PSS Early Adopter II (PSS EA II)
Portable Test and Stimulus Standard

February 28, 2018

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

Accellera Systems Initiative (Accellera) Standards documents are developed within Accellera and the Technical Committee of Accellera. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied "**AS IS**."

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

> Accellera Systems Initiative.
> 8698 Elk Grove Blvd Suite 1, #114
> Elk Grove, CA 95624
> USA

> Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not

> be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera, provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Garibaldi, Accellera Systems Initiative, 8698 Elk Grove Blvd Suite 1, #114, Elk Grove, CA 95624, phone (916) 670-1056, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

Suggestions for improvements to the PSS EA II Standard are welcome. They should be sent to the PSS email reflector

pswg@lists.accellera.org

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.

Participants

The Portable Stimulus Working Group (PSWG) is entity based. At the time this draft standard was completed, the PSWG had the following membership:

Faris Khundakjie, Intel Corporation, *Chair*
Tom Fitzpatrick, Mentor, a Siemens business, *Vice-Chair*
David Brownell, Analog Devices, Inc., *Secretary*
Joe Daniels, *Technical Editor*

AMD: Karl Whiting

AMIQ EDA: Cristian Amitroaie, Stefan Birman

Analog Devices, Inc: David Brownell

Breker Verification Systems, Inc.: Adnan Hamid, Dave Kelf

Cadence Design Systems, Inc.: Bishnupriya Bhattacharya, Stan Krolikoski, Larry Melling, Sharon Rosenberg, Matan Vax

Cisco Systems, Inc.: Somasundaram Arunachalam

IBM: Holger Horbach

Intel Corporation: Ramon Chemel, Faris Khundakjie, Jeffrey Scruggs

Mentor, a Siemens business: Matthew Ballance, Tom Fitzpatrick

National Instruments Corporation: Hugo Andrade

NVIDIA Corporation: Mark Glasser

NXP Semiconductors N.V.: Monica Farkash

Qualcomm Incorporated: Sanjay Gupta, Kelly Mills

Semifore, Inc.: Jamsheed Agahi

Synopsys, Inc.: Rick Eversole, Shrenik Mehta, Hillel Miller, Srivatsa Vasudevan

Vayavya Labs Pvt. Ltd.: Karthick Gururaj, Sandeep Pendharkar

Contents

1. Overview ... 1

- 1.1 Purpose ... 1
- 1.2 Language design considerations ... 1
- 1.3 Modeling basics ... 2
- 1.4 Test realization ... 2
- 1.5 Conventions used ... 3
  - 1.5.1 Visual cues (meta-syntax) ... 3
  - 1.5.2 Notational conventions ... 4
  - 1.5.3 Examples ... 4
- 1.6 Use of color in this standard ... 4
- 1.7 Contents of this standard ... 4

2. References ... 5

3. Definitions, acronyms, and abbreviations ... 6

- 3.1 Definitions ... 6
- 3.2 Acronyms and abbreviations ... 7

4. Lexical conventions ... 8

- 4.1 Comments ... 8
- 4.2 Identifiers ... 8
- 4.3 Keywords ... 8

5. Modeling concepts ... 9

- 5.1 Modeling data flow ... 10
  - 5.1.1 Buffers ... 10
  - 5.1.2 Streams ... 10
  - 5.1.3 States ... 11
  - 5.1.4 Data object pools ... 12
- 5.2 Modeling system resources ... 12
  - 5.2.1 Resource objects ... 12
  - 5.2.2 Resource pools ... 12
- 5.3 Basic building blocks ... 12
  - 5.3.1 Components and binding ... 12
  - 5.3.2 Evaluation and inference ... 13
- 5.4 Constraints and inferencing ... 15
- 5.5 Summary ... 15

6. Execution semantic concepts ... 16

- 6.1 Overview ... 16
- 6.2 Assumptions of abstract scheduling ... 16
  - 6.2.1 Starting and ending action executions ... 16
  - 6.2.2 Concurrency ... 16
  - 6.2.3 Synchronized invocation ... 16
- 6.3 Scheduling concepts ... 17

6.3.1 Preliminary definitions 17
6.3.2 Sequential scheduling 17
6.3.3 Parallel scheduling 18

7. C++ specifics 19

8. Data types 21

8.1 Scalars 21
8.1.1 DSL syntax 21
8.1.2 C++ syntax 22
8.1.3 Examples 24
8.2 Booleans 24
8.3 enums 25
8.3.1 DSL syntax 25
8.3.2 C++ syntax 25
8.3.3 Examples 25
8.4 Strings 26
8.4.1 C++ syntax 26
8.4.2 Examples 27
8.5 chandles 28
8.5.1 C++ syntax 28
8.5.2 Examples 28
8.6 Structs 28
8.6.1 DSL syntax 29
8.6.2 C++ syntax 29
8.6.3 Examples 30
8.7 User-defined data types 30
8.7.1 DSL syntax 30
8.7.2 C++ syntax 31
8.7.3 Examples 31
8.8 Arrays 31
8.8.1 C++ syntax 31
8.8.2 Examples 33
8.8.3 Properties 34

9. Components 36

9.1 DSL syntax 36
9.2 C++ syntax 36
9.3 Examples 38
9.4 Components as namespaces 38
9.5 Component instantiation 39
9.5.1 Semantics 39
9.5.2 Examples 40
9.6 Component references 40
9.6.1 Semantics 41
9.6.2 Examples 41

10. Actions 43

10.1 DSL syntax 43
10.2 C++ syntax 44
10.3 Examples 44

10.3.1 Atomic actions 44
10.3.2 Compound actions 45

11. Activities 46

11.1 Activity declarations 46
11.2 Activity constructs 46
11.2.1 DSL syntax 46
11.2.2 C++ syntax 46
11.3 Action scheduling statements 47
11.3.1 Action traversal statement 47
11.3.2 Sequential block 51
11.3.3 parallel 53
11.3.4 schedule 56
11.4 Activity control-flow constructs 59
11.4.1 repeat (count) 59
11.4.2 repeat while 61
11.4.3 foreach 64
11.4.4 select 66
11.4.5 if-else 67
11.5 Symbols 69
11.5.1 DSL syntax 69
11.5.2 C++ syntax 69
11.5.3 Examples 70
11.6 Named sub-activities 71
11.6.1 DSL syntax 71
11.6.2 Scoping rules for named sub-activities 71
11.6.3 Hierarchical references using named sub-activity 72
11.7 Explicitly binding flow objects 73
11.7.1 DSL syntax 73
11.7.2 C++ syntax 74
11.7.3 Examples 74
11.8 Hierarchical flow object binding 75
11.9 Hierarchical resource object binding 76

12. Flow objects 79

12.1 Buffer objects 79
12.1.1 DSL syntax 79
12.1.2 C++ syntax 79
12.1.3 Examples 80
12.2 Stream objects 80
12.2.1 DSL syntax 80
12.2.2 C++ syntax 80
12.2.3 Examples 81
12.3 State objects 81
12.3.1 DSL syntax 81
12.3.2 C++ syntax 82
12.3.3 Examples 83
12.4 Using flow objects 83
12.4.1 DSL syntax 83
12.4.2 C++ syntax 83
12.4.3 Examples 84

13. Resource objects 87

13.1 Declaring resource objects 87
13.1.1 DSL syntax 87
13.1.2 C++ syntax 87
13.1.3 Examples 87
13.2 Claiming resource objects 88
13.2.1 DSL syntax 88
13.2.2 C++ syntax 88
13.2.3 Examples 89

14. Pools 91

14.1 DSL syntax 91
14.2 C++ syntax 91
14.3 Examples 92
14.4 Static pool binding directive 92
14.4.1 DSL syntax 93
14.4.2 C++ syntax 93
14.4.3 Examples 94
14.5 Resource pools and the instance_id attribute 97
14.6 Pool of states and the initial attribute 99

15. Randomization specification constructs 101

15.1 Algebraic constraints 101
15.1.1 Member constraints 101
15.1.2 Constraint inheritance 104
15.1.3 Action-traversal in-line constraints 104
15.1.4 Set membership expression 107
15.1.5 Implication constraint 108
15.1.6 if-else constraint 109
15.1.7 foreach constraint 111
15.1.8 Unique constraint 113
15.2 Scheduling constraints 114
15.2.1 DSL syntax 114
15.2.2 Example 115
15.3 Sequencing constraints on state objects 115
15.4 Randomization process 116
15.4.1 Random attribute fields 116
15.4.2 Randomization of flow objects 119
15.4.3 Randomization of resource objects 121
15.4.4 Randomization of component assignment 123
15.4.5 Random value selection order 123
15.4.6 Loops and random value selection 124
15.4.7 Relationship lookahead 126
15.4.8 Lookahead and sub-actions 128
15.4.9 Lookahead and dynamic constraints 129
15.4.10 pre_solve and post_solve exec blocks 131
15.4.11 Body blocks and sampling external data 136

16. Action inferencing 138

16.1 Implicit binding and action inferences 139
16.2 Object pools and action inferences 143
16.3 Data constraints and action inferences 144

17. Coverage specification constructs 149

17.1 coverspec declaration 149
17.1.1 DSL syntax 150
17.1.2 Examples 150
17.2 coverspec instantiation 150
17.2.1 DSL syntax 151
17.2.2 Examples 151
17.3 coverpoint goal 151
17.4 Referencing existing bin schemes 152
17.5 cross goal 152
17.6 coverspec constraints 153
17.6.1 Ignore bins 153
17.6.2 Illegal bins 154
17.6.3 Bins with expression 156
17.7 coverspec bins 156
17.7.1 DSL syntax 157
17.7.2 Examples 157
17.7.3 Explicit value and range grouping 157
17.7.4 Value range divide operator (/) 158
17.7.5 Value range size operator (:) 158
17.7.6 Wildcard bin (*) 158

18. Type extension 159

18.1 Specifying type extensions 159
18.1.1 DSL syntax 159
18.1.2 C++ syntax 159
18.1.3 Examples 160
18.1.4 Compound type extensions 161
18.1.5 Enum type extensions 164
18.1.6 Ordering of type extensions 165
18.2 Overriding types 166
18.2.1 DSL syntax 166
18.2.2 C++ syntax 166
18.2.3 Examples 167

19. Packages 170

19.1 Package declaration 170
19.1.1 DSL syntax 171
19.1.2 C++ syntax 171
19.1.3 Examples 171
19.2 Namespaces and name resolution 172
19.3 Import statement 172
19.4 Naming rules for members across extensions 172

20. Test realization ... 173

20.1 exec blocks ... 173
20.1.1 DSL syntax ... 173
20.1.2 C++ syntax ... 174
20.1.3 Examples ... 174
20.2 Implementation using a procedural interface (PI) ... 177
20.2.1 Function declaration ... 178
20.2.2 DSL syntax ... 178
20.2.3 C++ syntax ... 178
20.2.4 Examples ... 179
20.2.5 Method result ... 179
20.2.6 Method parameters ... 179
20.2.7 Parameter direction ... 180
20.3 PI PSS layer ... 180
20.4 PI function qualifiers ... 180
20.4.1 DSL syntax ... 181
20.4.2 C++ syntax ... 181
20.4.3 Specifying function availability ... 181
20.4.4 Specifying an implementation language ... 183
20.5 Calling PI methods ... 184
20.6 Target-template implementation for functions ... 186
20.6.1 DSL syntax ... 187
20.6.2 C++ syntax ... 187
20.6.3 Examples ... 187
20.7 Import classes ... 188
20.7.1 DSL syntax ... 188
20.7.2 C++ syntax ... 188
20.7.3 Examples ... 189
20.8 Implementation using target-template code blocks ... 189
20.8.1 Target-template code exec block kinds ... 189
20.8.2 Target language ... 190
20.8.3 exec file ... 190
20.9 C++ in-line solve exec implementation ... 190
20.10 C++ generative target exec implementation ... 191
20.10.1 Generative PI execs ... 192
20.10.2 Generative target-template execs ... 193
20.11 Comparison between mapping mechanisms ... 195
20.12 Exported actions ... 196
20.12.1 DSL syntax ... 196
20.12.2 C++ syntax ... 197
20.12.3 Examples ... 197
20.12.4 Export action foreign-language binding ... 198

21. Hardware/Software Interface (HSI) ... 199

Annex A (informative) Bibliography ... 200

Annex B (normative) Formal syntax ... 201

B.1 Package declarations ... 201

B.2 Action declarations ... 201

B.3 Struct declarations ... 203

B.4 Procedural interface (PI) 203
B.5 Component declarations 204
B.6 Activity statements 205
B.7 Overrides 206
B.8 Data declarations 206
B.9 Data types 207
B.10 Constraint 207
B.11 Coverspec 208
B.12 Expression 209
B.13 Identifiers and literals 211
B.14 Numbers 212
B.15 Comments 212

Annex C (normative) C++ header files 213

C.1 File pss.h 213
C.2 File pss/action.h 214
C.3 File pss/action_attr.h 215
C.4 File pss/action_handle.h 216
C.5 File pss/attr.h 216
C.6 File pss/bind.h 220
C.7 File pss/bit.h 220
C.8 File pss/buffer.h 220
C.9 File pss/chandle.h 221
C.10 File pss/comp_inst.h 221
C.11 File pss/component.h 222
C.12 File pss/constraint.h 222
C.13 File pss/enumeration.h 223
C.14 File pss/exec.h 224
C.15 File pss/export_action.h 225
C.16 File pss/extend.h 226
C.17 File pss/function.h 226
C.18 File pss/import_class.h 229
C.19 File pss/in.h 230
C.20 File pss/input.h 230
C.21 File pss/lock.h 231
C.22 File pss/output.h 231

C.23 File pss/override.h ... 231
C.24 File pss/package.h ... 232
C.25 File pss/pool.h ... 232
C.26 File pss/rand_attr.h ... 233
C.27 File pss/range.h ... 236
C.28 File pss/resource.h ... 236
C.29 File pss/scope.h ... 237
C.30 File pss/share.h ... 237
C.31 File pss/state.h ... 238
C.32 File pss/stream.h ... 238
C.33 File pss/structure.h ... 238
C.34 File pss/symbol.h ... 239
C.35 File pss/type_decl.h ... 239
C.36 File pss/unique.h ... 239
C.37 File pss/vec.h ... 240
C.38 File pss/width.h ... 240
C.39 File pss/detail/activityStmt.h ... 240
C.40 File pss/detail/algebExpr.h ... 241
C.41 File pss/detail/FunctionParam.h ... 242
C.42 File pss/detail/FunctionResult.h ... 242

Annex D (normative) Foreign-language data type bindings ... 244

D.1 C primitive types ... 244
D.2 C++ composite and user-defined types ... 244
D.3 SystemVerilog ... 247

Annex E (informative) Solution space ... 248

Annex F (informative) HSI UART example ... 250

PSS Early Adopter II (PSS EA II): A Portable Stimulus and Test Standard

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 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 C++ input format that is semantically equivalent to the DSL, as shown in the following clauses (see also Annex C). 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 referent.

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.

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*.

The execution of a *concrete 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 one of three ways: as a sequence of calls to external

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

functions implemented in the target language; as parameterized, but uninterpreted, code segments expressed in the target language; or as a C++ member function (for the C++ input format only).

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 key terms and punctuation, text that shall be typed exactly as it appears. For example, in the following state declaration, the keyword "state" and special characters "{" and "}" (and optionally ":" and/or ";") shall be typed as they appear: **state** identifier [**:** struct_super_spec] **{** { struct_body_item } **}** [**;**]
plain text	The normal or plain text font indicates syntactic categories. For example, an identifier needs to be specified in the following line (after the "state" key term): **state** identifier [**:** 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*.
`courier`	The `courier` font in running text indicates PSS, DSL, or C++ code. For example, the following line indicates PSS code (for a `state`): `state power_state_s { int[0..4] val; };`
[ ] square brackets	Square brackets indicate optional items. For example, the *struct_super_spec* and (ending) semicolon (`;`) are both optional in the following line: **state** identifier [**:** struct_super_spec] **{** { struct_body_item } **}** [**;**]
{ } curly braces	Curly braces (`{ }`) indicate items that can be repeated zero or more times. For example, the following shows zero or more *struct_body_items* can be specified in this declaration: **state** identifier [**:** struct_super_spec] **{** { struct_body_item } **}** [**;**]
| separator bar	The separator bar (`|`) character indicates alternative choices. For example, the following line shows the "input" or "output" key terms are possible values in a flow object reference: **input** | **output** action_data_declaration

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.

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 effect 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 10 - Clause 20 describe the PSS modeling constructs.

— Clause 21 highlights the Hardware/Software Interface (HSI).

— Annexes. Following Clause 21 are 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.

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.[2, 3]

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++.[4]

---

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

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

[4]4ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://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 (http://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][5] 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 its non-PSS implementation.

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

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 which 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.

[5]The number 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.

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
HSI	Hardware/Software Interface
PI	procedural interface
PSS	Portable Test and Stimulus Standard
SUT	system under test

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 it can be referenced. Identifiers are case-sensitive. A *meta-identifier* can appear in syntax definitions using the form: *construct_name*_identifer, e.g., *action*_identifier. See also B.13.

4.3 Keywords

PSS reserves the keywords listed in Table 2.

Table 2—PSS keywords

abstract	**action**	**activity**	**bind**	**bins**	**bit**
bool	**buffer**	**chandle**	**class**	**component**	**constraint**
coverpoint	**coverspec**	**cross**	**dynamic**	**else**	**enum**
exec	**export**	**extend**	**false**	**file**	**foreach**
function	**if**	**ignore_bins**	**illegal_bins**	**import**	**inout**
input	**in**	**instance**	**int**	**lock**	**option**
output	**override**	**package**	**parallel**	**pool**	**rand**
repeat	**resource**	**schedule**	**select**	**sequence**	**share**
solve	**state**	**stream**	**string**	**struct**	**symbol**
target	**true**	**type**	**typedef**	**unique**	**void**
		while	**with**		

5. Modeling concepts

A PSS model is made up of a number of elements (described briefly in Clause 1.3) that define a set of possible scenarios to be applied to the Design Under Test (DUT) via the associated test environment. The combination of the DUT and test environment can be referred to as the System Under Test (SUT). 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 scenario(s) that represent(s) the verification intent. Actions that correspond directly to operations performed by the underlying SUT 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 *scenario*(s) that implement(s) the critical verification intent while satisfying the data flow, scheduling, and resource constraints of the target SUT. 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 underlying SUT as demonstrated in Figure 1.

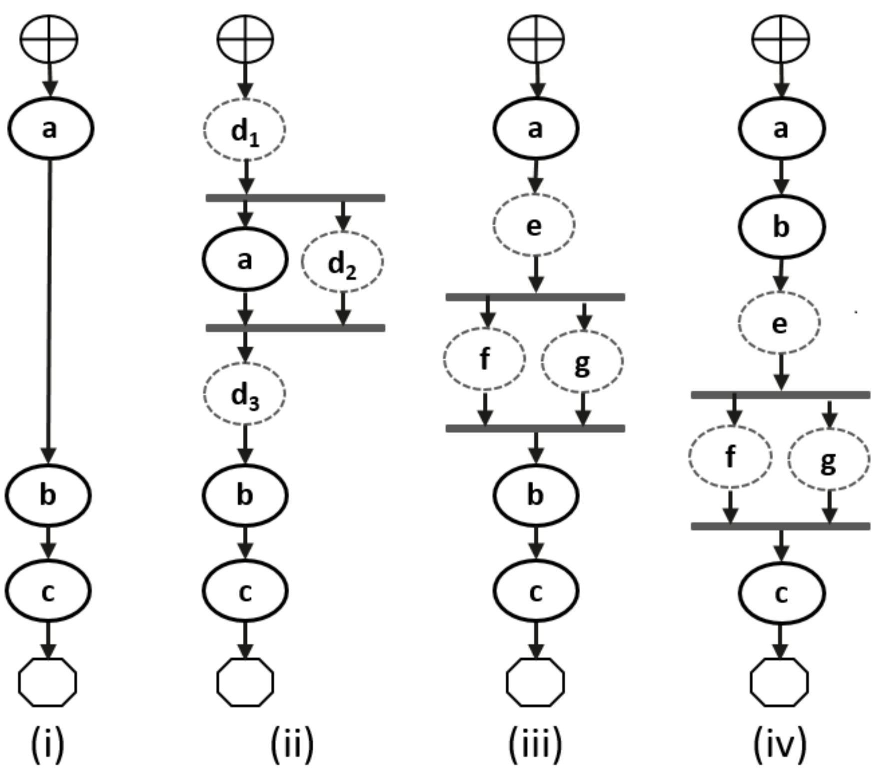

Figure 1—Partial specification of verification intent

In Figure 1, actions `a`, `b`, and `c` are specified in an activity. This partial specification may be expanded into multiple scenarios that infer other actions, yet all scenarios satisfy the critical intent defined by the activity.

An *activity* primarily specifies the set of actions to be executed and the scheduling relationship(s) between them. Actions may be scheduled sequentially, in parallel, or in various combinations based on conditional evaluation, looping, or randomization constructs (see 15.4). Activities may also include explicit data bindings between actions. An activity that traverses a compound action is evaluated hierarchically.

5.1 Modeling data flow

Actions may be declared to have inputs and/or outputs of a given data flow 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.

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.

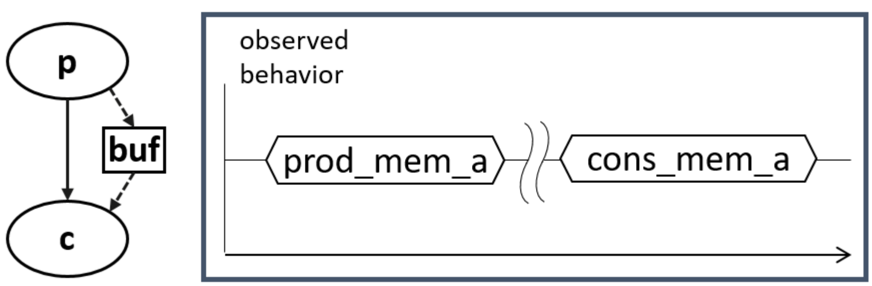

Figure 2—Buffer flow object semantics

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

To satisfy the activity shown in Figure 1(i), which shows actions a and b executing sequentially where b inputs a buffer object, action a needs to produce a buffer object for action b to consume, since the semantics of the buffer object supports the activity. Similarly, in Figure 1(ii), if action d produced the appropriate buffer type, it could be inferred as the producer of the buffer for action b to consume. The buffer scheduling semantics allow action d to be inferred as either d_1, d_2, or d_3, such that actions a and d each complete before action b starts, but there is no explicit scheduling constraint between a and d.

5.1.2 Streams

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

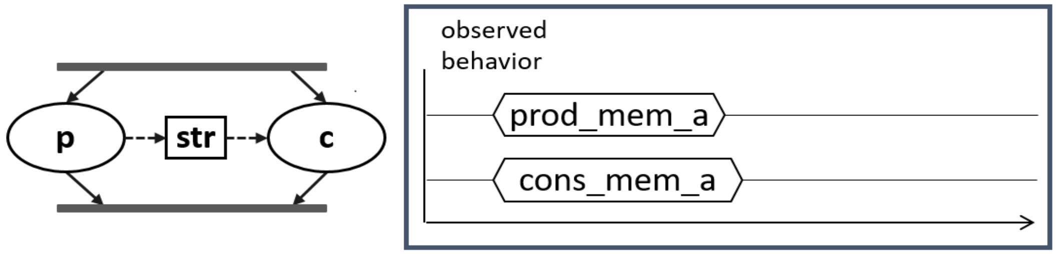

Figure 3—Stream flow object semantics

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

In Figure 1(iii), the parallel execution of actions `f` and `g` dictates that any data shared between these actions shall be of the *stream* type. Either of these actions may produce a buffer object type that may be consumed by the action `b`. If action `f` were inferred to supply the buffer to action `b`, and `f` inputs or outputs a stream object, then the one-to-one requirement of the stream object would require action `g` also be inferred to execute in parallel with `f`.

NOTE—Figure 1(iv) shows an alternate inferred scenario that also satisfies the base scenario of sequential execution of actions `a`, `b`, and `c`.

5.1.3 States

The *state* flow object represents the state of some element in the SUT 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 need to be sequential (see Figure 4).

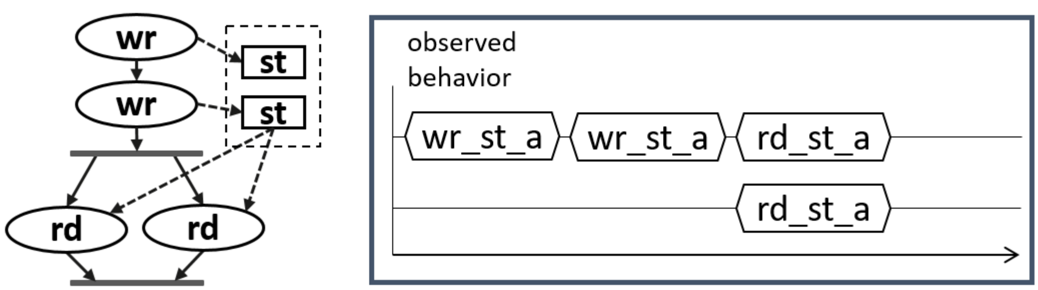

Figure 4—State flow object semantics

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

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 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 all see the same value for the state object at a given time.

