **struct** in **`addr_reg_pkg`**:

```
enum endianness_e {LITTLE_ENDIAN, BIG_ENDIAN};

struct packed_s <endianness_e e = LITTLE_ENDIAN> {};
```

Syntax 189—DSL: packed_s base struct

24.4.1.1 Packing rule

PSS uses the de facto packing algorithm from the GNU C/C++ compiler. The ordering of fields of structs follows the rules of the C language. This means that fields declared first would go in lower addresses. The layout of fields in a packed struct is defined by the endianness template parameter of the packed struct. Bit fields in PSS structs can be of any size.

For the packing algorithm, a register of size N bytes is used, where N*8 is greater than or equal to the number of bits in the packed struct.

For big-endian mode, fields are packed into registers from the most significant bit (MSB) to the least significant bit (LSB) in the order in which they are defined. Fields are packed in memory from the most significant byte (MSbyte) to the least significant byte (LSbyte) of the packed register. If the total size of the packed struct is not an integer multiple of bytes, don't-care bits are added at the LSB side of the packed register.

For little-endian mode, fields are packed into registers from the LSB to the MSB in the order in which they are defined and packed in memory from the LSbyte to the MSbyte of the packed register. If the total size of the packed struct is not an integer multiple of bytes, don't-care bits are added at the MSB side of the packed register.

24.4.1.2 Little-endian packing example

A packed struct is shown in Example 368. This struct has 30 bits. A register for packing this struct would have 4 bytes.

```
struct my_packed_struct : packed_s<LITTLE_ENDIAN> {
   bit[6] A;
   bit[2] B;
   bit[9] C;
   bit[7] D;
   bit[6] E;
}
```

Example 368—DSL: Packed PSS little-endian struct

Register packing will start from field `A`. The least significant bit of `A` would go in the least significant bit of the register, as shown in Figure 20. Field `B` would go after field `A`. The least significant bit of `B` would go in the lowest bit after `A` in the packed register, and so on. The layout of the packed struct in byte-addressable space is shown in Figure 21. (`X` means “don’t-care bit” in Figure 20 and Figure 21.)

```
MSB                                                            LSB
 X X E E E E E E D D D D D D D C C C C C C C C C B B A A A A A A
 X X 5 4 3 2 1 0 6 5 4 3 2 1 0 8 7 6 5 4 3 2 1 0 1 0 5 4 3 2 1 0
```

Figure 20—Little-endian struct packing in register

```
     byte 0           byte 1           byte 2           byte 3
B B A A A A A A  C C C C C C C C  D D D D D D D C  X X E E E E E E
1 0 5 4 3 2 1 0  7 6 5 4 3 2 1 0  6 5 4 3 2 1 0 8  X X 5 4 3 2 1 0
```

Figure 21—Little-endian struct packing in byte-addressable space

24.4.1.3 Big-endian packing example

A packed struct is shown in Example 369. This struct has 30 bits. A register for packing this struct would have 4 bytes.

```
struct my_packed_struct : packed_s<BIG_ENDIAN> {
   bit[6] A;
   bit[2] B;
   bit[9] C;
   bit[7] D;
   bit[6] E;
}
```

Example 369—DSL: Packed PSS big-endian struct

Register packing will start from field `A`. The most significant bit of `A` would go in the most significant bit of the register, as shown in Figure 22. Field `B` would go after field `A`. The most significant bit of `B` would go in the highest bit after `A` in the packed register, and so on. The layout of the packed struct in byte-addressable space is shown in Figure 23. (`X` means "don't-care bit" in Figure 22 and Figure 23.)

```
MSB                                                            LSB
 A A A A A A B B C C C C C C C C C D D D D D D D E E E E E E X X
 5 4 3 2 1 0 1 0 8 7 6 5 4 3 2 1 0 6 5 4 3 2 1 0 5 4 3 2 1 0 X X
```

Figure 22—Big-endian struct packing in register

```
     byte 0           byte 1            byte 2            byte 3
A A A A A A B B  C C C C C C C C C  D D D D D D D  E E E E E E X X
5 4 3 2 1 0 1 0  8 7 6 5 4 3 2 1 0  6 5 4 3 2 1 0  5 4 3 2 1 0 X X
```

Figure 23—Big-endian struct packing in byte-addressable space

24.4.2 sizeof_s

The template struct **`sizeof_s`** is used to query the physical storage size of a PSS data type. It applies to types that can be written to or read from a byte-addressable address space, namely scalars, packed structs, and arrays thereof.

24.4.2.1 Definition

```
struct sizeof_s<type T> {
   static const int nbytes = /* implementation-specific */;
   static const int nbits = /* implementation-specific */;
};
```

Syntax 190—DSL: sizeof_s struct

The static constant **`nbytes`** is initialized to the number of consecutive addresses required to store a value of type **`T`** in a byte-addressable address space. When using the read/write target functions (see 24.4.6), this number of bytes is assumed to be taken up by the data in the target storage. For types that are not byte-aligned in size, the number of bytes is rounded up. For the definition of packed struct layout in an address space, see 24.4.1.

The static constant **`nbits`** is initialized to the exact number of bits that are taken up by the representation of a value of type **`T`** in a byte-addressable address space.

`sizeof_s<>` shall not be parameterized with types other than scalars, packed structs, and arrays thereof.

24.4.2.2 Examples

The following code snippets show the value of **`nbytes`** of **`sizeof_s<>`** instantiated for several different types:

```
sizeof_s<int>::nbytes == 4

sizeof_s<int[3:0]>::nbytes == 1

sizeof_s<bit>::nbytes == 1
sizeof_s<bit[33]>::nbytes == 5

sizeof_s<array<int,10>>::nbytes == 40

struct my_packed_s : packed_s<> {bit[2] kind; int data;};
sizeof_s<my_packed_s>::nbytes  == 5
```

24.4.3 Address space handles

The built-in package **`addr_reg_pkg`** defines PSS types for *address space handles*.

```
typedef chandle addr_handle_t;

const addr_handle_t nullhandle = /* implementation-specific */;

struct sized_addr_handle_s <  int SZ, // in bits
                              int lsb = 0,
                              endianness_e e = LITTLE_ENDIAN
                            > : packed_s<e> {
    addr_handle_t hndl;
};
```

Syntax 191—DSL: Address space handle

24.4.3.1 Generic address space handle

`addr_handle_t` is the generic type for address handles within an address space. A variable of type **`addr_handle_t`** resolves to a concrete address value during test execution, on the target platform. However, the concrete value of an address handle cannot be obtained during the solve process, on the solve platform. A field of type **`addr_handle_t`** cannot be declared directly in a packed struct type. Packed structs are defined in 24.4.1.

24.4.3.2 nullhandle

`nullhandle` represents the address value `0` within the target address space, regardless of the actual mapping of regions.

24.4.3.3 sized address space handle

The wrapper struct **`sized_addr_handle_s`** is used for specifying the size of an address handle in a packed struct. An address field within a packed struct shall only be declared using **`sized_addr_handle_s`**, and not directly as a field of type **`addr_handle_t`**.

The **`SZ`** parameter specifies the size of the handle itself in bits when used in a packed struct. Note that the **`SZ`** parameter is not the size of the data it is pointing to.

The **`lsb`** parameter defines the starting bit in the resolved address that would become bit `0` of sized address handle in packed struct. For example, assume that the resolved address is 64 bits and the size of the handle is 30 bits, with the the **`lsb`** parameter set to 2. In this case, a sized handle in a packed struct would have bits 31 to 2 from the resolved address.

See an example in 24.4.7.

24.4.4 Obtaining an address space handle

A handle in an address space can be created from an address claim (with an optional offset value), from another handle (with an offset value), or from a region in an address space. An address claim is made using a claim struct declaration in actions and objects.

Some address space regions are non-allocatable. These regions can be used to represent memory-mapped I/O (MMIO) register spaces. A handle can be created from a region in an address space, in order to access non-allocatable regions.

A handle to a region is obtained when the region is added to the address space, using the **`add_region`** (see 24.2.1.2.1) or **`add_nonallocatable_region`** (see 24.2.1.2.2) functions. To create address handles from address claims or from other handles, the following functions are defined in the built-in package **`addr_reg_pkg`**.

24.4.4.1 make_handle_from_claim function

The function **`make_handle_from_claim()`** creates an address handle from a claim, with an optional offset value.

```
function addr_handle_t make_handle_from_claim
    (addr_claim_base_s claim, bit[64] offset = 0);
```

Syntax 192—DSL: make_handle_from_claim function

The **`make_handle_from_claim`** function arguments are:

— A claim struct instance declared in an action or a flow/resource object

— An optional offset value, of a 64-bit type

The returned handle's resolved address will be the sum of the claim's resolved address and the offset. The return value of the function is of type **`addr_handle_t`**.

24.4.4.1.1 Example

```
action my_action {
  rand transparent_addr_claim_s<> claim;

  constraint claim.size == 128;
  constraint claim.alignment == 2**4;

  exec body {
    int offset = 16;
    int data = 128;

    addr_handle_t h0 = make_handle_from_claim(claim);
    write32(h0, data); // access API defined in 24.4.6.1

    // Address handle from claim with an offset
    addr_handle_t h1 = make_handle_from_claim(claim, offset);
    write32(h1, data);
  }
};
```

Example 370—DSL: make_handle_from_claim example

24.4.4.2 make_handle_from_handle function

The function **`make_handle_from_handle()`** creates an address handle from another handle, given an offset.

```
function addr_handle_t make_handle_from_handle
    (addr_handle_t handle, bit[64] offset);
```

Syntax 193—DSL: make_handle_from_handle function

The **`make_handle_from_handle`** function arguments are:

— A handle that was created by a different call to a **`make_handle`** function

— An offset value, of a 64-bit type

The returned handle's resolved address will be the sum of the **`handle`** parameter’s resolved address and the offset. The return value of the function is of type **`addr_handle_t`**.

24.4.4.2.1 Example

```
action my_action {
   transparent_addr_claim_s<> claim;
   constraint claim.alignment == 2**4;

   exec body {
      int offset = 16;
      int data = 128;

      addr_handle_t h0 = make_handle_from_claim(claim, offset);
      write32(h0, data);

      // Make handle from another handle with an offset
      addr_handle_t h1 = make_handle_from_handle(h0, sizeof_s<int>::nbytes);
      write32(h1, data);
   }
};
```

Example 371—DSL: make_handle_from_handle example

24.4.5 addr_value function

The function **`addr_value()`** returns the resolved address of the parameter handle, as a numeric value. **`addr_value()`** is a target function and shall only be used in **exec body**, **run_start**, **run_end**, or functions called from these **exec** blocks.

```
function bit[64] addr_value (addr_handle_t hndl);
import target function addr_value;
```

Syntax 194—DSL: addr_value function

24.4.6 Access operations

Read/write operations of PSS data from/to byte-addressable address space are defined as a set of target functions. Target **exec** blocks (**exec body**, **run_start**, **run_end**), and functions called from them, may call these core library functions to access allocated addresses.

Access functions use an address handle to designate the required location within an address space.

PSS provides a way to customize the implementation of access functions for different executors (see 24.4.6.5).

24.4.6.1 Primitive read operations

Syntax 195 defines read operations for numeric types from byte addressable address spaces to read one, two, four or eight consecutive bytes starting at the address indicated by the **`addr_handle_t`** argument.

```
function bit[8]   read8(addr_handle_t hndl);
function bit[16] read16(addr_handle_t hndl);
function bit[32] read32(addr_handle_t hndl);
function bit[64] read64(addr_handle_t hndl);
```

Syntax 195—DSL: Primitive read operations for byte addressable spaces

The first byte goes into bits [7:0], then the next byte goes into bits [15:8], and so on.

24.4.6.2 Primitive write operations

Syntax 196 defines write operations for numeric types to byte addressable address spaces to write one, two, four or eight consecutive bytes from the **data** argument starting at the address indicated by the **addr_handle_t** argument.

```
function void write8 (addr_handle_t hndl, bit[8]  data);
function void write16(addr_handle_t hndl, bit[16] data);
function void write32(addr_handle_t hndl, bit[32] data);
function void write64(addr_handle_t hndl, bit[64] data);
```

Syntax 196—DSL: Primitive write operations for byte addressable spaces

Bits [7:0] of the input **data** go into the starting address specified by the **addr_handle_t** argument, bits [15:8] go into the next address (starting address + 1), and so on.

24.4.6.3 Read and write N consecutive bytes

Syntax 197 defines operations to read and write a series of consecutive bytes from byte addressable space.

For a read operation, the read data is stored in the argument **data**. For function **read_bytes()**, the **size** argument indicates the number of consecutive bytes to read. The returned list is resized accordingly, and its previous values, if any, are overwritten.

For a write operation, the input data is taken from the argument **data**. For function **write_bytes()**, the number of bytes to write is determined by the list size of the **data** parameter.

```
function void read_bytes (addr_handle_t hndl, list<bit[8]> data, int size);
function void write_bytes(addr_handle_t hndl, list<bit[8]> data);
```

Syntax 197—DSL: Read and write series of bytes

The first byte read comes from the address indicated by the **hndl** argument. This byte is stored at the first location (index 0) in the **data** list. The second byte comes from the address incremented by one and is stored at the second location (index 1) in the **data** list, and so on. The same semantics apply to **write_bytes**().

24.4.6.4 Read and write packed structs

Read and write operations to access packed structs are defined in Syntax 198. Argument **packed_struct** of functions **read_struct()** and **write_struct()** shall be a subtype of the **packed_s** struct. The **packed_struct** argument is read from or written to the address specified by the **hndl** argument.

```
function void read_struct (addr_handle_t hndl, struct packed_struct);
function void write_struct(addr_handle_t hndl, struct packed_struct);
```

Syntax 198—DSL: Read and write packed structs

The PSS implementation shall convert calls to **read_struct()** and **write_struct()** to one or more invocations of the primitive read and write operations (see 24.4.6.1 and 24.4.6.2). Reading and writing of structs of up to 64 bits stored at a correspondingly aligned address shall be implemented with a single primitive operation, and in other cases may be partitioned into multiple operations of any size.

24.4.6.5 Executor-based customization of read/write functions

PSS tools may provide built-in implementations of read and write operations for mainstream execution contexts. However, users can optionally customize the implementation of these operations for their own purposes and execution contexts.

Calls to primitive read/write functions (defined above in 24.4.6.1 and 24.4.6.2), and calls to byte list read/write functions (defined above in 24.4.6.3), are delegated to functions with the identical prototype in the executor instance assigned to the evaluation action or flow/resource object. Syntax 199 below shows the declarations of the executor implementation functions.

```
extend component executor_base_c {
   function bit[8]   read8(addr_handle_t hndl);
   function bit[16] read16(addr_handle_t hndl);
   function bit[32] read32(addr_handle_t hndl);
   function bit[64] read64(addr_handle_t hndl);

   function void write8 (addr_handle_t hndl, bit[8]  data);
   function void write16(addr_handle_t hndl, bit[16] data);
   function void write32(addr_handle_t hndl, bit[32] data);
   function void write64(addr_handle_t hndl, bit[64] data);

   function void read_bytes (addr_handle_t hndl, list<bit[8]> data,
                                                       int size);
   function void write_bytes(addr_handle_t hndl, list<bit[8]> data);
};
```

Syntax 199—DSL: Primitive operation implementation functions

Note that struct read/write functions (defined above in 24.4.6.4) and register read/write functions (defined below in 24.5.1) are implemented in terms of their respective primitive operations. Therefore, custom implementations of the primitive operations in an executor apply similarly to struct and register read/write functions.

The code in Example 372 below illustrates how a PSS implementation may define the delegation of one of the primitive read/write functions to the corresponding function in the current executor. The actual implementation does not necessarily take this form, but should have equivalent observable behavior. See 24.1.2.5 for more on the semantics of function **`executor()`**.

```
function bit[32] read32(addr_handle_t hndl) {
   if (executor() != null ) {
      return executor().read32(hndl);
   } else {
      // return value per default implementation
   }
}
```

Example 372—DSL: Illustration of read32()

Example 373 below demonstrates how primitive operations **`read32()`** and **`write32()`** are mapped to calls to functions of a C bus transactor in the context of a user-defined executor type.

```
function bit[32] my_transactor_read_word(bit[64] addr);
import target C function my_transactor_read_word;

function void my_transactor_write_word(bit[64] addr, bit[32] data);
import target C function my_transactor_write_word;

component my_transactor_executor_c<struct TRAIT : executor_trait_s =
    empty_executor_trait_s> : executor_c<TRAIT> {
  function bit[32] read32(addr_handle_t hndl) {
    return my_transactor_read_word(addr_value(hndl));
  }

  function void write32(addr_handle_t hndl, bit[32] data) {
    my_transactor_write_word(addr_value(hndl), data);
  }
};
```

Example 373—DSL: Mapping of primitive operations to foreign C functions

In Example 374 below, executor type `uvm_ubus_executor_c` corresponds to a UVM bus master. The **`write8()`** function is defined in terms of a SystemVerilog imported function (task) that starts a write-byte sequence on the agent designated by the path parameter. The executor type is instantiated twice under **`pss_top`**, and each instance is associated with a different UVM agent in the target environment using the UVM path.

```
import target SV function void ubus_write8(string uvm_path, bit[64] addr,
   bit[8] data);

component uvm_ubus_executor_c : executor_c<bus_trait_s> {
   string uvm_path;

   function void write8(addr_handle_t hndl, bit[8] data) {
      ubus_write8(uvm_path, addr_value(hndl), data);
   }
};

extend component pss_top {
  uvm_ubus_executor_c masters[2];
  executor_group_c<bus_trait_s> bus_group;
  exec init_down {
    foreach (m: masters) {
      bus_group.add_executor(m);
    }
    masters[0].uvm_path = "uvm_test_top.env.ubus_master0";
    masters[1].uvm_path = "uvm_test_top.env.ubus_master1";
  }
};
```

Example 374—DSL: Mapping of primitive operations to UVM sequences

In Example 375 below, an executor corresponding to a 32-bit architecture CPU customizes the **`read64()`** and **`write64()`** operations to be implemented in terms of the built-in **`read32()`** and **`write32()`** operations.

```
component my_32bit_cpu_c : executor_c<my_core_trait_s> {
  function bit[64] read64(addr_handle_t hndl) {
    bit[64] result;
    result[31: 0] = read32(hndl);
    result[63:32] = read32(make_handle_from_handle(hndl,4));
    return result;
   }

   function void write64(addr_handle_t hndl, bit[64] data) {
      write32(hndl, data[31:0]);
      write32(make_handle_from_handle(hndl,4), data[63:32]);
   }
};
```

Example 375—DSL: Implementing primitive operations in terms of other operations

24.4.7 Target data structure setup example

The following example demonstrates use of packed PSS data written to allocations on byte addressable space. It also demonstrates the use of address handles to construct complex data structures in target memory. Lifetime of allocation is extended by using address handles in flow objects.

```
buffer data_buff {
   rand addr_claim_s<> mem_seg;
};

component dma_c {

   struct descriptor_s : packed_s<> {
      sized_addr_handle_s<32> src_addr;
      sized_addr_handle_s<32> dst_addr;
      int size;
      sized_addr_handle_s<32> next_descr;
   };

   state descr_chain_state {
      list<addr_handle_t> handle_list;
   };

   pool descr_chain_state descr_chain_statevar;
   bind descr_chain_statevar *;

   action alloc_first_descr {
      output descr_chain_state out_chain;

      rand addr_claim_s<> next_descr_mem;
      constraint next_descr_mem.size == sizeof_s<descriptor_s>::nbytes;

      exec post_solve {
         out_chain.handle_list.push_back(
                             make_handle_from_claim(next_descr_mem));
      }
   };
```

Example 376—DSL: Example using complex data structures

```
action chained_xfer {
   input  data_buff src_buff;
   output data_buff dst_buff;
   constraint dst_buff.mem_seg.size == src_buff.mem_seg.size;

   input  descr_chain_state in_chain;
   output descr_chain_state out_chain;

   rand bool last;

   descriptor_s descr;

   rand addr_claim_s<> next_descr_mem;
   constraint next_descr_mem.size == sizeof_s<descriptor_s>::nbytes;

   addr_handle_t descr_hndl;

   exec post_solve {
      descr.src_addr.hndl = make_handle_from_claim(src_buff.mem_seg);
      descr.dst_addr.hndl = make_handle_from_claim(dst_buff.mem_seg);
      descr.size = src_buff.mem_seg.size;
      if (last) {
         descr.next_descr.hndl = nullhandle;
      } else {
         descr.next_descr.hndl = make_handle_from_claim(next_descr_mem);
      }

      // tail of current list
      descr_hndl = in_chain.handle_list[in_chain.handle_list.size()-1];

      // copy over list from input to output
      out_chain.handle_list = in_chain.handle_list;
      // add next pointer
      out_chain.handle_list.push_back(
                           make_handle_from_claim(next_descr_mem));
   }

   exec body {
      write_struct(descr_hndl,descr);
   }
};

action execute_xfer {
   input descr_chain_state in_chain;

   addr_handle_t descr_list_head;

   exec post_solve {
      descr_list_head = in_chain.handle_list[0]; // head of list
   }

   exec body {
      // Initiate chained-transfer with descr_list_head
      // Wait for the chained-transfer to complete
   }
};
```

Example 376—DSL: Example using complex data structures (cont.)

```
      action multi_xfer {
         rand int in [1..10] num_of_xfers;

         activity {
            do alloc_first_descr;
            repeat (i: num_of_xfers) {
               do chained_xfer with {last == (i == num_of_xfers-1);};
            }
            do execute_xfer;
         }
      };
   };
```

Example 376—DSL: Example using complex data structures (cont.)