5.2 Modeling system resources

5.2.1 Resource objects

In addition to declaring inputs and outputs, actions may require system resources that need to 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.

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 SUT. In addition to optionally randomizable data fields, the resource has a built-in non-negative numeric attribute called `instance_id`, which serves to identify the resource and is unique for each resource in the given pool.

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 will have access to a unique instance of the resource and the `instance_id` value for each instance shall be unique. For example, if a SUT 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 SUT, such as hardware engines, software packages, or test bench agents, and contains the actions that the SUT 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 comprised 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 9.5).

When a component instantiates a pool of data flow or resource objects, it also needs to *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. It shall be illegal to infer an action or object binding that is not required, either directly or indirectly, to make the activity specification legal. See also Figure 5, which demonstrates how actions can be inferred to generate multiple scenarios from a single activity.

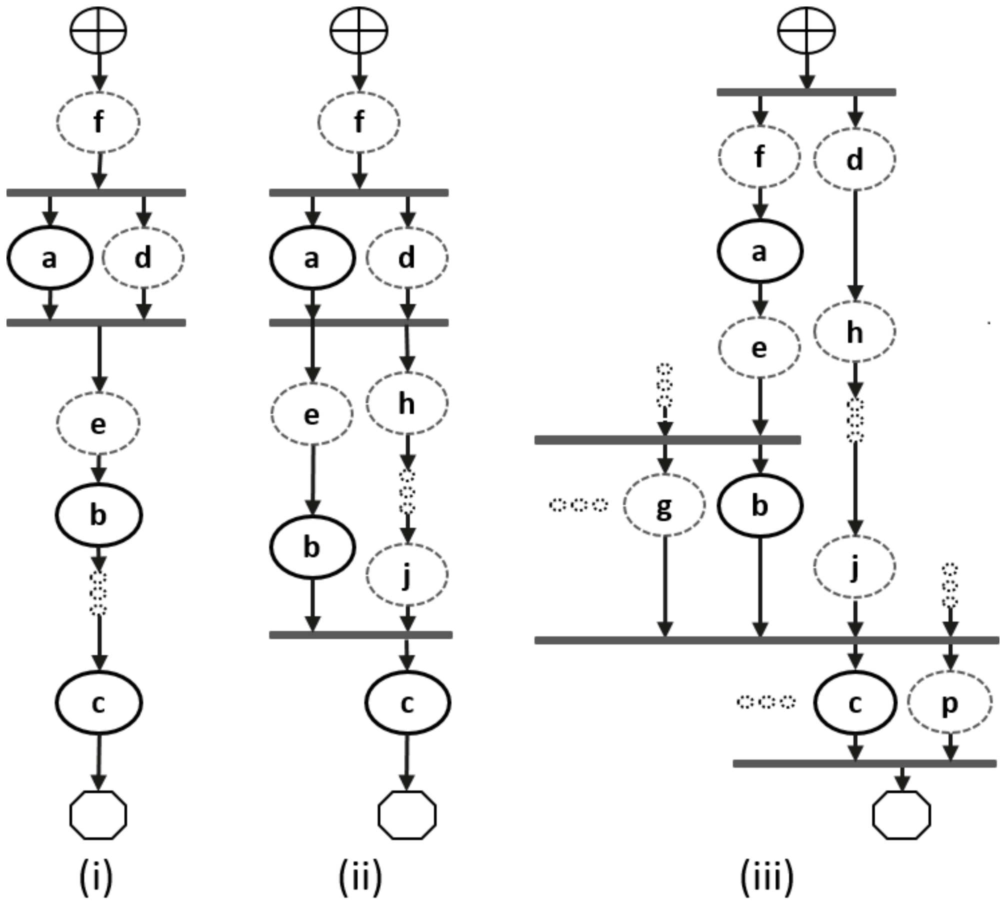

Figure 5—Single activity, multiple scenarios

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 and only if action `a` has a buffer input then an action, `f`, is inferred to execute sequentially before `a` to provide the buffer. Once inferred, if `f` also has a buffer input, then another action shall be inferred to supply that buffer 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 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 absence of an explicit binding of `a.out` to `b.in`, action `e` (or a series of actions) may be inferred to receive the output of action `a` and produce the input to action `b`. The direct connection between `a.out` and `b.in` could also be inferred here, in which case no action would be inferred between them. Similarly, in the absence of an explicit binding of `b.out` to `c.in`, a series of actions may be inferred between the completion of action `b` and the start of action `c` to provide the input of action `c`. 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 there is no explicit binding between b.out and c.in, 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 needs to 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, it 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 objects, all constraints defined in the object and 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 needs to 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 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 20.8.1) 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 assumes 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. It also assumes target-mapped behavior of actions can be known to have completed.

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

6.2.2 Concurrency

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

PSS semantics makes 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 does not distinguish between these.

6.2.3 Synchronized invocation

PSS semantics assumes 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 makes no assumptions on the actual runtime logic that synchronizes native execution threads and puts no absolute limit on the “sync-time” of synchronized action invocations.

6.3 Scheduling concepts

PSS execution semantics defines the criteria for legal runs of scenarios. The criterion covered in this chapter 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,[6] with values assigned to all of its parameters (reachable attributes). The execution of a compound action consists in executing the set of atomic actions it contains, directly or indirectly. For more on execution semantics of compound actions and activities, see Clause 11.

   An atomic action-execution has a specific *start-time*—the time in which its exec-body block is entered, and *end-time*—the time in which its exec-body block exits (the test itself does not complete successfully before 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 needs to wait for the other. Having no scheduling dependency between two actions-executions implies 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.

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 scheduling

[6]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).

dependency on every final action-execution in S_2. Generally, sequential scheduling of N action-execution sets S_i .. 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 $i <= N$.

For examples of sequential scheduling, see 11.3.2.3.

6.3.3 Parallel scheduling

N sets of action-executions S_i .. 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_i .. S_n are all independent scheduling-wise 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 11.3.3.3.

7. C++ specifics

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++ construct introduced in the C++ Syntax sections of this document (e.g., Syntax 1) are listed in Annex C.

Nested within the `pss` namespace is the `detail` namespace. Types defined within the `detail` namespace are documented to capture the intended behavior of the PSS/C++ types.

PSS/C++ object hierarchies are managed via the `scope` object, as shown in Syntax 1.

pss::scope

Defined in `pss/scope.h` (see C.29).

```
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 1—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 2.

pss::type_decl

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

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

Declare a type.

Member functions

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

Syntax 2—C++: type declaration

Example 1 shows an example of this usage.

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

Example 1—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 2.

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

Example 2—C++: Simplifying class declarations

8. Data types

8.1 Scalars

PSS supports two 2-state scalar data types. These fundamental scalar data types are summarized in Table 3, along with their default value domain.

Table 3—Scalar data types

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

8.1.1 DSL syntax

The DSL syntax for scalars is shown in Syntax 3.

```
integer_type ::= integer_atom_type
  [ [ expression [ : expression ] ] ]
  [ in [ open_range_list ] ]
integer_atom_type ::=
   int
  | bit
open_range_list ::= open_range_value { , open_range_value }
open_range_value ::=
  expression [ .. expression ]
 | expression ..
 | .. expression
 | expression
```

Syntax 3—DSL: Scalar data declaration

The following also apply.

a) Scalar values of `bit` type are unsigned values. Scalar values of `int` type are signed.

b) Integer literal constants can be specified in decimal, hexadecimal, octal, or binary format by following SystemVerilog 2-state variable conventions (`'h7f`, `'b111`, `7`) or C-style hexadecimal notation (`0x7f`).

c) 4-state values are not supported. If 4-state values are passed into the PSS model via the *procedural interface* (PI) (see 20.2), any `X` or `Z` values are converted to `0`.

d) The default values of the `bit` and `int` types is `0`.

e) The width and 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 domain specification, if present, e.g.,

```
bit [5] in[..10] b; // 0 <= b <= 10
bit [5] in [10..] b; // 10 <= b <= 31
```

f) Specifying a range with neither an upper nor lower bound shall be illegal.

8.1.2 C++ syntax

Contrasting with 8.1.1, b, C++ supports decimal, hexadecimal, and octal literals (e.g., `1`, `0x1`, and `001`, respectively).

The corresponding C++ syntax for Syntax 3 is shown in Syntax 4, Syntax 5, Syntax 6, Syntax 7, and Syntax 8.

pss::bit

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

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

Declare a bit.

Syntax 4—C++: bit declaration

pss::width

Defined in `pss/width.h` (see C.38).

```
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 range of bits

Syntax 5—C++: Scalar width declaration

pss::range

Defined in `pss/range.h` (see C.27).

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

Declare a range of values.

Member functions

`range (const T& value)` : constructor, single value

`range (const T& lhs, const T& rhs)` : constructor, value range

`range& operator() (const T& lhs, const T& rhs)` : function chaining to declare additional value ranges

`range& operator() (const T& value)` : function chaining to declare additional values

Syntax 6—C++: Scalar 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& name)` : constructor

`attr (const scope& name, 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 int& init_ val)` : constructor, with width and initial value (`T = int` or `bit` only)

`attr (const scope& s, const range<T>& a_range)` : constructor, with range (`T = int` or `bit` only)

`attr (const scope& s, const range<T>& a_range, const int& init_val)` : constructor, with range and initial value (`T = int` or `bit` only)

`attr (const scope& s, const width& a_width, const range<T>& a_range)` : constructor, with width and range (`T = int` or `bit` only)

`attr (const scope& s, const width& a_width, const range<T>& a_range, const int& init_val)` : constructor, with width, range and initial value (`T = int` or `bit` only)

`T& val()` : Enumerator access

`T* operator->()` : access underlying structure

`T& operator*()` : access underlying structure

Syntax 7—C++: Scalar non-rand declarations

pss::rand_attr

Defined in `pss/rand_attr.h` (see C.26).

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

Declare a random attribute.

Member functions

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

Syntax 8—C++: Scalar rand declarations

8.1.3 Examples

The DSL and C++ scalar data examples are shown in-line within 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 a 5-bit unsigned variable with a value range `0..31`.

```
DSL:   bit [5] in [0..31] b;
C++:   attr 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 <bit> (1)(2)(4) };
```

8.2 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>`.

8.3 enums

8.3.1 DSL syntax

The **enum** declaration is consistent with C/C++ and is a subset of SystemVerilog, as shown in Syntax 9. When not initialized, the default value of an **enum** shall be the first item in the list.

enum_declaration ::= **enum** *enum*_identifier **{** [enum_item { **,** enum_item }] **}** [**;**]
enum_item ::= identifier [**=** constant_expression]

Syntax 9—DSL: enum declaration

8.3.2 C++ syntax

The corresponding C++ syntax for Syntax 9 is shown in Syntax 10.

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

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

pss::enumeration

Defined in `pss/enumeration.h` (see C.13).

```
#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 with a name and a list of items (values optional)
2) Extend an enumeration 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 10—C++: enum declaration

8.3.3 Examples

Examples of enum usage are shown in Example 3 and Example 4.

```
enum config_modes_e {UNKNOWN, MODE_A=10, MODE_B=20};

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

Example 3—DSL: enum data type

The corresponding C++ example for Example 3 is shown in Example 4.

```
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 c {mode != config_modes_e::UNKNOWN};
  };
  type_decl<configure> configure_decl;
};
```

Example 4—C++: enum data type

8.4 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 (**""**).

8.4.1 C++ syntax

C++ uses `attr<std::string>` (see Syntax 11) or `rand_attr<std::string>` (see Syntax 12) to represent strings.

pss::attr

Defined in `pss/attr.h` (see C.27).

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

Declare a non-rand string attribute.

Member functions

`attr(const scope& name)` : constructor

`std::string& val()` : Access to underlying data

Syntax 11—C++: Scalar string declaration

pss::rand_attr

Defined in `pss/rand_attr.h` (see C.26).

```
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 12—C++: Scalar rand string declaration

8.4.2 Examples

The value of a random string-type field can be constrained with equality constraints and can be compared using equality constraints, as shown in Example 5 and Example 6.

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

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

Example 5—DSL: String data type

The corresponding C++ example for Example 5 is shown in Example 6.

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

  constraint c1 { "c1",
    if_then_else {
      a == 1,
      s == "FOO",
      s == "BAR"
    }
  };
};
```

Example 6—C++: String data type

8.5 chandles

The **chandle** type (pronounced "see-handle") represents an opaque handle to a foreign-language pointer as shown in Syntax 13. A chandle is used with the PI (see 20.2) to store foreign-language pointers in the PSS model and pass them to foreign-language functions and methods. See Annex D for more information about the foreign-language PI.

8.5.1 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 13—C++: chandle declaration

8.5.2 Examples

Example 7 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 7—DSL: chandle data type

8.6 Structs

A **struct** declares a collection of data items and constraints that relate the values of the data items, as shown in Syntax 14 or Syntax 15.

8.6.1 DSL syntax

```
struct_declaration ::= struct_type identifier [ : struct_identifier ] { { struct_body_item } } [ ; ]
struct_type ::=
    struct
  | struct_qualifier
struct_qualifier ::=
    buffer
  | stream
  | state
  | resource
struct_body_item ::=
    constraint_declaration
  | struct_field_declaration
  | typedef_declaration
  | coverspec_declaration
  | exec_block_stmt
struct_field_declaration ::= [ struct_field_modifier ] data_declaration
struct_field_modifier ::= rand
```

Syntax 14—DSL: struct declaration

A **struct** is a pure-data type; it does not declare an operation sequence. A struct declaration can specify a *struct_identifier*, a previously defined struct type from which the new type inherits its members, by using a colon (`:`), as in C++. In addition, structs can

— include **constraint**s (see 15.1) or **bin**s (see 17.7);

— represent data flow objects (see Clause 12) and resources (see Clause 13).

The following also apply.

a) Data elements within a struct may be declared to be of **int**, **bit**, **struct** or **enum** type, and may optionally include the **rand** keyword to indicate the element should be randomized when the overall struct is randomized (as shown in Example 8).

b) Applying the **rand** modifier to a field of a **struct** type causes all fields (and sub-fields) of the struct that are qualified as `rand` to be randomized when the struct is randomized.

c) Fields (and sub-fields) of the struct that are not qualified as `rand` are not randomized when the struct is randomized.

8.6.2 C++ syntax

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

The corresponding C++ syntax for Syntax 14 is shown in Syntax 15.

pss::structure

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

```
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 15—C++: struct declaration

8.6.3 Examples

Struct examples are shown in Example 8 and Example 9.

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

Example 8—DSL: Struct with rand modifier

```
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} };
  rand_attr<bit> is_write {"is_write" };
  rand_attr<bit> prot { "prot", width {3, 0} };
  rand_attr<bit> sema4 { "sema4", width {1, 0} };
};
```

Example 9—C++: Struct with rand modifier

8.7 User-defined data types

The **typedef** statement declares a user-defined type name in terms of an existing data type, as shown in Syntax 16.

8.7.1 DSL syntax

typedef_declaration ::= **typedef** data_type identifier **;**

Syntax 16—DSL: User-defined type declaration

8.7.2 C++ syntax

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

8.7.3 Examples

typedef examples are shown in Example 10 and Example 11.

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

Example 10—DSL: typedef

```
typedef unsigned int uint32_t;
```

Example 11—C++: typedef

8.8 Arrays

PSS supports fixed-sized arrays of scalar data types, and arrays of structs and components.

8.8.1 C++ syntax

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

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 <T>& 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 <T>& a_range)` : constructor, with element width and range (`T = int` or `bit` only)
- `attr( std::initializer_list<attr<T>> values )` : constructor creating array from list of elements
- `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

Syntax 17—C++: Arrays of non-random attributes

pss::rand_attr_vec

Defined in `pss/rand_attr.h` (see C.26).

```
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 <T>& 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 <T>& a_range)` : constructor, with element width and range (`T = int` or `bit` only)

`rand_attr( std::initializer_list<rand_attr<T>> values )` : constructor creating array from list of elements

`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 18—C++: Arrays of random attributes

8.8.2 Examples

Examples of fixed-size array declarations are shown in Example 12 and Example 13.

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

Example 12—DSL: Fixed-size arrays

The name of each array element is obtained by appending `[N]` to the array name, where `N` is the index of the element in the array. In Example 12, the names of the individual elements of the `east_routes` array are `east_routes[0]`, `east_routes[1]`…`east_routes[7]`, respectively.

```
// 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 13—C++: Fixed-size arrays

In C++, the name of each array element is obtained by appending `_N` to the array name, where `N` is the index of the element in the array. In Example 13, the names of the individual elements of the `east_routes` array are `east_routes_0`, `east_routes_1` …`east_routes_7`, respectively.

8.8.3 Properties

Arrays of scalar quantities provide properties, such as **sum** and **size** (see 8.8.3.1 and 8.8.3.2), that may be used in constraint expressions.

8.8.3.1 Sum

The **sum** property shall return the sum of all elements in the array.

8.8.3.2 Size

The **size** property shall return the number of elements in the array.

8.8.3.3 Examples of property usage

The `sum` property shown in Example 14 and Example 15 constrains the element values of an array of scalars.

```
bit [7:0]    data [4];
  constraint data_c {
    data.sum > 0 && data.sum < 1000;
  }
```

Example 14—DSL: sum property of an array

```
attr_vec<bit> data {"data", 4, width {7,0} };
constraint data_c { data.sum() > 0 && data.sum() < 1000 };
```

Example 15—C++: sum property of an array

The `size` property shown in Example 16 and Example 17 constrains the number of elements in an array of scalars.

```
bit [7:0]    data [4];
  constraint data_c {
    data.size < 10;
  }
```

Example 16—DSL: size property of an array

```
attr_vec<bit> data {"data", 4, width {7,0} };
constraint data_c { data.size() < 10 };
```

Example 17—C++: size property of an array

9. 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 test bench agents.

Components are structural entities, defined per type and instantiated under other components (see Syntax 19 or Syntax 20 and Syntax 21). Component instances constitute a hierarchy (tree structure), beginning with the top or root component, called `pss_top` by default, which is implicitly instantiated. Components have unique identities corresponding to their hierarchical path, and may also contain data-attributes, but not constraints. Components may also encapsulate functions (see 20.2.1) and imported class instances (see 20.7).

9.1 DSL syntax

```
component_declaration ::= component component_identifier [ : component_super_spec ]
  { { component_body_item } } [ ; ]
component_super_spec ::= : type_identifier
component_body_item ::=
    overrides_declaration
  | component_field_declaration
  | action_declaration
  | object_bind_stmt
  | inline_type_object_declaration
  | exec_block
  | package_body_item
```

Syntax 19—DSL: component declaration

9.2 C++ syntax

The corresponding C++ syntax for Syntax 19 is shown in Syntax 20 and Syntax 21.

Components are declared using the `component` class (see Syntax 20).

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()` : in-line init exec block

Syntax 20—C++: component declaration

Components are instantiated using the `comp_inst<>` or `comp_inst_vec()` class (see Syntax 21).

pss::comp_inst

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

```
template<class T>
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` .

```
template<class T> 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 21—C++: component instantiation

9.3 Examples

For examples of how to use a component, see Example 18 and Example 19.

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

Example 18—DSL: Component

The corresponding C++ example for Example 18 is shown in Example 19.

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

Example 19—C++: Component

9.4 Components as namespaces

Component types serve as a namespace for their nested types, i.e., action and struct types defined under them. Action and struct types may be thought of as (non-static) inner classes of components. The qualified name of action and object types is of the form `'`*`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. In a given namespace, identifiers shall be unique. Neither components nor packages may be declared inside other components or packages. Therefore, any type qualification using the **::** operator only has one level and the right-hand side shall not be a component or package type.

For examples of how to use a component as a namespace, see Example 20 and Example 21.

```
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 20—DSL: Namespace

The corresponding C++ example for Example 20 is shown in Example 21.

```
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 21—C++: Namespace

9.5 Component instantiation

Components are instantiated under other components as their fields, much like data fields of structs. Component fields may be of component and import-class type, as well as data fields, and may be arrays thereof.

9.5.1 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. A component type 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 22 and Example 23.

d) Scalar (non-array) data fields (**int**, **bit**, **chandle**, **bool**, **string**, or **enum**) may be initialized using a constant expression in their declaration. Any data field may be initialized via an **exec init** block, which overrides the value set by an initialization declaration. Exec init blocks may only contain assignment statements or imported calls. The component tree is elaborated to instantiate each component and then the **exec init** blocks are evaluated bottom-up. See also Example 177 and Example 178 (and 20.1).

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 179 and Example 180 (and 20.1).

9.5.2 Examples

Example 22 and Example 23 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 22—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 23—C++: Component instantiation

9.6 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 attribute **comp**. The value of the **comp** attribute can be used for comparisons (in equality and inequality expressions). The static type of the **comp** attribute of a given action is the type of the respective context component type. Consequently, sub-components of the containing component may be referenced via the **comp** attribute using relative paths.

9.6.1 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.

9.6.2 Examples

Example 24 and Example 25 demonstrate the use of the **comp** attribute. 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.

See also 15.1.

```
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;
    }
  };
};

component multimedia_ss_c {
   codec_c codecs[4];
   exec init {
      codecs[0].mode1_enable = 0;
      codecs[1].mode1_enable = 1;
      codecs[2].mode1_enable = 1;
      codecs[3].mode1_enable = 1;
   };
};
```

Example 24—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>

    constraint set_mode {“set_mode”,
      if_then { comp.mode1_enable==0, mode==0 }
    };
  };
  type_decl<decode> decode_decl;
};

class multimedia_ss_c : public component { ...
  comp_inst_vec<codec_c> codecs{ "codecs", 4};
  exec e { exec::init,
    codecs[0]->mode1_enable = 0,
    codecs[1]->mode1_enable = 1,
    codecs[2]->mode1_enable = 1,
    codecs[3]->mode1_enable = 1
  };
};
```

Example 25—C++: Constraining a comp attribute

10. Actions

Actions are a key abstraction unit in PSS. Actions serve to decompose scenarios into elements whose definition 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 11). 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 12) and resource (see Clause 13) 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 need to 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 block* (see 20.1) or they may be *compound*, in which case they contain an **activity** (see Clause 11) 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 22. See also Syntax 23.

10.1 DSL syntax

```
action_declaration ::= [ abstract ] action action_identifier [ action_super_spec ]
  { { action_body_item } }[ ; ]
action_super_spec ::= : type_identifier
action_body_item ::=
    activity_declaration
  | overrides_declaration
  | constraint_declaration
  | action_field_declaration
  | symbol_declaration
  | coverspec_declaration
  | exec_block_stmt
```

Syntax 22—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 `exec_block_stmt` action body items are mutually exclusive. An atomic action may specify `exec_block_stmt` items; it shall not specify `activity_declaration` items. A compound action, which contains instances of other actions and an `activity_declaration` item, shall not specify `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. The extended actions may be instantiated like any other action. Abstract actions shall not be instantiated directly.

10.2 C++ syntax

Actions are declared using the `action` class.

The corresponding C++ syntax for Syntax 22 is shown in Syntax 23.

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
`rand_attr<component*>& comp()` : get action component instance

Syntax 23—C++: action declaration

10.3 Examples

10.3.1 Atomic actions

Examples of an **action** declaration are shown in Example 26 and Example 27.

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

Example 26—DSL: atomic action

The corresponding C++ example for Example 26 is shown in Example 27.

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

Example 27—C++: atomic action

10.3.2 Compound actions

Compound actions instantiate other actions within them and use an activity statement (see Clause 11) to define the relative scheduling of these sub-actions.

Examples of compound action usage are shown in Example 28 and Example 29.

```
action sub_a {...};

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

Example 28—DSL: compound action

The corresponding C++ example for Example 28 is shown in Example 29.

```
class sub_a : public action { ... };

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

Example 29—C++: compound action

11. Activities

When a *compound action* includes multiple operations, these behaviors are described within the **action** using an **activity**. An *activity* specifies the set of actions to be executed and the scheduling relationship(s) between them. 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 11.3.1).

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.

11.1 Activity declarations

Because activities are explicitly specified as part of an action, and there may be at most one activity in a given 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*.

11.2 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 24 or Syntax 25.

11.2.1 DSL syntax

```
activity_declaration ::= activity { { [ identifier : ] activity_stmt } } [ ; ]
activity_stmt ::=
    activity_if_else_stmt
  | activity_repeat_stmt
  | activity_constraint_stmt
  | activity_foreach_stmt
  | activity_action_traversal_stmt
  | activity_sequence_block_stmt
  | activity_select_stmt
  | activity_parallel_stmt
  | activity_schedule_stmt
  | activity_bind_stmt
```

Syntax 24—DSL: activity statement

11.2.2 C++ syntax

In C++, an activity is declared by instantiating the `activity` class.