In this example, the `chained_xfer` **action** represents the data flow (source/destination buffers) associated with this transaction. It populates the descriptor, including a pointer to the next descriptor, which it allocates. Its runtime execution writes the full descriptor out to memory, in the location allocated for it by the previous link in the chain.

24.5 Registers

A PSS model will often specify interaction with the hardware SUT to control how the PSS tool-generated code will read/write to programmable registers of the SUT. This section shows how to associate meaningful identifiers with register addresses that need to be specified in the PSS model description, as well as manipulation of the value of register fields by name.

All the core library constructs in this section are declared in the **`addr_reg_pkg`** package. For brevity, the definitions below do not include the package name.

24.5.1 PSS register definition

A *register* is a logical aggregation of fields that are addressed as a single unit.

The **`reg_c`** component is a base type for specifying the programmable registers of the DUT. Note that it is a **pure** component (see 10.7). It shall be illegal to extend the **`reg_c`** class.

```
enum reg_access {READWRITE, READONLY, WRITEONLY};

pure component reg_c < type R,
                       reg_access ACC = READWRITE,
                       int SZ = (8*sizeof_s<R>::nbytes)> {

    function R read();                  // Read register as type R
    import target function read;

    function void write(R r);           // Write register as type R
    import target function write;

    function bit[SZ] read_val();        // Read register value as bits
    import target function read_val;

    function void write_val(bit[SZ] r);  // Write register value as bits
    import target function write_val;
};
```

Syntax 200—DSL: PSS register definition

Component **`reg_c`** is parameterized by:

a) A type **`R`** for the value (referred to as the *register-value type*) that can be read/written from/to the register, which can be:
   1) A packed structure type (that represents the register structure)
   2) A bit-vector type (**`bit[N]`**)

b) Kind of access allowed to the register, which by default is **`READWRITE`**

c) Width of the register (**`SZ`**) in number of bits, which by default equals the size of the register-value type **`R`** (rounded up to a multiple of 8)

`SZ`, if specified by the user, shall be greater than or equal to the size of the register-value type **`R`**. If the size of the register-value type **`R`** is less than the width of the register, it will be equivalent to having **`SZ – sizeof_s<R>::nbits`** reserved bits at the end of the structure.

The **`read()`**/**`read_val()`**/**`write()`**/**`write_val()`** functions may be called from the test-realization layer of a PSS model. Being declared as target functions, these need to be called in an **exec body** context.

The **`read()`** and **`read_val()`** functions return the value of the register in the DUT (the former returns an instance of register-value type and the latter returns a bit vector). The **`write()`** and **`write_val()`** functions update the value of a register in a DUT (the former accepting an instance of register-value type and the latter a bit vector). If the register-value type is a bit vector, then the functions **`read()`** and **`read_val()`** are equivalent, as are **`write()`** and **`write_val()`**.

See 24.5.4 for a description of the implementation of these functions. It shall be an error to call **`read()`** and **`read_val()`** on a register object whose access is set to **`WRITEONLY`**. It shall be an error to call **`write()`** and **`write_val()`** on a register object whose access is set to **`READONLY`**.

A template instantiation of the class **`reg_c`** (i.e., **`reg_c<R, ACC, SZ>`** for some concrete values for **`R`**, **`ACC`** and **`SZ`**) or a component derived from such a template instantiation (directly or indirectly) is a *register type*. An object of register type can be instantiated only in a *register group* (see 24.5.2).

Example 377 shows examples of register declarations.

```
struct my_reg0_s : packed_s<> { // (1)
   bit [16] fld0;
   bit [16] fld1;
};

pure component my_reg0_c : reg_c<my_reg0_s> {} // (2)

struct my_reg1_s : packed_s<> {

   bit     fld0;
   bit [2] fld1;
   bit [2] fld2[5];                 // (3)
};

pure component my_reg1_c : reg_c<my_reg1_s, READWRITE, 32> {} // (4)
```

Example 377—DSL: Examples of register declarations

Notes:

1) `my_reg0_s` is the register-value type. The endianness can be explicitly specified if needed.
2) `my_reg0_c` is the register type. Since it derives from **`reg_c`**`<my_reg0_s>`, it inherits the **`reg_c`** read/write functions. Note that the access is **`READWRITE`** by default and the width equals the size of the associated register-value type, `my_reg0_s`.
3) Fixed-size arrays are allowed.
4) **`sizeof_s<`**`my_reg1_s`**`>::nbits`** `= 13`, which is less than the specified register width (32). This is allowed and is equivalent to specifying a field of size 32 – 13 = 19 bits after `fld2[5]`. This reserved field cannot be accessed using **`read()`**/**`write()`** functions on the register object. In the numeric value passed to **`write_val()`** and in the return value of **`read_val()`**, the value of these bits is not defined by this standard.

It is recommended to declare the register type as **pure**. This allows the PSS implementation to optimally handle large static register components.

24.5.2 PSS register group definition

A *register group* aggregates instances of registers and of other register groups.

The **`reg_group_c`** component is the base type for specifying register groups. Note that it is a **pure** component (see 10.7). It shall be illegal to extend the **`reg_group_c`** class.

```
struct node_s {
   string name;
   int    index;
};

pure component reg_group_c {
   pure function bit[64] get_offset_of_instance(string name);
   pure function bit[64] get_offset_of_instance_array(string name,
                                                       int index);
   pure function bit[64] get_offset_of_path(list<node_s> path);

   function void set_handle(addr_handle_t addr);
   import solve function set_handle;
};
```

Syntax 201—DSL: PSS register group definition

A register group may instantiate registers and instances of other register groups. An instance of a register group may be created in another register group, or directly in a non-register-group component. In the latter case, the register group can be *associated* with an address region. The **`set_handle()`** function associates the register group with an address region. The definition of this function is implementation-defined. See 24.5.3 for more details on use of this function.

Each element in a register group (whether an instance of a register or an instance of another group) has a user-defined address offset relative to a notional base address of the register group.

The function **`get_offset_of_instance()`** retrieves the offset of a non-array element in a register group, by name of the element. The function **`get_offset_of_instance_array()`** retrieves the offset of an array element in a register group, by name of the element and index in the array.

For example, suppose `a` is an instance of a register group that has the following elements:

— A register instance, `r0`

— A register array instance, `r1[4]`

Calling `a.get_offset_of_instance("r0")` returns the offset of the element `r0`. Calling `a.get_offset_of_instance_array("r1", 2)` returns the offset at index `2` of element `r1`.

The function **`get_offset_of_path()`** retrieves the offset of a register from a hierarchical path of the register, starting from a given register group. The hierarchical path of the register is specified as a **list** of **`node_s`** objects. Each **`node_s`** object provides the name of the element (as a string) and an index (applicable if and only if the element is of array type). The first element of the list corresponds to an object directly instantiated in the given register group. Successive elements of the list correspond to an object instantiated in the register group referred by the predecessor node. The last element of the list corresponds to the final register instance.

For example, suppose `b` is an instance of a register group that has the following elements: a register group array instance `grp0[10]`, which in turn has a register group instance `grp1`, which in turn has a register instance, `r0`. The hierarchical path of register `r0` in `grp1` within `grp0[5]` within `b` will then be the list (e.g., `path_to_r0`) with the following elements in succession:

— `[0]`: **`node_s`** object with **`name`** = `"grp0"` and **`index`** = 5

— `[1]`: **`node_s`** object with **`name`** = `"grp1"` (**`index`** is not used)

— `[2]`: **`node_s`** object with **`name`** = `"r0"` (**`index`** is not used)

Calling `b.get_offset_of_path(path_to_r0)` will return the offset of register `r0` relative to the base address of `b`.

For a given register group, users shall provide the implementation of either **`get_offset_of_path()`** or of both functions **`get_offset_of_instance()`** and **`get_offset_of_instance_array()`**. It shall be an error to provide an implementation of all three functions. These may be implemented as native PSS functions, or foreign-language binding may be used. These functions (when implemented) shall provide the relative offset of *all* the elements in the register group. These functions are called by a PSS tool to compute the offset for a register access (as described later in 24.5.4). Note that these functions are declared **pure** —the implementation shall not have side-effects.

Example 378 shows an example of a register group declaration.

```
pure component my_reg_grp0_c : reg_group_c {
   my_readonly_reg0_c   reg0;            // (1)
   my_reg1_c            reg1[4];         // (2)
   my_sub_reg_grp_c     sub;             // (3)
   reg_c<my_regx_s, WRITEONLY, 32> regx; // (4)

   // May be foreign, too
   function bit[64] get_offset_of_instance(string name) {
      match(name) {
        ["reg0"]:  return 0x0;
        ["sub"]:   return 0x20;
        ["regx"]:  return 0x0;            // (5)
        default: return -1; // Error case
      }
   }

   function bit[64] get_offset_of_instance_array(string name, int index) {
      match(name) {
        ["reg1"]:  return (0x4 + index*4);
        default: return -1; // Error case
      }
   }
}
```

Example 378—DSL: Example of register group declaration

Notes:

1) `my_readonly_reg0_c`, `my_reg1_c`, etc., are all register types (declarations not shown in the example).
2) Arrays of registers are allowed.
3) Groups may contain other groups (declaration of `my_sub_reg_grp_c` not shown in the example).
4) A direct instance of **`reg_c<>`** may be created in a register group.
5) Offsets of two elements may be same. A typical use case for this is when a **`READONLY`** and a **`WRITEONLY`** register share the same offset.

24.5.3 Association with address region

Before the read/write functions can be invoked on a register, the top-level register group (under which the register object has been instantiated) must be associated with an address region, using the **`set_handle()`**

function in that register group. This is done from within an **exec init_up** or **init_down** context. Only the top-level register group shall be associated with an address region; it shall be an error to call **`set_handle()`** on other register group instances. An example is shown in Example 379.

```
component my_component_c
{
   my_reg_grp0_c  grp0; // Top-level group

   transparent_addr_space_c<> sys_mem;

   exec init_up {
     transparent_addr_region_s<> mmio_region;
     addr_handle_t h;
     mmio_region.size = 1024;
     mmio_region.addr = 0xA0000000;

     h = sys_mem.add_nonallocatable_region(mmio_region);

     grp0.set_handle(h);
   }
};
```

Example 379—DSL: Top-level group and address region association

24.5.4 Translation of register read/write

The PSS implementation shall convert **`read`**/**`read_val`**/**`write`**/**`write_val`** function calls on a register to invocations of the primitive read/write operations on its associated address (see 24.4.6.1 and 24.4.6.2). The conversion shall proceed as follows:

a) The read/write function is selected based on the size of the register. For example, if the size of the register is 32, the function **`read32(addr_handle_t hndl)`** will be called for a register read.

b) The total offset is calculated by summing the offsets of all elements starting from the top-level register group to the register itself.

   1) If the function **`get_offset_of_path()`** is available in any intermediate register group instance, the PSS implementation will use that function to find the offset of the register relative to the register group.

   2) Otherwise, the function **`get_offset_of_instance_array()`** or **`get_offset_of_instance()`** is used, depending on whether or not the register instance or register group instance is an array.

   For example, in the expression (where `a`, `b`, `c`, and `d` are all instances of register groups and `reg` is a register object):

   ```
   comp.a.b.c.d[4].reg.write_val(10)
   ```

   if the function **`get_offset_of_path()`** is implemented in the type of element `c`, then the offset is calculated as:

   ```
   offset = comp.a.get_offset_of_instance("b") +
            comp.a.b.get_offset_of_instance("c") +
            comp.a.b.c.get_offset_of_path(path)
   ```

   where `path` is the **list** `[ {"d", 4}, {"reg", 0} ].`

c) The handle for the access is calculated as **`make_handle_from_handle(h, offset)`**, where `h` is the handle set using **`set_handle()`** on the top-level register group.

24.5.5 Recommended packaging

It is recommended that all the register (and register group) definitions of a device be placed in a separate file and in a separate package by themselves, as shown in Example 380.

```
// In my_IP_regs.pss
package my_IP_regs {
   import addr_reg_pkg::*;
   struct my_reg0_s : packed_s<> { ... };
   pure component my_reg0_c : reg_c<my_reg0_s, READWRITE, 32> { ... };
   // ... etc: other registers

   pure component my_reg_group_c : reg_group_c {
      my_reg0_c r0;
      // ... etc: other registers
   };
}
```

Example 380—DSL: Recommended packaging

This ensures that the register file can be easily generated from a register specification (e.g., IP-XACT).

Annex A

(informative)

Bibliography

[B1] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.

Annex B

(normative)

Formal syntax

The PSS formal syntax is described using Backus-Naur Form (BNF). The syntax of the PSS source is derived from the starting symbol `Model`. If there is a conflict between a grammar element shown anywhere in this standard and the material in this annex, the material shown in this annex shall take precedence.

```
Model ::= { portable_stimulus_description }

portable_stimulus_description ::=
      package_body_item
    | package_declaration
    | component_declaration
```

B.1 Package declarations

```
package_declaration ::= package package_id_path  { { package_body_item } }

package_id_path ::= package_identifier { :: package_identifier }

package_body_item ::=
      abstract_action_declaration
    | struct_declaration
    | enum_declaration
    | covergroup_declaration
    | function_decl
    | import_class_decl
    | procedural_function
    | import_function
    | target_template_function
    | export_action
    | typedef_declaration
    | import_stmt
    | extend_stmt
    | const_field_declaration
    | component_declaration
    | package_declaration
    | compile_assert_stmt
    | package_body_compile_if
    | stmt_terminator

import_stmt ::= import package_import_pattern ;

package_import_pattern ::= type_identifier [ package_import_qualifier ]

package_import_qualifier ::=
      package_import_wildcard
    | package_import_alias

package_import_wildcard ::= :: *
```

```
package_import_alias ::= as package_identifier

extend_stmt ::=
      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 } ] }

const_field_declaration ::= [ static ] const data_declaration

stmt_terminator ::= ;
```

B.2 Action declarations

```
action_declaration ::= action action_identifier
   [ template_param_decl_list ] [ action_super_spec ] { { action_body_item } }

abstract_action_declaration ::= abstract action_declaration

action_super_spec ::= : type_identifier

action_body_item ::=
      activity_declaration
    | override_declaration
    | constraint_declaration
    | action_field_declaration
    | symbol_declaration
    | covergroup_declaration
    | exec_block_stmt
    | activity_scheduling_constraint
    | attr_group
    | compile_assert_stmt
    | covergroup_instantiation
    | action_body_compile_if
    | stmt_terminator

activity_declaration ::= activity { { [ label_identifier : ] activity_stmt } }

action_field_declaration ::=
      attr_field
    | activity_data_field
    | action_handle_declaration
    | object_ref_field_declaration

object_ref_field_declaration ::=
      flow_ref_field_declaration
    | resource_ref_field_declaration

flow_ref_field_declaration ::=
    ( input | output ) flow_object_type object_ref_field { , object_ref_field } ;

resource_ref_field_declaration ::=
   ( lock | share ) resource_object_type object_ref_field {, object_ref_field } ;
```

```
flow_object_type ::=
      buffer_type_identifier
    | state_type_identifier
    | stream_type_identifier

resource_object_type ::= resource_type_identifier

object_ref_field ::= identifier [ array_dim ]

action_handle_declaration ::= action_type_identifier action_instantiation ;

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

activity_data_field ::= action data_declaration

activity_scheduling_constraint ::= constraint ( parallel | sequence )
      { hierarchical_id , hierarchical_id { , hierarchical_id } };
```

B.3 Struct declarations

```
struct_declaration ::= struct_kind struct_identifier
   [ template_param_decl_list ] [ struct_super_spec ] { { struct_body_item } }

struct_kind ::=
      struct
    | object_kind

object_kind ::=
      buffer
    | stream
    | state
    | resource

struct_super_spec ::= : type_identifier

struct_body_item ::=
      constraint_declaration
    | attr_field
    | typedef_declaration
    | exec_block_stmt
    | attr_group
    | compile_assert_stmt
    | covergroup_declaration
    | covergroup_instantiation
    | struct_body_compile_if
    | stmt_terminator
```

B.4 Exec blocks

```
exec_block_stmt ::=
      exec_block
    | target_code_exec_block
    | target_file_exec_block
    | stmt_terminator

exec_block ::= exec exec_kind { { exec_stmt } }

exec_kind ::=
      pre_solve
    | post_solve
    | body
    | header
    | declaration
    | run_start
    | run_end
    | init_down
    | init_up
    | init

exec_stmt ::=
      procedural_stmt
    | exec_super_stmt

exec_super_stmt ::= super ;

target_code_exec_block ::= exec exec_kind language_identifier = string_literal;

target_file_exec_block ::= exec file filename_string = string_literal ;
```

B.5 Functions

```
procedural_function ::= [ platform_qualifier ] [ pure ] function
   function_prototype { { procedural_stmt } }

function_decl ::= [ pure ] function function_prototype ;

function_prototype ::=
   function_return_type function_identifier function_parameter_list_prototype

function_return_type ::=
      void
    | data_type

function_parameter_list_prototype ::=
      ( [ function_parameter { , function_parameter } ] )
    | ( { function_parameter , } varargs_parameter )

function_parameter ::=
      [ function_parameter_dir ] data_type identifier [ = constant_expression ]
    | ( type | ref type_category | struct ) identifier
```

```
function_parameter_dir ::=
      input
    | output
    | inout

varargs_parameter ::=
      ( data_type | type | ref type_category | struct ) ... identifier
```

B.6 Foreign procedural interface

```
import_function ::=
      import [ platform_qualifier ] [ language_identifier ]
          function type_identifier ;
    | import [ platform_qualifier ] [ language_identifier ]
          function function_prototype ;

platform_qualifier ::=
      target
    | solve

target_template_function ::=
    target language_identifier function function_prototype = string_literal ;

import_class_decl ::= import class import_class_identifier
    [ import_class_extends ] { { import_class_function_decl } }

import_class_extends ::= : type_identifier { , type_identifier }

import_class_function_decl ::= function_prototype ;

export_action ::= export [ platform_qualifier ] action_type_identifier
    function_parameter_list_prototype ;
```

B.7 Procedural statements

```
procedural_stmt ::=
      procedural_sequence_block_stmt
    | procedural_data_declaration
    | procedural_assignment_stmt
    | procedural_void_function_call_stmt
    | procedural_return_stmt
    | procedural_repeat_stmt
    | procedural_foreach_stmt
    | procedural_if_else_stmt
    | procedural_match_stmt
    | procedural_break_stmt
    | procedural_continue_stmt
    } stmt_terminator

procedural_sequence_block_stmt ::= [ sequence ] { { procedural_stmt } }

procedural_data_declaration ::= data_type procedural_data_instantiation
    { , procedural_data_instantiation } ;
```

```
procedural_data_instantiation ::= identifier [ array_dim ] [ = expression ]

procedural_assignment_stmt ::= ref_path assign_op expression ;

procedural_void_function_call_stmt ::= [ ( void ) ] function_call ;

procedural_return_stmt ::= return [ expression ] ;

procedural_repeat_stmt ::=
      repeat ( [ index_identifier : ] expression ) procedural_stmt
    | repeat procedural_stmt while ( expression ) ;
    | while ( expression ) procedural_stmt

procedural_foreach_stmt ::=
    foreach ( [ iterator_identifier : ] expression  [ [ index_identifier ] ] )
     procedural_stmt

procedural_if_else_stmt ::=
    if ( expression ) procedural_stmt [ else procedural_stmt ]

procedural_match_stmt ::=
    match ( match_expression )
      { procedural_match_choice { procedural_match_choice } }

procedural_match_choice ::=
      [ open_range_list ] : procedural_stmt
    | default : procedural_stmt

procedural_break_stmt ::= break ;

procedural_continue_stmt ::= continue ;
```

B.8 Component declarations

```
component_declaration ::=
    [ pure ] component component_identifier [ template_param_decl_list ]
      [ component_super_spec ] { { component_body_item } }

component_super_spec ::= : type_identifier

component_body_item ::=
      override_declaration
    | component_data_declaration
    | component_pool_declaration
    | action_declaration
    | abstract_action_declaration
    | object_bind_stmt
    | exec_block
    | struct_declaration
    | enum_declaration
    | covergroup_declaration
    | function_decl
    | import_class_decl
    | procedural_function
    | import_function
    | target_template_function
    | export_action
    | typedef_declaration
    | import_stmt
    | extend_stmt
    | compile_assert_stmt
    | attr_group
    | component_body_compile_if
    | stmt_terminator

component_data_declaration ::=
    [ access_modifier ] [ static const ] data_declaration

component_pool_declaration ::=
    pool [ [ expression ] ] type_identifier identifier ;

object_bind_stmt ::= bind hierarchical_id object_bind_item_or_list ;

object_bind_item_or_list ::=
      object_bind_item_path
    | { object_bind_item_path { , object_bind_item_path } }

object_bind_item_path ::= { component_path_elem . } object_bind_item

component_path_elem ::= component_identifier [ [ constant_expression ] ]

object_bind_item ::=
      action_type_identifier . identifier [ [ constant_expression ] ]
    | *
```

B.9 Activity statements

```
activity_stmt ::=
      [ label_identifier : ] labeled_activity_stmt
    | activity_data_field
    | activity_bind_stmt
    | action_handle_declaration
    | activity_constraint_stmt
    | activity_scheduling_constraint
    | stmt_terminator

labeled_activity_stmt ::=
      activity_action_traversal_stmt
    | activity_sequence_block_stmt
    | activity_parallel_stmt
    | activity_schedule_stmt
    | activity_repeat_stmt
    | activity_foreach_stmt
    | activity_select_stmt
    | activity_if_else_stmt
    | activity_match_stmt
    | activity_replicate_stmt
    | activity_super_stmt
    | symbol_call
```

```
activity_action_traversal_stmt ::=
      identifier [ [ expression ] ] inline_constraints_or_empty
    | do type_identifier inline_constraints_or_empty

inline_constraints_or_empty ::=
      with constraint_set
    | ;

activity_sequence_block_stmt ::= [ sequence ] { { activity_stmt } }

activity_parallel_stmt ::= parallel [ activity_join_spec ] { { activity_stmt } }

activity_schedule_stmt ::= schedule [ activity_join_spec ] { { activity_stmt } }

activity_join_spec ::=
      activity_join_branch
    | activity_join_select
    | activity_join_none
    | activity_join_first

activity_join_branch ::= join_branch ( label_identifier { , label_identifier } )

activity_join_select ::= join_select ( expression )

activity_join_none ::= join_none

activity_join_first ::= join_first ( expression )

activity_repeat_stmt ::=
      repeat ( [ index_identifier : ] expression ) activity_stmt
    | repeat activity_stmt while ( expression ) ;

activity_foreach_stmt ::= foreach ( [ iterator_identifier : ] expression
   [ [ index_identifier ] ] ) activity_stmt

activity_select_stmt ::= select { select_branch select_branch { select_branch } }

select_branch ::= [[ ( expression ) ][ [ expression ] ] : ] activity_stmt

activity_if_else_stmt ::= if ( expression ) activity_stmt [ else activity_stmt ]

activity_match_stmt ::=
    match ( match_expression ) { match_choice  { match_choice } }

match_expression ::= expression

match_choice ::=
      [ open_range_list ] : activity_stmt
    | default : activity_stmt

activity_replicate_stmt ::= replicate ( [ index_identifier : ] expression )
   [ label_identifier [ ] : ] labeled_activity_stmt

activity_super_stmt ::= super ;

activity_bind_stmt ::= bind hierarchical_id activity_bind_item_or_list ;
```

```
activity_bind_item_or_list ::=
      hierarchical_id
    | { hierarchical_id_list }

activity_constraint_stmt ::= constraint constraint_set

symbol_declaration ::=
    symbol symbol_identifier [ ( symbol_paramlist ) ] { { activity_stmt } }

symbol_paramlist ::= [ symbol_param { , symbol_param } ]

symbol_param ::= data_type identifier
```

B.10 Overrides

```
override_declaration ::= override { { override_stmt } }

override_stmt ::=
      type_override
    | instance_override
    | stmt_terminator

type_override ::= type type_identifier with type_identifier ;

instance_override ::= instance hierarchical_id with type_identifier ;
```

B.11 Data declarations

```
data_declaration ::= data_type data_instantiation { , data_instantiation } ;

data_instantiation ::= identifier [ array_dim ] [ = constant_expression ]

array_dim ::= [ constant_expression ]

attr_field ::= [ access_modifier ] [ rand | static const ] data_declaration

access_modifier ::= public | protected | private

attr_group ::= access_modifier :
```

B.12 Template types

```
template_param_decl_list ::= < template_param_decl { , template_param_decl } >

template_param_decl ::= type_param_decl | value_param_decl

type_param_decl ::= generic_type_param_decl | category_type_param_decl

generic_type_param_decl ::= type identifier [ = type_identifier ]

category_type_param_decl ::=
    type_category identifier [ type_restriction ] [ = type_identifier ]
```

```
type_restriction ::= : type_identifier

type_category ::=
        action
      | component
      | struct_kind

value_param_decl ::= data_type identifier [ = constant_expression ]

template_param_value_list ::=
    < [ template_param_value { , template_param_value } ] >

template_param_value ::= constant_expression | data_type
```

B.13 Data types

```
data_type ::=
        scalar_data_type
      | collection_type
      | reference_type
      | type_identifier

scalar_data_type ::=
        chandle_type
      | integer_type
      | string_type
      | bool_type
      | enum_type

casting_type ::=
        integer_type
      | bool_type
      | enum_type
      | type_identifier

chandle_type ::= chandle

integer_type ::= integer_atom_type
    [ [ constant_expression [ : 0 ] ] ]
    [ in [ domain_open_range_list ] ]

integer_atom_type ::=
        int
      | bit

domain_open_range_list ::=
    domain_open_range_value { , domain_open_range_value }

domain_open_range_value ::=
        constant_expression [ .. constant_expression ]
      | constant_expression ..
      | .. constant_expression

string_type ::= string [ in [ string_literal { , string_literal } ] ]
```

```
bool_type ::= bool

enum_declaration ::= enum enum_identifier { [ enum_item { , enum_item } ] }

enum_item ::= identifier [ = constant_expression ]

enum_type ::= enum_type_identifier [ in [ domain_open_range_list ] ]

collection_type ::=
      array < data_type , array_size_expression >
    | list < data_type >
    | map < data_type , data_type >
    | set < data_type >

array_size_expression ::= constant_expression

reference_type ::= ref entity_type_identifier

typedef_declaration ::= typedef data_type identifier ;
```

B.14 Constraints

```
constraint_declaration ::=
      constraint constraint_set
    | [ dynamic ] constraint identifier constraint_block

constraint_set ::=
      constraint_body_item
    | constraint_block

constraint_block ::= { { constraint_body_item } }

constraint_body_item ::=
      expression_constraint_item
    | foreach_constraint_item
    | forall_constraint_item
    | if_constraint_item
    | implication_constraint_item
    | unique_constraint_item
    | default hierarchical_id == constant_expression ;
    | default disable hierarchical_id ;
    | stmt_terminator

expression_constraint_item ::= expression ;

foreach_constraint_item ::=
    foreach ( [ iterator_identifier : ] expression [ [ index_identifier ] ] )
      constraint_set

forall_constraint_item ::=
   forall ( iterator_identifier : type_identifier [ in ref_path ] ) constraint_set

if_constraint_item ::= if ( expression ) constraint_set [ else constraint_set ]

implication_constraint_item ::= expression -> constraint_set
```

```
unique_constraint_item ::= unique { hierarchical_id_list } ;
```

B.15 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
    | stmt_terminator

covergroup_option ::=
      option . identifier = constant_expression ;
    | type_option . identifier = constant_expression ;

covergroup_instantiation ::=
      covergroup_type_instantiation
    | inline_covergroup

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

covergroup_type_instantiation ::=
    covergroup_type_identifier covergroup_identifier
      ( covergroup_portmap_list ) covergroup_options_or_empty

covergroup_portmap_list ::=
      covergroup_portmap { , covergroup_portmap }
    | hierarchical_id_list

covergroup_portmap ::= . identifier ( hierarchical_id )

covergroup_options_or_empty ::=
      with { { covergroup_option } }
    | ;
covergroup_coverpoint ::= [ [ data_type ] coverpoint_identifier : ] coverpoint
    expression [ iff ( expression ) ] bins_or_empty

bins_or_empty ::=
      { { covergroup_coverpoint_body_item } }
    | ;

covergroup_coverpoint_body_item ::=
      covergroup_option
    | covergroup_coverpoint_binspec

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

```
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

covergroup_expression ::= expression

covergroup_cross ::=
    covercross_identifier : cross coverpoint_identifier
      { , coverpoint_identifier }[ iff ( expression ) ] cross_item_or_null

cross_item_or_null ::=
      { { covergroup_cross_body_item } }
    | ;

covergroup_cross_body_item ::=
      covergroup_option
    | covergroup_cross_binspec

covergroup_cross_binspec ::= bins_keyword identifier = covercross_identifier
   with ( covergroup_expression ) ;
```

B.16 Conditional compilation

```
package_body_compile_if ::= compile if ( constant_expression )
   package_body_compile_if_item [ else package_body_compile_if_item ]

package_body_compile_if_item ::=
      package_body_item
    | { { package_body_item } }

action_body_compile_if ::= compile if ( constant_expression )
   action_body_compile_if_item [ else action_body_compile_if_item ]

action_body_compile_if_item ::=
      action_body_item
    | { { action_body_item } }

component_body_compile_if ::= compile if ( constant_expression )
   component_body_compile_if_item [ else component_body_compile_if_item ]

component_body_compile_if_item ::=
      component_body_item
    | { { component_body_item } }
```

```
struct_body_compile_if ::= compile if ( constant_expression )
    struct_body_compile_if_item [ else struct_body_compile_if_item ]

struct_body_compile_if_item ::=
       struct_body_item
     | { { struct_body_item } }

compile_has_expr ::= compile has ( static_ref_path )

compile_assert_stmt ::=
     compile assert ( constant_expression [ , string_literal ] ) ;
```

B.17 Expressions

```
constant_expression ::= expression

expression ::=
       primary
     | unary_operator primary
     | expression binary_operator expression
     | conditional_expression
     | in_expression

unary_operator ::= - | ! | ~ | & | | | ^

binary_operator ::=
       * | / | % | + | - | << | >> | == | != | < | <= | > | >= | || | && | |
     | ^ | & | **

assign_op ::= = | += | -= | <<= | >>= | |= | &=

conditional_expression ::= cond_predicate ? expression : expression

cond_predicate ::= expression

in_expression ::=
       expression in [ open_range_list ]
     | expression in collection_expression

open_range_list ::= open_range_value { , open_range_value }

open_range_value ::= expression [ .. expression ]

collection_expression ::= expression

primary ::=
       number
     | aggregate_literal
     | bool_literal
     | string_literal
     | null_ref
     | paren_expr
     | cast_expression
     | ref_path
     | compile_has_expr
```

```
paren_expr ::= ( expression )

cast_expression ::= ( casting_type ) expression

ref_path ::=
      static_ref_path [ . hierarchical_id ] [ bit_slice ]
    | [ super . ] hierarchical_id [ bit_slice ]

static_ref_path ::= [ :: ] { type_identifier_elem :: } member_path_elem

bit_slice ::= [ constant_expression : constant_expression ]

function_call ::=
      super . function_ref_path
    | [ :: ] { type_identifier_elem :: } function_ref_path

function_ref_path ::= { member_path_elem . } identifier function_parameter_list

symbol_call ::= symbol_identifier function_parameter_list ;

function_parameter_list ::= ( [ expression { , expression } ] )
```

B.18 Identifiers

```
identifier ::=
      ID
    | ESCAPED_ID

hierarchical_id_list ::= hierarchical_id { , hierarchical_id }

hierarchical_id ::= member_path_elem { . member_path_elem }

member_path_elem ::= identifier [ function_parameter_list ] [ [ expression ] ]

action_identifier ::= identifier

component_identifier ::= identifier

covercross_identifier ::= identifier

covergroup_identifier ::= identifier

coverpoint_identifier ::= identifier

enum_identifier ::= identifier

function_identifier ::= identifier

import_class_identifier ::= identifier

index_identifier ::= identifier

iterator_identifier ::= identifier

label_identifier ::= identifier
```

```
language_identifier ::= identifier

package_identifier ::= identifier

struct_identifier ::= identifier

symbol_identifier ::= identifier

type_identifier ::= [ :: ] type_identifer_elem { :: type_identifer_elem }

type_identifier_elem ::= identifier [ template_param_value_list ]

action_type_identifier ::= type_identifier

buffer_type_identifier ::= type_identifier

component_type_identifier ::= type_identifier

covergroup_type_identifier ::= type_identifier

enum_type_identifier ::= type_identifier

resource_type_identifier ::= type_identifier

state_type_identifier ::= type_identifier

stream_type_identifier ::= type_identifier

entity_type_identifier ::=
      action_type_identifier
    | component_type_identifier
    | flow_object_type
    | resource_object_type
```

B.19 Numbers and literals

```
number ::=
      oct_number
    | dec_number
    | hex_number
    | based_bin_number
    | based_oct_number
    | based_dec_number
    | based_hex_number

bin_digit ::= [0-1]

oct_digit ::= [0-7]

dec_digit ::= [0-9]

hex_digit ::= [0-9] | [a-f] | [A-F]

oct_number ::= 0 { oct_digit | _ }

dec_number ::= [1-9] { dec_digit | _ }
```

```
hex_number ::= 0[x|X] hex_digit { hex_digit | _ }

BASED_BIN_LITERAL ::= '[s|S]b|B bin_digit { bin_digit | _ }

BASED_OCT_LITERAL ::= '[s|S]o|O oct_digit { oct_digit | _ }

BASED_DEC_LITERAL ::= '[s|S]d|D dec_digit { dec_digit | _ }

BASED_HEX_LITERAL ::= '[s|S]h|H hex_digit { hex_digit | _ }

based_bin_number ::= [ dec_number ] BASED_BIN_LITERAL

based_oct_number ::= [ dec_number ] BASED_OCT_LITERAL

based_dec_number ::= [ dec_number ] BASED_DEC_LITERAL

based_hex_number ::= [ dec_number ] BASED_HEX_LITERAL

aggregate_literal ::=
      empty_aggregate_literal
    | value_list_literal
    | map_literal
    | struct_literal

empty_aggregate_literal ::= { }

value_list_literal ::= { expression { , expression } }

map_literal ::= { map_literal_item { , map_literal_item } }

map_literal_item ::= expression : expression

struct_literal ::= { struct_literal_item { , struct_literal_item } }

struct_literal_item ::= . identifier = expression

bool_literal ::=
      true
    | false

null_ref ::= null
```

B.20 Additional lexical conventions

```
SL_COMMENT ::= //{any_ASCII_character_except_newline}\n

ML_COMMENT ::= /*{any_ASCII_character}*/

string_literal ::=
      QUOTED_STRING
    | TRIPLE_QUOTED_STRING

QUOTED_STRING ::= " { unescaped_character | escaped_character } "

unescaped_character ::= any_printable_ASCII_character
```

```
escaped_character ::= \('|"|?|\|a|b|f|n|r|t|v|[0-7][0-7][0-7])

TRIPLE_QUOTED_STRING ::= """{any_ASCII_character}"""

filename_string ::= QUOTED_STRING

ID ::= [a-z]|[A-Z]|_ {[a-z]|[A-Z]|_|[0-9]}

ESCAPED_ID ::= \{any_printable_ASCII_character_except_whitespace} whitespace

whitespace ::= space | tab | newline | end_of_file
```

Annex C

(normative)

C++ header files

This annex contains the header files for the C++ input. If there is a conflict between a C++ class declaration shown anywhere in this standard and the material in this annex, the material shown in this annex shall take precedence.