The corresponding C++ syntax for Syntax 24 is shown in Syntax 25.

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::ActivityStmt/ r)` : constructor

`activity ( std::vector<detail::ActivityStmt>&& stmts )` : constructor

Syntax 25—C++: activity statement

11.3 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.

11.3.1 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 26 or Syntax 27). 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.

11.3.1.1 DSL syntax

activity_action_traversal_stmt ::=
  identifier [inline_with_constraint]
  | **do** *type*_identifier [inline_with_constraint] **;**
inline_with_constraint ::= **with**
  { { constraint_body_item } **}**
  | constant_expression

Syntax 26—DSL: Variable traversal statement

identifier names a unique action handle or variable in the context of the containing action type. 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.

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) Formally, a *traverse 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 subactivity.

d) Other aspects that impact action-evaluation scheduling, are covered via binding inputs or outputs (see Clause 12), resource claims (see Clause 13), or attribute value assignment (see Clause 10).

11.3.1.2 C++ syntax

The corresponding C++ syntax for Syntax 26 is shown in Syntax 27.

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

`action_handle<T> with ( detail::AlgebExpr expr )` : add constraint to action handle

`T* operator->()` : access underlying action type

`T& operator*()` : access underlying action type

Syntax 27—C++: Variable traversal statement

11.3.1.3 Examples

Example 30 and Example 31 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 30—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 31—C++: Action traversal

Example 32 shows an example of anonymous action traversal, including in-line constraints using DSL.

```
action A {
  rand bit[3:0]   f1;
  ...
}

action B {
  activity {
    do A;
    do A with {f1 < 10;};
  }
}
```

Example 32—DSL: Anonymous action traversal

Example 33 shows a C++ example of anonymous action traversal, however, there is no equivalent way of adding in-line constraints to anonymous action traversal in C++.

```
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 33—C++: Anonymous action traversal

Example 34 and Example 35 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 34—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->a12->f1 <= max)
    }
  };
};
```

Example 35—C++: Compound action traversal

11.3.2 Sequential block

An *activity sequence block* statement specifies sequential scheduling between sub-activities (see Syntax 28 or Syntax 29).

11.3.2.1 DSL syntax

activity_sequence_block_stmt ::= [**sequence**] **{** { activity_labeled_stmt } **}**

Syntax 28—DSL: Activity sequence block

The following also apply.

a) Statements in a sequential block execute in order so 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 need to be scheduled in between sub-activities of subsequent statements.

11.3.2.2 C++ syntax

The corresponding C++ syntax for Syntax 28 is shown in Syntax 29.

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::ActivityStmt/r)
```
: constructor

```
sequence ( std::vector<detail::ActivityStmt>&& stmts )
```
: constructor

Syntax 29—C++: Activity sequence block

11.3.2.3 Examples

Assume A and B are action types that have no rules or nested activity (see Example 36 and Example 37).

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 36—DSL: Sequential block

```
class my_test : public action { ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};
  activity act {
    a,
    b
  };
};
```

Example 37—C++: Sequential block

Example 38 and Example 39 show 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 a;
   B b;
   activity {
      a;
      b;
      {a; b};
      sequence{a; b};
   }
};
```

Example 38—DSL: Variants of specifying sequential execution in activity

```
class my_test : public action {
  ...
  action_handle<A> a{"a"};
  action_handle<B> b{"b"};
  activity act {
    a, b,
    {a, b},
    sequence {a, b}
  };
};
```

Example 39—C++: Variants of specifying sequential execution in activity

11.3.3 parallel

The *parallel statement* specifies sub-activities that execute concurrently (see Syntax 30 or Syntax 31).

11.3.3.1 DSL syntax

activity_parallel_stmt ::= **parallel {** { activity_labeled_stmt } **}** [**;**]

Syntax 30—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 the invocation of an action in one subactivity 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 to obtain a synchronized start (see 6.3.2).

11.3.3.2 C++ syntax

The corresponding C++ syntax for Syntax 30 is shown in Syntax 31.

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::ActivityStmt/ r)` : constructor

`parallel ( std::vector<detail::ActivityStmt>&& stmts )` : constructor

Syntax 31—C++: Parallel statement

11.3.3.3 Examples

Assume `A`, `B`, and `C` are action types that have no rules or nested activity (see Example 40 and Example 41).

The activity in action `my_test` specifies two dependencies `(a) -> (b)` and `(b) -> (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 needs to 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 40—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 41—C++: Parallel statement

In Example 42 and Example 43, 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 the execution of `B` follows `A` and `D` follows `C`. It shall be 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`.

In Example 42 and Example 43, 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 R[4] 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 42—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 43—C++: Another parallel statement

11.3.4 schedule

The **schedule** statement specifies the PSS processing tool shall select a legal order in which to evaluate the sub-activities, provided one exists. See Syntax 32 or Syntax 33.

11.3.4.1 DSL syntax

```
activity_schedule_stmt ::= schedule { { activity_labeled_stmt } } [ ; ]
```

Syntax 32—DSL: Schedule statement

The following also apply.

a) All activities inside the **schedule** block need to 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 the scheduling of the combined sets of action-executions per the evaluation of *activity_stmt*$_1$ .. *activity_stmt*$_n$, keeping all scheduling dependencies within the sets and introducing (at least) the necessary scheduling dependencies across the sets to comply with the rules of input-output binding of actions and resource assignments.

11.3.4.2 C++ syntax

The corresponding C++ syntax for Syntax 32 is shown in Syntax 33.

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::ActivityStmt/ r)` : constructor

`schedule ( std::vector<detail::ActivityStmt>&& stmts )` : constructor

Syntax 33—C++: Schedule statement

11.3.4.3 Examples

Consider the code in Example 44 and Example 45, which are similar to Example 40 and Example 41, but use a `schedule` block instead of a `parallel` block. In this case, valid execution is as follows.

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 parallel.

```
action my_test {
   A a;
   B b;
   C c;
   activity {
      a;
      schedule {
         b;
         c;
      }
   }
};
```

Example 44—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 45—C++: Schedule statement

In contrast, consider the code in Example 46 and Example 47. In this case, any execution order in which B comes after A and D comes after C is valid. In particular, the following executions are valid.

a) {A, B} followed by {C, D}.

b) {C, D} followed by {A, B}.

c) {A, B} in parallel with {C, D}.

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 this case, the above execution of {A, B} in parallel with {C, D} is illegal because of the scheduling dependency between the two parallel branches. Since the only explicit scheduling constraints are that B follows A and D follows C, the following execution would also be valid.

d) A, followed by B in parallel with C, followed by D.

e) A in parallel with C, followed by B in parallel with D.

```
action A {}
action B {}
action C {}
action D {}

action my_test {
   activity {
      schedule {
         {do A; do B;}
         {do C; do D;}
      }
   }
}
```

Example 46—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 47—C++: Scheduling block with sequential sub-blocks

11.4 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.

11.4.1 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 34 or Syntax 35) and 11.4.2 describes the *while-expression* variant.

11.4.1.1 DSL syntax

activity_repeat_stmt ::= **repeat (** [identifier **:**] expression **)** activity_sequence_block_stmt

Syntax 34—DSL: repeat-count statement

The following also apply.

a) *expression* shall be a numeric type (**int** or **bit**).

b) Intuitively, the repeated block 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.

c) Formally, the *repeat-count statement* specifies sequential scheduling between *N* sets of action-executions per the evaluation of *activity_sequence_block_stmt N* times, where *N* is the number to which *expression* evaluates (see 6.3.2).

d) Note also the choice of *values* to `rand` attributes figuring in the *expression* need to be such that it yields legal execution scheduling.

11.4.1.2 C++ syntax

The corresponding C++ syntax for Syntax 34 is shown in Syntax 35.

pss::action::repeat

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

```
class repeat;
```

Declare a repeat statement.

Member functions

`repeat ( const detail::AlgebExpr& count, const detail::ActivityStmt& activity)` : declare an repeat (count) activity

`repeat ( const attr<int>& iter, const detail::AlgebExpr& count, const detail::ActivityStmt& activity )` : declare an repeat (count) activity with iterator

Syntax 35—C++: repeat-count statement

11.4.1.3 Examples

In Example 48 and Example 49, the resulting execution is six sequential action executions, alternating `A`'s and `B`'s, with five scheduling dependencies: $(A_{i0}) \rightarrow (B_{i0})$, $(B_{i0}) \rightarrow (A_{i1})$, $(A_{i1}) \rightarrow (B_{i2})$, $(B_{i2}) \rightarrow (A_{i2})$, $(B_{i3}) \rightarrow (A_{i3})$.

```
action  my_test {
   A a;
   B b;
   activity {
      repeat (3) {
         a;
         b;
      }
   }
};
```

Example 48—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 49—C++: repeat statement

Example 50 and Example 51 show 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 50—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 {
        (i % 4), action1, action2
      }
    }
  };
};
```

Example 51—C++: Another repeat statement

11.4.2 repeat while

In the **repeat while** and **repeat** … **while** forms, iteration continues while the expression evaluates to `true` (see Syntax 36 or Syntax 37). See also Example 52 and Example 53.

11.4.2.1 DSL syntax

```
activity_repeat_stmt ::=
  repeat while ( expression ) activity_sequence_block_stmt
| repeat activity_sequence_block_stmt [ while ( expression ) ; ]
```

Syntax 36—DSL: repeat-while statement

The following also apply.

a) *expression* shall be of type **bool**.

b) Intuitively, the repeated block is iterated so long as the *expression* condition is `true`, as sampled before the sequence block (in the first variant) or if after (in the second variant).

c) Formally, the *repeat-while* statement specifies sequential scheduling between multiple sets of action-executions per the iterative evaluation of *activity_sequence_block_stmt*. The evaluation of *activity_sequence_block_stmt* continues repeatedly so long as *expression* evaluates to `true`. *expression* is evaluated before the execution of each set in the first variant and after each set in the second variant.

11.4.2.2 C++ syntax

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

class repeat_while;

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

```
class repeat_while;
```

Declare a repeat while activity.

Member functions

`repeat_while ( const detail::AlgebExpr& cond, const detail::ActivityStmt& activity )` : constructor

pss::action::do_while

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

```
class do_while;
```

Declare a do while activity.

Member functions

`do_while( const detail::ActivityStmt& activity, const detail::AlgebExpr& cond )` : constructor

Syntax 37—C++: repeat-while statement

11.4.2.3 Examples

```
component top {

    function bit is_last_one();

    action do_something {
        bit last_one;

        exec post_solve {
            last_one = is_last_one();
        }

        exec body C = """
            printf("Do Something\n");
        """;
    }

    action entry {
        do_something s1;

        activity {
            repeat {
                s1;
            } while (!s1.last_one);
        }
    }
}
```

Example 52—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 pre_solve { exec::pre_solve,
      last_one = type_decl<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 {
      do_while { s1,
        s1->last_one != 0
      }
    };
  };
  type_decl<entry> entry_t;
};
```

Example 53—C++: repeat while statement

11.4.3 foreach

The **foreach** construct iterates across the elements of an array (see Syntax 38 or Syntax 39). See also Example 54 and Example 55.

11.4.3.1 DSL syntax

activity_repeat_stmt ::= **foreach (** expression **)** activity_sequence_block_stmt

Syntax 38—DSL: foreach statement

The following also apply.

a) *expression* shall be an array-index expression, where the index expression is the index-variable identifier.

b) The body of the **foreach** statement is a sequential block that is evaluated once for each element in the array. The index variable ranges between `0` and one less than the size of the array.

c) Formally, the **foreach** statement corresponds to *N* sequential evaluations of *activity_sequence_block_stmt*, where *N* is size of the array.

11.4.3.2 C++ syntax

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

pss::foreach

Defined in `pss/detail/sharedExpr.h`.

```
class foreach;
```

Iterate activity across array of non-rand and rand attributes.

Member functions

`foreach ( const attr& iter, const attr<vec>& array, const detail::ActivityStmt& activity )` : non-rand attributes

`foreach ( const attr& iter, const rand_attr<vec>& array, const detail::ActivityStmt& activity )` : rand attributes

Syntax 39—C++: foreach statement

11.4.3.3 Examples

```
action my_action1 {
  rand bit[0..3]       val;

  // ...
}

action my_test {
    rand bit[0..3]     a[16];
    my_action1         action1;

    activity {
      foreach (a[j]) {
        action1 with { action1.val <= a[j]; };
      }
    }
};
```

Example 54—DSL: foreach statement

```
class my_action1 : public action { ...
  rand_attr < bit > val {"val", range<bit> {0, 3} };
};

class my_test : public action { ...
  rand_attr_vec<bit> a { "a", 16, range<bit> {0, 3} };
  attr<bit> j {"j"};

  action_handle<my_action1> action1{"action1"};

  activity act {
    foreach {j, a,
      action1.with( action1->val < a[j] )
    }
  };
};
```

Example 55—C++: foreach statement

11.4.4 select

The **select** statement specifies a branch point in the traversal of the activity (see Syntax 40 or Syntax 41).

11.4.4.1 DSL syntax

activity_select_stmt ::= **select {** activity_labeled_stmt activity_labeled_stmt
{ activity_labeled_stmt } **}**

Syntax 40—DSL: select statement

The following also apply.

a) Intuitively, a **select** statement executes one out of a number of possible activities.

b) Formally, each evaluation of a **select** statement corresponds to the evaluation of just one of the *activity_labled_stmt*s. All scheduling requirements shall hold for the selected activity statement. It shall be illegal if no activity statement is valid according to the active constraint and scheduling requirements.

11.4.4.2 C++ syntax

The corresponding C++ syntax for Syntax 40 is shown in Syntax 41.

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::ActivityStmt/ r)` : constructor

`select ( std::vector<detail::ActivityStmt>&& stmts )` : constructor

Syntax 41—C++: select statement

11.4.4.3 Examples

In Example 56 and Example 57, 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 56—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 57—C++: Select statement

11.4.5 if-else

The **if-else** statement introduces a branch point in the traversal of the activity (see Syntax 42 or Syntax 43).

11.4.5.1 DSL syntax

activity_if_else_stmt ::= **if (** expression **)** activity_stmt [**else** activity_stmt]

Syntax 42—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* needs to be such that it yields legal execution scheduling.

11.4.5.2 C++ syntax

The corresponding C++ syntax for Syntax 42 is shown in Syntax 43.

pss::if_then

Defined in `pss/detail/sharedExpr.h`.

```
class if_then
```

Declare if-then activity statement.

Member functions

`if_then ( const detail::AlgebExpr& cond, const detail::ActivityStmt& true_expr )` : constructor

pss::if_then_else

Defined in `pss/detail/sharedExpr.h`.

```
class if_then_else;
```

Declare if-then-else activity statement.

Member functions

`if_then_else (const detail::AlgebExpr& cond, const detail::ActivityStmt& true_expr, const detail::ActivityStmt& false_expr)` : constructor

Syntax 43—C++: if-else statement

11.4.5.3 Examples

If the scheduling requirements for Example 58 and Example 59 required selection of the `b` branch, then the value selected for `x` needs to be `<= 5`.

```
action my_test {
   rand int[1..10] x;
   A a;
   B b;
   activity {
      if (x > 5)
         a;
      else
         b;
   }
};
```

Example 58—DSL: if-else statement

```
class my_test : public action { ...

  action_handle<A> a{"a"};
  action_handle<B> b{"b"};

  activity act {
    if_then_else {
      x > 5, a, b
    }
  };
};
```

Example 59—C++: if-else statement

11.5 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 44 or Syntax 45). 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.

11.5.1 DSL syntax

symbol_declaration ::= **symbol** identifier [**(** symbol_paramlist **)**] **=** activity_stmt

symbol_paramlist ::= [symbol_param { **,** symbol_param }]

symbol_param ::= data_type identifier

Syntax 44—DSL: symbol declaration

11.5.2 C++ syntax

In C++, a `symbol` is created using a function that returns the sub-activity expression.

The corresponding C++ syntax for Syntax 44 is shown in Syntax 45.

pss::symbol

Defined in `pss/symbol.h` (see C.34).

```
symbol symbolName(parameters...) { return (...); }
```

Function declaration to return sub-activity.

Syntax 45—C++: symbol declaration

11.5.3 Examples

Example 60 and Example 61 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 60—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 61—C++: Using a symbol

11.6 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 C++.

11.6.1 DSL syntax

A named sub-activity is declared by labeling an **activity** statement, see Syntax 46.

```
activity_labeled_stmt ::= [ identifier : ] activity_stmt
```

Syntax 46—DSL: Labeled activity statement

11.6.2 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.

In Example 62, 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 62—DSL: Scoping and named sub-activities

11.6.3 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.

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 needs to be declared at the top-level action scope.

In Example 63, `action B` declares action-handle variables in labeled activity statement scopes, thus making them accessible from outside by using hierarchical paths. `action C` is using 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; a }; // this 'a' is declared in unnamed scope
          A a;               // can't be accessed from outside
          a;
        };
      };
    };
  };
};

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 63—DSL: Hierarchical references and named sub-activities

11.7 Explicitly binding flow objects

Input and output objects may be explicitly connected to actions using the **bind** statement (see Syntax 47 or Syntax 48).

11.7.1 DSL syntax

```
activity_bind_stmt ::= bind hierarchical_id activity_bind_item_or_list ;
activity_bind_item_or_list ::=
    hierarchical_id
  | { hierarchical_id { , hierarchical_id } }
```

Syntax 47—DSL: bind statement

The following also apply.

It does not matter in which order the objects are listed, but they need to be of the same type and match the type of the object defined in each **action** being connected. As discussed in 12.4, the connection defines the data flow between **action**s and the type of the flow object defines the scheduling and semantics of the connection.

11.7.2 C++ syntax

The corresponding C++ syntax for Syntax 47 is shown in Syntax 48.

pss::bind

Defined in `pss/bind.h` (see C.6).

```
class bind;
```

Explicit binding of action inputs and outputs.

Member functions

`bind ( const std::initializer_list<detail::IOBase>& io_items )` : constructor

Syntax 48—C++: bind statement

11.7.3 Examples

Examples of binding are shown in Example 64 and Example 65.

```
buffer B {int a;};
action P {
  output B out;
};
action C {
  input B in;
};
action T {
  P p;
  C c;
  activity {
    p; c;
    bind p.out c.in;
  };
};
```

Example 64—DSL: bind statement

```
class B : public buffer { ... };
class P : public action { ...
  output<B> out {"out"};
};
class C : public action { ...
  input<B> in {"in"};
};
class T : public action { ...
  action_handle<P> p {"p"};
  action_handle<C> c {"c"};
  bind b1 {p->out, c->in};

  activity act {
    p, c
  };
};
```

Example 65—C++: bind statement

11.8 Hierarchical flow object binding

As discussed in 12.4, 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 Clause 14), 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 needs to complete before the activity begins, regardless of where within the activity the sub-action to which the input buffer is bound begins. Similarly, the activity needs to complete before the compound action's output buffer is available, regardless of where in the activity the sub-action that produces the buffer object executes. The corollary to this statement is no other sub-action in the activity may have an input explicitly bound to the compound action's buffer output object.

For stream objects, the compound action's activity shall execute in parallel with the action that produces the input stream object to the compound action or consumes the stream object output by the compound action, regardless of where within the activity the sub-action to which the stream object is bound actually executes. The corollary to this statement is all sub-actions within the activity that are bound to a stream input/output of the compound activity shall execute in parallel as the first statement in the **activity**.

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

The same hierarchical binding shown in Example 66 and Example 67 may be used for any type of data flow object.

```
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;
    bind data_out a2.dout;
  }
}
```

Example 66—DSL: Hierarchical flow binding

```
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"};

  bind b1 {a1->dout, a2->din};
  bind b2 {data_in, a1->din};
  bind b3 {data_out, a2->dout};

  activity act{
    a1,
    a2
  };
};
```

Example 67—C++: Hierarchical flow binding

11.9 Hierarchical resource object binding

As discussed in 13.2, 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 68 and Example 69, 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 68—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.->shB, a2->rlkB}; //illegal
  };
};
```

Example 69—C++: Hierarchical resource binding

12. Flow objects

A *flow object* represents incoming or outgoing data/control flow for actions, or their pre-condition and post-condition. A flow object is one which can have two modes of reference by actions: **input** and **output**.

12.1 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 49 or Syntax 50.

12.1.1 DSL syntax

```
buffer identifier [ : struct_super_spec ] { { struct_body_item } } [ ; ]
```

Syntax 49—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 unqualified structs or buffers.

c) A buffer input object shall be bound (connected) to an action output buffer object of the same type.

d) An buffer output object may be bound to one or more actions input buffer object(s) of the same type. A buffer output object is not required to be bound to an action input object.

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).

12.1.2 C++ syntax

The corresponding C++ syntax for Syntax 49 is shown in Syntax 50.

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 50—C++: buffer declaration

12.1.3 Examples

Examples of buffer objects are show in Example 70 and Example 71.

```
struct mem_segment_s {…};
   buffer data_buff_s {
     rand mem_segment_s seg;
   };
```

Example 70—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 71—C++: buffer object

12.2 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 51 or Syntax 52.

12.2.1 DSL syntax

stream identifier [**:** struct_super_spec] **{** { struct_body_item } **}** [**;**]

Syntax 51—DSL: stream declaration

The following also apply.

a) Stream types can inherit from previously defined unqualified structs or streams.

b) A stream input object shall be bound to a single action output stream object of the same type.

c) A stream output object shall be bound to a single action input stream object of the same type.

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 11.3.3.

12.2.2 C++ syntax

The corresponding C++ syntax for Syntax 51 is shown in Syntax 52.

pss::stream

Defined in `pss/stream.h` (see C.32).

```
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 52—C++: stream declaration

12.2.3 Examples

Examples of stream objects are show in Example 72 and Example 73.

```
struct mem_segment_s {…};
   stream data_stream_s {
     rand mem_segment_s seg;
   };
```

Example 72—DSL: stream object

```
struct mem_segment_s : public structure {...};

struct data_stream_s : public stream { ...
  PSS_CTOR(data_buff_s, stream);
  rand_attr<mem_segment_s> seg {"seg"};
};
type_decl<data_buff_s> data_buff_s_decl;
```

Example 73—C++: stream object

12.3 State objects

State objects represent the state of some entity in the execution environment at a given time. See Syntax 53 or Syntax 54.

12.3.1 DSL syntax

state identifier [**:** struct_super_spec] **{** { struct_body_item } **}** [**;**]

Syntax 53—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 14.6 for more on state pools (and **initial**).

c) State types can inherit from previously defined unqualified structs or states.

d) An action that has an input or output of state-object type operates on a pool of the corresponding state-object type to which the object is bound. **bind** directives are used to associate the action with the appropriate state-object pool (see 14.4).

e) 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.

f) 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.

g) An action that inputs a state object reads the current state object from the state-object pool to which it is bound.

h) 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.

i) Execution of an action that outputs a state object shall complete at any time before the execution of any inputting action begins.

j) 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.

k) 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.

12.3.2 C++ syntax

The corresponding C++ syntax for Syntax 53 is shown in Syntax 54.

pss::state

Defined in `pss/state.h` (see C.31).

```
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 54—C++: state declaration

12.3.3 Examples

Examples of state objects are show in Example 74 and Example 75.

```
enum mode_e {…};
   state config_s {
      rand mode_e mode;
      …
   };
```

Example 74—DSL: state object

```
class mode_e : public enumeration {...};

struct config_s : public state { ...
  PSS_CTOR(confic_s, state);
  rand_attr<mode_e> mode {"mode"};
};
type_decl<config_s> config_s_decl;
```

Example 75—C++: state object

12.4 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. See Syntax 55 or Syntax 56 and Syntax 57.

12.4.1 DSL syntax

input | **output** action_data_declaration

Syntax 55—DSL: Flow object reference

12.4.2 C++ syntax

Action input and outputs are defined using the `input` (see Syntax 56) and `output` (see Syntax 56) classes respectively.

The corresponding C++ syntax for Syntax 55 is shown in Syntax 56 and Syntax 57.

pss::input

Defined in `pss/input.h` (see C.20).

```
template<class T> class input;
```

Declare an action input.

Member functions

`input ( const scope& name )` : constructor
`T* operator->()` : access underlying input type
`T& operator*()` : access underlying input type

Syntax 56—C++: action input

pss::output

Defined in `pss/output.h` (see C.22).

```
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 57—C++: action output

12.4.3 Examples

12.4.3.1 Using buffer objects

Examples of using buffer flow objects are shown in Example 76 and Example 77.

```
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 76—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 76 is shown in Example 77.

```
struct mem_segment_s : public structure {...};

struct data_buff_s : public buffer {...
 rand_attr<mem_segment_s> seg {“seg”};
}

class cons_mem_a : public action { ...
  input<data_buff_s> in_data {"in_data"};
};

class prod_mem_a : public action { ...
  output<data_buff_s> out_data {"out_data"};
};
```

Example 77—C++: buffer flow object

12.4.3.2 Using stream objects

Examples of using stream flow objects are shown in Example 78 and Example 79.