C.1 File pss.h

```
#pragma once
#include "pss/scope.h"
#include "pss/type_decl.h"
#include "pss/bit.h"
#include "pss/cond.h"
#include "pss/vec.h"
#include "pss/enumeration.h"
#include "pss/chandle.h"
#include "pss/width.h"
#include "pss/range.h"
#include "pss/attr.h"
#include "pss/rand_attr.h"
#include "pss/component.h"
#include "pss/comp_inst.h"
#include "pss/covergroup.h"
#include "pss/covergroup_bins.h"
#include "pss/covergroup_coverpoint.h"
#include "pss/covergroup_cross.h"
#include "pss/covergroup_iff.h"
#include "pss/covergroup_inst.h"
#include "pss/covergroup_options.h"
#include "pss/structure.h"
#include "pss/buffer.h"
#include "pss/stream.h"
#include "pss/state.h"
#include "pss/resource.h"
#include "pss/lock.h"
#include "pss/share.h"
#include "pss/symbol.h"
#include "pss/action.h"
#include "pss/input.h"
#include "pss/output.h"
#include "pss/constraint.h"
#include "pss/in.h"
#include "pss/unique.h"
#include "pss/default_value.h"
#include "pss/default_disable.h"
#include "pss/action_handle.h"
#include "pss/action_attr.h"
#include "pss/pool.h"
#include "pss/bind.h"
#include "pss/exec.h"
#include "pss/foreach.h"
```

```
#include "pss/forall.h"
#include "pss/if_then.h"
#include "pss/function.h"
#include "pss/import_class.h"
#include "pss/export_action.h"
#include "pss/extend.h"
#include "pss/override.h"
#include "pss/ctrl_flow.h"
```

C.2 File pss/action.h

```
#pragma once
#include <vector>
#include "pss/detail/actionBase.h"
#include "pss/detail/algebExpr.h"
#include "pss/detail/activityBase.h"
#include "pss/detail/Stmt.h"
#include "pss/detail/sharedExpr.h"
#include "pss/detail/comp_ref.h"
namespace pss {
  class component; // forward declaration
  /// Declare an action
  class action : public detail::ActionBase {
  protected:
    /// Constructor
    action ( const scope& s );
    /// Destructor
    ~action();
  public:
    template <class T=component> detail::comp_ref<T> comp();
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
    /// Declare an activity
    class activity : public detail::ActivityBase {
    public:
      // Constructor
      template < class... R >
      activity(R&&... /* detail::Stmt */ r);
      // Destructor
      ~activity();
    };
      // Specifies the guard condition for a select branch
      class guard {
      public:
          guard(const detail::AlgebExpr &cond);
      };
      // Specifies the weight for a select branch
      class weight {
      public:
          weight(const detail::AlgebExpr &w);
      };
    class branch {
    public:
      // Specifies a select-branch statement with no guard
      // condition and no weight
      template <class R> branch(
                  const R& /* detail::Stmt */ r);
      // Specifies a select-branch statement with a guard
      // condition and no weight
      template <class R> branch(const guard &g,
                  const R& /* detail::Stmt */ r);
      // Specifies a select-branch statement with both a
      // guard condition and a weight
      template <class R> branch(
                  const guard                       &g,
                  const weight                      &w,
                  const R& /* detail::Stmt */ r);
      // Specifies a select-branch statement with a weight and
      // no guard condition
      template <class R> branch(
                  const weight                      &w,
                  const R& /* detail::Stmt */ r);
    };
    // select() must be inside action declaration to disambiguate
    // from built in select()
    /// Declare a select statement
    class select : public detail::Stmt {
    public:
      template < class... R >
      select(R&&... /* detail::Stmt|branch */ r);
    };
    /// Declare a schedule block
    class schedule : public detail::Stmt {
    public:
      // Constructor
      template < class... R >
      schedule(R&&... /* detail::Stmt */ r);
    };
    /// Declare a parallel block
    class parallel : public detail::Stmt {
    public:
      // Constructor
      template < class... R >
      parallel(R&&... /* detail::Stmt */ r);
    };
    /// Declare a replicate statement
    class replicate : public detail::Stmt {
    public:
      /// Declare a replicate statement
      replicate( const detail::AlgebExpr& count,
                 const detail::Stmt& activity);
      /// Declare a replicate statement with iterator variable
      replicate( const attr<int>& iter,
                 const detail::AlgebExpr& count,
                 const detail::Stmt& activity );
    };
  }; // class action
}; // namespace pss
#include "pss/timpl/action.t"
```

C.3 File pss/action_attr.h

```
#pragma once
#include "pss/rand_attr.h"
namespace pss {
  template < class T >
  class action_attr : public rand_attr<T> {
  public:
    /// Constructor
    action_attr (const scope& name);
    /// Constructor defining width
    action_attr (const scope& name, const width& a_width);
    /// Constructor defining range
    action_attr (const scope& name, const range& a_range);
    /// Constructor defining width and range
    action_attr (const scope& name, const width& a_width,
                 const range& a_range);
  };
}; // namespace pss
#include "pss/timpl/action_attr.t"
```

C.4 File pss/action_handle.h

```
#pragma once
#include "pss/detail/actionHandleBase.h"
#include "pss/detail/algebExpr.h"
namespace pss {
  /// Declare an action handle
  template<class T>
  class action_handle : public detail::ActionHandleBase {
  public:
    action_handle();
    action_handle(const scope& name);
    action_handle(const action_handle<T>& a_action_handle);
    template <class... R> action_handle<T> with (
             const R&... /* detail::AlgebExpr */ constraints );
    T* operator-> ();
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/action_handle.t"
```

C.5 File pss/attr.h

```
#pragma once

#include <string>
#include <memory>
#include <list>

#include "pss/bit.h"
#include "pss/vec.h"
#include "pss/scope.h"
#include "pss/width.h"
#include "pss/range.h"
```

```
#include "pss/structure.h"
#include "pss/component.h"

#include "pss/detail/attrTBase.h"
#include "pss/detail/attrIntBase.h"
#include "pss/detail/attrBitBase.h"
#include "pss/detail/attrStringBase.h"
#include "pss/detail/attrBoolBase.h"
#include "pss/detail/attrCompBase.h"

#include "pss/detail/attrVecTBase.h"
#include "pss/detail/attrVecIntBase.h"
#include "pss/detail/attrVecBitBase.h"
#include "pss/detail/attrVecStringBase.h"

#include "pss/detail/algebExpr.h"
#include "pss/detail/execStmt.h"

namespace pss {
  template <class T>
  class rand_attr; // forward reference

  /// Primary template for enums and structs
  template < class T>
  class attr : public detail::AttrTBase {
  public:
    /// Constructor
    attr (const scope& s);
    /// Constructor with initial value
    attr (const scope& s, const T& init_val);
    /// Constructor defining range
    attr (const scope& s, const range& a_range);
    /// Constructor defining range and (const) initial value
    attr (const scope& s, const range& a_range, const T& init_val);
    /// Copy constructor
    attr(const attr<T>& other);
    /// Struct access
    T* operator-> ();
    /// Struct access
    T& operator* ();
    /// Enumerator access
    T& val();
    /// Exec statement assignment
    detail::ExecStmt operator= (const detail::AlgebExpr& value);
  };

  /// Template specialization for int
  template <>
  class attr<int> : public detail::AttrIntBase {
  public:
    /// Constructor
    attr (const scope& s);
    /// Constructor with (const) initial value
    attr (const scope& s, const int& init_val);
    /// Constructor with (non const) initial expression
    attr (const scope& s, const detail::AlgebExpr& init_val);
    /// Constructor defining width
    attr (const scope& s, const width& a_width);
    /// Constructor defining width and (const) initial value
    attr (const scope& s, const width& a_width, const int& init_val);
    /// Constructor defining width and (non const) initial expression
    attr (const scope& s, const width& a_width, const detail::AlgebExpr&
   init_val);
    /// Constructor defining range
    attr (const scope& s, const range& a_range);
    /// Constructor defining range and (const) initial value
    attr (const scope& s, const range& a_range,
          const int& init_val);
    /// Constructor defining range and (non const) initial expression
    attr (const scope& s, const range& a_range, const detail::AlgebExpr&
   init_val);
    /// Constructor defining width and range
    attr (const scope& s, const width& a_width,
          const range& a_range);
    /// Constructor defining width and range and (const) initial value
    attr (const scope& s, const width& a_width,
          const range& a_range,
          const int& init_val);
   /// Constructor defining width and range and (non const) initial expression
    attr (const scope& s, const width& a_width,
          const range& a_range,
          const detail::AlgebExpr& init_val);
    /// Copy constructor
    attr(const attr<int>& other);
    /// Access to underlying data
    int& val();
    /// Exec statement assignment
    detail::ExecStmt operator= (const detail::AlgebExpr& value);
    detail::ExecStmt operator+= (const detail::AlgebExpr& value);
    detail::ExecStmt operator-= (const detail::AlgebExpr& value);
    detail::ExecStmt operator<<= (const detail::AlgebExpr& value);
    detail::ExecStmt operator>>= (const detail::AlgebExpr& value);
    detail::ExecStmt operator&= (const detail::AlgebExpr& value);
    detail::ExecStmt operator|= (const detail::AlgebExpr& value);
  };


  /// Template specialization for bit
  template <>
  class attr<bit> : public detail::AttrBitBase {
  public:
    /// Constructor
    attr (const scope& s);
    /// Constructor with (const) initial value
    attr (const scope& s, const bit& init_val);
    /// Constructor with (non const) initial expression
    attr (const scope& s, const detail::AlgebExpr& init_val);
    /// Constructor defining width
    attr (const scope& s, const width& a_width);
    /// Constructor defining width and (const) initial value
    attr (const scope& s, const width& a_width, const bit& init_val);
    /// Constructor defining width and (non const) initial expression
    attr (const scope& s, const width& a_width, const detail::AlgebExpr&
   init_val);
    /// Constructor defining range
    attr (const scope& s, const range& a_range);
    /// Constructor defining range and (const) initial value
    attr (const scope& s, const range& a_range,
          const bit& init_val);
    /// Constructor defining range and (non const) initial expression
    attr (const scope& s, const range& a_range, const detail::AlgebExpr&
   init_val);
    /// Constructor defining width and range
    attr (const scope& s, const width& a_width,
          const range a_range);
    /// Constructor defining width and range and (const) initial value
    attr (const scope& s, const width& a_width,
          const range& a_range,
          const bit& init_val);
   /// Constructor defining width and range and (non const) initial expression
    attr (const scope& s, const width& a_width,
          const range& a_range,
          const detail::AlgebExpr& init_val);
    /// Copy constructor
    attr(const attr<bit>& other);
    /// Access to underlying data
    bit& val();
    /// Exec statement assignment
    detail::ExecStmt operator= (const detail::AlgebExpr& value);
    detail::ExecStmt operator+= (const detail::AlgebExpr& value);
    detail::ExecStmt operator-= (const detail::AlgebExpr& value);
    detail::ExecStmt operator<<= (const detail::AlgebExpr& value);
    detail::ExecStmt operator>>= (const detail::AlgebExpr& value);
    detail::ExecStmt operator&= (const detail::AlgebExpr& value);
    detail::ExecStmt operator|= (const detail::AlgebExpr& value);
  };


  /// Template specialization for string
  template <>
  class attr<std::string> : public detail::AttrStringBase {
  public:
    /// Constructor
    attr (const scope& s);
    /// Constructor and (const) initial value
    attr (const scope& s, const std::string& init_val);
    /// Constructor with (non const) initial expression
    attr (const scope& s, const detail::AlgebExpr& init_val);
    /// Constructor defining range
    attr (const scope& s, const range& a_range);
    /// Constructor defining range and (const) initial value
    attr (const scope& s, const range& a_range,
          const std::string& init_val);
    /// Constructor defining range and (non const) initial expression
    attr (const scope& s, const range& a_range, const detail::AlgebExpr&
   init_val);
    /// Copy constructor
    attr(const attr<std::string>& other);
    /// Access to underlying data
    std::string& val();
    /// Exec statement assignment
    detail::ExecStmt operator= (const detail::AlgebExpr& value);
  };

  /// Template specialization for bool
  template <>
  class attr<bool> : public detail::AttrBoolBase {
```

```
public:
  /// Constructor
  attr (const scope& s);
  /// Constructor and (const) initial value
  attr (const scope& s, const bool init_val);
  /// Constructor with (non const) initial expression
  attr (const scope& s, const detail::AlgebExpr& init_val);
  /// Copy constructor
  attr(const attr<bool>& other);
  /// Access to underlying data
  bool& val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
};

/// Template specialization for component*
template <>
class attr<component*> : public detail::AttrCompBase {
public:
  /// Copy constructor
  attr(const attr<component*>& other);
  /// Access to underlying data
  component* val();
};


/// Template specialization for array of ints
template <>
class attr<vec<int>> : public detail::AttrVecIntBase {
public:
  /// Constructor defining array size
  attr(const scope& name, const std::size_t count);
  /// Constructor defining array size and element width
  attr(const scope& name, const std::size_t count,
       const width& a_width);
  /// Constructor defining array size and element range
  attr(const scope& name, const std::size_t count,
       const range& a_range);
  /// Constructor defining array size and element width and range
  attr(const scope& name, const std::size_t count,
       const width& a_width, const range& a_range);
  /// Access to specific element
  attr<int>& operator[](const std::size_t idx);
  /// Constraint on randomized index
  detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
  /// Get size of array
  std::size_t size() const;
  /// Constraint on sum of array
  detail::AlgebExpr sum() const;
};

/// Template specialization for array of bits
template <>
class attr<vec<bit>> : public detail::AttrVecBitBase {
public:
  /// Constructor defining array size
  attr(const scope& name, const std::size_t count);
  /// Constructor defining array size and element width
  attr(const scope& name, const std::size_t count,
          const width& a_width);
      /// Constructor defining array size and element range
      attr(const scope& name, const std::size_t count,
          const range& a_range);
      /// Constructor defining array size and element width and range
      attr(const scope& name, const std::size_t count,
          const width& a_width, const range& a_range);
      /// Access to specific element
      attr<bit>& operator[](const std::size_t idx);
      /// Constraint on randomized index
      detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
      /// Get size of array
      std::size_t size() const;
      /// Constraint on sum of array
      detail::AlgebExpr sum() const;
    };

    /// Template specialization for array of strings
    template <>
    class attr<vec<std::string>> : public detail::AttrVecStringBase {
    public:
      /// Constructor defining array size
      attr(const scope& name, const std::size_t count);
      /// Constructor defining array size and element range
      attr(const scope& name, const std::size_t count,
          const range& a_range);
      /// Access to specific element
      attr<std::string>& operator[](const std::size_t idx);
      /// Constraint on randomized index
      detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
      /// Get size of array
      std::size_t size() const;
    };

    /// Template specialization for arrays of enums and arrays of structs
    template <class T>
    class attr<vec<T>> : public detail::AttrVecTBase {
    public:
      attr(const scope& name, const std::size_t count);
      attr(const scope& name, const std::size_t count, const range& a_range);
      attr<T>& operator[](const std::size_t idx);
      detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
      std::size_t size() const;

    };

    template < class T >
    using attr_vec = attr< vec <T> >;

  }; // namespace pss

  #include "pss/timpl/attr.t"
```

C.6 File pss/bind.h

```
#pragma once
#include "pss/pool.h"
#include "pss/detail/bindBase.h"
#include "pss/detail/ioBase.h"
namespace pss {
  /// Declare a bind
  class bind : public detail::BindBase {
  public:
    /// Bind a type to multiple targets
    template <class R /*type*/, typename... T /*targets*/ >
    bind (const pool<R>& a_pool, const T&... targets);
    /// Explicit binding of action inputs and outputs
    template <class... R>
    bind ( const R&... /* input|output|lock|share */ io_items );
    /// Destructor
    ~bind();
  };
}; // namespace pss
#include "pss/timpl/bind.t"
```

C.7 File pss/bit.h

```
#pragma once
namespace pss {
  using bit = unsigned int;
}; // namespace pss
```

C.8 File pss/buffer.h

```
#pragma once
#include "pss/detail/bufferBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a buffer object
  class buffer : public detail::BufferBase {
  protected:
    /// Constructor
    buffer (const scope& s);
    /// Destructor
    ~buffer();
  public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
  };
}; // namespace pss
```

C.9 File pss/chandle.h

```
#pragma once
#include "pss/detail/algebExpr.h"
#include "pss/detail/chandleBase.h"
namespace pss {
  class chandle : public detail::ChandleBase {
  public:
    chandle& operator= ( detail::AlgebExpr val );
  };
};
```

C.10 File pss/comp_inst.h

```
#pragma once
#include "pss/detail/compInstBase.h"
#include "pss/detail/compInstVecBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a component instance
  template<class T>
  class comp_inst : public detail::CompInstBase {
  public:
    /// Constructor
    comp_inst (const scope& s);
    /// Copy Constructor
    comp_inst (const comp_inst& other);
    /// Destructor
    ~comp_inst();
    /// Access content
    T* operator-> ();
    /// Access content
    T& operator* ();
  };
  /// Template specialization for array of components
  template<class T>
  class comp_inst<vec<T> > : public detail::CompInstVecBase {
  public:
    comp_inst(const scope& name, const std::size_t count);
    comp_inst<T>& operator[](const std::size_t idx);
    std::size_t size() const;
  };
  template < class T >
  using comp_inst_vec = comp_inst< vec <T> >;
}; // namespace pss
#include "pss/timpl/comp_inst.t"
```

C.11 File pss/component.h

```
#pragma once
#include "pss/detail/componentBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a component
  class component : public detail::ComponentBase {
  protected:
    /// Constructor
    component (const scope& s);
    /// Copy Constructor
    component (const component& other);
    /// Destructor
    ~component();
  public:
    /// In-line exec block
    virtual void init_down();
    virtual void init_up();
    virtual void init();
  };
}; // namespace pss
```

C.12 File pss/cond.h

```
#pragma once
namespace pss {
namespace detail {
      class AlgebExpr;
}
class cond {
public:
      cond(const detail::AlgebExpr &c);
};
}
```

C.13 File pss/constraint.h

```
#pragma once
#include <vector>
#include "pss/detail/constraintBase.h"
namespace pss {
  namespace detail {
    class AlgebExpr;            // forward reference
  }
  /// Declare a member constraint
  class constraint : public detail::ConstraintBase {
  public:
    /// Declare an unnamed member constraint
    template <class... R> constraint (
      const R&&... /*detail::AlgebExpr*/ expr
      );
    /// Declare a named member constraint
    template <class... R> constraint (
      const std::string& name,
      const R&&... /*detail::AlgebExpr*/ expr
      );
  };
  /// Declare a dynamic member constraint
  class dynamic_constraint : public detail::DynamicConstraintBase {
  public:
    /// Declare a named dynamic member constraint
    template <class... R> dynamic_constraint (
      const std::string& name,
      const R&&... /*detail::AlgebExpr*/ expr
      );
  };
}; // namespace pss
#include "pss/timpl/constraint.t"
```

C.14 File pss/covergroup.h

```
#pragma once
#include <stdint.h>
#include <string>
#include "pss/scope.h"
namespace pss {
class covergroup {
public:
      covergroup(const scope &s);
      virtual ~covergroup();
};
}
```