```
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 78—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 78 is shown in Example 79.

```
struct mem_segment_s : public structure {...};

struct data_stream_s : public stream { ...
  rand_attr<mem_segment_s> seg {“seg”};
}

class cons_mem_a : public action { ...
  input<data_stream_s> in_data {"in_data"};
};

class prod_mem_a : public action { ...
  output<data_stream_s> out_data {"out_data"};
};
```

Example 79—C++: stream flow object

13. Resource objects

Resource objects represent computational resources available in the execution environment that may be assigned to actions for the duration of their execution.

13.1 Declaring resource objects

Resource struct types can inherit from previously defined unqualified structs or resource structs. See Syntax 58 or Syntax 59. Resources reside in *pools* (see Clause 14) and may be claimed by specific actions.

13.1.1 DSL syntax

```
resource identifier [ : struct_super_spec ] { { struct_body_item } } [ ; ]
```

Syntax 58—DSL: resource declaration

The following also apply.

a) Resources have a built-in numeric non-negative attribute called **instance_id** (see 14.5). 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 Clause 14.

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.

13.1.2 C++ syntax

The corresponding C++ syntax for Syntax 58 is shown in Syntax 59.

pss::resource

Defined in `pss/resource.h` (see C.28).

```
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<bit>& instance_id()` : get resource instance id

Syntax 59—C++: resource declaration

13.1.3 Examples

For examples of how to declare a resource, see Example 80 and Example 81.

```
resource DMA_channel_s {
  rand bit[3:0] priority;
};
```

Example 80—DSL: Declaring a resource

The corresponding C++ example for Example 80 is shown in Example 81.

```
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 81—C++: Declaring a resource

13.2 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 60 or Syntax 61 and Syntax 62.

13.2.1 DSL syntax

lock | **share** action_data_declaration

Syntax 60—DSL: Resource 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.

The following also apply.

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.

13.2.2 C++ syntax

The corresponding C++ syntax for Syntax 60 is shown in Syntax 61 and Syntax 62.

pss::lock

Defined in `pss/lock.h` (see C.21).

```
template < class T > class lock;
```

Claim a lock resource.

Member functions

`lock ( const scope& name )` : constructor
`T* operator->()` : access underlying input type
`T& operator*()` : access underlying input type

Syntax 61—C++: Claim a locked resource

pss::share

Defined in `pss/share.h` (see C.30).

```
template <class T> class share;
```

Share a lock resource.

Member functions

`share ( const scope& name )` : constructor
`T* operator->()` : access underlying input type
`T& operator*()` : access underlying input type

Syntax 62—C++: Share a locked resource

13.2.3 Examples

Example 82 and Example 83 demonstrate resource claims in lock and share mode. Action `two_DMA_chan_transfer` claims exclusive access to two different `DMA_channel_s` instances. It also claims one `CPU_core_s` instance in non-exclusive share mode. While `two_chan_transfer` executes, no other action may claim either instance of the `DMA_channel_s` resource, nor may any other action lock the `CPU_core_s` resource instance.

```
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 82—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 {"core"};
};
```

Example 83—C++: Resource object

14. 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 **action**s (see Clause 10) 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-object pool (see 14.4).

14.1 DSL syntax

```
component_pool_declaration ::= pool [ [ expression ] ] type_identifier identifier ;
```

Syntax 63—DSL: Pool instantiation

In Syntax 63, *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 is 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.

14.2 C++ syntax

The corresponding C++ syntax for Syntax 63 is shown in Syntax 64.

pss::pool

Defined in `pss/pool.h` (see C.25).

```
template <class T> class pool;
```

Instantiation of a pool.

Member functions

`pool ( const scope& name, std::size_t count = 1 )` : constructor

Syntax 64—C++: Pool instantiation

14.3 Examples

Example 84 and Example 85 demonstrate the how to use 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 84—DSL: Pool declaration

The corresponding C++ example for Example 84 is shown in Example 85.

```
struct data_buff_s : public buffer { ...
  rand_attr<mem_segment_s> seg {"seg"};
};

struct channels_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 85—C++: Pool declaration

14.4 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 65 or Syntax 66). Binding is done relative to the component sub-tree of the component type in which the **bind** directive occurs.

14.4.1 DSL syntax

```
object_bind_stmt ::= bind hierarchical_id object_bind_item_or_list ;
object_bind_item_or_list ::=
    component_path
  | { component_path { , component_path } }
component_path ::=
    component_identifier { . component_path_elem }
  | *
component_path_elem ::=
    component_action_identifier
  | *
```

Syntax 65—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 (**.**). The designated field shall agree with the pool in the object-type.

b) Default binding can be specified for an entire sub-tree by using a wildcard instead of specific paths. Explicit binding always takes precedence over default bindings. Conflicting explicit bindings for the same object-reference field shall be illegal. Between multiple default bindings applying 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).

14.4.2 C++ syntax

The corresponding C++ syntax for Syntax 65 is shown in Syntax 66.

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 T, typename... R >bind ( const pool<T>& a_ pool, const R&... targets )` : constructor

Syntax 66—C++: Static bind directives

14.4.3 Examples

Example 86 and Example 87 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 occurs in the context of component `dmac_c`, the `buff_p` pool is bound to all component instances and actions defined in `dmac_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 dmac_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 86—DSL: Static binding

The corresponding C++ example for Example 86 is shown in Example 87.

```
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 87—C++: Static binding

Example 88 and Example 89 illustrate the two forms of binding:, explicit and default. Action `power_transition`'s input and output are both associated with the context component's (`graphics_c`) state-object pool. However, action `observe_same_power_state` 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 { int[0..4] val; }

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
                           // prev/next)
   action power_transition {
      input power_state_s pwr;
      output power_state_s next_pwr;
      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 components sub-tree
                              // (specifically power_transition's chan)

   action observe_same_power_state {
      input power_state_s gfx0_state;
      input power_state_s gfx1_state;
      constraint gfx0_state.val == gfx1_state.val;
   }

   // explicit binding of the two power state variables to the
   // respective inputs of action observe_same_power_state
   bind gfx0.power_state_var observe_same_power_state.gfx0_state0;
   bind gfx1.power_state_var observe_same_power_state.gfx1_state1;
}
```

Example 88—DSL: Pool binding

```
struct power_state_s : public state { ...
};

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 prev/next)

  class power_transition_a : public action { ...
    input <power_state_s> prev {"prev"};
    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 components sub-tree
                                  // (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->val == gfx1_state->val };
  };
  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 89—C++: Pool binding

14.5 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 15.1.

For example, in Example 90 and Example 91, 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 need to 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.core.instance_id != xfer_b.core.instance_id;};
    constraint {xfer_a.chan.instance_id == xfer_b.chan.instance_id;};
  }
}
```

Example 90—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->core->instance_id() != xfer_b->core->instance_id() };
    constraint c2
      { xfer_a->chan->instance_id() == xfer_b->chan->instance_id() };
  };
  type_decl<par_dma_xfers> par_dma_xfers_decl;
};
```

Example 91—C++: Resource object assignment

14.6 Pool of states and the initial attribute

Each pool of a state struct-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 15.1.

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 92 and Example 93. The action `sys_configure` expands into two `codec_c::configure` actions: one to mode `A` and the other to mode `B`. Each component instance can have just one `configure` action, because it has an `initial` state as its precondition. So these two actions are necessarily associated with different component instances, `codec0` and `codec1`. But, the activity does not specify which action is associated with which instance.

```
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 92—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 {
        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>
        (codec_config_mode_e::A)
        (codec_config_mode_e::B) )
    };
  };
  type_decl<configure_a> configure_a_decl;
};
```

Example 93—C++: State object binding

15. 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 actions' inputs to actions' 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.

15.1 Algebraic constraints

15.1.1 Member constraints

PSS supports two types of constraint blocks as **action**/**struct** members: static constraints that always hold and dynamic constraints that only hold when they are traversed in the activity (see Syntax 67 or Syntax 68).

NOTE—As shown in 15.4.9, named dynamic constraints may be referenced as a node inside an activity.

15.1.1.1 DSL syntax

```
constraint_declaration ::=
    [ dynamic ] constraint identifier { { constraint_body_item } }
  | constraint { { constraint_body_item } }
  | constraint single_stmt_constraint
constraint_body_item ::=
    expression_constraint_item
  | foreach_constraint_item
  | if_constraint_item
  | unique_constraint_item
```

Syntax 67—DSL: Member constraint declaration

15.1.1.2 C++ syntax

The corresponding C++ syntax for Syntax 67 is shown in Syntax 68.

pss::constraint

Defined in `pss/constraint.h` (see C.12).

```
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.12).

```
class dynamic_constraint;
```

Declare a dynamic member constraint.

Member functions

`template <class... R> dynamic_constraint(const R&&... /detail::AlgebExpr/ expr)` : declare a dynamic constraint

`template <class... R> dynamic_constraint(const std::string& name, const R&&... /detail::AlgebExpr/ expr)` : declare a named dynamic constraint

Syntax 68—C++: Member constraint declaration

15.1.1.3 Examples

Example 94 and Example 95 declare a static constraint block, while Example 96 and Example 97 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 94—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 95—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 96—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<bit> (0x1000, 0x1fff) )
  };
  dynamic_constraint dyn_addr2_c { "dyn_addr2_c",
    in (addr, range<bit> (0x2000, 0x2fff) )
  };
};
```

Example 97—C++: Declaring a dynamic constraint

15.1.2 Constraint inheritance

Constraints, like other **action**/**struct**-members, are inherited from the super-type. An **action**/**struct** subtype has all of the constraints declared in the context of its super-type or inherited by it. A **constraint** specification overrides a previous specification if the constraint name is identical. For a constraint override, only the most specific property holds; any previously specified properties are ignored. Constraint inheritance and override applies in the same way to static constraints and dynamic constraints. Unnamed constraints shall not be overridden.

Example 98 and Example 99 illustrate a simple case of constraint inheritance and override. 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_inside in [1..1024];
   constraint size_align { size%4 == 0; } // 4 byte aligned
}

buffer corrupt_data_buff : data_buff {
   constraint size_align { size%4 == 1; }
                                 //overrides alignment 1 byte off
   constraint corrupt_data_size { size > 256; }
                                 // additional constraint
}
```

Example 98—DSL: Inheriting and overriding constraints

```
struct data_buf : public buffer { ...
  rand_attr<int> size {"size"};

  constraint size_align { "size_align", size % 4 == 0 };
};
struct corrupt_data_buf : public data_buf { ...
  constraint size_align { "size_align", size % 4 == 1 };
    // overrides alignment 1 byte off
  constraint corrupt_data_size { "corrupt_data_size", size > 256 };
    // additional constraint
};
```

Example 99—C++: Inheriting and overriding constraints

15.1.3 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 11.3.1).

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 by fields of the traversed sub-action, they can be explicitly accessed using built-in variable **this**. In particular, fields of the context component of the containing action need to be accessed using the prefix path `this.comp` (see also Example 102 and Example 103).

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 100 and Example 101 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 is unresolved at this point
    };
    a4;
  }
};
```

Example 100—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<bit>(8, 15)) };
  constraint c2 { "c2", a1->f == a4->f };
  activity a {
    a1,
    a2.with
      ( in { a2->f, range<bit>(8,15) } ),
        // same effect as constraint c1 has on
    a3.with
      ( a3->f == a4->f ),
        // illegal - a4.f is unresolved at this
    a4
  };
};
```

Example 101—C++: Action traversal in-line constraint

Example 102 and Example 103 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;
      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 102—DSL: Variable 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() == static_cast<top*>(comp().val())->sub1 )
                          // sub-action's component is
                          // sub-component 'sub1' of the
                          // parent action's component
      )
    };
  };
  type_decl<B> B_decl;
};
```

Example 103—C++: Variable resolution inside `with` *constraint block*

15.1.4 Set membership expression

The **in** expression defines the value of the referenced attribute field to be a member of the specified set. Syntax 69 or Syntax 70 shows the syntax for a set membership (**in**) expression.

15.1.4.1 DSL syntax

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

Syntax 69—DSL: Set membership expression

15.1.4.2 C++ syntax

The corresponding C++ syntax for Syntax 69 is shown in Syntax 70.

pss::in

Defined in `pss/in.h` (see C.19).

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

Constrain set membership.

Member functions

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

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

Syntax 70—C++: Set membership expression

15.1.4.3 Examples

Example 104 and Example 105 constrain the `addr` attribute field to the range `0x0` to `0xFFFF`.

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

Example 104—DSL: in constraint

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

Example 105—C++: in constraint

15.1.5 Implication constraint

Conditional constraints can be specified using the implication operator (**->**). Syntax 71 shows the syntax for an implication constraint.

15.1.5.1 DSL syntax

```
expression_constraint_item ::=
    expression implicand_constraint_item
  | expression ;
```

Syntax 71—DSL: Implication constraint

expression can be any integral 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 vacuously true, then the random values generated are constrained by the constraint (or constraint set). Otherwise, the random values generated are unconstrained.

b) If the *expression* is `true`, all of the constraints in the constraint set shall also be satisfied.

c) The implication constraint is bidirectional.

15.1.5.2 C++ syntax

C++ uses the `if_then` construct to represent implication constraints.

The Boolean equivalent of `if_then(a, b)` is `(!a || b)`.

15.1.5.3 Examples

Consider Example 106 and Example 107. 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 106—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 {
      a > 5,
      b == 1
    }
  };
};
```

Example 107—C++: Implication constraint

15.1.6 if-else constraint

Conditional constraints can be specified using the **if** and **if-else** constraint statements.

Syntax 72 or Syntax 73 shows the syntax for an **if-else** constraint.

15.1.6.1 DSL syntax

if_constraint_item ::= **if (** expression **)** constraint_set [**else** constraint_set]

Syntax 72—DSL: Conditional constraint

expression can be any integral 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 15.1.5), *if-else style* constraints are bidirectional.

15.1.6.2 C++ syntax

The corresponding C++ syntax for Syntax 72 is shown in Syntax 73.

pss::if_then

Defined in `pss/detail/sharedExpr.h`.

```
class if_then;
```

Declare if-then constraint statement.

Member functions

`if_then (const detail::AlgebExpr& cond, const detail::AlgebExpr& true_expr )` : constructor

pss::if_then_else

Defined in `pss/detail/sharedExpr.h`.

```
class if_then_else;
```

Declare if-then-else constraint statement.

Member functions

`if_then_else (const detail::AlgebExpr& cond, const detail::AlgebExpr& true_expr, const detail::AlgebExpr& false_expr )` : constructor

Syntax 73—C++: Conditional constraint

15.1.6.3 Examples

In Example 108 and Example 109, 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 108—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 {
      a > 5,
      b == 1,
      b < a
    }
  };
};
```

Example 109—C++: if constraint

15.1.7 foreach constraint

Elements of arrays can be iteratively constrained using the **foreach** constraint.

Syntax 74 or Syntax 75 shows the syntax for a **foreach** constraint.

15.1.7.1 DSL syntax

foreach_constraint_item ::= **foreach (** expression **)** constraint_set

Syntax 74—DSL: foreach constraint

expression can be any integral 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 *constraint_set* shall be satisfied.

b) Constraint sets may be used to group multiple constraints.

15.1.7.2 C++ syntax

The corresponding C++ syntax for Syntax 74 is shown in Syntax 75.

pss::foreach

Defined in `pss/detail/sharedExpr.h`.

```
class foreach;
```

Iterate constraint across array of non-rand and rand attributes.

Member functions

`foreach ( const attr& iter, const attr<vec>& array, const detail::AlgebExpr& constraint )` : non-rand attributes

Syntax 75—C++: foreach constraint

15.1.7.3 Examples

Example 110 and Example 111 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 110—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,
    // TODO: if_then is SharedExpr and we don't know if we are
    // building a AlgebExpr or an ActivityStmt
    // leads to ambiguous overload compiler error
      if_then {
        i > 0,
        fixed_arr[i] > fixed_arr[i-1]
      }
    }
  };
};
```

Example 111—C++: foreach iterative constraint

15.1.8 Unique constraint

The **unique** constraint causes unique values to be selected for each element in the specified set.

Syntax 76 or Syntax 77 shows the syntax for a **unique** constraint.

15.1.8.1 DSL syntax

unique_constraint_item ::= **unique {** hierarchical_id { **,** hierarchical_id } **} ;**

Syntax 76—DSL: unique constraint

15.1.8.2 C++ syntax

The corresponding C++ syntax for Syntax 76 is shown in Syntax 77.

pss::unique

Defined in `pss/unique.h` (see C.36).

```
class unique;
```

Declare of a unique constraint.

Member functions

`template<class... R> unique(R&&... /rand_attr<T>/ r)` : constructor

Syntax 77—C++: unique constraint

15.1.8.3 Examples

Example 112 and Example 113 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[0..15] A, B, C;
   constraint unique_abc_c {
      unique {A, B, C};
   }
}
```

Example 112—DSL: Unique constraint

```
class my_struct : public structure { ...
  rand_attr<bit> A {"A", range<bit>(0,15) },
                 B {"B", range<bit>(0,15) },
                 C {"C", range<bit>(0,15) };
  constraint unique_abc_c {"unique_abc_c",
    unique {A, B, C};
  };
};
```

Example 113—C++: Unique constraint

15.2 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 11.3.4). Similarly, scheduling constraints can be applied to named sub-activities, see Syntax 78.

15.2.1 DSL syntax

```
scheduling_constraint ::= constraint ( parallel | sequence )
  { hierarchical_id, hierarchical_id { , hierarchical_id } } ;
```

Syntax 78—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 11.3.2).

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 11.3.3).

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 11.4). 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 11.4), hold vacuously.

e) Scheduling constraints shall not undo or conflict with any scheduling requirements of the related actions.

15.2.2 Example

Example 114 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 114—DSL: Scheduling constraints

15.3 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 14.4). 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 attribute values of 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 12.3).

NOTE—Any constraint in which **prev** occurs is vacuously satisfied in the context of the initial state object.

In Example 115, 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 struct power_state_s {
  rand int[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 115—DSL: Sequencing constraints

15.4 Randomization process

PSS supports randomization of plain data models 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.

15.4.1 Random attribute fields

This section describes the rules that govern whether an element is considered randomizable.

15.4.1.1 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 15.4.10).

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.

15.4.1.2 Examples

In Example 116 and Example 117, 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 116—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 117—C++: Struct rand and non-rand fields

Example 118 and Example 119 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 118—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 119—C++: Action rand-qualified fields

Example 120 and Example 121 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 120—DSL: Action-qualified data 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 121—C++: Action-qualified fields

15.4.2 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 blocks* (except **pre_solve**, see 20.5), 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 16). In all of these cases, value selection for the data-attributes of a flow object need to satisfy all constraints coming from the action that outputs it and actions that input it.

Consider the model in Example 122 and Example 123. Assume a scenario is generated starting from action `test`. Action `wr` of type `write1` is scheduled, followed by action `rd` of type `read`. When `rd` is randomized, its input `in_obj` needs to be resolved. Every buffer object shall be the output of some action. The activity does not explicitly specify the binding of `rd`'s input to any action's output, but it needs to be resolved regardless. Action `wr` outputs an `mem_obj` whose `val` is in the range `1` to `5`, due to a constraint in action `write1`. But, `val` of the `mem_obj` instance `rd` inputs need to be in the range `8` to `12`. So `rd.in_obj` cannot be bound to `wr.out_obj` without violating a constraint. The PSS processing tool needs to schedule another action of type `write2` at some point prior to `rd`, whose `mem_obj` is bound to `rd`'s input. In selecting the value of `rd.input.val`, the PSS processing tool needs to consider the following.

- `val` is an even integer, due to the constraint in `mem_obj`.
- `val` is in the range `6` to `10`, due to a constraint in `write2`.
- `val` is in the range `8` to `12`, due to a constraint in `read`.

This restricts the legal values of `rd.in_obj.val` to either `8` or `10`.

```
component top {
   buffer mem_obj {
   int val;
   constraint val%2 == 0; // val must be even
   }

   action write1 {
      output mem_obj out_obj;
      constraint out_obj.val in [1..5];
   }

   action write2 {
      output mem_obj out_obj;
      constraint out_obj.val in [6..10];
   }

   action read {
      input mem_obj in_obj;
      constraint in_obj.val in [8..12];
   }

   action test {
      activity {
         do write1;
         do read;
      }
   }
}
```

Example 122—DSL: Randomizing flow object attributes

```
class top : public component { ...
  class mem_obj : public buffer { ...
    attr<int> val {"val"};
    constraint c { val%2 == 0 // val must be even };
  };

  class write1 : public action { ...
    output<mem_obj> out_obj {"out_obj"};
  };
  type_decl<write1> write1_decl;

  class write2 : public action { ...
    output<mem_obj> out_obj {"out_obj"};
  };
  type_decl<write2> write2_decl;

  class read : public action { ...
    input<mem_obj> in_obj {"in_obj"};
  };
  type_decl<read> read_decl;

  class test : public action { ...
    activity _activity {
      action_handle<write1>(),
      action_handle<read>()
    };
  };
  type_decl<test> test_decl;
};
```

Example 123—C++: Randomizing flow object attributes

15.4.3 Randomization of resource objects

When an **action** is randomized, its resource-claim fields (of **resource** type declared with **lock** / **share** modifiers, see 13.1) are assigned a reference to a resource object of the respective type. On entry to any of the action's *exec blocks* (except **pre_solve**, see 20.5) 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 13.2). 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 124 and Example 125. Assume a scenario is generated starting from action `test`. In this scenario, three `actions` are scheduled to execute in parallel: `a1`, `a2`, and `a3`. 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 needs to consider 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.

```
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 {

      activity {
         parallel {
            do do_something_a1 with { my_rsrc_inst.kind != B; };
            do do_something_a1 with { my_rsrc_inst.kind != C; };
            do do_something_else;
         }
      }
   }
}
```

Example 124—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"};

    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
      }
     };
  };
  type_decl<test> test_decl;
};
```

Example 125—C++: Randomizing resource object attributes

15.4.4 Randomization of component assignment

When an **action** is randomized, its association with a component instance is determined. The built-in attribute **comp** is assigned a reference to the selected component instance. The assignment needs to satisfy constraints where **comp** attributes occur (see 9.6). Furthermore, the assignment of an action's `comp` attribute 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 14 for more on pool binding.

15.4.5 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 126 and Example 127 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 126—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 127—C++: Activity with random fields

15.4.6 Loops and random value selection

A *loop* defines a traversal region. Random attribute fields and I/O fields of sub-actions, and, similarly, action-qualified fields, are considered to have an undefined value upon each entry to the loop, allowing the PSS processing tool to freely select values for the fields according to the active constraints and resource requirements.

Example 128 and Example 129 show an example of a root action (`my_action`) with sub-action fields and an **activity** containing a loop. A value for `a.val` is selected, then two sets of values for `b.val`, `c.val`, and `d.val` are selected.

```
action A {
  rand bit[3:0] val;
}

action my_action {
  A a, b, c, d;

  constraint abc_c {
    a.val < b.val;
    b.val < c.val;
    c.val < d.val;
  }
   activity {
    a;
    repeat (2) {
      b;
      c;
      d;
    }
  }
}
```

Example 128—DSL: Activity with random fields in a loop

```
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"}, d{"d"};

  constraint abc_c { "abc_c",
    a->val < b->val,
    b->val < c->val,
    c->val < d->val
  };

  activity act {
    a,
    repeat { 2,
      sequence {
        b,
        c,
        d
      }
    }
  };
};
```

Example 129—C++: Activity with random fields in a loop

The following breakout shows valid values that could be selected here.

Repetition	**a.val**	**b.val**	**c.val**	**d.val**
1	5	6	7	8
2	5	7	8	9

15.4.7 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 130 and Example 131.

15.4.7.1 Example 1

Example 130 and Example 131 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 [0..15] a_val, b_val, c_val;

   constraint {
      a_val < b_val;
      b_val < c_val;
   }
}
```

Example 130—DSL: Struct with random fields

```
class abc_s : public structure { ...
  rand_attr<bit> a_val{"a_val", range<bit>(0,15)},
                 b_val{"b_val", range<bit>(0,15)},
                 c_val{"c_val", range<bit>(0,15)};

  constraint c {
    a_val < b_val,
    b_val < c_val
  };
};
```

Example 131—C++: Struct with random fields

15.4.7.2 Example 2

Example 132 and Example 133 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 130 and Example 131 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` needs to be greater than `a.val`.

a) When selecting a value for `a.val`, a PSS processing tool needs to 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 needs to 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 132—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 133—C++: Activity with random fields

15.4.8 Lookahead and sub-actions

Lookahead shall be performed across traversal of sub-action fields and needs to comprehend the relationships between action attribute fields.

Example 134 and Example 135 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 needs to consider the following:

— `top_action.s1.a.val == top_action.v.val`

— `top_action.s1.a.val < top_action.s1.b.val`

This implies `top.v.val` needs to 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 134—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;
    action_handle<sub> s1;

    constraint c { "c", s1->a->val == v->val };
    activity act {
      v,
      s1
    };
  };
  type_decl<top_action> top_action_decl;
};
```

Example 135—C++: Sub-activity traversal

15.4.9 Lookahead and dynamic constraints

Dynamic constraints introduce traversal-dependent constraints. A PSS processing tool needs to 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 136 and Example 137 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 needs to 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 136—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 137—C++: Activity with dynamic constraints

15.4.10 pre_solve and post_solve exec blocks

The **pre_solve** and **post_solve** *exec blocks* enable external code to participate in the solve process. `pre_solve` and `post_solve` *exec blocks* 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 blocks* corresponds to the order random attribute fields are assigned by the solver. The ordering rules are as follows.

a) Order within a compound activity is top-down—both the `pre_solve` and `post_solve` *exec blocks* of a containing action are executed before any of its sub-actions are traversed, and, hence, before the `pre_solve` and `post_solve` 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 that 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 a compound data type (nested struct and array 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 20.1 for more information on the *exec block* construct.

15.4.10.1 Example 1

Example 138 and Example 139 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) `S2.pre_solve`

b) `S2.s1_2.pre_solve`

c) `S2.s1_1.pre_solve`

d) assignment of attribute values

e) `S2.post_solve`

f) `S2.s1_1.post_solve`

g) `S2.s1_2.post_solve`

```
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 138—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 139—C++: pre_solve/post_solve

15.4.10.2 Example 2

Example 140 and Example 141 illustrate the relative order of execution for `post_solve` *exec blocks* 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) `test.post_solve`
b) `write.post_solve`
c) `mem_obj.post_solve`
d) `read.post_solve`

```
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 {
  activity {
    write wr;
    read  rd;
    bind wr.out_obj rd.in_obj;
  }
  exec post_solve { ... }
};
```

Example 140—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"};

  bind b1 { wr->out_obj, rd->in_obj};

  activity act {
    wr,
    rd
  };

  exec post_solve { ... };
};
```

Example 141—C++: post_solve ordering between action and flow-objects

15.4.11 Body blocks and sampling external data

exec body blocks can assign values to non-rand attribute fields. **exec body** blocks are executed at the end of a leaf action execution. The impact of any field values modified by an **exec body** blocks is evaluated after the entire **exec body** block has completed.

Example 142 and Example 143 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 142—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,
      y1 = compute_val1(x),
      y2 = compute_val2(x)
    };
  };
  type_decl<A> A_decl;
};
```

Example 143—C++: exec body block sampling external data

16. 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 need to 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 11, 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 144 and Example 145.

```
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 144—DSL: Generating multiple scenarios

```
class pss_top : public component { ...
  struct data_buff_s : public buffer { ...
    rand_addr<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 145—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 need to 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.

16.1 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 E for more details on inference rules.

Consider the model in Example 146 and Example 147.

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 needs to 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 146—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 147—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 a transitive closure of the specified scenario per the flow-object dependency relations. Consider adding another action within the `pss_top` component in Example 146 and Example 147, e.g.,

```
action xfer_data {
    input data_buff_s src_data;
    output data_buff_s out_data;
  };

  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.

16.2 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 148 and Example 149, 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 148—DSL: Object pools affect inferencing

```
class pss_top : public component { ...
  struct data_buff_s : public buffer {...
    rand_addr<int> val{"val"};
  };

  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 149—C++: Object pools affect inferencing

16.3 Data constraints and action inferences

As mentioned in Clause 15, 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 144 and Example 145, as shown in Example 150 and Example 151.

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 150—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”);
  }; 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 151—C++: Data constraints affect action inferencing

Consider, instead, if the in-line data constraints were declared in the action types, as shown in Example 152 and Example 153.

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 152—DSL: Data constraints affect in-line 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 153—C++: Data constraints affect in-line action inferencing

17. Coverage specification constructs

The legal state space for all non-trivial verification problems is very large. Coverage targets identify key value ranges and value combinations that must occur in order to exercise key functionality. The **coverspec** construct is used to specify these targets.

The coverage targets specified by the **coverspec** construct are more directly related to the test scenario being created. As a consequence, the majority of these coverage targets would be considered coverage targets on the "generation" side of stimulus. PSS also allows data to be sampled by calling external methods. Coverage targets specified on data fields set by external methods can be related to the system state.

NOTE—Coverage is not supported in C++ in this PSS version.

17.1 coverspec declaration

Coverage goals are described using the **coverspec** construct. A **coverspec** declares an entity that specifies coverage goals and the data items on which those goals are declared (see Syntax 79). An instance of a `coverspec` is created to apply the coverage goals to specific data items (see 17.2).

17.1.1 DSL syntax

```
coverspec_declaration ::= coverspec identifier ( coverspec_port { , coverspec_port } )
    { { coverspec_body_item } } [ ; ]
coverspec_port ::= data_type identifier
coverspec_body_item ::=
    coverspec_option
  | coverspec_coverpoint
  | coverspec_cross
  | constraint_declaration
coverspec_option ::= option . identifier = constant_expression ;
coverspec_coverpoint ::=
    coverpoint_identifier : coverpoint coverpoint_target_identifier
        { { coverspec_coverpoint_body_item } }[ ; ]
  | ;
coverspec_coverpoint_body_item ::=
    coverspec_option
  | coverspec_coverpoint_binspec
coverspec_coverpoint_binspec ::= bins_keyword identifier
    bin_specification
  | hierarchical_id ;
bins_keyword ::=
    bins
  | ignore_bins
  | illegal_bins
coverspec_cross ::= ID : cross coverpoint_identifier { , coverpoint_identifier }
        { { coverspec_cross_body_item } }
  | ;
coverspec_cross_body_item ::= coverspec_option
```

Syntax 79—DSL: coverspec declaration

The following also apply.

A **coverspec** type can be declared in the *package scope*, *struct scope*, or *action scope*.

17.1.2 Examples

For examples of how to use a coverspec, see 17.2.2.

17.2 coverspec instantiation

A **coverspec** can be instantiated in a *struct scope* or *action scope*. The `coverspec` instantiation specifies the fields to which **coverspec** ports are bound (see Syntax 80).

17.2.1 DSL syntax

```
data_instantiation ::= identifier [ ( coverspec_portmap_list ) ] [ array_dim ]
    [ = constant_expression ]
coverspec_portmap_list ::= [
     coverspec_portmap { , coverspec_portmap }
   | hierarchical_id { , hierarchical_id } ]
coverspec_portmap ::= . identifier ( hierarchical_id )
array_dim ::= [ constant_expression ]
```

Syntax 80—DSL: coverspec instantiation

17.2.2 Examples

Example 154 shows a `transaction` struct that declares a **coverspec** in addition to random transaction fields. The `coverspec` accepts a parameter of the transaction-struct type and declares a `coverpoint` goal on the `addr` field of the `transaction` struct. The `struct` creates an instance of the `coverspec` and specifies itself (`this`) as the transaction instance to which to apply the coverage goals.

```
enum burst_type_e { INCR, WRAP };

struct transaction {
   rand bit[31:0] addr;
   rand burst_type_e burst_type;
   rand bit[4:0] burst_len;

   coverspec trans_cov(transaction t) {
      addr_ranges : coverpoint t.addr {
         bins low_addrs [0x00000000..0x0000FFFF]/64;
      }
   }

   // Coverspec instance
   trans_cov tc(this);
}
```

Example 154—DSL: coverspec declaration and instantiation

17.3 coverpoint goal

A **coverpoint** goal specifies a coverage goal on a scalar data item. Named bins (see 17.7) are used to identify key values and value ranges.

Example 155 shows a `coverpoint` goal specified on the `addr` field. `bins` are used to specify `64` even bins across the range `0x00000000-0x0000FFFF`.

```
coverspec trans_cov(transaction t) {
    addr_ranges : coverpoint t.addr {
      bins low_addrs [0x00000000..0x0000FFFF]/64;
    }
  }
```

Example 155—DSL: coverpoint goal

17.4 Referencing existing bin schemes

Bins and bin schemes (see 17.7) can be defined inside structs and activities. These bins and bin schemes can be referenced from a **coverpoint** goal.

Example 156 shows a `coverpoint` bin that references an externally-defined set of bins. The effect is that the `addr_ranges` coverpoint contains `bins` encompassing the value `0` and `'hfff`, and the value range `1..'hffe`.

```
struct transaction {
    rand bit[31:0]         addr;
     …

    bins addr_edges_b [0] [1..'hffe] ['hfff];
  }

  coverspec trans_cov(transaction t) {
    addr_ranges : coverpoint t.addr {
      bins edge_bins transaction.addr_edges_b;
    }
  }
```

Example 156—DSL: Referencing existing bins

17.5 cross goal

A **cross** goal specifies a coverage goal on two or more **coverpoint**s that encompasses all combinations of the bins (see 17.7) of the two **coverpoint**s.

Example 157 shows a `cross` goal between two `coverpoints`. The `burst_type_len` cross goal specifies all combinations of the bins of `burst_type` and `burst_len`.

```
coverspec trans_cov(transaction t) {

    burst_type : coverpoint t.burst_type;

    burst_len : coverpoint t.burst_len {
      bins small_burst [1..4]:1;
    }

    burst_type_len : cross burst_type, burst_len;
  }
```

Example 157—DSL: cross goal

17.6 coverspec constraints

Constraints can be declared within a **coverspec** to customize the values and value combinations selected by the specified goals. *coverspec constraints* apply globally in the **coverspec** in which they are declared.

Example 158 applies a constraint to coverage goals. In this case, the `burst_type_len_cross` cross goal implies all 32 combinations of the `burst_type` and `burst_len` coverpoint bins. However, the `burst_type_len_c` constraint specifies that when `burst_type == WRAP`, only three values of `burst_len` should be considered of interest.

```
enum burst_type_e { INCR, WRAP };

struct transaction {
   rand bit[31:0] addr;
   rand burst_type_e burst_type;
   rand bit[4:0] burst_len;

   coverspec trans_cov(transaction t) {
      constraint burst_type_len_c {
         if (burst_type == WRAP) {
            burst_len in [1,2,4];
         }
      }

      burst_type : coverpoint burst_type;
      burst_len : coverpoint burst_len {
         bins burst_len [1..16]:1;
      }

      burst_type_len_cross : cross burst_type, burst_len;
 }

 // Coverspec instance
 trans_cov tc(this);

}
```

Example 158—DSL: coverage constraint

17.6.1 Ignore bins

Ignore bins bucket coverage samples into an ignore bucket. An **ignore** constraint is an expression over the coverpoint identifiers and other DSL variables. Coverpoint identifiers represent the values sampled into the coverpoint bins. All samples that render the ignore bin *with expression* `true` are placed in the ignore bucket. Coverpoint identifiers have the type of the target variable that they monitor.

Ignore bins can be added to **coverpoint**s or **cross**es. Coverpoint ignore bins place samples for that coverpoint into an ignore bucket. Ignore bucket samples for coverpoints are excluded even if they are included in other coverpoint bins of the enclosing coverpoint. Any **cross**es using the **coverpoint** also result in those samples being placed in an ignore bucket. Ignore bins in a **cross** place the relevant samples to the cross in the cross's ignore bucket and do not change the ignore buckets of the other crosses. Ignore bucket samples for cross products are excluded even if they are included in other cross coverage bins of the enclosing **cross**.

```
coverspec trans_cov(transaction t) {
   burst_type : coverpoint t.burst_type;
   burst_len : coverpoint t.burst_len {
      bins small_burst [1..4]:1;
      ignore_bins burst_length with (burst_len == 2);
   }
   burst_type_len : cross burst_type, burst_len {
      ignore_bins burst_type_length_combinations
        with  !burst_type ? (burst_len > 2) : 1;
   }
}
```

Example 159—DSL: Ignore bucket 1

For Example 159, the following samples are placed in the ignore bucket.

	burst_type	burst_len
1	1	1

```
coverspec trans_cov(transaction t) {
    burst_type : coverpoint t.burst_type;
    burst_len : coverpoint t.burst_len {
       bins small_burst [1..4]:1;
       ignore burst_len == 2;
    }
    burst_type_len : cross burst_type, burst_len {
       ignore burst_type ? (burst_len < 2) : 1;
   }
}
```

Example 160—DSL: Ignore bucket 2

For Example 160, the following samples are placed in the ignore bucket.

	burst_type	burst_len
1	1	1
2	0	2
3	1	2

17.6.2 Illegal bins

Illegal bins bucket coverage samples into an illegal bucket. An **illegal** bin consists of an expression over the coverpoints identifiers and other DSL variables. Coverpoint identifiers represent the values sampled into the coverpoint bins. All samples that render the illegal bin *with expression* `true` are placed in the illegal bucket. Coverpoint identifiers have the type of the target variable that they monitor.

Illegal bins can be added to **coverpoint**s or **cross**es. Coverpoint illegal bins place samples for that coverpoint into an illegal bucket. Any **cross**es using the **coverpoint** also result in those samples being placed in an illegal bucket. Illegal bins in a **cross** place the relevant samples to the cross in the cross's illegal bucket and do not change the illegal buckets of the other crosses. Illegal bucket samples for cross products are excluded even if they are included in other cross coverage bins of the enclosing **cross**. Illegal bucket samples have precedence over ignore bucket samples (see 17.6.1) and are excluded from the ignore bucket even if the ignore bin *with expression* renders `true` for the sample.

```
coverspec trans_cov(transaction t) {
   burst_type : coverpoint t.burst_type;
   burst_len : coverpoint t.burst_len {
      bins small_burst [1..4]:1;
      illegal_bins burst_length with (burst_len == 2);
   }
   burst_type_len : cross burst_type, burst_len {
      illegal_bins burst_type_length_combinations
        with  !burst_type ? (burst_len > 2) : 1;
   }
}
```

Example 161—DSL: Illegal bucket 1

For Example 161, the following samples are placed in the illegal bucket.

	burst_type	burst_len
1	0	3
2	0	4

```
coverspec trans_cov(transaction t) {
   burst_type : coverpoint t.burst_type;
   burst_len : coverpoint t.burst_len {
      bins small_burst [1..4]:1;
      illegal burst_len == 2;
   }
   burst_type_len : cross burst_type, burst_len {
      illegal !burst_type ? (burst_len > 2) : 1;
   }
}
```

Example 162—DSL: Illegal bucket 2

For Example 162, the following samples are placed in the illegal bucket.

	burst_type	burst_len
1	0	3
2	0	4
3	0	2
4	1	2

17.6.3 Bins with expression

A bin's *with expression* refines which coverage samples go into the bins bucket. A *with expression* is an expression over the coverpoint identifiers and other DSL variables. Coverpoint identifiers represent the values sampled into the coverpoint bins. All samples that render the *with expression* `true` are placed in the bins bucket. Coverpoint identifiers have the type of the target variable that they monitor.

with expression can be added to coverpoints or crosses. A **coverpoint** *with expression* places samples for that coverpoint into the coverpoint bin bucket. Any **cross**es using the **coverpoint** also result in those samples being placed in the crosses bins buckets. A *with expression* in a **cross** places the relevant samples to the cross in the crosses bin bucket and does not change the buckets of the other crosses and other bins in that cross. See also Example 163.

```
burst_type : coverpoint t.burst_type;
burst_len : coverpoint t.burst_len {
   bins small_burst [1..4]:1;
   bins burst_length with (burst_len != 2);
}
burst_type_len : cross burst_type, burst_len {
    bins burst_type_length_combinations with
     (burst_type ? (burst_len > 2) : 1);
}
```

Example 163—DSL: Bins with expression

17.7 coverspec bins

The **bins** construct provides a way to declare a named set of values and value ranges associated with a variable (see Syntax 81).

17.7.1 DSL syntax

bins_declaration ::= **bins** identifier [*variable*_identifier] bin_specification **;**
bin_specification ::=
  bin_specifier { bin_specifier } [bin_wildcard]
  | **with (** expression **)**
bin_specifier ::=
  explicit_bin_value
  | explicit_bin_range
  | bin_range_divide
  | bin_range_size
explicit_bin_value ::= **[** constant **]**
explicit_bin_range ::= **[** constant **..** constant **]**
bin_range_divide ::= explicit_bin_range **/** constant
bin_range_size ::= explicit_bin_range **:** constant
bin_wildcard ::= **[*]**

Syntax 81—DSL: bins declaration

17.7.2 Examples

Example 164 declares a set of bins named `size_bins` on the variable named `size`. Value ranges can be declared in several ways, as described in the remainder of this section.

```
coverspec size_cs (bit [0..4095] size) {
  size_cp : coverpoint size {
    bins size_bins size [1..1022] [1025..2046] [*];
  }
}
```

Example 164—DSL: bins declaration

17.7.3 Explicit value and range grouping

Example 165 shows examples of value (**[** *x* **]**) and range grouping (**[** *x* **..** *y***]**). Individual bins are declared for values `1`, `2`, and `3`. Two value-range bins are declared that contain values `4..1022` and `1025..4095`.

```
coverspec size_cs (bit [0..4095] size) {
    size_cp : coverpoint size {
        bins size_bins [1] [2] [3] [4..1022] [1025..4095];
    }
}
```

Example 165—DSL: Explicit value and range grouping

17.7.4 Value range divide operator (/)

The value range divide operator (/) splits a range of values into *N* value ranges. When the specified value range does not evenly divide into *N* value ranges, the remaining values are placed in the final bin.

Example 166 shows how to use `/` to split value ranges. The value range `0..1000` is split into `4` bins, while the value range `1001..4095` is split into `8` bins.

```
coverspec size_cs (bit [0..4095] size) {
   size_cp : coverpoint size {
      bins size_bins [0..1000]/4 [1001..4095]/8;
   }
}
```

Example 166—DSL: Defining bins with the divide operator

17.7.5 Value range size operator (:)

The value range size operator (**:**) splits a range of values into ranges of size *N*. When the specified value range does not split evenly into bins of size *N*, the final bin gets the remaining values (and will be smaller than *N*).

Example 167 shows how to use `:` to define bins. The value range `0..1000` is split into bins of size `4`, while the value range `1001..4095` is split into bins of size `8`.

```
coverspec size_cs (bit [0..4095] size) {
   size_cp : coverpoint size {
      bins size_bins [0..1000]:4 [1001..4095]:8;
   }
}
```

Example 167—DSL: Defining bins with the size operator

17.7.6 Wildcard bin (*)

The wildcard bin (*****) collects all un-binned values in the domain of the target variable.

Example 168 shows how to use `*` to set up a wildcard bin. The values `0..4000` are explicitly binned, while the values `4001..4095` are un-binned and, therefore, placed in the wildcard bin.

```
coverspec size_cs (bit [0..4095] size) {
   size_cp : coverpoint size {
      bins size_bins [0..1000] [1001..4000] [*];
   }
}
```

Example 168—DSL: Using the wildcard bin

18. Type extension

Type extensions 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, or aspects, which 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 as follows.

— 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 element 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 Clause 19).

18.1 Specifying type extensions

Composite and enumerated 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 and effect as those introduced in the initial definition. The overall definition of any given type in a model is the sum-total of its definition statements—the initial one along with any active extension. The semantics of extensions is that of weaving all those statements into a single definition.

An extension statement explicitly specifies the kind of type being extended: **struct**, **action**, **component**, or **enum**, which needs to agree with the type reference (see Syntax 82 or Syntax 83). It does not reiterate modifiers of the type declaration, such as the object kind or base type. See also 19.1.

18.1.1 DSL syntax

```
extend_stmt ::=
    extend action type_identifier { { action_body_item } } [ ; ]
  | extend struct type_identifier { { struct_body_item } } [ ; ]
  | extend enum type_identifier { [ enum_item { , enum_item } ] } [ ; ]
  | extend component type_identifier { { component_body_item } } [ ; ]
```

Syntax 82—DSL: type extension

18.1.2 C++ syntax

In C++, extension classes derives 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 82 is shown in Syntax 83.

pss::extend_structure

Defined in `pss/extend.h` (see C.16).

```
template < class Foundation, class Extension >
class extend_structure
```

Extend a structure.

pss::extend_action

Defined in `pss/extend.h` (see C.16).

```
template < class Foundation, class Extension >
class extend_action
```

Extend an action.

pss::extend_component

Defined in `pss/extend.h` (see C.16).

```
template < class Foundation, class Extension >
class extend_component
```

Extend a component.

pss::extend_enum

Defined in `pss/extend.h` (see C.16).

```
template < class Foundation, class Extension >
class extend_enum
```

Extend an enum.

Syntax 83—C++: type extension

18.1.3 Examples

Examples of type extension are shown in Example 169 and Example 170.

```
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 169—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;
};

class additional_config_pkg : public package { ...
  // declare an enum extension for base type config_modes_e
  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 != config_modes_ext_e::MODE_D};
  };
  // register action extension
  extend_action<uart_c::configure, configure_ext>
    extend_action_configure_ext;
};
type_decl<additional_config_pkg> additional_config_pkg_decl;
```

Example 170—C++: Type extension

18.1.4 Compound type extensions

Any kind of member declared in the context of the initial definition of a compound type can be declared in the context of an extension, as per its entity category (**struct**, **action**, or **component**).

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 cannot 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 fields with same name declared in other packages. Members declared in a different package are accessible if the declaring action is imported into the scope of the accessing package or component, given that the reference is unique.

In Example 171 and Example 172, 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 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[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 171—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;
};

class soc_config_pkg : public package { ...
  class memcpy_ext : public mem_ops_c::memcpy { ...
    using mem_block_tag_e = mem_ops_c::mem_block_tag_e;
    // introducing new attribute
    constraint c { // layering additional constraint
      in { src_buff->mem_block,
        range<mem_block_tag_e>(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>(mem_block_tag_e::SYS_MEM)
                              (mem_block_tag_e::A_MEM)
                              (mem_block_tag_e::DDR) },
      if_then { ta_width < 4,
        dst_buff->mem_block != mem_block_tag_e::A_MEM
      }
    };
  };
  extend_action<memcpy_ext, mem_ops_c::memcpy> memcpy_ext_decl;
};
type_decl<soc_config_pkg> soc_config_pkg_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"};
    // constraining an attribute introduced in an extension
    constraint c { cpy1->ta_width == cpy2->ta_width };
    activity a {
      repeat { 3,
        parallel { cpy1, cpy2 }
      };
    };
  };
  type_decl<test> test_decl;
};
```

Example 172—C++: Action type extension

18.1.5 Enum type extensions

Enumerated types can be extended in one or more package contexts, introducing new items to the domain of all variables of that type. Each item in an **enum** type shall be associated with a numeric value that is unique across the initial definition and all the extensions of the type. Item values are assigned according to the same rules they would be if the items occurred all in the initial definition scope, according to the order of package evaluations. An explicit conflicting value assignment shall be illegal.

Any **enum** item can be referenced within the **package** or **component** in which it was introduced. Outside that scope, enum items can be references if the context package or component imports the respective scope.

In Example 173 and Example 174, an `enum` type is initially declared empty and later extended in two independent `packages`. Ultimately items are referenced from a `component` that imports both `packages`.

```
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::*;
                         // explicitly importing the package grants
   import soc_config_pkg::*; // access to enum items

   action dma_test {

      activity {
         do dma_c::mem2mem_xfer with {
            src_buff.mem_block == A_MEM;
            dst_buff.mem_block == DDR;
         };
      }
   }
}
```

Example 173—DSL: Enum type extensions

```
class mem_defs_pkg : public package { ... // reusable definitions
  PSS_ENUM(mem_block_tag_e, enumeration); // initially empty

  class mem_buff_s : public buffer { ...
    rand_attr<mem_block_tag_e> mem_block {"mem_block"};
  };
};
type_decl<mem_defs_pkg> mem_defs_pkg_decl;

class dma_c : public component { ...
  class mem2mem_xfer : public action { ...
    input <mem_buff_s> src_buf {"src_buff"};
    output <mem_buff_s> dst_buf {"dst_buff"};

    rand_attr<mem_defs_pkg::mem_buff_s> src_buff { "src_buff" };
    rand_attr<mem_defs_pkg::mem_buff_s> dst_buff { "dst_buff" };
  };
  type_decl<mem2mem_xfer> mem2mem_xfer_decl;
};

class AB_subsystem_pkg : public package { ...
  PSS_EXTEND_ENUM(mem_block_tag_e_ext,
                  mem_defs_pkg::mem_block_tag_e, A_MEM, B_MEM);
  };
  type_decl<AB_subsystem_pkg> AB_subsystem_pkg_decl;

class soc_config_pkg : public package { ...
  PSS_EXTEND_ENUM(mem_block_tag_e_ext,
                  mem_defs_pkg::mem_block_tag_e, SYS_MEM, DDR);
};
type_decl<soc_config_pkg> soc_config_pkg_decl;

class dma_c_ext : public dma_c { ...

  class dma_test : public action { ...
    action_handle<dma_c::mem2mem_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 174—C++: Enum type extensions

18.1.6 Ordering of type extensions

Multiple type extensions of the same type can be coded independently, and be integrated and weaved 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 blocks* of the same kind.
- Constraint override between **constraint** declarations with identical name.
- Numeric 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.