C.15 File pss/covergroup_bins.h

```
#pragma once
#include <string>
#include "pss/covergroup.h"
#include "pss/range.h"
#include "pss/covergroup_coverpoint.h"
namespace pss {
namespace detail {
    class AlgebExpr;
}
template <class T> class bins {
public:
};
template <> class bins<int> {
public:
    // default bins
    bins(const std::string      &name);
    bins(
        const std::string       &name,
        const range             &ranges);
    bins(
        const std::string       &name,
        const coverpoint        &cp);
    const bins<int> &with(const detail::AlgebExpr &expr);
};
template <> class bins<bit> {
public:
    // default bins
    bins(const std::string      &name);
    bins(
        const std::string       &name,
        const range             &ranges);
    bins(
        const std::string       &name,
        const coverpoint        &cp);
    bins(
        const std::string       &name,
        const rand_attr<bit>    &var);
    bins(
        const std::string       &name,
        const attr<bit>         &var);
    const bins<bit> &with(const detail::AlgebExpr &expr);
};
template <> class bins<vec<int>> {
public:
    // default bins
    bins(
        const std::string       &name,
        uint32_t                size);
    bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    bins(
        const std::string       &name,
        const range             &ranges);
    bins(
        const std::string       &name,
        const coverpoint        &cp);
    bins(
        const std::string       &name,
        const rand_attr<int>    &var);
    bins(
        const std::string       &name,
        const attr<int>         &var);
    const bins<vec<int>> &with(const detail::AlgebExpr &expr);
};
template <> class bins<vec<bit>> {
public:
    // default bins
    bins(
        const std::string       &name);
    bins(
        const std::string       &name,
        const range             &ranges);
    bins(
        const std::string       &name,
        const coverpoint        &cp);

    bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    bins(
        const std::string       &name,
        const rand_attr<bit>    &var);
    bins(
        const std::string       &name,
        const attr<bit>         &var);
    bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    const bins<vec<bit>> &with(const detail::AlgebExpr &expr);
};
template <class T> class ignore_bins {
public:
};
template <> class ignore_bins<int> {
public:
    // default bins
    ignore_bins(const std::string   &name);
    ignore_bins(
        const std::string       &name,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        const coverpoint        &cp);
    const ignore_bins<int> &with(const detail::AlgebExpr &expr);
};
template <> class ignore_bins<bit> {
public:
    // default bins
    ignore_bins(const std::string       &name);
    ignore_bins(
        const std::string       &name,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        const coverpoint        &cp);
    const ignore_bins<bit> &with(const detail::AlgebExpr &expr);
};
template <> class ignore_bins<vec<int>> {
public:
    ignore_bins(const std::string   &name);
    ignore_bins(
        const std::string       &name,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        const coverpoint        &cp);
    ignore_bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    const ignore_bins<vec<int>> &with(const detail::AlgebExpr &expr);
};
template <> class ignore_bins<vec<bit>> {
public:
    // default bins
    ignore_bins(const std::string       &name);
    ignore_bins(
        const std::string       &name,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        const coverpoint        &cp);
    ignore_bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    ignore_bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    const ignore_bins<vec<bit>> &with(const detail::AlgebExpr &expr);
};
template <class T> class illegal_bins {
public:
};
template <> class illegal_bins<int> {
public:
    // Default bins
    illegal_bins(const std::string      &name);
    illegal_bins(
        const std::string       &name,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        const coverpoint        &cp);
    const illegal_bins<int> &with(const detail::AlgebExpr &expr);
};
template <> class illegal_bins<bit> {
public:
    // Default bins
    illegal_bins(const std::string      &name);
    illegal_bins(
        const std::string       &name,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        const coverpoint        &cp);
    const illegal_bins<bit> &with(const detail::AlgebExpr &expr);
};
template <> class illegal_bins<vec<int>> {
public:
    // Default bins
    illegal_bins(const std::string      &name);
    illegal_bins(
        const std::string       &name,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        const coverpoint        &cp);
    illegal_bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    const illegal_bins<vec<int>> &with(const detail::AlgebExpr &expr);
};
```

```
template <> class illegal_bins<vec<bit>> {
public:
    // Default bins
    illegal_bins(const std::string      &name);
    illegal_bins(
        const std::string       &name,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        const coverpoint        &cp);
    illegal_bins(
        const std::string       &name,
        uint32_t                size,
        const range             &ranges);
    illegal_bins(
        const std::string       &name,
        uint32_t                size,
        const coverpoint        &cp);
    const illegal_bins<vec<bit>> &with(const detail::AlgebExpr &expr);
};
}
```

C.16 File pss/covergroup_coverpoint.h

```
#pragma once
#include "pss/covergroup.h"
#include "pss/covergroup_options.h"
#include "pss/covergroup_iff.h"
#include "pss/detail/algebExpr.h"
#include "pss/detail/coverpointBase.h"
namespace pss {
namespace detail {
      class AlgebExpr;
}
class coverpoint : public detail::CoverPointBase {
public:
   template <class... T> coverpoint(
         const std::string         &name,
         const detail::AlgebExpr   &target,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const std::string         &name,
         const detail::AlgebExpr   &target,
         const iff                 &cp_iff,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const std::string         &name,
         const detail::AlgebExpr   &target,
         const options             &cp_options,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const std::string         &name,
         const detail::AlgebExpr   &target,
         const iff                 &cp_iff,
         const options             &cp_options,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const detail::AlgebExpr   &target,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const detail::AlgebExpr   &target,
         const iff                 &cp_iff,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const detail::AlgebExpr   &target,
         const options             &cp_options,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
   template <class... T> coverpoint(
         const detail::AlgebExpr   &target,
         const iff                 &cp_iff,
         const options             &cp_options,
         const T&... /*iff|bins|ignore_bins|illegal_bins */ bin_items);
};
}
```

C.17 File pss/covergroup_cross.h

```
#pragma once
#include "pss/covergroup.h"
#include "pss/covergroup_options.h"
#include "pss/covergroup_iff.h"
#include "pss/covergroup_coverpoint.h"
namespace pss {
class cross : public coverpoint {
public:
   template <class... T> cross(
         const std::string      &name,
         const T&...
    /*coverpoint|attr|rand_attr|bins|ignore_bins|illegal_bins */    items);
   template <class... T> cross(
         const std::string      &name,
         const iff            &cp_iff,
         const T&...
    /*coverpoint|attr|rand_attr|bins|ignore_bins|illegal_bins */    items);
   template <class... T> cross(
         const std::string      &name,
         const options         &cp_options,
         const T&...
    /*coverpoint|attr|rand_attr|bins|ignore_bins|illegal_bins */    items);
   template <class... T> cross(
         const std::string      &name,
         const iff            &cp_iff,
         const options         &cp_options,
         const T&...
    /*coverpoint|attr|rand_attr|bins|ignore_bins|illegal_bins */    items);
};
}
```

C.18 File pss/covergroup_iff.h

```
#pragma once
#include "pss/detail/algebExpr.h"
namespace pss {
class iff {
public:
   iff(const detail::AlgebExpr &expr);
};
}
```

C.19 File pss/covergroup_inst.h

```
#pragma once
#include "covergroup.h"
#include "covergroup_options.h"
#include <functional>
namespace pss {
template <class T=covergroup> class covergroup_inst {
public:
   covergroup_inst(
         const std::string      &name,
         const options         &opts);
   template <class... R> covergroup_inst(
         const std::string      &name,
         const options         &opts,
         const R&...            ports);
   template <class... R> covergroup_inst(
         const std::string      &name,
         const R&...            ports);
};
template <> class covergroup_inst<covergroup> {
public:
   template <class... R> covergroup_inst(
         const std::string         &name,
         std::function<void(void)>   body);
};
}
```

C.20 File pss/covergroup_options.h

```
#pragma once
#include "covergroup.h"
namespace pss {
class weight {
public:
   weight(uint32_t w);
};
class goal {
public:
   goal(uint32_t w);
};
class name {
public:
   name(const std::string &name);
};
class comment {
public:
   comment(const std::string &name);
};
class detect_overlap {
public:
   detect_overlap(bool l);
};
class at_least {
public:
   at_least(uint32_t w);
};
class auto_bin_max {
public:
   auto_bin_max(uint32_t m);
};
class per_instance {
public:
   per_instance(bool is_per_instance);
};
class options {
public:
   template <class... O> options(
         const O&... /*
            weight
            | goal
            | name
            | comment
            | detect_overlap
            | at_least
            | auto_bin_max
            | per_instance */ options);
};
class type_options {
public:
   template <class... O> type_options(
         const O&... /*
            weight
            | goal
            | comment */ options);
};
}
```

C.21 File pss/ctrl_flow.h

```
#pragma once

#include "pss/detail/Stmt.h"

namespace pss {

    class choice {
    public:
        // Specifies a case-branch statement
        template <class... R>
        choice( const range &range,
               R&&... /*detail::Stmt|std::function<void(void)>|sequence&&*/
   stmts) { /* Implementation specific */ }
    };

    class default_choice {
    public:
        template <class... R>
        default_choice( R&&... /* detail::Stmt | std::function<void(void)> |
   sequence&&*/ stmts) { /* Implementation specific */ }
    };

    class match : public detail::Stmt {
    public:

        template <class... R>
        match( const cond &c,
               R&&... /* choice|default_choice */ stmts);

    };

    /// Declare a sequence block
    class sequence : public detail::Stmt {
    public:
      // Constructor
      template < class... R >
    sequence(R&&... /* detail::Stmt|detail::ExecStmt */ r) { /* Implementation
   specific */ }
    };

    /// Declare a repeat statement
    class repeat : public detail::Stmt {
    public:
      /// Declare a repeat statement
      repeat(const detail::AlgebExpr& count,
             const detail::Stmt& activity
        );

      /// Declare a repeat statement
      repeat(const attr<int>& iter,
             const detail::AlgebExpr& count,
             const detail::Stmt& activity
        );

      /// Declare a procedural repeat (count) statement
      repeat(const detail::AlgebExpr& count,
             std::function<void(void)> loop_stmts
        );

      /// Declare a procedural repeat (count) statement with iterator
      repeat(const attr<int>& iter,
             const detail::AlgebExpr& count,
             std::function<void(void)> loop_stmts
        );
    };

    /// Declare a repeat-while statement
    class repeat_while : public detail::Stmt {
    public:
      /// Declare a repeat-while statement
```

```
      repeat_while(const detail::Stmt& activity,
                   const cond& a_cond
        );

      /// Declare a procedural repeat-while statement
      repeat_while(std::function<void(void)> loop_stmts,
                   const cond& a_cond
        );
    };

    /// Declare a while_do statement
    class while_do : public detail::Stmt {
    public:
      /// Declare a procedural while_do statement
      while_do(const cond& a_cond,
               const detail::Stmt& loop_stmts
        );

      /// Declare a procedural while_do statement
      while_do(const cond& a_cond,
               std::function<void(void)> loop_stmts
        );
    };

    /// Declare pss_return
    class pss_return {
    public :
       // Constructor
       pss_return(void);
       // Constructor
       pss_return(const detail::AlgebExpr& expr);
    };

    /// Declare pss_break
    class pss_break {
    public :
       // Constructor
       pss_break(void);
    };

    /// Declare pss_continue
    class pss_continue {
    public :
       // Constructor
       pss_continue(void);
    };

} // namespace pss
```

C.22 File pss/default_disable.h

```
#pragma once
#include "pss/detail/algebExpr.h"
namespace pss {
   /// Declare default disable constraint
   template < class T >
   class default_disable : public detail::AlgebExpr {
```

```
   public:
      default_disable (const rand_attr<T>& attribute);
   };
}; // namespace pss
```

C.23 File pss/default_value.h

```
#pragma once
#include "pss/detail/algebExpr.h"
namespace pss {
   /// Declare default value constraint
   template < class T >
   class default_value : public detail::AlgebExpr {
   public:
      default_value (const rand_attr<T>& attribute,
                     const detail::AlgebExpr& default_expr);
   };
}; // namespace pss
```

C.24 File pss/enumeration.h

```
#pragma once
#include "pss/detail/enumerationBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare an enumeration
  class enumeration : public detail::EnumerationBase {
  public:
    /// Constructor
    enumeration ( const scope& s);
    /// Default Constructor
    enumeration ();
    /// Destructor
    ~enumeration ();
  protected:
    class __pss_enum_values {
    public:
      __pss_enum_values (enumeration* context, const std::string& s);
    };
    template <class T>
    enumeration& operator=( const T& t);
  };
}; // namespace pss
#define PSS_ENUM(class_name, ...)                                   \
class class_name : public enumeration {                             \
public:                                                             \
  class_name (const scope& s) : enumeration (this){}                \
                                                                    \
  enum __pss_##class_name {                                         \
    __VA_ARGS__                                                     \
      };                                                            \
                                                                    \
  __pss_enum_values __pss_enum_values_ {this, #__VA_ARGS__};        \
                                                                    \
  class_name() {}                                                   \
  class_name (const __pss_##class_name e) {                         \
    enumeration::operator=(e);                                 \
  }                                                            \
                                                               \
  class_name& operator=(const __pss_##class_name e){           \
    enumeration::operator=(e);                                 \
    return *this;                                              \
  }                                                            \
}
#define PSS_EXTEND_ENUM(ext_name, base_name, ...)              \
class ext_name : public base_name {                            \
public:                                                        \
  ext_name (const scope& s) : base_name (this){}               \
                                                               \
  enum __pss_##ext_name {                                      \
    __VA_ARGS__                                                \
      };                                                       \
                                                               \
  __pss_enum_values __pss_enum_values_ {this, #__VA_ARGS__};   \
                                                               \
  ext_name() {}                                                \
  ext_name (const __pss_##ext_name e) {                        \
    enumeration::operator=(e);                                 \
  }                                                            \
                                                               \
  ext_name& operator=(const __pss_##ext_name e){               \
    enumeration::operator=(e);                                 \
    return *this;                                              \
  }                                                            \
};                                                             \
extend_enum<base_name, ext_name> __pss_##ext_name
#include "pss/timpl/enumeration.t"
```

C.25 File pss/exec.h

```
#pragma once
#include <functional>
#include "pss/detail/execBase.h"
#include "pss/detail/attrCommon.h"
namespace pss {
  class sequence; // forward declaration
  /// Declare an exec block
  class exec : public detail::ExecBase {
  public:
    /// Types of exec blocks
    enum ExecKind {
      run_start,
      header,
      declaration,
      init_down,
      init_up,
      init,
      pre_solve,
      post_solve,
      body,
      run_end,
      file
    };
    /// Declare inline exec
    exec(
      ExecKind kind,
      std::initializer_list<detail::AttrCommon>&& write_vars
      );
    /// Declare target template exec
    exec(
      ExecKind kind,
      const char* language_or_file,
      const char* target_template );
    exec(
      ExecKind kind,
      std::string&& language_or_file,
      std::string&& target_template );
    /// Declare native exec - with single exec statement
    exec(
      ExecKind kind,
      const detail::ExecStmt& r
      );

    /// Declare native exec - with single AlgebExpr statement
    exec(
      ExecKind kind,
      const detail::AlgebExpr& r
      );
    /// Declare native exec - with sequence statement
    exec(
      ExecKind kind,
      const detail::Stmt& /* sequence & */ r
      );
    /// Declare generative procedural-interface exec
    exec(
      ExecKind kind,
      std::function<void()> genfunc
      );
    /// Declare generative target-template exec
    exec(
      ExecKind kind,
      std::string&& language_or_file,
      std::function<void(std::ostream&)> genfunc
      );
  };
}; // namespace pss
#include "pss/timpl/exec.t"
```

C.26 File pss/export_action.h

```
#pragma once
#include <vector>
#include "pss/scope.h"
#include "pss/bit.h"
#include "pss/width.h"
#include "pss/range.h"
#include "pss/detail/exportActionParam.h"
namespace pss {
  class export_action_base {
  public:
    // Export action kinds
    enum kind { solve, target };
    template <class T> class in : public detail::ExportActionParam {
    public:
    };
  };
  /// Declare an export action
  template <class T=int> class export_action
    : public export_action_base {
  public:
    using export_action_base::in;
    export_action(
      const std::vector<detail::ExportActionParam> &params ) {};
    export_action(
      kind,
      const std::vector<detail::ExportActionParam> &params ) {};
  };
  template <> class export_action_base::in<bit>
    : public detail::ExportActionParam {
  public:
    in(const scope &name) {};
    in(const scope &name, const width &w) {};
    in(const scope &name, const range &rng) {};
    in(const scope &name, const width &w, const range &rng) {};
  };
  template <> class export_action_base::in<int>
    : public detail::ExportActionParam {
  public:
    in(const scope &name) {};
    in(const scope &name, const width &w) {};
    in(const scope &name, const range &rng) {};
    in(const scope &name, const width &w, const range &rng) {};
  };
  template <> class export_action_base::in<std::string>
    : public detail::ExportActionParam {
  public:
    in(const scope &name) {};
    in(const scope &name, const range &rng) {};
  };
}
```

C.27 File pss/extend.h

```
  #pragma once
  namespace pss {
    /// Extend a structure
    template < class Foundation, class Extension>
    class extend_structure  {
    public:
      extend_structure();
    };
    /// Extend an action
    template < class Foundation, class Extension>
    class extend_action  {
    public:
      extend_action();
    };
    /// Extend a component
    template < class Foundation, class Extension>
    class extend_component  {
    public:
      extend_component();
    };
    /// Extend an enum
    template < class Foundation, class Extension>
    class extend_enum {
    public:
      extend_enum();
    };
  }; // namespace pss
  #include "pss/timpl/extend.t"
```

C.28 File pss/forall.h

```
#pragma once
#include "pss/detail/sharedExpr.h"
#include "pss/iterator.h"
namespace pss {
   /// Declare a forall constraint item
   template < class T >
   class forall : public detail::SharedExpr {
   public:
      forall ( const iterator<T>& iter_var,
               const detail::AlgebExpr& constraint
        );
   };
}; // namespace pss
#include "pss/timpl/forall.t"
```

C.29 File pss/foreach.h

```
#pragma once
#include "pss/bit.h"
#include "pss/vec.h"
#include "pss/detail/sharedExpr.h"
namespace pss {
  template <class T> class attr; // forward declaration
  template <class T> class rand_attr; // forward declaration
  namespace detail {
    class AlgebExpr;            // forward reference
    class Stmt;         // forward reference
  };
  /// Declare a foreach statement
  class foreach : public detail::SharedExpr {
  public:
    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<int>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<bit>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach activity statement
    template < class T >
    foreach( const attr<int>& iter,
             const rand_attr<vec<T>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const attr<vec<int>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const attr<vec<bit>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach activity statement
    template < class T >
    foreach( const attr<int>& iter,
             const attr<vec<T>>& array,
             const detail::Stmt& activity
      );
    /// Declare a foreach constraint statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<int>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Declare a foreach constraint statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<bit>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Declare a foreach constraint statement
    template < class T >
    foreach( const attr<int>& iter,
             const rand_attr<vec<T>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Declare a foreach constraint statement
    foreach( const attr<int>& iter,
             const attr<vec<int>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Declare a foreach constraint statement
    foreach( const attr<int>& iter,
             const attr<vec<bit>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Declare a foreach constraint statement
    template < class T >
    foreach( const attr<int>& iter,
             const attr<vec<T>>& array,
             const detail::AlgebExpr& constraint
      );
    /// Disambiguate a foreach sharedExpr statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<int>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Disambiguate a foreach sharedExpr statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<bit>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Disambiguate a foreach sharedExpr statement
    template < class T >
    foreach( const attr<int>& iter,
             const rand_attr<vec<T>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Disambiguate a foreach sharedExpr statement
    foreach( const attr<int>& iter,
             const attr<vec<int>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Disambiguate a foreach sharedExpr statement
    foreach( const attr<int>& iter,
             const attr<vec<bit>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Disambiguate a foreach sharedExpr statement
    template < class T >
    foreach( const attr<int>& iter,
             const attr<vec<T>>& array,
             const detail::SharedExpr& sharedExpr
      );
    /// Declare a foreach procedural construct
    foreach( const attr<int> &iter,
             const attr<vec<int>> &array,
             std::function<void(void)> loop_stmts
      );
    foreach( const attr<int> &iter,
             const rand_attr<vec<int>> &array,
             std::function<void(void)> loop_stmts
      );

    /// Declare a foreach procedural construct
    foreach( const attr<int> &iter,
             const attr<vec<bit>> &array,
             std::function<void(void)> loop_stmts
      );
    foreach( const attr<int> &iter,
             const rand_attr<vec<bit>> &array,
             std::function<void(void)> loop_stmts
      );

    /// Declare a foreach procedural construct
    template < class T >
    foreach( const attr<T> &iter,
             const attr<vec<T>> &array,
             std::function<void(void)> loop_stmts
      );
    template < class T >
    foreach( const attr<T> &iter,
             const rand_attr<vec<T>> &array,
             std::function<void(void)> loop_stmts
```

```
      );
    };
  }; // namespace pss
  #include "pss/timpl/foreach.t"
```