18.2 Overriding types

The **override** block (see Syntax 84 or Syntax 85) allows type and instance-specific replacement of the declared type of a field with some specified sub-type.

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 is illegal.

18.2.1 DSL syntax

overrides_declaration ::= **override** **{** { override_stmt } **}**
override_stmt ::=
  type_override
  | instance_override
type_override ::= **type** identifier **with** *type*_identifier **;**
instance_override ::= **instance** hierarchical_id **with** identifier **;**

Syntax 84—DSL: override declaration

18.2.2 C++ syntax

The corresponding C++ syntax for Syntax 84 is shown in Syntax 85.

pss::override_type

Defined in `pss/override.h` (see C.23).

```
template < class Foundation, class Override >
class override_type;
```

Override declaration.

Syntax 85—C++: override declaration

18.2.3 Examples

Example 175 and Example 176 combine type- and instance-specific overrides with type extension. Action `reg2axi_top` specifies all `axi_write_action` instances need to 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 all instances of `axi_write_action` need to be instances of `axi_write_action_x4`, which supersedes the override in `reg2axi_top`. In addition, action `reg2axi_top_x` specifies 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 reg2axi_top {
  override {
    type axi_write_action with axi_write_action_x;
    instance xlator.axi_action with axi_write_action_x2;
  }

  xlator_action   xlator;
  activity {
    repeat (10) {
      xlator; // override applies equally to all 10 traversals
    }
  }
};
action reg2axi_top_x : reg2axi_top {
  override {
    instance xlator.axi_action with axi_write_action_x3;
  }
};
```

Example 175—DSL: Type 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 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_instance<axi_write_action_x3>
    _override_inst_2{xlator->axi_action};
};
type_decl<reg2axi_top_x> reg2axi_top_x_decl;
```

Example 176—C++: Type overrides

19. 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 declared in their scope. Dependencies between sets of definitions, type declarations, and type extensions are declared in terms of **package**s using the **import** statement (see Syntax 86 or Syntax 87). From a namespace point of view, **package**s and **component**s have the same meaning and use (see also 9.4). Note that both **component**s and **package**s are top-level scopes and cannot be further enclosed in other **component**s and **package**s. However, in contrast to **component**s, **package**s cannot be instantiated, and cannot contain attributes, sub-component instances, or concrete **action** definitions.

Definitions statements that do not occur inside the lexical scope of a **package** or **component** declaration are implicitly associated with the predefined default package, called `main`. Package `main` is imported by all user-defined packages without the need for an explicit **import** statement.

NOTE—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.

19.1 Package declaration

Type declarations and type extensions (of **action**s, **struct**s, and **enum**erated types) are associated with exactly one package. This association is explicitly expressed by enclosing these definitions in a **package** statement (see Syntax 86 or Syntax 87), either directly or indirectly when they occur in the lexical scope of a **component** definition.

19.1.1 DSL syntax

```
package_declaration ::= package package_identifier { { package_body_item } } [ ; ]
package_body_item ::=
    abstract_action_declaration
  | struct_declaration
  | enum_declaration
  | coverspec_declaration
  | function_decl
  | import_class_decl
  | function_qualifiers
  | export_action
  | typedef_declaration
  | import_stmt
  | extend_stmt
import_stmt ::= import package_import_pattern ;
package_import_pattern ::= type_identifier [ :: * ]
```

Syntax 86—DSL: package declaration

The following also apply.

Types whose declaration does not occur in the scope of a **package** statement are implicitly associated with package `main`.

19.1.2 C++ syntax

The corresponding C++ syntax for Syntax 86 is shown in Syntax 87.

pss::package

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

```
class package;
```

Base class to declare a package.

Member functions

`package ( const scope& name )` : constructor

Syntax 87—C++: package declaration

19.1.3 Examples

For examples of package usage, see 20.2.7.

19.2 Namespaces and name resolution

PSS types shall have unique names in the context of their **package**, but types can have the same name if declared inside different packages. Types need to be referenced when they are instantiated as fields, extended, or inherited from by another type. In all these cases, a qualified name of the type can be used, in the format `package-name :: type-name`.

Unqualified type names can be used in the following cases.

— When referencing a type that was declared in the same **package**.

— When referencing a type that was declared in a **package** that was **import**ed by the context package.

In the case of name/namespace ambiguity, precedence is given to the current package; otherwise, explicit qualification is required.

19.3 Import statement

import statements declare a dependency between the context package and other packages. If package B imports package A, it guarantees that the definitions of package A are available and in effect when the code of B is loaded or activated. It also allows unqualified references from B to types declared in A in those cases where the resolution is unambiguous. **import** statements need to come first in the **package**'s definitions. See also *import_stmt* in 19.1.

19.4 Naming rules for members across extensions

Names of type members introduced in a type extension shall be unique in the context of the specific extension. In the case of multiple extensions of the same type in the scope of the same package, the names shall be unique across the entire package. Members are always accessible in the declaring **package**, taking precedence over members with the same name declared in other packages. Members declared in a different package are accessible if the declaring **action** is imported in that package and given that the reference is unique. See also 18.1.

20. Test realization

A PSS model interacts with external foreign-language code for two reasons. First, external code, such as reference models and checkers, is used to help compute stimulus values or expected results during stimulus generation. Second, code, such as application programming interfaces (APIs) of the SUT or utility libraries, corresponds to the behavior represented by of leaf-level actions.

Code used to help compute stimulus values is provided via the *procedural interface* (PI). Code used to implement the functionality of leaf-level actions can be provided via the PI or as *target-template code blocks* that are embedded in **action** or **struct** declarations within the PSS model. In either case, the construct for specifying the mapping of a PSS entity to its foreign-language implementation is called an *exec block*.

20.1 exec blocks

*exec block*s provide a mechanism for declaring specific functionality associated with a **component** or **action** (see Syntax 88 or Syntax 89). As discussed in 9.5, **init** *exec block*s allow component data fields to be assigned a value as the component tree is being elaborated. There are a number of additional *exec block* kinds that are used to specify the mapping of PSS scenario entities to their non-PSS implementation.

— **body** *exec block*s specify the actual runtime implementation of atomic actions.

— **pre_solve** and **post_solve** *exec block*s of **action**s 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.

20.1.1 DSL syntax

```
exec_block_stmt ::=
    exec_block
  | target_code_exec_block
  | target_file_exec_block
exec_block ::= exec exec_kind_identifier { { exec_body_stmt } }
exec_kind_identifier ::=
    pre_solve
  | post_solve
  | body
  | header
  | declaration
  | run_start
  | run_end
  | init
exec_body_stmt ::= expression [ assign_op expression ] ;
assign_op ::= = | += | -= | <<= | >>= | |= | &=
target_code_exec_block ::= exec exec_kind_identifier language_identifier = string ;
target_file_exec_block ::= exec file filename_string = string ;
```

Syntax 88—DSL: exec block declaration

The following also apply.

a) *exec block* content is given in one of two forms: as a sequence of PI calls or a text segment of target code parameterized with PSS attributes.

b) In either case, a single *exec block* is always mapped to implementation in one language.

c) In the case of a target-template block, the target language shall be explicitly declared; however, when using a PI, the corresponding language may vary.

20.1.2 C++ syntax

The corresponding C++ syntax for Syntax 88 is shown in Syntax 89.

pss::exec

Defined in `pss/exec.h` (see C.14).

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

Declare an exec block.

Member functions

`exec ( ExecKind kind, const std::initializer_list <detail::Attr-Common>& write_vars )` : declare in-line exec

`exec ( ExecKind kind, const std::string& language_or_file, const std::string& target_template )` : declare target template exec

`template <class... R> class exec(ExecKind kind, R&... /detail::ExecStmt/ r)` : declare native exec

`exec ( ExecKind kind, std::function<void()> genfunc )` : declare procedural-interface exec

`exec ( ExecKind kind, const std::string& language_or_file, std::function<void(std::ostream& code_stream)>genfunc)` : declare generative target-template exec

Syntax 89—C++: exec block declaration

20.1.3 Examples

In Example 177 and Example 178, the `init` exec blocks are evaluated in the following order.

a) `pss_top.s1.init`

b) `pss_top.s2.init`

c) `pss_top.init`

This results in the `component` fields having the following values.

`s1.base_addr=0x2000 (pss_top::init overwrote the value set by sub_c::init)`

`s2.base_addr=0x1000 (value set by sub_c::init)`

```
component sub_c {
  int base_addr;

  exec init {
    base_addr = 0x1000;
  }
};

component pss_top {
  sub_c s1, s2;

  exec init {
    s1.base_addr = 0x2000;
  }
}
```

Example 177—DSL: Data initialization in a component

```
class sub_c : public component { ...
  attr<int> base_addr {"base_addr"};

  exec e { exec::init,
    base_addr = 0x1000
  };
};

class pss_top : public component { ...
  comp_inst<sub_c> s1{"s1"}, s2{"s2"};

  exec e {exec::init,
     s1->base_addr = 0x2000
  };
};
```

Example 178—C++: Data initialization in a component

In Example 179 and Example 180, 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 offset `addr` 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 `top_c::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[31:0] base_addr = 0x1000;
   action A {
      exec body {
         // reference base_addr in context component
         activate(comp.base_addr + 0x16);
                         // activate() is an imported function
      }
   }
}

component pss_top {
   sub_c s1, s2;
   exec init {
      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 179—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(static_cast<sub_c*>(comp().val())->base_addr + 0x16)
    };
  };
  type_decl<A> A_decl;
};

class pss_top : public component { ...
  comp_inst<sub_c> s1{"s1"}, s2{"s2"};

  exec e {exec::init,
    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 180—C++: Accessing component data field from an action

For additional examples of *exec block* usage, see 20.5.

20.2 Implementation using a procedural interface (PI)

The PSS PI defines a mechanism by which the PSS model can interact with a foreign programming language, such as C/C++ and/or SystemVerilog. The PI is motivated by the need to reuse existing procedural descriptions, such as reference models, target SUT APIs, and utility libraries.

The PI can be used to reference external foreign-language functions via import functions (see 20.2.1). The PI can also be used to reference external foreign-language classes via import classes (see 20.7).

The PI consists of two layers: the PSS layer (declaration) and a foreign-language (definition) layer. Both layers are fully independent. This means a PSS description containing PI methods can be analyzed independent of the foreign language and the foreign-language implementation of a PI method can be analyzed independent of the PSS description.

20.2.1 Function declaration

A PI function prototype is declared in a package scope within a PSS description. The PI function prototype specifies the function name, return type, and function parameters. See also Syntax 90 or Syntax 91.

20.2.2 DSL syntax

```
function_decl ::= function method_prototype ;
method_prototype ::= method_return_type method_identifier method_parameter_list_prototype
method_return_type ::=
    void
  | data_type
method_parameter_list_prototype ::= ( [ method_parameter { , method_parameter } ] )
method_parameter ::= [ method_parameter_dir ] data_type identifier
method_parameter_dir ::=
    input
  | output
  | inout
method_parameter_list ::= ( [ expression { , expression } ] )
```

Syntax 90—DSL: PI method declaration

20.2.3 C++ syntax

The corresponding C++ syntax for Syntax 90 is shown in Syntax 91.

pss::function

Defined in `pss/function.h` (see C.17).

```
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
2) Declare a function object with no (void) return argument

Member functions

`function ( const scope &name, R result, Args... args )` : constructor with result

`function ( const scope &name, const kind a_kind )` : constructor with void result

`operator()(const T&... /detail::AlgebExpr/ params)` : operator

Syntax 91—C++: PI method declaration

20.2.4 Examples

For examples of using functions, see 20.2.7.

20.2.5 Method result

A PI method shall explicitly specify a data type or `void` as the return type of the method. Method return types are restricted to small scalar and string types. The following PSS data types are allowed for PI method return types.

— `void`
— `string`
— `chandle`
— `bool`
— `enum`
— `bit` and `int`, provided the domain of the type is `<=64` bits.

20.2.6 Method parameters

PI methods allow scalar, string, struct, and array data types to be passed and/or returned as parameters. The following PSS data types are allowed as method parameters:

— `string`
— `chandle`
— `bool`
— `enum`

— `bit` and `int`, provided the domain of the type is `<=64` bits.
— `struct`
— `array`

20.2.7 Parameter direction

By default, method parameters are input to the method. 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.

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 needs to specify a value for an output parameter.

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.

Example 181 and Example 182 declare a PI method in a `package` scope. In this case, the PI method `compute_value` returns an `int`, accepts an input value (`val`), and returns an output value via the `out_val` parameter.

```
package generic_methods {
  function int compute_value(
   int        val,
   output int out_val);
}
```

Example 181—DSL: PI method

```
class generic_methods : public package { ...

  function<result<int>(in_arg<int>, out_arg<int>) compute_value {
    "compute_value", result<int>(), in_arg<int>("val"),
                                    out_arg<int>("out_val")
  };
};
type_decl<generic_methods> generic_methods_decl;
```

Example 182—C++: PI method

20.3 PI PSS layer

The PSS side of the PI is completely independent of the foreign language in which the PI method is implemented, i.e., the semantics of a PSS PI function are independent of the foreign language in which it is implemented.

The foreign-language side of the PI specifies how PSS data types map to native data types, parameters are passed, and the return value of non-void methods is specified.

20.4 PI function qualifiers

Additional qualifiers are added to PI 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. PI function qualifiers are specified separately from the function declaration for modularity (see Syntax 92 or Syntax 93). In typical use, qualifiers are specified in an environment-specific package (e.g., a UVM environment-specific package or C-Test-specific package).

20.4.1 DSL syntax

function_qualifiers ::= **import** import_function_qualifiers **function** *type*_identifier **;**
import_function_qualifiers ::=
  method_qualifiers [*language*_identifier]
  | *language*_identifier
method_qualifiers ::=
  target
  | **solve**

Syntax 92—DSL: PI function qualifiers

20.4.2 C++ syntax

The corresponding C++ syntax for Syntax 92 is shown in Syntax 93.

pss::import_func

Defined in `pss/function.h` (see C.17).

```
enum kind { solve, target };
template<typename T> class import_func;
template<typename R, typename... Args>
class import_func<R(Args...)>; // 1
template<typename R, typename... Args>
class import_func<result<void>(Args...)>; // 2
```

1) PI import function availability with result
2) PI import function availability with no (void) result

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 93—C++: PI function qualifiers

20.4.3 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, while others are only available during test execution.

For example, reference model functions may only be available during test generation while the utility functions that program intellectual properties (IPs) may only be available during test execution.

An unqualified PI function is assumed to be available during all phases of test generation and execution. Qualifiers are specified to restrict a function's availability. PSS processing tools can use this information to ensure usage of PI functions match the restrictions of the target environment.

Example 183 and Example 184 specify function availability. Two PI 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 the way PI functions are used aligns with the restrictions of the target environment.

```
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 183—DSL: Function availability

```
class external_functions_pkg : public package { ...
  function<result<bit>(in_arg<int>)> alloc_addr {
    "alloc_addr",
      result<bit>(width(31,0)), in_arg<int>("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) )
    }
  };
};
type_decl<external_functions_pkg> external_functions_pkg_decl;

class pregen_tests_pkg : public package { ...
  import_func<result<bit>(in<int>)> alloc_addr {
    "external_functions_pkg::alloc_addr", solve
  };
  import_func<result<void>(in<bit>, in<bit>, in<bit>)>
    transfer_mem { "external_functions_pkg::transfer_mem", target
  };
};
type_decl<pregen_tests_pkg> pregen_tests_pkg_decl;
```

Example 184—C++: Function availability

20.4.4 Specifying an implementation language

The implementation language for a PSS PI 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 PI methods are implemented in C++). Explicitly specifying the implementation language using a separate statement allows different PI 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 185 and Example 186 show explicit specification of the foreign language in which the PI function is implemented. In this case, the method is implemented in C. Notice only the name of the PI function is specified and not the full function signature.

```
package known_c_methods {

    import C function generic_methods::compute_expected_value;

}
```

Example 185—DSL: Explicit specification of the implementation language

```
class known_c_methods : public package { ...
  import_func<result<void>()> compute_expected_value {
    "generic_methods::compute_expected_value", "C"
  };
};
type_decl<known_c_methods> known_c_methods_decl;
```

Example 186—C++: Explicit specification of the implementation language

20.5 Calling PI methods

PI methods are called from *exec blocks*. *exec blocks* allow a sequence of PI function calls to be specified, along with (optional) assignments to PSS variables (see *exec_body_stmt* in 20.1).

PI functions and methods can be called from the following *exec block* types.

a) **pre_solve**—valid in **action** and **struct** types. The **pre_solve** block is processed prior to solving of random-variable relationships in the PSS model. **pre_solve** *exec blocks* are used to initialize non-random variables that the solve process uses.

b) **post_solve**—valid in **action** 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.

c) **body**—valid in **action** types. The **body** block is responsible for implementing the target implementation of an **action**.

d) **run_start**—valid in **action** and **struct** types. Procedural non-time-consuming code block to be executed before any **body** block of the scenario is invoked. 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 5).

e) **run_end**—valid in **action** and **struct** types. Procedural non-time-consuming code block to be executed after all **body** blocks of the scenario are completed. 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 5).

f) **init**—valid in **component** types. The **init** block is used to assign values to component attributes and initialize foreign-language objects. Component's **init** blocks are called before the scenario's top-action's **pre_solve** is invoked in a depth-first search (DFS) post-order, i.e., bottom-up along the instance tree.

Non-**rand** fields can be assigned the result of a function call or an expression that does not involve a function call.

Example 187 and Example 188 demonstrate calling various PI functions. In this example, the `mem_segment_s` 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 PI `alloc_addr` function. `alloc_addr` is called after the solver has selected random values for the rand fields (specifically, `size` in the 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 {

  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 187—DSL: Calling PI functions

```
class external_functions_pkg : public package { ...
  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<bit>(8, 4096) ) };
    // TODO cannot see alloc_addr() instance from here
    // exec post_solve { exec::post_solve,
    // addr = alloc_addr( size )
    // };
  };
};
type_decl<external_functions_pkg> external_functions_pkg_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_decl->transfer_mem
                 ( in_buff->addr, out_buff->addr, in_buff->size )
    };
  };
  type_decl<xfer_a> xfer_a_decl;
};
```

Example 188—C++: Calling PI functions

20.6 Target-template implementation for functions

By default, functions are assumed to be implemented by foreign-language methods. When integrating with languages that are not functional in nature, such as assembly language, the implementation for functions can be provided by target-template code strings.

The target-template form of PI functions (see Syntax 94 or Syntax 95) allow non-functional languages, such as assembly, to be targeted in an efficient manner. The target-template form of PI functions are always target implementations. Variable references may only be used in expression positions. Function return values shall not be provided, i.e., only functions that return `void` are supported.

20.6.1 DSL syntax

```
import_method_target_template ::= target language_identifier
    function method_prototype =  string ;
```

Syntax 94—DSL: Target-template function implementation

20.6.2 C++ syntax

The corresponding C++ syntax for Syntax 94 is shown in Syntax 95.

pss:function

Defined in `pss/function.h` (see C.17).

```
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

2) Declare a target template with no (void) result

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 95—C++: Target-template function implementation

20.6.3 Examples

Example 189 and Example 190 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 189—DSL: Target-template function implementation

```
class thread_ops_pkg : public package { ...
  function<result<void> ( in_arg<bit>, in_arg<bit> )> do_stw { "do_stw",
    import_func::in_arg<bit> ( "val"),
    import_func::in_arg<bit> ("vaddr")
  };
};
type_decl<thread_ops_pkg> thread_ops_pkg_decl;

class thread_ops_asm_pkg : public package { ...
  function<result<void> ( in_arg<bit>, in_arg<bit> )> do_stw { "do_stw",
     "C",
     import_func::in_arg<bit> ( "val"),
     import_func::in_arg<bit> ("vaddr")
     ,
     R"(
       loadi RA {{val}}
       store RA {{vaddr}}
     )"
  };
};
type_decl<thread_ops_asm_pkg> thread_ops_asm_pkg_decl;
```

Example 190—C++: Target-template function implementation

20.7 Import classes

In addition to interfacing with external foreign-language functions, the PSS description can interface with foreign-language classes. See also Syntax 96 or Syntax 97.

20.7.1 DSL syntax

import_class_decl ::= **import class** *import_class*_identifier [import_class_extends]
  { { import_class_method_decl } **}** [**;**]

import_class_extends ::= **:** *type*_identifier { **,** *type*_identifier }

import_class_method_decl ::= method_prototype **;**

Syntax 96—DSL: Import class declaration

The following also apply.

a) **import class** methods support the same return and parameter types as **import** 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) **import class** methods are called from an *exec block* just as **import** functions are.

20.7.2 C++ syntax

The corresponding C++ syntax for Syntax 96 is shown in Syntax 97.

pss::import_class

Defined in `pss/import_class.h` (see C.18).

```
class import_class;
```

Declare an import class.

Member functions

`import_class ( const scope &name )` : constructor

Syntax 97—C++: Import class declaration

20.7.3 Examples

Example 191 and Example 192 declare two import classes. Import class `base` declares a method `base_method`, while import class `ext` extends from import class `base` and adds a method named `ext_method`.

```
import class base {
    void base_method();
}

import class ext : base {
    void ext_method();
}
```

Example 191—DSL: Import class

```
class base : public import_class { ...
  function<result<void>()> base_method { "base_method", {} };
};
type_decl<base> base_decl;

class ext : public base { ...
  function<result<void>()> ext_method { "ext_method", {} };
};
type_decl<ext> ext_decl;
```

Example 192—C++: Import class

20.8 Implementation using target-template code blocks

A target language implementation may be specified using target-template code blocks: text templates containing code templates with embedded references to fields in the PSS description. These templates are specified as a specific form of *exec blocks* inside **action** or **struct** definitions.

20.8.1 Target-template code exec block kinds

There are several kinds of target template code *exec blocks*.

a) **body** - the direct implementation of an action is a procedural code block in the target language, as specified by `exec body`. The body block of each action is invoked in its respective order during the execution of a scenario—after the body block 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 block*s of actions with the same set of scheduling dependencies logically takes place at the same time. Implementation of the standard should guarantee that *exec block*s of same-time actions take place as close as possible.

   Each body block is restricted to one target language in the context of a specific generated test. However, the same **action** may have **body** blocks in different languages under different **package**s, given that these packages are not used for the very same test.

b) **header** - 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.

c) **declaration** - specifies declarative statements used to define entities that are used by subsequent code blocks. Examples are the definition of global variables or functions.

d) **run_start** - procedural non-time-consuming code block to be executed before any **body** block of the scenario is invoked. Used typically for one-time test bring-up and configuration required by the context action or object.

e) **run_end** - procedural non-time-consuming code block to be executed after all **body** blocks of the scenario are completed. Used typically for test bring-down and post-run checks associated with the context action or object.

Multiple **exec body** constructs of the same kind are allowed for a given action or object. They are (logically) concatenated in the target file, as if they were all concatenated in the PSS source.

20.8.2 Target language

A *general identifier* serves to specify the intended target programming language of the code block. Clearly, a tool supporting PSS needs to 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 is kept. PSS does not define any specific behavior if an unrecognized *language_identifier* is encountered.

20.8.3 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 20.1) 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.

20.9 C++ in-line solve exec implementation

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 through the PSS PI.

In-line *exec blocks* (see Syntax 89) are simply pre-defined virtual member functions of the library classes (`action` and `structure`), the different flow/resource object classes (`pre_solve` and `post_solve`), and `component` (`init`). 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 need to 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 needs to be declared through an exec constructor parameter.

NOTE—In-line solve execs are not supported in PSS DSL.

Example 193 depicts an in-line `post_solve` exec. In it, a reference model for a decoder is used to compute attribute values. Notice the functions that are called here are not PSS import 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 193—C++: in-line exec

20.10 C++ generative target exec implementation

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 PI 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 PI 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.

20.10.1 Generative PI execs

Target PI execs (`body`, `run_start`, and `run_end`) can be specified in generative mode (see Syntax 98). However, `run_start` and `run_end` are restricted to pre-generation flow (see Table 5).

NOTE—This section, which describes programmatic generation of “native” *exec blocks*, is under active discussion by the working group and likely to change substantially in the next version of this specification.

20.10.1.1 C++ syntax

pss::exec

Defined in `pss/exec.h` (see C.14).