C.30 File pss/function.h

```
#pragma once
#include "pss/scope.h"
#include "pss/bit.h"
#include "pss/width.h"
#include "pss/range.h"
#include "pss/detail/FunctionParam.h"
#include "pss/detail/FunctionResult.h"
#include "pss/attr.h"
namespace pss {
  template <class T> class arg;
  template <class T> class in_arg;
  template <class T> class out_arg;
  template <class T> class inout_arg;
  template <class T> class result;
  /// Import function availability
  enum kind { solve, target };
  template<typename T> class function;
  template<typename R, typename... Args>
  class function<R(Args...)> {
  public:
    // CTOR for the case with no procedural specification
    function(const scope &name
             , R result
             , Args... args
      );
    // CTOR for the case with pure modifier and no procedural specification
    function(const scope &name
             , bool is_pure
             , R result
             , Args... args
      );
    template <class... T> R operator() (
      const T&... /* detail::AlgebExpr */ params);
    /// Declare target-template function
    function(const scope   &name
             , const std::string &language
             , R result
             , Args... args
             , const std::string &target_template
      );
     // Declare function specified procedurally
     function(const scope   &name
             , R result
             , Args... args
             , std::function<void(Args...)> ast_builder
      );
     // Declare function specified procedurally with pure modifier
     function(const scope   &name
             , bool is_pure
             , R result
             , Args... args
             , std::function<void(Args...)> ast_builder
      );
  };
  template<typename T> class import_func;
  template<typename R, typename... Args>
  class import_func<function<R(Args...)>> {
  public:
    /// Declare import function availability
    import_func(const scope   &name
                , const kind a_kind
      );
    /// Declare import function language
    import_func(const scope   &name
                , const std::string &language
      );
    /// Declare import function language and availability
    import_func(const scope   &name
                , const kind a_kind
                , const std::string &language
      );
    template <class... T> R operator() (
      const T&... /* detail::AlgebExpr */ params);
  };
  // Some simplifications when R = result<void>
  template<typename... Args>
  class function<result<void>(Args...)> {
  public:
    // CTOR for the case with no procedural specification
    function(const scope &name
             , Args... args
      );
    // CTOR for the case with pure modifier and no procedural specification
    function(const scope &name
             , bool is_pure
             , Args... args
      );
    template <class... T> result<void> operator() (
      const T&... /* detail::AlgebExpr */ params);
    /// Declare target-template function
    function(const scope   &name
             , const std::string &language
             , Args... args
             , const std::string &target_template
      );
     // Declare function specified procedurally
     function(const scope   &name
             , Args... args
             , std::function<void(Args...)> ast_builder
      );
     // Declare function specified procedurally with pure modifier
     function(const scope   &name
             , bool is_pure
             , Args... args
             , std::function<void(Args...)> ast_builder
      );
};
  template<typename... Args>
  class import_func<function<result<void>(Args...)>> {
```

```
public:
  /// Declare import function availability
  import_func(const scope   &name
              , const kind a_kind
    );
  /// Declare import function language
  import_func(const scope   &name
              , const std::string &language
    );
  /// Declare import function language and availability
  import_func(const scope   &name
              , const kind a_kind
              , const std::string &language
    );
  template <class... T> result<void> operator() (
    const T&... /* detail::AlgebExpr */ params);
};
/// Template specialization for arg
template <> class arg<bit> : public detail::FunctionParam, public attr<bit> {
public:
  arg(const scope &name);
  arg(const scope &name, const width &w);
  arg(const scope &name, const range &rng);
  arg(const scope &name, const width &w, const range &rng);
  using attr<bit>::operator=;
};

template <> class arg<int> : public detail::FunctionParam,public attr<int> {
public:
  arg(const scope &name);
  arg(const scope &name, const width &w);
  arg(const scope &name, const range &rng);
  arg(const scope &name, const width &w, const range &rng);
  using attr<int>::operator=;
};

template <> class arg<attr_vec<bit>> : public detail::FunctionParam, public
 attr_vec<bit> {
public:
  arg(const scope& name, const std::size_t count);
  arg(const scope &name, const std::size_t count, const width &w);
  arg(const scope &name, const std::size_t count, const range &rng);
  arg(const scope &name, const std::size_t count, const width &w,
                                                   const range &rng);
};

template <> class arg<attr_vec<int>> : public detail::FunctionParam, public
 attr_vec<int> {
public:
  arg(const scope& name, const std::size_t count);
  arg(const scope &name, const std::size_t count, const width &w);
  arg(const scope &name, const std::size_t count, const range &rng);
  arg(const scope &name, const std::size_t count, const width &w,
                                                   const range &rng);
};
/// Template specialization for inputs
template <> class in_arg<bit> : public detail::FunctionParam {
public:
  in_arg(const scope &name);
  in_arg(const scope &name, const width &w);
  in_arg(const scope &name, const range &rng);
  in_arg(const scope &name, const width &w, const range &rng);
  in_arg(const scope &name, const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const width &w,
                            const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const range &rng,
                            const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const width &w, const range &rng,
                            const detail::AlgebExpr &default_param);
};
template <> class in_arg<int> : public detail::FunctionParam {
public:
  in_arg(const scope &name);
  in_arg(const scope &name, const width &w);
  in_arg(const scope &name, const range &rng);
  in_arg(const scope &name, const width &w, const range &rng);
  in_arg(const scope &name, const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const width &w,
                            const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const range &rng,
                            const detail::AlgebExpr &default_param);
  in_arg(const scope &name, const width &w, const range &rng,
                            const detail::AlgebExpr &default_param);
};
/// Template specialization for outputs
template <> class out_arg<bit> : public detail::FunctionParam {
public:
  out_arg(const scope &name);
  out_arg(const scope &name, const width &w);
  out_arg(const scope &name, const range &rng);
  out_arg(const scope &name, const width &w, const range &rng);
};
template <> class out_arg<int> : public detail::FunctionParam {
public:
  out_arg(const scope &name);
  out_arg(const scope &name, const width &w);
  out_arg(const scope &name, const range &rng);
  out_arg(const scope &name, const width &w, const range &rng);
};
/// Template specialization for inout_args
template <> class inout_arg<bit> : public detail::FunctionParam {
public:
  inout_arg(const scope &name);
  inout_arg(const scope &name, const width &w);
  inout_arg(const scope &name, const range &rng);
  inout_arg(const scope &name, const width &w, const range &rng);
};
template <> class inout_arg<int> : public detail::FunctionParam {
public:
  inout_arg(const scope &name);
  inout_arg(const scope &name, const width &w);
  inout_arg(const scope &name, const range &rng);
  inout_arg(const scope &name, const width &w, const range &rng);
};
/// Template specialization for results
template <> class result<bit> : public detail::FunctionResult {
public:
  result();
```

```
    result(const width &w);
    result(const range &rng);
    result(const width &w, const range &rng);
  };
  template <> class result<int> : public detail::FunctionResult {
  public:
    result();
    result(const width &w);
    result(const range &rng);
    result(const width &w, const range &rng);
  };
  template <> class result<void> : public detail::FunctionResult {
  public:
    result();
  };
}
#include "pss/timpl/function.t"
```

C.31 File pss/if_then.h

```
#pragma once
#include "pss/detail/sharedExpr.h"
#include <functional>
namespace pss {
  class sequence; // forward declaration
  namespace detail {
    class AlgebExpr;            // forward reference
    class Stmt;         // forward reference
  };
  /// Declare if-then statement
  class if_then : public detail::SharedExpr {
  public:
    /// Declare if-then activity statement
    if_then (const cond& a_cond,
             const detail::Stmt& true_expr
      );
    /// Declare if-then constraint statement
    if_then (const cond& a_cond,
             const detail::AlgebExpr& true_expr
      );
    /// Disambiguate if-then sharedExpr statement
    if_then (const cond& a_cond,
             const detail::SharedExpr& true_expr
      );
    ///Declare if-then procedural statement
    if_then (const cond& a_cond,
             std::function<void(void)> true_stmts
      );
  };
  /// Declare if-then-else statement
  class if_then_else : public detail::SharedExpr {
  public:
    /// Declare if-then-else activity statement
    if_then_else (const cond& a_cond,
                  const detail::Stmt& true_expr,
                  const detail::Stmt& false_expr
      );
    /// Declare if-then-else constraint statement
    if_then_else (const cond& a_cond,
                  const detail::AlgebExpr& true_expr,
                  const detail::AlgebExpr& false_expr
      );
    /// Disambiguate if-then-else sharedExpr activity statement
    if_then_else (const cond& a_cond,
                  const detail::SharedExpr& true_expr,
                  const detail::Stmt& false_expr
      );
    /// Disambiguate if-then-else sharedExpr activity statement
    if_then_else (const cond& a_cond,
                  const detail::Stmt& true_expr,
                  const detail::SharedExpr& false_expr
      );
    /// Disambiguate if-then-else sharedExpr constraint statement
    if_then_else (const cond& a_cond,
                  const detail::SharedExpr& true_expr,
                  const detail::AlgebExpr& false_expr
      );
    /// Disambiguate if-then-else sharedExpr constraint statement
    if_then_else (const cond& a_cond,
                  const detail::AlgebExpr& true_expr,
                  const detail::SharedExpr& false_expr
      );
    /// Disambiguate if-then-else sharedExpr statement
    if_then_else (const cond& a_cond,
                  const detail::SharedExpr& true_expr,
                  const detail::SharedExpr& false_expr
      );
    /// Declare if-then-else procedural statement
    if_then_else (const cond& a_cond,
                  std::function<void(void)> true_stmts,
                  std::function<void(void)> false_stmts
      );

    /// Declare if-then-else procedural statement
    if_then_else (const cond& a_cond,
                  const detail::Stmt& /* sequence & */ true_stmts,
                  std::function<void(void)> false_stmts
      );

    /// Declare if-then-else procedural statement
    if_then_else (const cond& a_cond,
                  std::function<void(void)> true_stmts,
                  const detail::Stmt& /* sequence & */ false_stmts
      );
  };
}; // namespace pss
```

C.32 File pss/import_class.h

```
#pragma once
#include "pss/scope.h"
#include "pss/detail/importClassBase.h"
namespace pss {
  /// Declare an import class
  class import_class : public detail::ImportClassBase {
  public:
    /// Constructor
    import_class(const scope &name);
    /// Destructor
    ~import_class();
  };
}
```

C.33 File pss/in.h

```
#pragma once
#include "pss/range.h"
#include "pss/attr.h"
#include "pss/rand_attr.h"
namespace pss  {
  /// Declare a set membership
  class in : public detail::AlgebExpr {
  public:
    in ( const attr<int>& a_var,
         const range& a_range
      );
    in ( const attr<bit>& a_var,
         const range& a_range
      );
    in ( const rand_attr<int>& a_var,
         const range& a_range
      );
    in ( const rand_attr<bit>& a_var,
         const range& a_range
      );
    template < class T>
    in ( const rand_attr<T>& a_var,
         const range& a_range
      );
  };
}; // namespace pss
```

C.34 File pss/input.h

```
#pragma once
#include "pss/detail/inputBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare an action input
  template<class T>
  class input : public detail::InputBase {
  public:
    /// Constructor
    input (const scope& s);
    /// Destructor
    ~input();
    /// Access content
    T* operator-> ();
    /// Access content
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/input.t"
```

C.35 File pss/iterator.h

```
#pragma once
#include "pss/detail/attrTBase.h"

namespace pss {
   /// Declare an action handle
   template<class T>
   class iterator : public detail::AttrTBase {
   public:
      iterator (const scope& name);
   };
}; // namespace pss
#include "pss/timpl/iterator.t"
```

C.36 File pss/lock.h

```
#pragma once
#include "pss/detail/lockBase.h"
namespace pss {
  /// Claim a locked resource
  template<class T>
  class lock : public detail::LockBase {
  public:
    /// Constructor
    lock(const scope& name);
    /// Destructor
    ~lock();
    /// Access content
    T* operator-> ();
    /// Access content
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/lock.t"
```

C.37 File pss/output.h

```
#pragma once
#include "pss/detail/outputBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare an action output
  template<class T>
  class output : public detail::OutputBase {
  public:
    /// Constructor
    output (const scope& s);
    /// Destructor
    ~output();
    /// Access content
    T* operator-> ();
    /// Access content
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/output.t"
```

C.38 File pss/override.h

```
#pragma once
namespace pss {
  /// Override a type
  template < class Foundation, class Override>
  class override_type  {
  public:
    override_type();
  };
  /// Override an instance
  template < class Override >
  class override_instance {
  public:
    /// Override an instance of a structure
    template <class T>
    override_instance ( const attr<T>& inst);
    /// Override an instance of a rand structure
    template <class T>
    override_instance ( const rand_attr<T>& inst);
    /// Override an instance of a component instance
    template <class T>
    override_instance ( const comp_inst<T>& inst);
    /// Override an action instance
    template <class T>
    override_instance ( const action_handle<T>& inst);
  };
}; // namespace pss
#include "pss/timpl/override.t"
```

C.39 File pss/pool.h

```
#pragma once
#include <string>
#include "pss/detail/poolBase.h"
namespace pss {
  /// Declare a pool
  template <class T>
  class pool : public detail::PoolBase {
  public:
    /// Constructor
    pool (const scope& name, std::size_t count = 1);
    /// Destructor
    ~pool();
  };
}; // namespace pss
#include "pss/timpl/pool.t"
```

C.40 File pss/rand_attr.h

```
#pragma once

#include <string>
#include <memory>
#include <list>

#include "pss/bit.h"
#include "pss/vec.h"
#include "pss/scope.h"
#include "pss/width.h"
#include "pss/range.h"
#include "pss/structure.h"

#include "pss/detail/randAttrTBase.h"
#include "pss/detail/randAttrIntBase.h"
#include "pss/detail/randAttrBitBase.h"
#include "pss/detail/randAttrStringBase.h"
#include "pss/detail/randAttrBoolBase.h"
#include "pss/detail/randAttrCompBase.h"

#include "pss/detail/randAttrVecTBase.h"
#include "pss/detail/randAttrVecIntBase.h"
#include "pss/detail/randAttrVecBitBase.h"
#include "pss/detail/randAttrVecStringBase.h"

#include "pss/detail/algebExpr.h"
#include "pss/detail/execStmt.h"

namespace pss {

  template <class T>
  class attr; // forward reference

  /// Primary template for enums and structs
  template <class T>
  class rand_attr : public detail::RandAttrTBase {
  public:
    /// Constructor
    rand_attr (const scope& name);
    /// Constructor defining range
    rand_attr (const scope& name, const range& a_range);
    /// Copy constructor
    rand_attr(const rand_attr<T>& other);
    /// Struct access
    T* operator-> ();
    /// Struct access
    T& operator* ();
    /// Enumerator access
    T& val();
    /// Exec statement assignment
    detail::ExecStmt operator= (const detail::AlgebExpr& value);
  };

  /// Template specialization for rand int
  template <>
  class rand_attr<int> : public detail::RandAttrIntBase {
```

```
public:
  /// Constructor
  rand_attr (const scope& name);
  /// Constructor defining width
  rand_attr (const scope& name, const width& a_width);
  /// Constructor defining range
  rand_attr (const scope& name, const range& a_range);
  /// Constructor defining width and range
 rand_attr (const scope& name, const width& a_width, const range& a_range);
  /// Copy constructor
  rand_attr(const rand_attr<int>& other);
  /// Access to underlying data
  int& val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
  detail::ExecStmt operator+= (const detail::AlgebExpr& value);
  detail::ExecStmt operator-= (const detail::AlgebExpr& value);
  detail::ExecStmt operator<<= (const detail::AlgebExpr& value);
  detail::ExecStmt operator>>= (const detail::AlgebExpr& value);
  detail::ExecStmt operator&= (const detail::AlgebExpr& value);
  detail::ExecStmt operator|= (const detail::AlgebExpr& value);
};

/// Template specialization for rand bit
template <>
class rand_attr<bit> : public detail::RandAttrBitBase {
public:
  /// Constructor
  rand_attr (const scope& name);
  /// Constructor defining width
  rand_attr (const scope& name, const width& a_width);
  /// Constructor defining range
  rand_attr (const scope& name, const range& a_range);
  /// Constructor defining width and range
 rand_attr (const scope& name, const width& a_width, const range& a_range);
  /// Copy constructor
  rand_attr(const rand_attr<bit>& other);
  /// Access to underlying data
  bit& val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
  detail::ExecStmt operator+= (const detail::AlgebExpr& value);
  detail::ExecStmt operator-= (const detail::AlgebExpr& value);
  detail::ExecStmt operator<<= (const detail::AlgebExpr& value);
  detail::ExecStmt operator>>= (const detail::AlgebExpr& value);
  detail::ExecStmt operator&= (const detail::AlgebExpr& value);
  detail::ExecStmt operator|= (const detail::AlgebExpr& value);
};


/// Template specialization for rand string
template <>
class rand_attr<std::string> : public detail::RandAttrStringBase {
public:
  /// Constructor
  rand_attr (const scope& name);
  /// Constructor defining range
  rand_attr (const scope& s, const range& a_range);
  /// Copy constructor
  rand_attr(const rand_attr<std::string>& other);
  /// Access to underlying data
  std::string& val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
};

/// Template specialization for rand bool
template <>
class rand_attr<bool> : public detail::RandAttrBoolBase {
public:
  /// Constructor
  rand_attr (const scope& name);
  /// Copy constructor
  rand_attr(const rand_attr<bool>& other);
  /// Access to underlying data
  bool val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
};


/// Template specialization for array of rand ints
template <>
class rand_attr<vec<int>> : public detail::RandAttrVecIntBase {
public:
  /// Constructor defining array size
  rand_attr(const scope& name, const std::size_t count);
  /// Constructor defining array size and element width
  rand_attr(const scope& name, const std::size_t count,
            const width& a_width);
  /// Constructor defining array size and element range
  rand_attr(const scope& name, const std::size_t count,
            const range& a_range);
  /// Constructor defining array size and element width and range
  rand_attr(const scope& name, const std::size_t count,
            const width& a_width, const range& a_range);
  /// Access to specific element
  rand_attr<int>& operator[](const std::size_t idx);
  /// Constraint on randomized index
  detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
  /// Get size of array
  std::size_t size() const;
  /// Constraint on sum of array
  detail::AlgebExpr sum() const;
};

/// Template specialization for array of rand bits
template <>
class rand_attr<vec<bit>> : public detail::RandAttrVecBitBase {
public:
  /// Constructor defining array size
  rand_attr(const scope& name, const std::size_t count);
  /// Constructor defining array size and element width
  rand_attr(const scope& name, const std::size_t count,
            const width& a_width);
  /// Constructor defining array size and element range
  rand_attr(const scope& name, const std::size_t count,
            const range& a_range);
```

```
    /// Constructor defining array size and element width and range
    rand_attr(const scope& name, const std::size_t count,
              const width& a_width, const range& a_range);
    /// Access to specific element
    rand_attr<bit>& operator[](const std::size_t idx);
    /// Constraint on randomized index
    detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
    /// Get size of array
    std::size_t size() const;
    /// Constraint on sum of array
    detail::AlgebExpr sum() const;
  };

  /// Template specialization for array of rand strings
  template <>
  class rand_attr<vec<std::string>> : public detail::RandAttrVecStringBase {
  public:
    /// Constructor defining array size
    rand_attr(const scope& name, const std::size_t count);
    /// Constructor defining array size and element range
    rand_attr(const scope& name, const std::size_t count,
         const range& a_range);
    /// Access to specific element
    rand_attr<std::string>& operator[](const std::size_t idx);
    /// Constraint on randomized index
    detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
    /// Get size of array
    std::size_t size() const;
  };

 // Template specialization for arrays of rand enums and arrays of rand structs
  template <class T>
  class rand_attr<vec<T>> : public detail::RandAttrVecTBase {
  public:
    rand_attr(const scope& name, const std::size_t count);
    rand_attr(const scope& name, const std::size_t count, const range&
   a_range);
    rand_attr<T>& operator[](const std::size_t idx);
    detail::AlgebExpr operator[](const detail::AlgebExpr& idx);
    std::size_t size() const;

  };

  template < class T >
  using rand_attr_vec = rand_attr< vec <T> >;

}; // namespace pss

#include "pss/timpl/rand_attr.t"
```