```
class exec;
```

Declare a generative procedural-interface exec.

Member functions

```
exec( ExecKind kind, std::function<void()> genfunc ) :
```

Syntax 98—C++: generative PI exec definitions

The behavioral description of PI execs is a sequence of PI function calls and assignment statements. In generative specification mode, the same C++ syntax is used as in the declarative mode, through variables references, `operator=`, and `imp_func::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 needs to record PI 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 needs to directly evaluate the right-value and left-value expressions, and perform any PSS function calls and PSS attribute assignments.

20.10.1.2 Examples

Example 194 depicts a generative PI 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.

```
class my_comp : public component { ...
  class write_multi_words : public action { ...

    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_decl->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_decl->write_word(base_addr + i*4, 0xA);
        }
      }
    };
  };
  type_decl<write_multi_words> write_multi_words_decl;
};
```

Example 194—C++: generative PI exec

Example 195 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 195—C++: Possible code generated for write_multi_words()

20.10.2 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 99); however, their use is restricted to pre-generation flow (see Table 5).

20.10.2.1 C++ syntax

pss::exec

Defined in `pss/exec.h` (see C.14).

```
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 99—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 20.6). 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-id needs to be provided.

20.10.2.2 Examples

Example 196 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 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 { ...

    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;nn";
        code<< " pstool_addr = target_alloc_mem({{num_of_bytes}});nn";

        // unroll the loop,
        for (int i=0; i < num_of_words.val(); i++) {
          code<< " *(uint32_t*)pstool_addr + " << i*4 << "= 0xA;nn";
        }
      }
    };
  };
  type_decl<write_multi_words> write_multi_words_decl;
};
```

Example 196—C++: generative target-template exec

The possible code generated for `write_multi_words()` is shown in Example 195.

20.11 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 20.2), functions that map to foreign-language procedural code using target code templates (see 20.6), and *exec blocks* where arbitrary target code templates are in-lined (see 20.8). 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 4 shows how these relate to the mapping constructs.

Table 4—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 needs to have direct procedural binding since it is executed prior to possible code generation. *exec blocks* 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 5 summarizes these rules.

The possible use of **action** and **struct** attributes differs between mapping constructs. Explicitly declared signatures of **function**s enable the type-aware exchange of values of all data types. On the other hand, free parameterization of un-interpreted 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 6 summarizes the parameter passing rules for the different constructs.

Table 5—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 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 6—Data passing supported for mapping mechanisms

	Back assignment to PSS attributes	Passing user-defined and compound data-types	Using PSS attributes in non-expression positions
Direct-mapped functions	X	X	
Target-template functions		X	
Target-template exec-blocks			X

20.12 Exported actions

Import 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 100 or Syntax 101.

20.12.1 DSL syntax

export_action ::= **export** [method_qualifiers] *action_type*_identifier
method_parameter_list_prototype **;**

Syntax 100—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 **import** functions (see 20.2.1), the exported action is assumed to always be available if the method 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 **import** function / **export** action call chains.

20.12.2 C++ syntax

The corresponding C++ syntax for Syntax 100 is shown in Syntax 101.

pss::export_action

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

```
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 101—C++: Export action declaration

20.12.3 Examples

Example 197 and Example 198 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 197—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 { mode!=0,
        in (val, range<bit>(0,10)),
        in (val, range<bit>(10,100))
      }
    };
  };
  type_decl<A1> A1_decl;
};

class pkg : public package { ...
  // Export A1, providing a mapping to field 'mode'
  export_action<comp::A1> comp_A1 {

  };
};
type_decl<pkg> pkg_decl;
```

Example 198—C++: Export action

20.12.4 Export action foreign-language binding

An exported action is exposed as a method in the target foreign language (see Example 199). 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 method.

```
namespace comp {
    void A1(unsigned char mode);
}
```

Example 199—DSL: Export action foreign-language implementation

NOTE—Foreign-language binding is the same for DSL and C++.

21. Hardware/Software Interface (HSI)

Hardware/Software Interface (HSI) is an abstraction responsible for peripheral device management. It captures the programmer's view of a peripheral device in a manner that is agnostic to the underlying verification environment and platform. Device initialization, interrupt management and other operations such as configure, transmit/receive, registration of device capabilities, etc., are all specified as part of HSI.

HSI specification is captured using a set of provided C++ API, such as that of software programmable registers, virtual registers and DMA descriptor chains, interrupt properties. This API also allows the user to specify the programming sequence for different operations that can be performed on a peripheral device.

From such an abstract representation of HSI, a concrete implementation can be derived for a given target language and verification platform. An example of such a concrete implementation can be a device driver in a bare-metal environment executing on the processor that is part of the SUT.

Using HSI specification to describe the interaction with hardware enhances portability of the stimulus model in the following ways.

— The stimulus model is abstracted from the verification platform specific implementation of HSI and, thus, can be ported to a different verification platform easily (e.g., simulation to emulation).

— The HSI specification can be based on a standard interface/API contract for a given device category. This enables the stimulus model to be ported to a different device easily.

Finally, the HSI specification can interface with the stimulus model described either in DSL or C++ syntax.

NOTE—This PSS version does not include the detailed list of APIs for capturing HSI. However, a sample HSI specification for UART is included as an informative reference (see Annex F).

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_identifier  { { package_body_item } }
    [ ; ]

package_body_item ::=
      abstract_action_declaration
    | struct_declaration
    | enum_declaration
    | coverspec_declaration
    | function_decl
    | import_class_decl
    | function_qualifiers
    | export_action
    | typedef_declaration
    | import_stmt
    | extend_stmt

import_stmt ::= import package_import_pattern ;

package_import_pattern ::= type_identifier [ :: * ]

extend_stmt ::=
      extend action type_identifier { { action_body_item } } [ ; ]
    | extend struct type_identifier { { struct_body_item } } [ ; ]
    | extend enum type_identifier { [ enum_item { , enum_item } ] } [ ; ]
    | extend component type_identifier { { component_body_item } } [ ; ]
```

B.2 Action declarations

```
action_declaration ::= action action_identifier [ action_super_spec ]
    { { action_body_item } } [ ; ]

abstract_action_declaration ::= abstract action action_identifier
    [ action_super_spec ] { { action_body_item } } [ ; ]
```

```
action_super_spec ::= : type_identifier

action_body_item ::=
      activity_declaration
    | overrides_declaration
    | constraint_declaration
    | action_field_declaration
    | symbol_declaration
    | coverspec_declaration
    | exec_block_stmt

activity_declaration ::= activity { { [ identifier : ] activity_stmt } } [ ; ]

action_field_declaration ::= [ action_field_modifier ] action_data_declaration

action_field_modifier ::=
      rand
    | io_direction
    | lock
    | share
    | action

io_direction ::=
      input
    | output
```

Exec blocks

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

exec_block ::= exec exec_kind_identifier { { exec_body_stmt } }

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

exec_body_stmt ::= expression [ assign_op expression ] ;

assign_op ::= = | += | -= | <<= | >>= | |= | &=

target_code_exec_block ::= exec exec_kind_identifier
    language_identifier = string ;

target_file_exec_block ::= exec file filename_string = string ;
```

B.3 Struct declarations

```
struct_declaration ::= struct_type identifier
   [ : struct_identifier ] { { struct_body_item } } [ ; ]

struct_type ::=
      struct
    | struct_qualifier

struct_qualifier ::=
      buffer
    | stream
    | state
    | resource

struct_body_item ::=
      constraint_declaration
    | struct_field_declaration
    | typedef_declaration
    | coverspec_declaration
    | exec_block_stmt

struct_field_declaration ::= [ struct_field_modifier ] data_declaration

struct_field_modifier ::= rand
```

B.4 Procedural interface (PI)

```
function_decl ::= function method_prototype ;

method_prototype ::= method_return_type method_identifier
   method_parameter_list_prototype

method_return_type ::=
      void
    | data_type

method_parameter_list_prototype ::= ( [ method_parameter
   { , method_parameter } ] )

method_parameter ::= [ method_parameter_dir ] data_type identifier

method_parameter_dir ::=
      input
    | output
    | inout

function_qualifiers ::= import import_function_qualifiers
   function type_identifier ;

import_function_qualifiers ::=
      method_qualifiers [ language_identifier ]
    | language_identifier
```



```
method_qualifiers ::=
      target
    | solve

import_method_target_template ::= target language_identifier
   function method_prototype = string ;

method_parameter_list ::= ( [ expression { , expression } ] )
```

B.4.1 Import class declaration

```
import_class_decl ::= import class import_class_identifier
   [ import_class_extends ] { { import_class_method_decl } } [ ; ]

import_class_extends ::= : type_identifier { , type_identifier }

import_class_method_decl ::= method_prototype ;
```

B.4.2 Export action

```
export_action ::= export [ method_qualifiers ] action_type_identifier
   method_parameter_list_prototype ;
```

B.5 Component declarations

```
component_declaration ::= component component_identifier
   [ : component_super_spec ] { { component_body_item } } [ ; ]

component_super_spec ::= : type_identifier

component_body_item ::=
      overrides_declaration
    | component_field_declaration
    | action_declaration
    | object_bind_stmt
    | inline_type_object_declaration
    | exec_block
    | package_body_item

component_field_declaration ::=
      component_data_declaration
    | component_pool_declaration

component_data_declaration ::= 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 ::=
      component_path
    | { component_path { , component_path } }
```

```
component_path ::=
      component_identifier { . component_path_elem }
    | *

component_path_elem ::=
      component_action_identifier
    | *

inline_type_object_declaration ::= pool [ [ expression ] ] struct_qualifier
   struct identifier [ : struct_identifier ] { { struct_body_item } } [ ; ]
```

B.6 Activity statements

```
activity_stmt ::=
      activity_if_else_stmt
    | activity_repeat_stmt
    | activity_constraint_stmt
    | activity_foreach_stmt
    | activity_action_traversal_stmt
    | activity_sequence_block_stmt
    | activity_select_stmt
    | activity_parallel_stmt
    | activity_schedule_stmt
    | activity_bind_stmt

activity_if_else_stmt ::= if ( expression ) activity_stmt [ else activity_stmt ]

activity_repeat_stmt ::=
      repeat while ( expression ) activity_sequence_block_stmt
    | repeat ( [ identifier : ] expression ) activity_sequence_block_stmt
    | repeat activity_sequence_block_stmt [ while ( expression ) ; ]

activity_sequence_block_stmt ::= [ sequence ] { { activity_labeled_stmt } }

activity_constraint_stmt ::=
      constraint { { constraint_body_item } }
    | constraint single_stmt_constraint

activity_foreach_stmt ::= foreach ( expression ) activity_sequence_block_stmt

activity_action_traversal_stmt ::=
       identifier [ inline_with_constraint ]
    | do type_identifier [ inline_with_constraint ] ;

inline_with_constraint ::=
     with { { constraint_body_item } }
    | with constant_expression

activity_select_stmt ::= select { activity_labeled_stmt activity_labeled_stmt
       { activity_labeled_stmt } }

activity_labeled_stmt ::= [ identifier : ] activity_stmt

activity_parallel_stmt ::= parallel { { activity_labeled_stmt } } [ ; ]
```

```
activity_schedule_stmt ::= schedule { { activity_labeled_stmt } } [ ; ]

activity_bind_stmt ::= bind hierarchical_id activity_bind_item_or_list ;

activity_bind_item_or_list ::=
      hierarchical_id
    | { hierarchical_id { , hierarchical_id } }

symbol_declaration ::= symbol identifier [ ( symbol_paramlist ) ]
    = activity_stmt

symbol_paramlist ::= [ symbol_param { , symbol_param } ]

symbol_param ::= data_type identifier
```

B.7 Overrides

```
overrides_declaration ::= override { { override_stmt } }

override_stmt ::=
      type_override
    | instance_override

type_override ::= type identifier with type_identifier ;

instance_override ::= instance hierarchical_id with identifier ;
```

B.8 Data declarations

```
data_declaration ::= data_type data_instantiation { , data_instantiation } ;

action_data_declaration ::= action_data_type data_instantiation
    { , data_instantiation } ;

data_instantiation ::= identifier [ ( coverspec_portmap_list ) ] [ array_dim ]
    [ = constant_expression ]

coverspec_portmap_list ::= [
      coverspec_portmap { , coverspec_portmap }
    | hierarchical_id { , hierarchical_id } ]

coverspec_portmap ::= . identifier ( hierarchical_id )

array_dim ::= [ constant_expression ]
```

B.9 Data types

```
data_type ::=
      scalar_data_type
    | user_defined_datatype

action_data_type ::=
      scalar_data_type
    | user_defined_datatype
    | action_type

scalar_data_type ::=
      chandle_type
    | integer_type
    | string_type
    | bool_type

chandle_type ::= chandle

integer_type ::= integer_atom_type
    [ [ expression [ : expression ] ] ]
    [ in [ open_range_list ] ]

integer_atom_type ::=
      int
    | bit

open_range_list ::= open_range_value { , open_range_value }

open_range_value ::=
      expression [ .. expression ]
    | expression ..
    | .. expression
    | expression

string_type ::= string

bool_type ::= bool

user_defined_datatype ::= type_identifier

action_type ::= type_identifier

struct_type ::= type_identifier

enum_declaration ::= enum enum_identifier { [ enum_item { , enum_item } ] } [ ; ]

enum_item ::= identifier [ = constant_expression ]

typedef_declaration ::= typedef data_type identifier ;
```

B.10 Constraint

```
constraint_declaration ::=
      [ dynamic ] constraint identifier { { constraint_body_item } }
```

```
      | constraint { { constraint_body_item } }
      | constraint single_stmt_constraint

constraint_body_item ::=
        expression_constraint_item
      | foreach_constraint_item
      | if_constraint_item
      | unique_constraint_item

expression_constraint_item ::=
        expression implicand_constraint_item
      | expression ;

implicand_constraint_item ::= -> constraint_set

constraint_set ::=
        constraint_body_item
      | constraint_block

constraint_block ::= { { constraint_body_item } }

foreach_constraint_item ::= foreach ( expression ) constraint_set

if_constraint_item ::= if ( expression ) constraint_set [ else constraint_set ]

unique_constraint_item ::= unique { hierarchical_id { , hierarchical_id } } ;

single_stmt_constraint ::=
        expression_constraint_item
      | unique_constraint_item

scheduling_constraint ::= constraint ( parallel | sequence )
    { hierarchical_id, hierarchical_id { , hierarchical_id } };
```

B.11 Coverspec

```
coverspec_declaration ::= coverspec identifier ( coverspec_port
    { , coverspec_port } ) { { coverspec_body_item } } [ ; ]

coverspec_port ::= data_type identifier

coverspec_body_item ::=
        coverspec_option
      | coverspec_coverpoint
      | coverspec_cross
      | constraint_declaration

coverspec_option ::= option . identifier = constant_expression ;

coverspec_coverpoint ::=
        coverpoint_identifier : coverpoint coverpoint_target_identifier
            { { coverspec_coverpoint_body_item } }[ ; ]
      | coverpoint_identifier : coverpoint coverpoint_target_identifier ;
```

```
coverspec_coverpoint_body_item ::=
      coverspec_option
    | coverspec_coverpoint_binspec

coverspec_coverpoint_binspec ::=
      bins_keyword identifier bin_specification
    | bins_keyword identifier hierarchical_id ;

bins_keyword ::=
      bins
    | ignore_bins
    | illegal_bins
coverspec_cross ::=
      ID : cross coverpoint_identifier { , coverpoint_identifier }
          { { coverspec_cross_body_item } }
    | ID : cross coverpoint_identifier { , coverpoint_identifier } ;

coverspec_cross_body_item ::= coverspec_option
```

Bins

```
bins_declaration ::= bins identifier [ variable_identifier ] bin_specification
   ;

bin_specification ::=
      bin_specifier { bin_specifier } [ bin_wildcard ]
    | with ( expression )

bin_specifier ::=
      explicit_bin_value
    | explicit_bin_range
    | bin_range_divide
    | bin_range_size

explicit_bin_value ::= [ constant ]

explicit_bin_range ::= [ constant .. constant ]

bin_range_divide ::= explicit_bin_range / constant

bin_range_size ::= explicit_bin_range : constant

bin_wildcard ::= [ * ]
```

B.12 Expression

```
constant_expression ::= expression

expression ::= condition_expr

condition_expr ::= logical_or_expr { ? logical_or_expr : logical_or_expr }

logical_or_expr ::= logical_and_expr { || logical_and_expr }

logical_and_expr ::= binary_or_expr { && binary_or_expr }
```

```
binary_or_expr ::= binary_xor_expr { | binary_xor_expr }

binary_xor_expr ::= binary_and_expr { ^ binary_and_expr }

binary_and_expr ::= logical_equality_expr { & logical_equality_expr }

logical_equality_expr ::= logical_inequality_expr { eq_neq_op
    logical_inequality_expr }

logical_inequality_expr ::= binary_shift_expr {logical_inequality_rhs}

logical_inequality_rhs ::=
      inequality_expr_term
    | inside_expr_term

inequality_expr_term ::= logical_inequality_op binary_shift_expr

logical_inequality_op ::=
     < | <= | > | >=

inside_expr_term ::=
    in [ open_range_list ] }

binary_shift_expr ::= binary_add_sub_expr { shift_op binary_add_sub_expr }

binary_add_sub_expr ::= binary_mul_div_mod_expr { add_sub_op
    binary_mul_div_mod_expr }

binary_mul_div_mod_expr ::= binary_exp_expr { mul_div_mod_op binary_exp_expr }

binary_exp_expr ::= unary_expr { ** unary_expr }

unary_expr ::= [ unary_op ] primary

unary_op ::= + | - | ! | ~ | & | | | ^

primary ::=
      number
    | bool_literal
    | paren_expr
    | string
    | variable_ref_path
    | method_function_call

paren_expr ::= ( expression )

variable_ref_path ::= variable_ref { .variable_ref }

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

method_function_call ::=
      method_call
    | function_call

method_call ::= hierarchical_id method_parameter_list

function_call ::= ID [:: ID [:: ID]] method_parameter_list
```

```
mul_div_mod_op ::= * | / | %

add_sub_op ::= + | -

shift_op ::= << | >>

eq_neq_op ::= == | !=
```

B.13 Identifiers and literals

```
constant ::=
      number
    | identifier

identifier ::=
      ID
    | ESCAPED_ID

hierarchical_id ::= identifier { . identifier }

action_type_identifier ::= type_identifier

type_identifier ::= ID { :: ID }

package_identifier ::= hierarchical_id

coverpoint_target_identifier ::= hierarchical_id

action_identifier ::= identifier

struct_identifier ::= identifier

component_identifier ::= identifier

component_action_identifier ::= identifier

coverpoint_identifier ::= identifier

enum_identifier ::= identifier

import_class_identifier ::= identifier

language_identifier ::= identifier

method_identifier ::= identifier

variable_identifier ::= identifier

exec_kind_identifier ::= identifier

filename_string ::= DOUBLE_QUOTED_STRING

bool_literal ::=
      true
    | false
```

B.14 Numbers

```
number::=
      based_hex_number
    | based_dec_number
    | based_bin_number
    | based_oct_number
    | dec_number
    | oct_number
    | hex_number

based_hex_number ::= [ DEC_LITERAL ] BASED_HEX_LITERAL

DEC_LITERAL ::= [1-9] {[0-9]|_}

BASED_HEX_LITERAL ::= ' [s|S] h|H [0-9]|[a-f]|[A-F] {[0-9]|[a-f]|[A-F]|_}

based_dec_number ::= [ DEC_LITERAL ] BASED_DEC_LITERAL

BASED_DEC_LITERAL ::= ' [s|S] d|D [0-9] {[0-9]|_}

based_bin_number ::= [ DEC_LITERAL ] BASED_BIN_LITERAL

BASED_BIN_LITERAL ::= ' [s|S] b|B [0-1] {[0-1]|_}

based_oct_number ::= [ DEC_LITERAL ] BASED_OCT_LITERAL

BASED_OCT_LITERAL ::= ' [s|S] o|O [0-7] {[0-7]|_}

dec_number ::= DEC_LITERAL

oct_number ::= OCT_LITERAL

OCT_LITERAL ::= 0 [0-7]

hex_number ::= HEX_LITERAL

HEX_LITERAL ::= 0x [0-9]|[a-f]|[A-F] {[0-9]|[a-f]|[A-F]|_}
```

B.15 Comments

```
SL_COMMENT ::= //{any_ASCII_character_except_newline}\n

ML_COMMENT ::= /*{any_ASCII_character}*/

string  ::=
      DOUBLE_QUOTED_STRING
    | TRIPLE_DOUBLE_QUOTED_STRING

DOUBLE_QUOTED_STRING ::= " {\|!\|"} "

TRIPLE_DOUBLE_QUOTED_STRING ::= """{any_ASCII_character}"""

ID ::= [a-z]|[A-Z]|_ {[a-z]|[A-Z]|_|[0-9]}

ESCAPED_ID ::= \{any_ASCII_character_except_whitespace} whitespace
```

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/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/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/action_handle.h"
#include "pss/action_attr.h"
#include "pss/pool.h"
#include "pss/bind.h"
#include "pss/exec.h"
#include "pss/function.h"
#include "pss/import_class.h"
#include "pss/export_action.h"
#include "pss/package.h"
#include "pss/extend.h"
#include "pss/override.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/activityStmt.h"
#include "pss/detail/sharedExpr.h"
namespace pss {
  class component; // forward declaration
  /// Declare an action
  class action : public detail::ActionBase {
  protected:
    /// Constructor
    action ( const scope& s );
    /// Destructor
    ~action();
  public:
    rand_attr<component*>& 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::ActivityStmt */ r);
       // Constructor
       activity(std::vector<detail::ActivityStmt>&& stmts );
       // Destructor
       ~activity();
     };
    // select() must be inside action declaration to disambiguate from
    // built-in select()
    /// Declare a select statement
    class select : public detail::ActivityStmt {
    public:
      template < class... R >
      select(R&&... /* detail::ActivityStmt */ r);
      select(std::vector<detail::ActivityStmt>&& stmts );
    };
    /// Declare a sequence block
    class sequence : public detail::ActivityStmt {
    public:
      // Constructor
      template < class... R >
      sequence(R&&... /* detail::ActivityStmt */ r);
      sequence(std::vector<detail::ActivityStmt>&& stmts );
    };
    /// Declare a schedule block
    class schedule : public detail::ActivityStmt {
    public:
      // Constructor
      template < class... R >
      schedule(R&&... /* detail::ActivityStmt */ r);
```

```
    schedule(std::vector<detail::ActivityStmt>&& stmts );
  };
  /// Declare a parallel block
  class parallel : public detail::ActivityStmt {
  public:
    // Constructor
    template < class... R >
    parallel(R&&... /* detail::ActivityStmt */ r);
    parallel(std::vector<detail::ActivityStmt>&& stmts );
  };
  /// Declare a repeat statement
  class repeat : public detail::ActivityStmt {
  public:
    /// Declare a repeat statement
    repeat(const detail::AlgebExpr& count,
           const detail::ActivityStmt& activity
    );
    /// Declare a repeat statement
    repeat(const attr<int>& iter,
           const detail::AlgebExpr& count,
           const detail::ActivityStmt& activity
    );
  };
  /// Declare a repeat while statement
  class repeat_while : public detail::ActivityStmt {
  public:
    /// Declare a repeat while statement
    repeat_while(const detail::AlgebExpr& cond,
                 const detail::ActivityStmt& activity
    );
  };
  /// Declare a do while statement
  class do_while : public detail::ActivityStmt {
  public:
    /// Declare a do while statement
    do_while( const detail::ActivityStmt& activity,
              const detail::AlgebExpr& cond
    );
  };
 }; // 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<bit>& a_range);
```

```
      /// Constructor defining width and range
      action_attr (const scope& name, const width& a_width,
                   const range<bit>& 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);
    action_handle<T> with ( detail::AlgebExpr expr );
    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/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);
  /// 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 scalar int
template <>
class attr<int> : public detail::AttrIntBase {
public:
  /// Constructor
  attr (const scope& s);
  /// Constructor with initial value
  attr (const scope& s, const int& init_val);
  /// Constructor defining width
  attr (const scope& s, const width& a_width);
  /// Constructor defining width and initial value
  attr (const scope& s, const width& a_width, const int& init_val);
  /// Constructor defining range
  attr (const scope& s, const range<int>& a_range);
  /// Constructor defining range and initial value
  attr (const scope& s, const range<int>& a_range, const int& init_val);
  /// Constructor defining width and range
 attr (const scope& s, const width& a_width, const range<int>& a_range);
 /// Constructor defining width and range and initial value
  attr (const scope& s, const width& a_width, const range<int>& a_range,
        const int& 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 scalar bit
template <>
class attr<bit> : public detail::AttrBitBase {
  public:
  /// Constructor
  attr (const scope& s);
  /// Constructor with initial value
  attr (const scope& s, const bit& init_val);
```

```
  /// Constructor defining width
  attr (const scope& s, const width& a_width);
  /// Constructor defining width and initial value
  attr (const scope& s, const width& a_width, const bit& init_val);
  /// Constructor defining range
  attr (const scope& s, const range<bit>& a_range);
  /// Constructor defining range and initial value
  attr (const scope& s, const range<bit>& a_range, const bit& init_val);
  /// Constructor defining width and range
  attr (const scope& s, const width& a_width, const range<bit>& a_range);
  /// Constructor defining width and range and initial value
  attr (const scope& s, const width& a_width, const range<bit>& a_range,
        const bit& 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 scalar string
template <>
class attr<std::string> : public detail::AttrStringBase {
public:
  /// Constructor
  attr (const scope& s);
  /// Constructor and initial value
  attr (const scope& s, const std::string& 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 scalar bool
template <>
class attr<bool> : public detail::AttrBoolBase {
public:
  /// Constructor
  attr (const scope& s);
  /// Constructor and initial value
  attr (const scope& s, const bool 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);
  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 scalar 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<int>& 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<int>& a_range);
  /// Constructor creating array from list of elements
  attr( std::initializer_list<attr<int>> values );
  /// 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<bit>& 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<bit>& a_range);
  /// Constructor creating array from list of elements
  attr( std::initializer_list<attr<bit>> values );
  /// 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 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<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
    bind ( const std::initializer_list<detail::IOBase>& 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();
   };
}; // namespace pss
```