C.41 File pss/range.h

```
#pragma once
#include <vector>
#include "pss/detail/rangeBase.h"
namespace pss {
  class Lower {
```

```
  public:
  };
  // Used to specify a range that is bounded
  // by the domain minimum
  const Lower lower;
  class Upper {
  public:
  };
  // Used to specify a range that is bounded
  // by the domain maximum
  const Upper       upper;
  /// Declare domain of a numeric attribute
  class range : public detail::RangeBase {
  public:
    /// Declare a range of values
    range (const detail::AlgebExpr& lhs, const detail::AlgebExpr& rhs);
    range (const Lower& lhs, const detail::AlgebExpr& rhs);
    range (const detail::AlgebExpr& lhs, const Upper& rhs);
    /// Declare a single value
    range (const detail::AlgebExpr& value);
    /// Copy constructor
    range ( const range& a_range);
    /// Function chaining to declare another range of values
    range& operator() (const detail::AlgebExpr& lhs, const detail::AlgebExpr&
    rhs);
    /// Function chaining to declare another single value
    range& operator() (const detail::AlgebExpr& value);
  }; // class range
}; // namespace pss
```

C.42 File pss/resource.h

```
#pragma once
#include "pss/detail/resourceBase.h"
#include "pss/scope.h"
#include "pss/rand_attr.h"
namespace pss {
  /// Declare a resource object
  class resource : public detail::ResourceBase {
  protected:
    /// Constructor
    resource (const scope& s);
    /// Destructor
    ~resource();
  public:
    /// Get the instance id of this resource
    rand_attr<int> instance_id;
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
  };
}; // namespace pss
```

C.43 File pss/scope.h

```
#pragma once
#include <string>
#include "pss/detail/scopeBase.h"
namespace pss {
  /// Class to manage PSS object hierarchy introspection
  class scope : public detail::ScopeBase {
  public:
    /// Constructor
    scope (const char* name);
    /// Constructor
    scope (const std::string& name);
    /// Constructor
    template < class T > scope (T* s);
    /// Destructor
    ~scope();
  };
}; // namespace pss
/*! Convenience macro for PSS constructors */
#define PSS_CTOR(C,P) public: C (const scope& p) : P (this) {}
#include "pss/timpl/scope.t"
```

C.44 File pss/share.h

```
#pragma once
#include "pss/detail/shareBase.h"
namespace pss {
  /// Claim a shared resource
  template<class T>
  class share : public detail::ShareBase {
  public:
    /// Constructor
    share(const scope& name);
    /// Destructor
    ~share();
    /// Access content
    T* operator-> ();
    /// Access content
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/share.t"
```

C.45 File pss/state.h

```
#pragma once
#include "pss/detail/stateBase.h"
#include "pss/scope.h"
#include "pss/rand_attr.h"
namespace pss {
  /// Declare a state object
  class state : public detail::StateBase {
  protected:
    /// Constructor
    state (const scope& s);
    /// Destructor
    ~state();
  public:
    /// Test if this is the initial state
    rand_attr<bool> initial;
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
  };
  /// Return previous state of a state object
  template <class T>
  T* prev(T* this_);
}; // namespace pss
#include "pss/timpl/state.t"
```

C.46 File pss/stream.h

```
#pragma once
#include "pss/detail/streamBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a stream object
  class stream : public detail::StreamBase {
  protected:
    /// Constructor
    stream (const scope& s);
    /// Destructor
    ~stream();
  public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
  };
}; // namespace pss
```

C.47 File pss/structure.h

```
#pragma once
#include "pss/detail/structureBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a structure
  class structure : public detail::StructureBase {
  protected:
    /// Constructor
    structure (const scope& s);
    /// Destructor
    ~structure();
  public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
  };
}; // namespace pss
```

C.48 File pss/symbol.h

```
#pragma once
namespace pss {
  namespace detail {
    class Stmt;  // forward reference
  };
  using symbol = detail::Stmt;
};
```

C.49 File pss/type_decl.h

```
#pragma once
#include "pss/detail/typeDeclBase.h"
namespace pss {
  template<class T>
  class type_decl : public detail::TypeDeclBase {
  public:
    type_decl();
    T* operator-> ();
    T& operator* ();
  };
}; // namespace pss
#include "pss/timpl/type_decl.t"
```

C.50 File pss/unique.h

```
#pragma once
#include <iostream>
#include <vector>
#include <cassert>
#include "pss/range.h"
#include "pss/vec.h"
#include "pss/detail/algebExpr.h"
namespace pss  {
  /// Declare an unique constraint
  class unique : public detail::AlgebExpr {
  public:
    /// Declare unique constraint
    template < class ... R >
    unique ( R&&... /* rand_attr<T> */ r );
  };
}; // namespace pss
#include "pss/timpl/unique.t"
```

C.51 File pss/vec.h

```
#pragma once
#include <vector>
namespace pss {
  template < class T>
  using vec = std::vector <T>;
};
```

C.52 File pss/width.h

```
#pragma once
#include "pss/detail/widthBase.h"
namespace pss {
  /// Declare width of a numeric attribute
  class width : public detail::WidthBase {
  public:
    /// Declare width as a range of bits
    width (const std::size_t& lhs, const std::size_t& rhs);
    /// Declare width in bits
    width (const std::size_t& size);
    /// copy constructor
    width (const width& a_width);
  };
}; // namespace pss
```

C.53 File pss/detail/algebExpr.h

```
#pragma once
#include <iostream>
#include <vector>
#include <cassert>
#include "pss/range.h"
#include "pss/vec.h"
#include "pss/comp_inst.h"
#include "pss/component.h"
#include "pss/detail/exprBase.h"
#include "pss/detail/sharedExpr.h"
namespace pss  {
  template <class T> class attr; // forward declaration
  template <class T> class rand_attr; // forward declaration
  class coverpoint; // forward declaration
  class dynamic_constraint; // forward declaration
  template <class T> class result; // forward declaration
  class coverpoint; // forward declaration
  namespace detail {
    template <class T> class comp_ref; // forward declaration
    /// Construction of algebraic expressions
    class AlgebExpr : public ExprBase {
    public:
      /// Default constructor
      AlgebExpr();
      AlgebExpr(const coverpoint &cp);
      /// Recognize a rand_attr<>
      template < class T >
      AlgebExpr(const rand_attr<T>& value);
      /// Recognize an attr<>
      template < class T >
      AlgebExpr(const attr<T>& value);
      /// Recognize a range() for in()
      AlgebExpr(const range& value);
      /// Recognize a comp_inst<>
      template < class T >
      AlgebExpr(const comp_inst<T>& value);
      /// Recognize a comp_ref<>
      template <class T>
      AlgebExpr(const comp_ref<T> &value);
      /// Recognize a CompInstBase
      AlgebExpr(const CompInstBase& value);
      // Allow dynamic constraints to be referenced
      // in constraint expressions
      AlgebExpr(const dynamic_constraint &c);
      // /// Capture other values
      // template < class T >
      // AlgebExpr(const T& value);
      /// Recognize integers
      AlgebExpr(const int& value);
      /// Recognize strings
      AlgebExpr(const char* value);
      AlgebExpr(const std::string& value);
      /// Recognize shared constructs
      AlgebExpr(const SharedExpr& value);
      /// Recognize function return values
      template < class T >
      AlgebExpr(const result<T>& value);
    };
    /// Logical Or Operator
    const AlgebExpr operator|| ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Logical And Operator
    const AlgebExpr operator&& ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Bitwise Or Operator
    const AlgebExpr operator| ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Bitwise And Operator
    const AlgebExpr operator& ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Xor Operator
    const AlgebExpr operator^ ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Less Than  Operator
    const AlgebExpr operator< ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Less than or Equal  Operator
    const AlgebExpr operator<= ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Greater Than  Operator
    const AlgebExpr operator> ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Greater than or Equal  Operator
    const AlgebExpr operator>= ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Right Shift  Operator
    const AlgebExpr operator>> ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Left Shift  Operator
    const AlgebExpr operator<< ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Multiply Operator
    const AlgebExpr operator* ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Divide Operator
    const AlgebExpr operator/ ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Modulus  Operator
    const AlgebExpr operator% ( const AlgebExpr& lhs, const AlgebExpr& rhs);
```

```
    /// Add Operator
    const AlgebExpr operator+ ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Subtract Operator
    const AlgebExpr operator- ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Equal Operator
    const AlgebExpr operator== ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Not Equal Operator
    const AlgebExpr operator!= ( const AlgebExpr& lhs, const AlgebExpr& rhs);
    /// Unary bang Operator
    const AlgebExpr operator!(const AlgebExpr &e);
    /// Unary minus Operator
    const AlgebExpr operator-(const AlgebExpr &e);
    /// Unary tilde Operator
    const AlgebExpr operator~(const AlgebExpr &e);
    /// Unary AND reduction operator
    const AlgebExpr and_reduce(const AlgebExpr &e);
    /// Unary OR reduction operator
    const AlgebExpr or_reduce(const AlgebExpr &e);
    /// Unary XOR reduction operator
    const AlgebExpr xor_reduce(const AlgebExpr &e);
    const AlgebExpr pow(const AlgebExpr& base, const AlgebExpr &exp);
  }; // namespace detail
}; // namespace pss
#include "algebExpr.t"
```

C.54 File pss/detail/comp_ref.h

```
#pragma once
namespace pss {
   namespace detail {
   template <class T> class comp_ref {
   public:
      T* operator -> ();
   };
   }
}
```

C.55 File pss/detail/FunctionParam.h

```
#pragma once
namespace pss {
  namespace detail {
    class FunctionParam {
    };
  }; // namespace detail
}; // namespace pss
```

C.56 File pss/detail/FunctionResult.h

```
#pragma once
namespace pss {
  namespace detail {
    class FunctionResult {
    };
  }; // namespace detail
}; // namespace pss
```

C.57 File pss/detail/Stmt.h

```
#pragma once
#include<vector>
#include "pss/action_handle.h"
#include "pss/action_attr.h"
#include "pss/constraint.h"
#include "algebExpr.h"
#include "sharedExpr.h"
namespace pss {
  class bind;
  namespace detail {
    class Stmt
    {
    public:
      /// Recognize action_handle<>
      template<class T>
      Stmt(const action_handle<T>& value);
      /// Recognize action_attr<>
      template<class T>
      Stmt(const action_attr<T>& value);
      /// Recognize dynamic_constraint
      Stmt(const dynamic_constraint& value);
      /// Recognize constraint as an activity statement
      Stmt(const constraint& b);
      /// Recognize shared constructs
      Stmt(const SharedExpr& other);
      /// Recognize bind as an activity statement
      Stmt(const bind& b);
      // Default Constructor
      Stmt();
    };
  }; // namespace detail
}; // namespace pss
#include "Stmt.t"
```

Annex D

(normative)

Core library package

This annex contains the contents of the built-in core library packages **executor_pkg** and **addr_reg_pkg** described in Clause 24. If there is a conflict between core library package contents shown anywhere in this standard and the material in this annex, the material shown in this annex shall take precedence.

D.1 Package executor_pkg

```
package executor_pkg {

   struct executor_trait_s {};

   struct empty_executor_trait_s : executor_trait_s {};

   component executor_base_c {};

   component executor_c
                <struct TRAIT : executor_trait_s = empty_executor_trait_s>
                : executor_base_c {
     TRAIT trait;
   };

   component executor_group_c
                <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
      function void add_executor(ref executor_c<TRAIT> exe);
   };

   struct executor_claim_s
                <struct TRAIT : executor_trait_s = empty_executor_trait_s> {
      rand TRAIT trait;
   };

   function ref executor_base_c executor();
}
```

D.2 Package addr_reg_pkg

```
package addr_reg_pkg {
   import executor_pkg::* ;

   component addr_space_base_c {};

    struct addr_trait_s {};

    struct empty_addr_trait_s : addr_trait_s {};

    typedef chandle addr_handle_t;
```

```
component contiguous_addr_space_c
                    <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                    : addr_space_base_c {

   function addr_handle_t add_region(addr_region_s <TRAIT> r);
   function addr_handle_t add_nonallocatable_region(addr_region_s <> r);

   bool byte_addressable = true;
};

component transparent_addr_space_c
                    <struct TRAIT: addr_trait_s = empty_addr_trait_s>
                    : contiguous_addr_space_c<TRAIT> {};

struct addr_region_base_s {
   bit[64] size;
};

struct addr_region_s <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                     : addr_region_base_s {
   TRAIT trait;
};

struct transparent_addr_region_s
                    <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                    : addr_region_s<TRAIT> {
   bit[64] addr;
};

struct addr_claim_base_s {
   rand bit[64] size;
   rand bool permanent;
   constraint default permanent == false;
};

struct addr_claim_s <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                    : addr_claim_base_s {
   rand TRAIT trait;
   rand bit[64] in [64'd2**0, 64'd2**1, 64'd2**2, 64'd2**3 , 64'd2**4 ,
      64'd2**5 , 64'd2**6 , 64'd2**7 , 64'd2**8 , 64'd2**9 , 64'd2**10,
      64'd2**11, 64'd2**12, 64'd2**13, 64'd2**14, 64'd2**15, 64'd2**16,
      64'd2**17, 64'd2**18, 64'd2**19, 64'd2**20, 64'd2**21, 64'd2**22,
      64'd2**23, 64'd2**24, 64'd2**25, 64'd2**26, 64'd2**27, 64'd2**28,
      64'd2**29, 64'd2**30, 64'd2**31, 64'd2**32, 64'd2**33, 64'd2**34,
      64'd2**35, 64'd2**36, 64'd2**37, 64'd2**38, 64'd2**39, 64'd2**40,
      64'd2**41, 64'd2**42, 64'd2**43, 64'd2**44, 64'd2**45, 64'd2**46,
      64'd2**47, 64'd2**48, 64'd2**49, 64'd2**50, 64'd2**51, 64'd2**52,
      64'd2**53, 64'd2**54, 64'd2**55, 64'd2**56, 64'd2**57, 64'd2**58,
      64'd2**59, 64'd2**60, 64'd2**61, 64'd2**62, 64'd2**63] alignment;
};

struct transparent_addr_claim_s
                    <struct TRAIT : addr_trait_s = empty_addr_trait_s>
                    : addr_claim_s<TRAIT> {
   rand bit[64] addr;
};

enum endianness_e {LITTLE_ENDIAN, BIG_ENDIAN};
```

```
struct packed_s<endianness_e e = LITTLE_ENDIAN> {};

struct sizeof_s<type T> {
   static const int nbytes = /* implementation-specific */;
   static const int nbits = /* implementation-specific */;
};

const addr_handle_t nullhandle = /* implementation-specific */;

struct sized_addr_handle_s < int SZ, // in bits
                             int lsb = 0,
                             endianness_e e = LITTLE_ENDIAN >
                           : packed_s<e> {
   addr_handle_t hndl;
};

function addr_handle_t make_handle_from_claim  (addr_claim_base_s claim,
                                                bit[64] offset = 0);

function addr_handle_t make_handle_from_handle (addr_handle_t handle,
                                                bit[64] offset);

function bit[64] addr_value(addr_handle_t hndl);

import target function addr_value;

function bit[8]   read8(addr_handle_t hndl);
function bit[16] read16(addr_handle_t hndl);
function bit[32] read32(addr_handle_t hndl);
function bit[64] read64(addr_handle_t hndl);

function void write8 (addr_handle_t hndl, bit[8]  data);
function void write16(addr_handle_t hndl, bit[16] data);
function void write32(addr_handle_t hndl, bit[32] data);
function void write64(addr_handle_t hndl, bit[64] data);

function void read_bytes (addr_handle_t hndl, list<bit[8]> data,
                                                      int size);
function void write_bytes(addr_handle_t hndl, list<bit[8]> data);

function void read_struct (addr_handle_t hndl, struct packed_struct);
function void write_struct(addr_handle_t hndl, struct packed_struct);

extend component executor_base_c {
   function bit[8]   read8(addr_handle_t hndl);
   function bit[16] read16(addr_handle_t hndl);
   function bit[32] read32(addr_handle_t hndl);
   function bit[64] read64(addr_handle_t hndl);

   function void write8 (addr_handle_t hndl, bit[8]  data);
   function void write16(addr_handle_t hndl, bit[16] data);
   function void write32(addr_handle_t hndl, bit[32] data);
   function void write64(addr_handle_t hndl, bit[64] data);

   function void read_bytes (addr_handle_t hndl, list<bit[8]> data,
                                                         int size);
   function void write_bytes(addr_handle_t hndl, list<bit[8]> data);
};
```

```
    enum reg_access {READWRITE, READONLY, WRITEONLY};

    pure component reg_c < type R,
                           reg_access ACC = READWRITE,
                           int SZ = (8*sizeof_s<R>::nbytes)> {
        function R read();
        import target function read;

        function void write(R r);
        import target function write;

        function bit[SZ] read_val();
        import target function read_val;

        function void write_val(bit[SZ] r);
        import target function write_val;
    };

    struct node_s {
        string name;
        int    index;
    };

    pure component reg_group_c {
        pure function bit[64] get_offset_of_instance(string name);
        pure function bit[64] get_offset_of_instance_array(string name,
                                                           int index);
        pure function bit[64] get_offset_of_path(list<node_s> path);

        function void set_handle(addr_handle_t addr);
        import solve function set_handle;
    };
}
```

Annex E

(normative)

Foreign language bindings

E.1 Function prototype mapping

Let *f* be a function declared under hierarchical path *H* in PSS with type signature as below (with D_x as the direction, T_x as the type and p_x as the parameter name):

$$f(D_0\ T_0\ p_0,\ D_1\ T_1\ p_1,\ \ldots,\ D_n\ T_n\ p_n);$$

When *f* is bound to a foreign language API (see 22.4), it is mapped to the following function in the target language:

$$H'::f'(T'_0\ p_0,\ T'_1\ p_1,\ \ldots,\ T'_n\ p_n);$$

If the foreign language supports parameter directions, their directions are the same as in PSS.

NOTE—See E.5 for exceptions when mapping PSS functions to SystemVerilog tasks.

Each parameter in the PSS function is mapped to a corresponding parameter in the mapped function. The details of function name and data type binding are covered further below.

E.2 Data type mapping

PSS specifies data type bindings to C/C++ and SystemVerilog. The data type binding rules apply only to parameter and return types referenced (directly or indirectly) in the declaration of functions in PSS that are bound to foreign language APIs (see 22.4). The allowed types are specified in 22.4.1.1, namely:

— Primitive types: **bit** or **int** (width no more than 64 bits), **bool**, **string**, **chandle**.

— User-defined types: **enum** and **struct**, excluding packed structs (see 24.3.1) and excluding flow/resource objects. Fields of **struct**s shall be of these allowed types (recursively).

— Fixed-size arrays of these types.

The type binding is specified for parameter and return types.

E.3 C language bindings

E.3.1 Function names

PSS implementations shall support mapping a PSS function name to an identical function name in C, ignoring the hierarchical path in PSS. PSS implementations may define additional mapping schemes for function names.

E.3.2 Primitive types

The mapping between the PSS primitive types and C types is specified in Table E.1.

Table E.1—Mapping PSS primitive types and C types

PSS type	C input type	C output/inout type	C return type
string	**const char ***	**char ****	**char ***
bool	**unsigned int**	**unsigned int ***	**unsigned int**
chandle	**const void ***	**void ****	**void ***
bit (1-8-bit domain)	**unsigned char**	**unsigned char ***	**unsigned char**
bit (9-16-bit domain)	**unsigned short**	**unsigned short ***	**unsigned short**
bit (17-32-bit domain)	**unsigned int**	**unsigned int ***	**unsigned int**
bit (33-64-bit domain)	**unsigned long long**	**unsigned long long ***	**unsigned long long**
int (1-8-bit domain)	**char**	**char ***	**char**
int (9-16-bit domain)	**short**	**short ***	**short**
int (17-32-bit domain)	**int**	**int ***	**int**
int (33-64-bit domain)	**long long**	**long long ***	**long long**

Where pointers are used, the callee shall not allocate or de-allocate the memory region referenced by the pointer. Further, for non-void pointers, the callee shall assume that the memory location is valid only for the duration of the function execution, and shall not retain a reference to the parameter after the function call returns. For **string**s and **chandle**s, in the case of **inout**/**output** directions, the callee may return a pointer to storage it owns.

E.3.3 Arrays

Fixed-sized arrays are mapped to fixed-size arrays in C for function arguments. Mapping PSS fixed-sized arrays to C is not supported for function return types.

E.3.4 Structs

E.3.4.1 Name mapping

The mapping between a PSS **struct** type (T_{PSS}) defined in a hierarchical path H and a C type (T_C) is shown in Table E.2.

Table E.2—Mapping PSS struct types and C types

PSS type	C input type	C output/inout type	C return type
$H::T_{PSS}$	**const** T_C *****	T_C *****	T_C

In the general case, the name of the type in C (T_C), is derived from the PSS type name (T_{PSS}) and its hierarchical path (*H*). A PSS implementation shall support the name mapping scheme where the name of the C type is identical to the PSS type (ignoring the hierarchical path), i.e., $T_C == T_{PSS}$. A PSS implementation may support additional name mapping schemes.

E.3.4.2 Field mapping

Each PSS **struct** field is mapped to a corresponding field in C of the corresponding type and name in the same order. If the field type is itself a user-defined type (e.g., **struct** or **enum**), the mapping of the field entails the corresponding mapping of the type (recursively). For primitive types, the field is mapped as shown in Table E.3.

Table E.3—Mapping PSS struct field primitive types and C types

PSS field type	C field type
string	**char ***
bool	**unsigned int**
chandle	**void ***
bit (1-8-bit domain)	**unsigned char**
bit (9-16-bit domain)	**unsigned short**
bit (17-32-bit domain)	**unsigned int**
bit (33-64-bit domain)	**unsigned long long**
int (1-8-bit domain)	**char**
int (9-16-bit domain)	**short**
int (17-32-bit domain)	**int**
int (33-64-bit domain)	**long long**

Since the C language does not support type inheritance, if the PSS **struct** T_{PSS} derives from a PSS base type, then the fields of that base type are mapped directly into the mapped type T_C. The code listing below shows an example of **struct** type mapping in C.

```
// PSS code                        // C code

struct base_s {
  string f0;
};

struct sub_s {                     struct sub_s {
  int in [0..99] f1 = 2;             char f1;
  string f2;                         char *f2;
};                                 };

                                   struct my_struct_s {
struct my_struct_s : base_s {        char *f0;
  sub_s f3;                          sub_s f3;
  bit[16] f4;                        unsigned short f4;
};                                 };

my_struct_s function foo           my_struct_s foo
    (input my_struct_s x,             (const my_struct_s *x,
    output my_struct_s y);                   my_struct_s *y);
```

Example E.1—DSL struct mapping into C

Only the field name, its type and the position of the field inside a **struct** is relevant for mapping to the C type. Other field properties (such as initial value) and **struct** properties (such as constraints) are ignored.

E.3.4.3 Other mapping aspects

Tools may automatically generate C definitions for the required types, given PSS source code. Or, tools may utilize existing C declarations of the types. Regardless of whether these definitions are automatically generated or obtained in another way, PSS test generation tools may assume that these definitions are operative in the compilation of the C user implementation of the imported functions.

Note that the C declaration of a **struct** data type may have additional fields that are not reflected in the PSS type declaration. A PSS implementation may not assume that the C struct is size-compatible to the PSS **struct** type.

E.3.5 Enumeration types

A PSS enumeration type *E* is mapped to C as a plain numeric type *N* as follows:

Table E.4—Mapping PSS enum types and C types

PSS type	C input type	C output/inout type	C return type
E	***N***	***N****	***N***

where *N* is:

a) one of: **char**, **short**, **int**, or **long long**

b) the smallest type that includes the values of all the enum items in its domain

A PSS implementation will pass the value of the enumeration as an argument in the generated call to the function. These values can be either explicitly user-defined or assigned by a PSS implementation.

E.4 C++ language bindings

C++ is seen by the PSS standard as a primary language in the PSS domain. The PSS standard covers the projection of PSS arrays, enumeration types, strings, and struct types to their native C++ counterparts and requires that the naming of entities is kept identical between the two languages. This provides a consistent logical view of the data model across PSS and C++ code. The PSS language can be used in conjunction with C++ code without tool-specific dependencies.

Tools may automatically generate C++ definitions for the required types, given PSS source code. However, regardless of whether these definitions are automatically generated or obtained in another way, PSS test generation tools may assume these exact definitions are operative in the compilation of the C++ user implementation of the imported functions. In other words, the C++ functions are called by the PSS tool during test generation, with the actual parameter values in the C++ memory layout of the corresponding data types. Since actual binary layout is compiler dependent, PSS tool flows may involve compilation of some C++ glue code in the context of the user environment.

E.4.1 Function name mapping and namespaces

Generally, PSS user-defined types correspond to C++ types with identical names. In PSS, **packages** and **components** constitute namespaces for types declared in their scopes. The C++ type definition corresponding to a PSS type declared in a **package** or **component** scope shall be inside the namespace statement scope having the same name as the PSS **component**/**package**. Consequently, both the unqualified and qualified names of the C++ mapped type are the same as in PSS.

PSS implementations shall support mapping a PSS function name to an identical function name in C++, in the same namespace hierarchical path. PSS implementations may define additional mapping schemes for function names.

E.4.2 Primitive types

a) C++ type mapping for primitive numeric types is the same as that for C.
b) A PSS **bool** is a C++ **`bool`** and the values: **`false`**, **`true`** are mapped respectively from PSS to their C++ equivalents.
c) C++ mapping of a PSS **string** is **`std::string`** (typedef-ed by the Standard Template Library (STL) to **`std::basic_string<char>`** with default template parameters).

Table E.5 provides the mapping between PSS primitive types and C++ types. Note that **string** is passed as a reference.

Table E.5—Mapping PSS primitive types and C++ types

PSS type	C++ input type	C++ output/inout type	C++ return type
string	**const std::string &**	**std::string &**	**std::string**
bool	**bool**	**bool ***	**bool**
chandle	**const void ***	**void ****	**void ***
bit (1-8-bit domain)	**unsigned char**	**unsigned char ***	**unsigned char**
bit (9-16-bit domain)	**unsigned short**	**unsigned short ***	**unsigned short**
bit (17-32-bit domain)	**unsigned int**	**unsigned int ***	**unsigned int**
bit (33-64-bit domain)	**unsigned long long**	**unsigned long long ***	**unsigned long long**
int (1-8-bit domain)	**char**	**char ***	**char**
int (9-16-bit domain)	**short**	**short ***	**short**
int (17-32-bit domain)	**int**	**int ***	**int**
int (33-64-bit domain)	**long long**	**long long ***	**long long**

E.4.3 Arrays

The C++ mapping of a PSS array is `std::vector` of the C++ mapping of the respective element type (using the default allocator class). Fixed-sized arrays in PSS are mapped to the corresponding STL vector class, just like arrays of an unspecified size. However, if modified, they are resized to the original size upon return, filling the default values of the respective element type as needed.

E.4.4 Structs

E.4.4.1 Name mapping

The mapping between a PSS **struct** type (T_{PSS}) and a C++ type (T_{CPP}) is shown in Table E.6.

Table E.6—Mapping PSS struct types and C++ types

PSS type	C++_ input type	C++ output/inout type	C++ return type
T_{PSS}	**const** T_{CPP} **&**	T_{CPP} **&**	T_{CPP}

PSS **struct** types are mapped to C++ structs, along with their field structure and inherited base type, if specified.

The base type declaration of the **struct**, if any, is mapped to the (public) base struct type declaration in C++ and entails the mapping of its base type (recursively).

E.4.4.2 Field mapping

Each PSS field is mapped to a corresponding (public, non-static) field in C++ of the corresponding type and in the same order. If the field type is itself a user-defined type (**struct** or **enum**), the mapping of the field entails the corresponding mapping of the type (recursively).

For example, given the following imported function definitions:

```
function void foo(derived_s d);
import solve CPP function foo;
```

with the corresponding PSS definitions:

```
struct base_s {
   int in [0..99] f1;
};
struct sub_s {
   string f2;
};
struct derived_s : base_s {
   sub_s f3;
   bit[15:0] f4[4];
};
```

mapping type `derived_s` to C++ involves the following definitions:

```
struct base_s {
   int f1;
};
struct sub_s {
   std::string f2;
};
struct derived_s : base_s {
   sub_s f3;
   std::vector<unsigned short> f4;
};
```

Nested **structs** in PSS are instantiated directly under the containing **struct**, that is, they have value semantics. Mapped **struct** types have no member functions and, in particular, are confined to the default constructor and implicit copy constructor.

Mapping a **struct** type does not entail the mapping of any of its subtypes. However, **struct** instances are passed according to the type of the actual parameter expression used in an **import function** call. Therefore, the ultimate set of C++ mapped types for a given PSS model depends on its function calls, not just the function prototypes.

E.4.4.3 Other mapping aspects

In the case of **`output`** and **`inout`** composite parameters, if a different memory representation is used for the PSS tool vs. C++, the inner state shall be copied in upon calling it and any change shall be copied back out onto the PSS entity upon return.

E.4.5 Enumeration types

PSS enumeration types are mapped to C++ unscoped enumeration types (as opposed to enum classes), with the same set of enum items in the same order and identical names. When specified, explicit numeric constant values for an enum item correspond to the same value in the C++ definition.

For example, the PSS definition:

```
enum color_e {red = 0x10, green = 0x20, blue = 0x30};
```

is mapped to the C++ type as defined by this very same code.

Consequently, enum item names within types used in PSS-to-C++ type binding must be unique.

E.5 SystemVerilog language bindings

E.5.1 Function names

PSS implementations shall support mapping a PSS function name to an identical function or task name in SystemVerilog, ignoring the hierarchical path in PSS. PSS implementations may define additional mapping schemes for function names.

E.5.2 Primitive types

The mapping between the PSS primitive types and SystemVerilog types for both parameter and return types is specified in Table E.7.

Table E.7—Mapping PSS primitive types and SystemVerilog types

PSS type	SystemVerilog type
string	**string**
bool	**bit**
chandle	**chandle**
bit (1-8-bit domain)	**byte unsigned**
bit (9-16-bit domain)	**shortint unsigned**
bit (17-32-bit domain)	**int unsigned**
bit (33-64-bit domain)	**longint unsigned**
int (1-8-bit domain)	**byte**
int (9-16-bit domain)	**shortint**
int (17-32-bit domain)	**int**
int (33-64-bit domain)	**longint**

PSS functions designated with the **target** qualifier (see 22.4.1) may be mapped either to tasks or functions in SystemVerilog, and shall be mapped to tasks by default. PSS **solve** functions shall be mapped to SystemVerilog functions. If neither platform qualifier is used, the default mapping shall be to a function. PSS functions that are mapped to SystemVerilog tasks may not be called on the solve platform.

When a PSS function is mapped to a SystemVerilog function, the return type (if any) and arguments of the SystemVerilog function shall correspond to those of the PSS function prototype.

When a PSS function is mapped to a SystemVerilog task, the following apply:

a) If the PSS function is a **void** function, then all arguments of the SystemVerilog task shall correspond to the PSS prototype:

$$f(D_0\ T_0\ p_0,\ D_1\ T_1\ p_1,\ .\ .\ .,\ D_n\ T_n\ p_n);\ \Rightarrow\ t(D_0\ T'_0\ p_0,\ D_1\ T'_1\ p_1,\ .\ .\ .,\ D_n\ T'_n\ p_n);$$

b) If the PSS function returns a value, then the first argument of the SystemVerilog task shall be an output of the type corresponding to the return value. All other arguments shall correspond accordingly:

$$T_r\ f(D_0\ T_0\ p_0,\ D_1\ T_1\ p_1,\ .\ .\ .,\ D_n\ T_n\ p_n);\ \Rightarrow\ t(output\ T'_r\ p_r,\ D_0\ T'_0\ p_0,\ D_1\ T'_1\ p_1,\ .\ .\ .,\ D_n\ T'_n\ p_n);$$

E.5.3 Numeric value mapping

When a numeric type or value is passed from PSS to SystemVerilog, the value shall be expanded or truncated according to SystemVerilog rules (IEEE 1800-2017, section 10.7), treating the SystemVerilog type as the left-hand side of an assignment statement where the PSS value is the right-hand side.

When a numeric type of value is passed from SystemVerilog to PSS, the value shall be expanded or truncated according to the rules in 9.7 and 9.8, treating the SystemVerilog type as the right-hand side of an assignment statement where the PSS value is the left-hand side.

E.5.4 Arrays

Fixed-size arrays in PSS are mapped to SystemVerilog dynamic arrays of corresponding type. Arrays are passed by value between PSS and SystemVerilog.

E.5.5 Structs

PSS **struct** types are mapped to classes in SystemVerilog with fields whose types correspond and whose names match. Values of all fields are deep-copied between mapped elements.

The following also apply:

a) The target SystemVerilog class must contain all fields present in the PSS **struct**. The target SystemVerilog class may be derived from a base class type.

b) Inheritance relationships may or may not be the same across the boundary. Whether the PSS **struct** is derived from a base type has no bearing on whether the SystemVerilog class to which it is mapped is derived from a similar (or any) type.

c) Passing inheritance hierarchies with shadowed fields is not supported.

d) Tools shall ignore the containing namespace of mapped structs.

E.5.6 Enumeration types

A PSS enumeration type is mapped to a SystemVerilog enum type. The numeric values of the *enum_items* must match, but it is not required that the names of the *enum_items* match.

If a PSS enumeration type is passed to or from SystemVerilog, the **enum** value is passed as its numeric equivalent, according to E.5.3.

Annex F

(informative)

Solution space

Once a PSS model has been specified, the elements of the model must be processed in some way to ensure that resulting scenarios accurately reflect the specified behaviors. This annex describes the steps a processing tool may take to analyze a portable stimulus description and create a (set of) scenario(s). See also Clause 18.

a) Identify root action:
    1) Specified by the user.
    2) Unless otherwise specified, the designated root action shall be located in the root component. By default, the root component shall be `pss_top`.
    3) If the specified root action is an atomic action, consider it to be the initial action traversed in an implicit **activity** statement.
    4) If the specified root action is a compound action:
        i) Identify all **bind** statements in the activity and bind the associated object(s) accordingly. Identify all resulting scheduling dependencies between bound actions.
        i) For every compound action traversed in the activity, expand its activity to include each sub-action traversal in the overall activity to be analyzed.
        ii) Identify scheduling dependencies among all action traversals declared in the activity and add to the scheduling dependency list identified in a.4.i.
b) For each action traversed in the activity:
    1) For each resource locked or shared (i.e., claimed) by the action:
        i) Identify the resource pool of the appropriate type to which the resource reference may be bound.
        ii) Identify all other action(s) claiming a resource of the same type that is bound to the same pool.
        iii) Each resource object instance in the resource pool has an built-in `instance_id` field that is unique for that pool.
        iv) The algebraic constraints for evaluating field(s) of the resource object are the union of the constraints defined in the resource object type and the constraints in all actions ultimately connected to the resource object.
        v) Identify scheduling dependencies enforced by the claimed resource and add these to the set of dependencies identified in a.4.i.
            1. If an action locks a resource instance, no other action claiming that same resource instance may be scheduled concurrent with the locking action.
            2. If actions scheduled concurrently collectively attempt to lock more resource instances than are available in the pool, an error shall be generated.
            3. If the resource instance is not locked, there are no scheduling implications of sharing a resource instance.
    2) For each flow object declared in the action that is not already bound:
        i) If the flow object is not explicitly bound to a corresponding flow object, identify the object pool(s) of the appropriate type to which the flow object may be bound.

ii) The algebraic constraints for evaluating field(s) of the flow object are the union of the constraints defined in flow object type and the constraints in all actions ultimately connected to the flow object.

iii) Identify all other explicitly-traversed actions bound to the same pool that:

1. Declare a matching object type with consistent data constraints,
2. Meet the scheduling constraints from b.1.v, and
3. Are scheduled consistent with the scheduling constraints implied by the type of the flow object.

iv) The set of explicitly-traversed actions from b.2.iii shall compose the *inferencing candidate list* (*ICL*).

v) If no explicitly traversed action appears in the ICL, then an anonymous instance of each action type bound to the pool from b.2.i shall be added to the ICL.

vi) If the ICL is empty, an error shall be generated.

vii) For each element in the ICL, perform step b.2 until no actions in the ICL have any unbound flow object references or the tool's inferencing limit is reached (see c).

c) If the tool reaches the maximum inferencing depth, it shall infer a terminating action if one is available. Given the set of actions, flow and resource objects, scheduling and data constraints, and associated ICLs, pick an instance from the ICL and a value for each data field in the flow object that satisfies the constraints and bind the flow object reference from the action to the corresponding instance from the ICL.