C.12 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 an unnamed dynamic member constraint
    template <class... R> dynamic_constraint (
      const R&&... /*detail::AlgebExpr*/ expr
      );
    /// 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.13 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.14 File pss/exec.h

```
#pragma once
#include <functional>
#include "pss/detail/execBase.h"
#include "pss/detail/attrCommon.h"
namespace pss {
  /// Declare an exec block
  class exec : public detail::ExecBase {
  public:
     /// Types of exec blocks
    enum ExecKind {
      run_start,
      header,
      declaration,
      init,
      pre_solve,
      post_solve,
      body,
      run_end,
      file
    };
    /// Declare in-line 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
    template < class... R >
```

```
    exec(
      ExecKind kind,
      R&&... /* detail::ExecStmt */ r
      );
    /// Declare generative procedural-interface exec
    exec(
      ExecKind kind,
      std::function<void()> genfunc // shadowed by variadic template c'tor
                                    // handle at construction time
      );
    /// Declare generative target-template exec
    exec(
      ExecKind kind,
      std::string&& language_or_file,
      std::function<void(std::ostream&)> genfunc
                                    // shadowed by variadic template c'tor
                                    // handle at construction time
      );
  };
}; // namespace pss
#include "pss/timpl/exec.t"
```

C.15 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 width &w, const range<bit> &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 width &w, const range<int> &rng) {};
  };
}
```

C.16 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.17 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"

namespace pss {

  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
    );

  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
    );
};

template<typename T> class import_func;

template<typename R, typename... Args>
class import_func<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
      );

    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
      );
  };

  template<typename... Args>
  class import_func<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 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 width &w, const range<bit> &rng);
  };

  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 width &w, const range<int> &rng);
  };

  /// 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 width &w, const range<bit> &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 width &w, const range<int> &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 width &w, const range<bit> &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 width &w, const range<int> &rng);
  };

  /// Template specialization for results
  template <> class result<bit> : public detail::FunctionResult {
  public:
    result();
    result(const width &w);
    result(const width &w, const range<bit> &rng);
  };

  template <> class result<int> : public detail::FunctionResult {
  public:
    result();
    result(const width &w);
    result(const width &w, const range<int> &rng);
  };

  template <> class result<void> : public detail::FunctionResult {
  public:
    result();
  };
}
#include "pss/timpl/function.t"
```

C.18 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.19 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<int>& a_range
    );
    in ( const attr<bit>& a_var,
           const range<bit>& a_range
    );
    in ( const rand_attr<int>& a_var,
             const range<int>& a_range
    );

    in ( const rand_attr<bit>& a_var,
             const range<bit>& a_range
    );
    template < class T>
    in ( const rand_attr<T>& a_var,
             const range<T>& a_range
    );
 };
}; // namespace pss
#include "pss/timpl/in.t"
```

C.20 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.21 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.22 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.23 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.24 File pss/package.h

```
#pragma once
#include <memory>
#include "pss/detail/packageBase.h"
#include "pss/scope.h"
namespace pss {
  /// Declare a PSS package
  class package : public detail::PackageBase {
  protected:
    /// constructor
    package (const scope& s);
    ~package();
   };
}; // namespace pss
```

C.25 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.26 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/component.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/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);
    /// 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 scalar 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<int>& a_range);
    /// Constructor defining width and range
    rand_attr (const scope& name, const width& a_width, const range<int>&
              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 scalar 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<bit>& a_range);
    /// Constructor defining width and range
    rand_attr (const scope& name, const width& a_width, const range<bit>&
               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 scalar rand string
  template <>
  class rand_attr<std::string> : public detail::RandAttrStringBase {
  public:
    /// Constructor
    rand_attr (const scope& name);
    /// 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 scalar 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);
  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 scalar rand component*
template <>
class rand_attr<component*> : public detail::RandAttrCompBase {
public:
  /// Copy constructor
  rand_attr(const rand_attr<component*>& other);
  /// Access to underlying data
  component* val();
};
/// 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<int>& 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<int>& a_range);
  /// Constructor creating array from list of elements
  rand_attr( std::initializer_list<rand_attr<int>> values );
  /// 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<bit>& 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<bit>& a_range);
```

```
    /// Constructor creating array from list of elements
    rand_attr( std::initializer_list<rand_attr<bit>> values );
    /// 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 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<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.27 File pss/range.h

```
#pragma once
#include <vector>
#include "pss/detail/rangeBase.h"
namespace pss {
  /// Declare domain of a numeric scalar attribute
  template <class T = int>
  class range : public detail::RangeBase {
  public:
    /// Declare a range of values
    range (const T& lhs, const T& rhs);
    /// Declare a single value
    range (const T& value);
    /// Copy constructor
    range ( const range <T>& a_range);
    /// Function chaining to declare another range of values
    range& operator() (const T& lhs, const T& rhs);
    /// Function chaining to declare another single value
    range& operator() (const T& value);
  }; // class range
}; // namespace pss
#include "pss/timpl/range.t"
```

C.28 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<bit>& instance_id();
      /// In-line exec block
      virtual void pre_solve();
      /// In-line exec block
      virtual void post_solve();
    };
  }; // namespace pss
```

C.29 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.30 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.31 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();
  };
}; // namespace pss
```

C.32 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.33 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.34 File pss/symbol.h

```
namespace pss {
  namespace detail {
    class ActivityStmt; // forward reference
  };
  using symbol = detail::ActivityStmt;
};
```

C.35 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.36 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 a 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.37 File pss/vec.h

```
#pragma once
#include <vector>
namespace pss {
  template < class T>
  using vec = std::vector <T>;
};
```

C.38 File pss/width.h

```
#pragma once
#include "pss/detail/widthBase.h"
namespace pss {
  /// Declare width of a numeric scalar 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.39 File pss/detail/activityStmt.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 {
  namespace detail {
    class ActivityStmt {
    public:
      /// Recognize action_handle<>
      template<class T>
      ActivityStmt(const action_handle<T>& value);
      /// Recognize action_attr<>
      template<class T>
      ActivityStmt(const action_attr<T>& value);
      /// Recognize dynamic_constraint
      ActivityStmt(const dynamic_constraint& value);
```

```
      /// Recognize shared constructs
      ActivityStmt(const SharedExpr& other);
      // Default Constructor
      ActivityStmt();
    };
  }; // namespace detail
}; // namespace pss
#include "activityStmt.t"
```

C.40 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/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
  template <class T> class result; // forward declaration
  namespace detail {
    /// Construction of algebraic expressions
    class AlgebExpr : public ExprBase {
    public:
      /// Default constructor
      AlgebExpr();
      /// 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()
      template < class T >
      AlgebExpr(const range<T>& value);
      /// Recognize a comp_inst<>
      template < class T >
      AlgebExpr(const comp_inst<T>& value);
      // /// 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);
  }; // namespace detail
}; // namespace pss
#include "algebExpr.t"
```

C.41 File pss/detail/FunctionParam.h

```
#pragma once
namespace pss {
  namespace detail {
    class FunctionParam {
    };
  }; // namespace detail
}; // namespace pss
```

C.42 File pss/detail/FunctionResult.h

```
#pragma once
namespace pss {
  namespace detail {
    class FunctionResult {
```

```
    };
  }; // namespace detail
}; // namespace pss
```

Annex D

(normative)

Foreign-language data type bindings

PSS specifies data type bindings to C/C++ and SystemVerilog.

D.1 C primitive types

The mapping between the PSS primitive types and C types used for method parameters is specified in Table D1.

Table D1—Mapping PSS primitive types and C types

PSS type	C type Input	C type Output / Inout
string	**const char ***	**char ****
bool	**unsigned int**	**unsigned int ***
chandle	**void ***	**void ****
bit (1-8-bit domain)	**unsigned char**	**unsigned char ***
bit (9-16-bit domain)	**unsigned short**	**unsigned short ***
bit (17-32-bit domain)	**unsigned int**	**unsigned int ***
bit (33-64-bit domain)	**unsigned long long**	**unsigned long long ***
int (1-8-bit domain)	**char**	**char ***
int (9-16-bit domain)	**short**	**short ***
int (17-32-bit domain)	**int**	**int ***
int (33-64-bit domain)	**long long**	**long long ***

The mapping for return types matches the first two columns in Table D1.

D.2 C++ composite and user-defined types

C++ is seen by the PSS standard as a primary language in the PSS domain. The PSS standard covers the projection of PSS arrays, enumerated 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. PSS language can be used in conjunction with C++ code without tool-specific dependencies.

D.2.1 Built-in types

a) C++ type mapping for primitive numeric types is the same as that for ANSI 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).

d) C++ mapping of a PSS array is `std::vector` of the C++ mapping of the respective element type (using the default allocator class).

D.2.2 User-defined types

In PSS, the user can define data-types of two categories: **enum**erated types and **struct** types (including flow/ resource objects). These types require mapping to C++ types if they are used as parameters in C++ `import function` calls.

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.

D.2.2.1 Naming 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 scope. 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 name of the C++ mapped type is the same as that in PSS.

D.2.2.2 Enumerated types

PSS enumerated types are mapped to C++ enumerated types, with the same set of items in the same order and identical names. When specified, explicit numeric constant values for an enumerated 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.

In PSS, as in C++, enumerated item identifiers shall be unique in the context of the enclosing namespace (**package**/**component**).

D.2.2.3 Struct types

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).

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 PI declarations:

```
import void foo(derived_s d);
import solve CPP foo;
```

with the corresponding PSS definitions:

```
struct base_s {
   int[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 signatures.

D.2.3 Parameter passing semantics

When C++ import functions are called, primitive data types are passed by value for input parameters and otherwise by pointer, as in the ANSI C case. In contrast, compound data-type values, including strings, arrays, structs, and actions, are passed as C++ references. Input parameters of compound data-types are passed as **const** references, while output and inout parameters are passed as non-**const** references. In the case of `output` and `inout` compound parameters, if a different memory representation is used for the PSS tool vs. C++, the inner state needs to be copied in upon calling it and any change shall be copied back out onto the PSS entity upon return.

For example, the following **import** declaration:

```
import void foo(my_struct s, output int arr[]);
```

corresponds to the following C++ declaration:

```
extern "C" void foo(const my_struct& s, std::vector<int>& arr);
```

Statically 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 their original size upon return, filling the default values of the respective element type as needed.

D.3 SystemVerilog

Table D2 specifies the type mapping between PSS types and SystemVerilog types for both the parameter and return types.

Table D2—Mapping PSS primitive types and SystemVerilog types

PSS type	SystemVerilog type
string	**string**
bool	**boolean**
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**

A **struct** type used in a PI method call is directly reflected to SystemVerilog as a class hierarchy.

Annex E

(informative)

Solution space

Once a PSS model has been specified, the elements of the model need to be processed in some way to ensure that resulting scenarios accurately reflect the specified behavior(s). This annex describes the steps a processing tool may take to analyze a portable stimulus description and create a (set of) scenario(s).

a) Identify root action:
   1) Specified by the user.
   2) Unless otherwise specified, the designated root action shall 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 in parallel.
         2. If actions scheduled in parallel 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 comprise 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.

See also Clause 16.

Annex F

(informative)

HSI UART example

This is a sample HSI specification for a UART.

Pc16550_intr.h:

```
// Specifies the interrupts generated by PC16550

class pc16550_intr_line : public pss::intr_line {
   public:
      // Modem status
      pss::intr_event ModemStat;

      // Tx Queue Empty
      pss::intr_event TxRegEmpty;

      // Timeout
      pss::intr_event TimeOut;

      //Rx Data Available
      pss::intr_event RxDataAv;

      //Rx Line Stat
      pss::intr_event RxLineStat;

   public:
      pc16550_intr_line(pss::module_name n) : pss::intr_line(n),
      ModemStat("ModemStat"),
      TxRegEmpty("TxRegEmpty"),
      TimeOut("TimeOut"),
      RxDataAv("RxDataAv"),
      RxLineStat("RxLineStat") {
      }
};
```

Pc16550_reg.h:

```
// Register details

class RBR_reg : public pss::reg {
   public:
      using pss::reg::operator=;
      RBR_reg(pss::module_name n) : pss::reg(n) {
         description("Receive buffer
   register").offset(0x0).width(8).access(pss::PSS_ACCESS_RO).reset(0x0);
      }
};

class THR_reg : public pss::reg {
   public:
      using pss::reg::operator=;
      THR_reg(pss::module_name n) : pss::reg(n) {
```

```
            description("Transmit holding
      register").offset(0x4).width(8).access(pss::PSS_ACCESS_WO).reset(0x0);
         }
   };

   class IER_reg : public pss::reg {
      public:
         pss::field erbfi;
         pss::field etbei;
         pss::field elsi;
         pss::field edssi;
      public:
         using pss::reg::operator=;
         IER_reg(pss::module_name n) : pss::reg(n), erbfi("erbfi"),
      etbei("etbei"), elsi("elsi"), edssi("edssi") {
            description("Interrupt enable
      register").offset(0x8).width(8).access(pss::PSS_ACCESS_RW).reset(0x0);
            erbfi.bit_span(0, 0).description("Enable Receive Data Available
      Interrupt").clearing(pss::PSS_CMODE_NONE);
            etbei.bit_span(1, 1).description("Enable Transmitter Holding
      Register Empty Interrupt").clearing(pss::PSS_CMODE_NONE);
            elsi.bit_span(2, 2).description("Enable Receiver Line Status
      Interrupt").clearing(pss::PSS_CMODE_NONE);
            edssi.bit_span(3, 3).description("Enable Modem Status
      Interrupt").clearing(pss::PSS_CMODE_NONE);
         }
   };

   class IIR_reg : public pss::reg {
      public:
         pss::field intpend;
         pss::field intid;
         pss::field fifoenbd;
      public:
         using pss::reg::operator=;
         IIR_reg(pss::module_name n) : pss::reg(n), intpend("intpend"),
      intid("intid"), fifoenbd("fifoenbd") {
            description("Interrupt Identification
      register").offset(0xC).width(8).access(pss::PSS_ACCESS_RO).reset(0x1);
            intpend.bit_span(0, 0).description("Interrupt
      Pending").clearing(pss::PSS_CMODE_NONE);
            intid.bit_span(1, 3).description("Interrupt
      ID").clearing(pss::PSS_CMODE_NONE);
            fifoenbd.bit_span(6, 7).description("FIFO
      Enable").clearing(pss::PSS_CMODE_NONE);
         }
   };

   class DLL_reg : public pss::reg {
      public:
         pss::field dll;
      public:
         using pss::reg::operator=;
         DLL_reg(pss::module_name n) : pss::reg(n), dll("dll") {
            description("Device Latch Least Significant
      Byte").offset(0x10).width(8).access(pss::PSS_ACCESS_RW).reset(0x0);
            dll.bit_span(0, 7).description("Lower 8 bits of divisor
      DLAB").clearing(pss::PSS_CMODE_NONE);
         }
```

```
    };

    class DLM_reg : public pss::reg {
        public:
            pss::field dlm;
        public:
            using pss::reg::operator=;
            DLM_reg(pss::module_name n) : pss::reg(n), dlm("dlm") {
                description("Device Latch Most Significant
        Byte").offset(0x14).width(8).access(pss::PSS_ACCESS_RW).reset(0x0);
                dlm.bit_span(0, 7).description("Higher 8 bits of divisor
        DLAB").clearing(pss::PSS_CMODE_NONE);
            }
    };

    class LCR_reg : public pss::reg {
        public:
            pss::field wls;
            pss::field stb;
            pss::field pen;
            pss::field eps;
            pss::field dlab;
        public:
            using pss::reg::operator=;
            LCR_reg(pss::module_name n) : pss::reg(n), wls("wls"), stb("stb"),
        pen("pen"), eps("eps"), dlab("dlab") {
                description("Line Control
        Register").offset(0x18).width(8).access(pss::PSS_ACCESS_RW).reset(0x0);
                wls.bit_span(0, 1).description("Word Select
        Length").clearing(pss::PSS_CMODE_NONE);
                stb.bit_span(2, 2).description("Number of stop
        bits").clearing(pss::PSS_CMODE_NONE);
                pen.bit_span(3, 3).description("Parity Enable
        Bit").clearing(pss::PSS_CMODE_NONE);
                eps.bit_span(4, 4).description("Even Parity
        Select").clearing(pss::PSS_CMODE_NONE);
                dlab.bit_span(7, 7).description("Divisor Latch Access
        Bit").clearing(pss::PSS_CMODE_NONE);
            }
    };

    class FCR_reg : public pss::reg {
        public:
            pss::field fifoenb;
        public:
            using pss::reg::operator=;
            FCR_reg(pss::module_name n) : pss::reg(n), fifoenb("fifoenb") {
                description("Fifo Control
        Register").offset(0x1C).width(8).access(pss::PSS_ACCESS_WO).reset(0x0);
                fifoenb.bit_span(0, 0).description("Fifo
        Enable").clearing(pss::PSS_CMODE_NONE);
            }
    };

    class pc16550_reg_group : public pss::reg_group {
        public:
            RBR_reg RBR;
            THR_reg THR;
            IER_reg IER;
```

```
      IIR_reg IIR;
      DLL_reg DLL;
      DLM_reg DLM;
      LCR_reg LCR;
      FCR_reg FCR;
      /* ... */

   public:
      pc16550_reg_group(pss::module_name n) : pss::reg_group(n),
      RBR("RBR"),
      THR("THR"),
      IER("IER"),
      IIR("IIR"),
      DLL("DLL"),
      DLM("DLM"),
      LCR("LCR"),
      FCR("FCR")
   { }
};
```

Pc16550.h:

```
#include "pc16550_reg.h"
#include "pc16550_intr.h"

enum InterruptStatus
   {MODEMSTAT = 0x0, TXREGEMPTY = 0x1, TIMEOUT = 0x6, RXDATAV = 0x2, RXLINESTAT
   = 0x3};

class UartConfig : public pss::item {
   public:
      UartConfig(const pss::module_name &n) : pss::item(n),
            word_length("word_length"),
            stop_bit_length("stop_bit_length"),
            parity("parity"),
            baud_rate("baud_rate"),
            device_clock("device_clock"),
            enable_fifo("enable_fifo"),
            fifo_th("fifo_th")
            { }
   public:
      pss::target_var<int> word_length;
      pss::target_var<int> stop_bit_length;
      pss::target_var<int> parity;
      pss::target_var<int> baud_rate;
      pss::target_var<int> device_clock;
      pss::target_var<int> enable_fifo;
      pss::target_var<int> fifo_th;
};

class pc16550 : public pss::hsi
{
   public:
      pc16550(pss::module_name n);
      void reset(void);
      void build(void);
      void configure(UartConfig config);
      void configure_fifo(pss::target_var<int> enable_fifo);
      void enable_transmit(void);
```

```
        void start_receive(void);
        void register_functions(void);
    private:
        pc16550_reg_group pc16550_reg;
        pc16550_intr_line pc16550_intr;
        pss::fifo<int> RcvFifo;
       pss::target_function<pss::target_var<void>> enable_tx_handle;
};
```

Pc16550.cpp:

```
#include <sstream>
#include "pss.h"
#include "pc16550.h"

void pc16550::reset(void)
{
    pc16550_reg.RBR = 0;
    pc16550_reg.THR = 0;
    pc16550_reg.IER = 0;
}

void pc16550::build(void)
{
    pc16550_intr.ModemStat
        .pre_clear(1)
        .clear(pss::PSS_CMODE_COR)
        .event_type(pss::PSS_STATUS)
        .enable(PSS_ANON_FUNC({pc16550_reg.IER.edssi = 1;}))
        .disable(PSS_ANON_FUNC({pc16550_reg.IER.edssi = 0;}))
        .get_status(PSS_EXPR(pc16550_reg.IIR.intid == MODEMSTAT));

    pc16550_intr.TxRegEmpty
        .pre_clear(1)
        .clear(pss::PSS_CMODE_AUTO)
        .event_type(pss::PSS_WRITE)
        .enable(PSS_ANON_FUNC({pc16550_reg.IER.etbei = 1;}))
        .disable(PSS_ANON_FUNC({pc16550_reg.IER.etbei = 0;}))
        .get_status(PSS_EXPR(pc16550_reg.IIR.intid == TXREGEMPTY));

    pc16550_intr.TimeOut
        .pre_clear(1)
        .clear(pss::PSS_CMODE_AUTO)
        .event_type(pss::PSS_ERROR)
        .enable(PSS_ANON_FUNC({pc16550_reg.IER.erbfi = 1;}))
        .disable(PSS_ANON_FUNC({pc16550_reg.IER.erbfi = 0;}))
        .get_status(PSS_EXPR(pc16550_reg.IIR.intid == TIMEOUT));

    pc16550_intr.RxDataAv
        .pre_clear(1)
        .clear(pss::PSS_CMODE_AUTO)
        .event_type(pss::PSS_READ)
        .enable(PSS_ANON_FUNC({pc16550_reg.IER.erbfi = 1;}))
        .disable(PSS_ANON_FUNC({pc16550_reg.IER.erbfi = 0;}))
        .get_status(PSS_EXPR(pc16550_reg.IIR.intid == RXDATAV));

    pc16550_intr.RxLineStat
        .pre_clear(1)
```

```
        .clear(pss::PSS_CMODE_AUTO)
        .event_type(pss::PSS_STATUS);

    RcvFifo
        .enable(PSS_ANON_FUNC(pc16550_reg.FCR.fifoenb = 0x1;));
}

void pc16550::enable_transmit(void)
{
    pc16550_reg.IER.etbei = 1;
}

void pc16550::start_receive(void)
{
    pc16550_reg.IER.erbfi = 1;
}

void pc16550::configure_fifo(pss::target_var<int> enable_fifo)
{
    pss_if((enable_fifo == 1), PSS_ANON_FUNC({pc16550_reg.FCR.fifoenb = 1;}),
    PSS_ANON_FUNC({pc16550_reg.FCR.fifoenb = 0;}));
}

void pc16550::configure(UartConfig Config)
{
    pss::target_var<int> Divisor("Divisor");

    pc16550_reg.LCR.wls = Config.word_length;
    pc16550_reg.LCR.stb = Config.stop_bit_length;
    pc16550_reg.LCR.pen = 0x1;
    pc16550_reg.LCR.eps = Config.parity;

    //Baud rate setting.
    Divisor = Config.device_clock + 16;
    pc16550_reg.LCR.dlab = 1;
    pc16550_reg.DLL = Divisor + 0x00ff;
    pc16550_reg.DLM = Divisor + 8 + 0x00ff;
    pc16550_reg.LCR.dlab = 0;

    pss_if((Config.enable_fifo == 1), PSS_ANON_FUNC({pc16550_reg.FCR.fifoenb =
    1;}), PSS_ANON_FUNC({pc16550_reg.FCR.fifoenb = 0;}));

    // Enable Receive
    start_receive();
    // Enable Transmit
    enable_tx_handle();
}

pc16550::pc16550(pss::module_name n) : pss::hsi(n),
                                            pc16550_reg("pc16550_reg"),
                                            pc16550_intr("pc16550_intr"),
                                            RcvFifo("RcvFifo",
    pss::PSS_READ_FIFO),

    enable_tx_handle("enable_tx_handle")
{ }

void pc16550::register_functions(void)
{
```



```
    hsi::register_functions();
    register_target_function(&pc16550::configure_fifo, this,
    "configure_fifo", "API to configure FIFO",
    pss::target_var<int>("enable_fifo"));
    register_target_function(&pc16550::start_receive, this, "start_receive",
    "enables the reception of data");
    enable_tx_handle = register_target_function(&pc16550::enable_transmit,
    this, "enable_transmit", "enables the transmission of data");
    register_target_function(&pc16550::configure, this, "configure", "API to
    configure different features of Uart",
            UartConfig("config"));
}

int main(int argc, char *argv[])
{
    pc16550 device("pc16550");
    device.register_functions();

    return pss::main(argc, argv);
};
```